UNIVERSIDADE FEDERAL DO ABC

CECS

Centro de Engenharia, Modelagem e Ciências Sociais Aplicadas

LUCAS GRIMAUTH EVANGELISTA

ESTUDO DAS REDES ADVERSÁRIAS GENERATIVAS: GERAÇÃO E RESTAURAÇÃO DE IMAGENS

Trabalho apresentado ao curso Engenharia de Informação como requisito para obtenção do diploma do curso.

Orientador: Murilo Bellezoni Loiola.

Santo André
2021

SUMÁRIO

1. RESUMO

Considerando a importância da inteligência artificial nos dias de hoje, as redes neurais adversárias generativas são um tipo de rede revolucionária por apresentarem resultados consistentes, eficientes e de boa qualidade devido ao seu modelo adversarial, onde duas redes, uma geradora e uma discriminadora, competem entre si de forma a otimizar os resultados gerados. No presente trabalho, foi utilizada uma arquitetura que combina redes neurais convolucionais e redes neurais adversárias generativas voltada para o objetivo de gerar novas imagens semelhantes a imagens já existentes e restaurar imagens incompletas, gerando novas imagens completas a partir delas.

2. INTRODUÇÃO

> *"In the first half of the 20th century, science fiction familiarized the world with the concept of artificially intelligent robots. It began with the "heartless" Tin man from the Wizard of Oz and continued with the humanoid robot that impersonated Maria in Metropolis. By the 1950s, we had a generation of scientists, mathematicians, and philosophers with the concept of artificial intelligence (or AI) culturally assimilated in their minds. One such person was Alan Turing, a young British polymath who explored the mathematical possibility of artificial intelligence. Turing suggested that humans use available information as well as reason in order to solve problems and make decisions, so why can't machines do the same thing? This was the logical framework of his 1950 paper, Computing Machinery and Intelligence in which he discussed how to build intelligent machines and how to test their intelligence*[1]."

A inteligência artificial está presente no cotidiano de grande parte das pessoas, ajudando motoristas e pedestres a encontrarem melhores rotas até certos destinos, mostrando produtos aos usuários relacionados às suas atividades na internet, além dos assistentes virtuais de grandes empresas, como o Google Assistant no Android e o Watson da IBM, e sistemas de segurança com câmeras inteligentes que conseguem disparar alarmes se detectarem alguma atividade suspeita.

A humanidade está na era da "*big data*", onde vários dados são coletados e é difícil para as pessoas processarem essa enorme quantidade de dados por si mesmas. Além das artes, a inteligência artificial já deixou seus frutos nas áreas de tecnologia, marketing, entretenimento e bancária, desde os carros capazes de dirigirem a si mesmos, até IAs capazes de criarem trailers de um filme[2].

Estes, entre outros, são alguns exemplos de como a inteligência artificial está tomando cada vez mais espaço e importância no cotidiano e o presente trabalho visa experimentar com uma técnica de inteligência artificial em especial: redes adversárias generativas (GAN, do inglês *Generative Adversarial Network*).

A GAN é um modelo de rede neural que tem uma arquitetura com dois sub-modelos: um modelo gerador, responsável por gerar novos exemplos de dados com base em dados escolhidos e um modelo discriminador, responsável por

comparar os exemplos gerados pelo gerador com dados de treinamento e assim decidir se estes são válidos ou não. De forma sucinta, a rede geradora precisa competir contra a rede adversária, discriminadora, que tenta distinguir os dados que vieram dos dados de treinamento e os que foram gerados pela própria rede.

Assim, ambas as redes são treinadas a cada rodada: a discriminadora é atualizada para uma melhor separação entre dados reais e gerados e a geradora é atualizada dependendo do quão bem os dados gerados enganaram a rede discriminadora. Um exemplo de analogia: a rede discriminadora é a dona de uma loja e a rede geradora é um cliente que gera notas falsas. Cabe à discriminadora distinguir entre as notas reais e falsas e cabe à geradora produzir notas mais próximas do real possível.

Por conseguinte, a GAN é capaz de criar, por exemplo, obras de artes com características semelhantes aos de um artista escolhido, cujas obras são os dados a serem usados pela GAN. De forma interessante, uma obra de arte que foi criada por uma inteligência artificial seguindo o modelo de uma GAN foi vendida em outubro de 2018[3], um passo importante na inserção da inteligência artificial nas artes.

Outra aplicação, em que as GANs tiveram participação, é o *Inpainting*[4], onde a rede é capaz de editar uma imagem de forma a preencher espaços vazios, remover certos objetos, entre outros objetivos. Há diversos exemplos que mostram a importância e utilidade desta aplicação como: restauração de obras de arte, mostrando como certos espaços que foram desgastados com o tempo deveriam ser preenchidos; remoção de partes indesejáveis em uma imagem, uma aplicação bastante utilizada em aplicativos de edição de imagem, e recuperação de imagens corrompidas, podendo ser até utilizada para reparar filmes. Logo, considerando a importância da inteligência artificial nos dias de hoje e as diversas aplicações dela, o estudo das GANs é relevante.

3. OBJETIVO

O presente trabalho tem como objetivo o estudo das Redes Adversárias Generativas utilizando imagens. Assim, utilizando a linguagem Python e o programa open-source TensorFlow, foi utilizado um programa de forma a mostrar uma implementação de uma GAN com base em imagens escolhidas de um certo domínio e, assim, revelar as imagens geradas a partir dessas imagens e restaurar imagens incompletas de forma similar ao processo de *inpainting*, porém gerando imagens completas a partir de imagens incompletas.

Adicionalmente, acompanhar a aprendizagem dos sub-modelos gerador e discriminador durante a geração e restauração das imagens. Com este estudo, é possível compreender as diversas capacidades das GANs e como o uso delas pode ser útil em diferentes aplicações no cotidiano das pessoas, como melhorar a resolução de imagens e restaurar imagens corrompidas.

4. REDES ADVERSÁRIAS GENERATIVAS

4.1. História

As GANs foram introduzidas por Ian Goodfellow em 2014[5] quando escreveu um artigo propondo a idéia de fazer duas redes neurais, uma geradora e uma discriminadora, competirem, fazendo com que ambas melhorem seus métodos até que as amostras geradas sejam quase indistinguíveis das amostras reais. A partir daí, houveram diversas modificações e inovações feitas com GANs e alguns exemplos serão citados a seguir.

Considerando que as Redes Neurais Convolucionais (CNN, do inglês *Convolutional Neural Networks*) são um tipo de rede neural apropriado para análise de imagens[6], a implementação delas com GANs deu origem às DCGANs (*Deep Convolutional Generative Adversarial Networks*), utilizadas amplamente até hoje na geração e manipulação de imagens.

Mais tarde, as ProGAN[7] (*Progressive growing of Generative Adversarial Networks*) foram desenvolvidas visando estabilizar o aprendizado das redes neurais nas GANs na geração de imagens. Iniciando o processo com uma imagem de baixa resolução, são acrescentadas camadas na rede neural conforme a resolução da imagem aumenta durante o treinamento.

Em seguida, as CycleGANs[8] (*Cycle-Consistent Adversarial Networks*) foram desenvolvidas para realizar traduções entre imagens sem par de diferentes domínios, ou seja, propõe um método para descobrir como as características de uma imagem podem ser aplicadas em uma outra imagem de outro domínio sem exemplos de imagens pareadas para treinamento. Um exemplo é utilizar imagens de cavalos como fontes e de zebras como alvo e fazer com que a rede neural “transforme” os cavalos em zebras.

Por fim, as StyleGANs[9] (*Style-based Generative Adversarial Networks*) foram desenvolvidas visando controlar o processo de síntese de imagens, ajustando o “estilo” da imagem em cada camada de convolução, controlando seus aspectos em diferentes escalas.

4.2. Contextualização

Existem diversas aplicações em que a geração de amostras por uma inteligência artificial é útil e as GANs destacam-se por poder gerar amostras de ótima qualidade devido ao seu modelo adversarial. Alguns exemplos são:

- Aprimoramento de imagens: algoritmos para aprimoramento de imagens existem há algum tempo, porém é possível criar amostras de melhor qualidade utilizando GANs, como pode ser observado na Figura 1. *Upsampling* é um exemplo de técnica utilizada pela rede geradora de forma a aumentar o tamanho da imagem e sua qualidade através da interpolação dos pixels, gerando novos pixels entre as linhas e colunas da imagem. Esta tarefa de estimar uma imagem de alta resolução a partir de uma imagem de baixa resolução é chamada de super resolução (SR, do inglês super resolution).

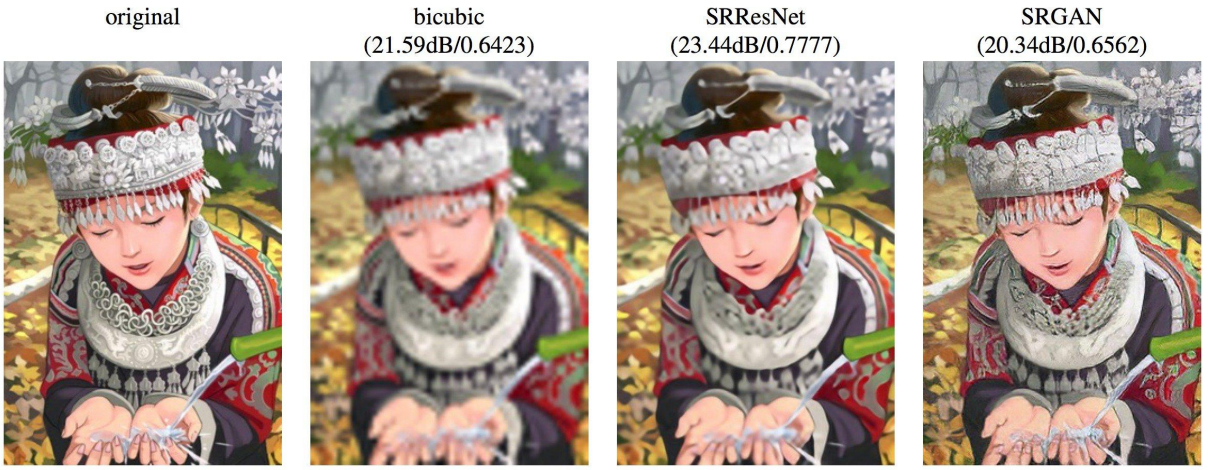

Figura 1 - Da esquerda para a direita: imagem original, imagem utilizando interpolação bicúbica (*bicubic interpolation*), imagem utilizando uma rede neural residual de super resolução (SRResNet) e a última utilizando uma GAN de super resolução (SRGAN). Nota-se, entre parênteses, o *peak signal-to-noise ratio* (PSNR) em dBs, que é um meio de avaliar e comparar algoritmos SR, e o valor da similaridade estrutural (SSIM), que é um método de prever a qualidade percebida de uma imagem ou vídeo[10].

A interpolação bicúbica é um método de interpolação de pontos de dados em uma grade bidimensional, como uma imagem. Basicamente envolve estimar certos pontos baseando-se nos outros já existentes, o que é útil no caso do

upsampling já que é necessário preencher os pixels gerados ao aumentar uma imagem. A SRResNet faz uso de uma rede de aprendizado profundo residual para reconhecimento de imagem (ResNet, do inglês *Deep Residual Learning Network*)[11] voltada para a tarefa de super resolução e a SRGAN é uma GAN voltada para a mesma tarefa de super resolução.

- Manipulação e tradução de imagens: GANs podem receber imagens simples e gerar imagens com mais detalhes partindo do que foi recebido, como por exemplo receber apenas o esboço de uma bolsa e gerar uma imagem preenchendo com cores e textura. Isso pode ser usado em profissões criativas como designers de roupas e acessórios. Outra utilização é gerar imagens mais simples contendo apenas informações essenciais, como no caso de gerar um mapa contendo apenas os contornos das ruas a partir de uma foto aérea, algo útil para aplicações de navegação em GPS, como Waze e Google Maps.
- Criação de arte: como dito anteriormente, GANs são capazes de criar arte gerando amostras a partir de outras obras.
- Criação de conteúdo em linguagem natural: robôs capazes de criar artigos, notícias, shows de TV, filmes e conteúdos no geral que conseguem capturar a atenção das pessoas ou até convencê-las a investir dinheiro é o sonho de muitos comerciantes. É uma aplicação que ainda está se desenvolvendo, mas certamente algo que chama a atenção. Um exemplo dessa aplicação é a técnica de escrita *Article Spinning* onde se gera o que parece ser novos conteúdos a partir de algo que já existe, e o interessante dessa técnica é que ela pode ser completamente automatizada. As GANs podem ajudar a produzir conteúdos mais críveis.

4.2.1. Inpainting

Como dito anteriormente, *Inpainting* é uma aplicação de redes neurais que visa preencher pixels faltantes numa imagem e possui diversos usos como restauração de imagens e remoção de objetos indesejáveis. As GANs foram utilizadas para tal aplicação por gerarem resultados consistentes, eficientes e de boa qualidade. Alguns exemplos desse uso são:

- Codificadores de Contexto (*Context Encoders*)[12]: assim como os seres humanos possuem a habilidade de enxergar o mundo e compreender imagens mesmo que alguma parte dela esteja faltando, imaginando o que poderia ser colocado nesta parte, uma rede convolucional pode ser treinada de forma a predizer as partes faltantes de uma imagem a partir dos seus arredores e seu contexto. Assim, codificadores de contexto são redes convolucionais treinadas por redes adversárias e que são capazes de gerar pixels faltantes em uma imagem de forma que faça sentido no contexto daquela imagem. Eles são formados por um codificador, que pega o contexto da imagem e passa para um decodificador, que gera a imagem preenchendo a parte faltante. Contudo, este método possui dificuldade em treinar imagens de alta resolução e a textura do buraco preenchido muitas vezes não encaixa bem no restante da imagem, podendo-se notar um artefato visível na borda do buraco na imagem. Pode-se observar um exemplo do uso de um codificador de contexto e comparações com outros métodos na figura 2.

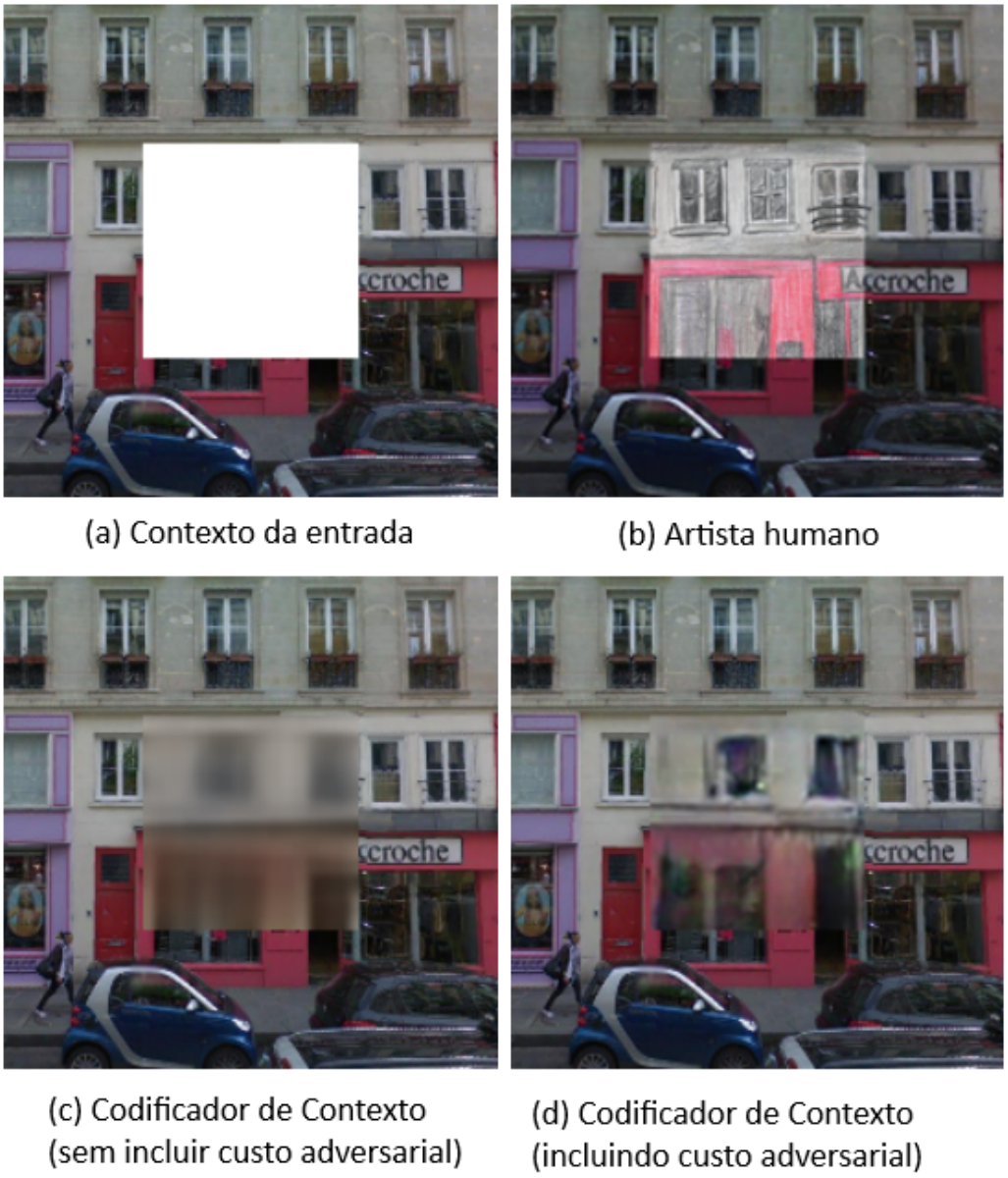

Figura 2 - No item (a) é possível observar a imagem com a região faltante. No item (b) tem-se um exemplo de *inpainting* feito por um artista humano. No item (c) foi utilizado o codificador de contexto, mas sem utilizar a rede adversarial para treinamento. No item (d) foi utilizado o codificador de contexto junto com a rede adversarial[12].

- *Multi-Scale Neural Patch Synthesis* (MSNPS)[13]: de forma a superar as limitações dos Codificadores de Contexto, as MSNPS combinaram esse método com o método de transferência de estilo (Neural Style Transfer)[14]. Primeiramente, o codificador de contexto é utilizado para obter a predição dos pixels faltantes na imagem. Depois, é utilizado o método de transferência de estilo através de uma rede de textura, assim nomeada por eles, com o

objetivo de verificar a consistência entre os pixels gerados e os pixels válidos da imagem de forma a manter a textura fluída. Também é utilizado um esquema de múltipla escala de forma que a rede consiga lidar com imagens de alta resolução, onde a imagem é reduzida e depois aumentada através de *upsampling*, passando por múltiplas redes em diferentes escalas, porém esse é um processo de pouca eficiência. Um outro problema que as MSNPS enfrentam é a dificuldade em preencher espaços faltantes em uma imagem com detalhes mais específicos e complicados. Na figura 3 é possível observar um comparativo das MSNPS com outros métodos de *Inpainting*.

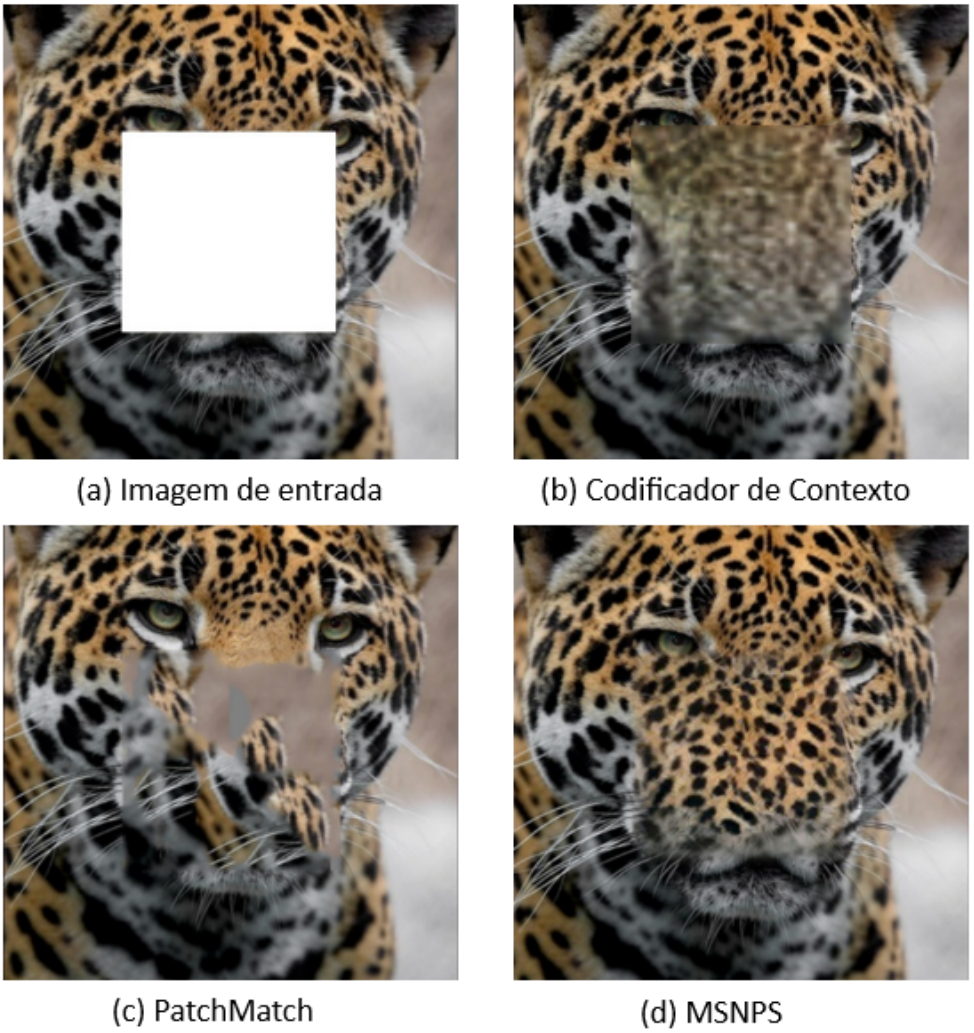

Figura 3 - No item (a) é possível observar a imagem com a região faltante. No item (b) foi utilizado um codificador de contexto. No item (c) foi utilizado um *Content-Aware Fill* usando *PatchMatch*. No item (d) foi utilizada uma MSNPS[13].

- *Globally and Locally Consistent Image Completion* (GLCIC)[15]: nos casos anteriores, a rede depende de encontrar fragmentos semelhantes na imagem a fim de preencher o vazio com esses fragmentos. Isso é plausível ao utilizar imagens, por exemplo, de paisagens naturais, onde há diversos fragmentos semelhantes. Contudo, no caso de uma face humana, por exemplo, se os olhos forem removidos, não há outro local na imagem que mostre olhos a fim de preencher esse vazio. Assim, a GLCIC mostra a importância na geração de novos fragmentos no *Inpainting*. Não é possível gerar algo que não tenha sido visto e, por isso, a base de dados de treinamento é muito importante e a GLCIC faz uso de três redes: uma inteiramente convolucional com convolução dilatada responsável por preencher a imagem, uma discriminadora focada na região faltante da imagem e outra discriminadora focada na imagem como um todo. A rede inteiramente convolucional permite que as redes de treinamento possam ser usadas para imagens de diversos tamanhos e a dinâmica entre as duas redes discriminadoras global e local, permite manter uma consistência global da imagem, enquanto mantém a qualidade da aparência do espaço preenchido.
- *Patch-Based Image Inpainting with Generative Adversarial Networks* (PGGAN-Res)[16]: este método combina uma rede ResNet com uma combinação de duas redes discriminadoras, a PatchGAN[17] e um discriminador GAN global (G-GAN) formando a rede discriminadora PGGAN. A PGGAN tem a mesma intenção das redes discriminadoras local e global da GLCIC e mostram um aprimoramento nos detalhes da textura local dos pixels gerados e melhores resultados de inpainting devido ao uso de blocos residuais dilatados pela ResNet. De modo geral, este método é bem similar ao GLCIC, porém dessa vez utilizando aprendizado residual na rede geradora e a PatchGAN como uma rede discriminadora local. Na figura 4 é possível observar um comparativo entre este método e alguns métodos mencionados anteriormente.

Figura 4 - Na primeira coluna estão as imagens de entrada com região faltante. Na segunda coluna em diante são mostrados resultados para os seguintes métodos, respectivamente: codificadores de contexto, *Globally and Locally Consistent Image Completion,* PGGAN com ResNet utilizando convoluções dilatadas (PGGAN-DRes) e PGGAN com ResNet (PGGAN-Res)[16].

4.3. Primeiros experimentos com GANs

As GANs, por utilizarem um modelo de rede geradora, são capazes de produzir conteúdo. Para ilustrar essa geração de conteúdo, Ian Goodfellow, conhecido como criador da técnica empregada nas GANs, e seus co-autores realizaram diversos

testes em conjuntos de dados de imagem diferentes de forma a mostrar as capacidades das GANs[5].

Figura 5 - Visualização de amostras geradas por GANs. Cada ilustração utiliza um conjunto diferente de dados: a) MNIST b) TFD c) CIFAR-10 (*fully connected model*) d) CIFAR-10 (*convolutional discriminator and "deconvolutional" generator*).

Os testes realizados encontram-se na Figura 5. A ilustração "a" utilizou o conjunto de dados MNIST (*Modified National Institute of Standards and Technology*), composto por imagens de dígitos escritos à mão. A ilustração "b" utilizou o conjunto de dados TFD (*Toronto Faces Dataset*), composto por imagens de rostos de diversas pessoas. As ilustrações "c" e "d" utilizaram o mesmo conjunto de dados CIFAR-10[18] (*Canadian Institute For Advanced Research*), porém utilizando técnicas diferentes nas GANs.

Em cada ilustração, a coluna destacada mais à direita contém os exemplos de treino que são mais próximos das amostras vizinhas, além disso as amostras são escolhidas de forma aleatória, demonstrando que o modelo não tem um conjunto de

dados de treinamento memorizado. De forma a conhecer melhor o funcionamento dessa geração de imagens, será apresentado a seguir a arquitetura e princípios básicos das GANs.

4.4. Arquitetura básica

As GANs possuem uma arquitetura composta por dois sub-modelos: um modelo gerador e um modelo discriminador. Pode-se visualizar essa arquitetura básica na Figura 3 e ter uma noção de seu funcionamento. Essa arquitetura pode sofrer alterações dependendo da aplicação da GAN, assim como utilizar vetores de entrada específicos ou outros métodos de treinamento da rede.

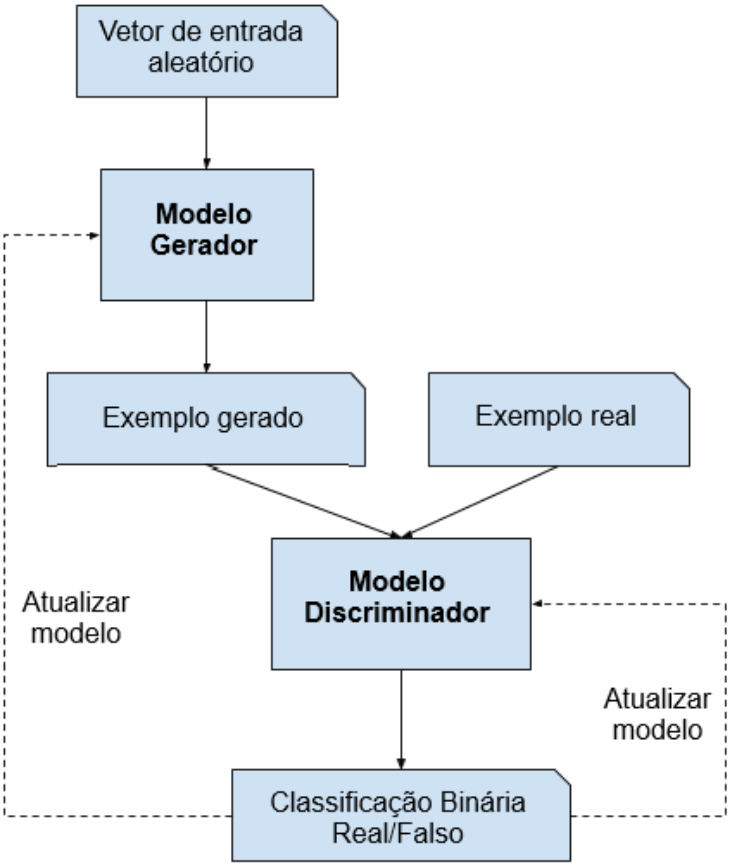

Figura 6 - Exemplo de arquitetura de uma *Generative Adversarial Network*[19].

Na Figura 6, o modelo gerador recebe um vetor aleatório de entrada para dar início ao processo de geração. Após o modelo gerador gerar exemplos, o modelo

discriminador recebe esses exemplos gerados junto com os exemplos vindo do conjunto de dados de treinamento composto por amostras “reais” das quais se pretende gerar exemplos similares. Então, o modelo discriminador separa todos os exemplos como “reais” ou “falsos” e, dependendo de como os exemplos foram classificados, os modelos são atualizados de acordo.

Se o discriminador consegue separar os exemplos perfeitamente entre reais e falsos, ele pode ser recompensado sem mudanças ao seu modelo, porém o gerador é penalizado com várias atualizações. De forma análoga, se, idealmente, o discriminador tiver uma incerteza máxima de 50% em decidir se os exemplos são reais ou falsos, então ele é penalizado com grandes atualizações e o gerador pode continuar sem mudanças. Isso pode ser observado na função custo de uma GAN, que será apresentada na seção seguinte.

4.5. Função Custo

A idéia geral de uma GAN é que a rede geradora e a rede discriminadora estão tentando otimizar coisas opostas. A discriminadora precisa classificar imagens como “real” ou “falsa”, ou seja, ela realiza uma classificação binária. Para calcular o custo de uma classificação binária é utilizada a função *Binary Cross Entropy*[20] dada abaixo:

$$F = -[t\ log\ p(t) + (1 - t)\ log(1 - p(t)] \quad (1)$$

A variável *t* representa as classes “real” e ‘falsa” nessa classificação das imagens, assumindo, por exemplo, o valor 1 se a imagem é real e o valor 0 se a imagem é falsa. Assim, a variável $p(t)$ representa a probabilidade da imagem ser real e, reciprocamente, $(1 - p(t))$ representa a probabilidade da imagem ser falsa. A função custo está basicamente somando as duas classes das imagens multiplicadas pelas suas respectivas probabilidades logarítmicas, representando suas entropias. Assim, considerando que *J* seja o valor do custo, tem-se:

$$J = -[t\ log\ p(t) + (1 - t)\ log(1 - p(t))] \quad (2)$$

$t\ log\ p(t)$ representa a entropia das imagens reais e onde *t* possui valor 1, enquanto $(1 - t)\ log\ (1 - p(t))$ representa a entropia das imagens falsas onde *t* possui valor 0. De forma a facilitar a leitura, considera-se a probabilidade $p(t)$ no custo da rede discriminadora como sendo $D(x)$, representando a probabilidade que uma imagem *x* pertença a classe "real" na rede. Assim, pode-se considerar *x* como uma imagem apenas real, *x** como uma imagem falsa e separar $D(x)$ e $D(x^*)$ na função custo em suas respectivas entropias, simplificando ainda mais a função e obtendo o custo da rede discriminadora $J^{(D)}$:

$$J^{(D)} = -[1\ logD(x) + (1 - 0)\ log(1 - D(x^*))] \quad (3)$$

$$J^{(D)} = -[\ logD(x) + log(1 - D(x^*))] \quad (4)$$

Nota-se que a variável *t* sumiu da função, assumindo valor 1 na entropia para imagens reais e valor 0 na entropia para imagens falsas. A respeito da rede geradora, as imagens geradas por ela são representadas por $G(z)$, onde *z* é uma amostra no espaço latente da rede. O espaço latente é uma região escondida que as redes neurais utilizam para guardar as principais características dos dados sendo passados por ela, ajudando a encontrar padrões nestes dados e a lidar com dados comprimidos.

Assim, sabendo-se que a imagem gerada pela rede geradora é uma imagem falsa para a rede discriminadora e *x** representa imagens falsas, pode-se substituir *x** por $G(z)$ na função custo, obtendo:

$$J^{(D)} = -[\ logD(x) + log(1 - D(G(z)))] \quad (5)$$

Considerando agora a função custo da rede geradora $J^{(G)}$, como o objetivo dela é enganar a rede discriminadora, então pode-se dizer que sua função custo seria maximizar o custo da discriminadora, levando a um jogo de soma-zero ou minimax:

$$J^{(G)} = -J^{(D)} \tag{6}$$

$$J^{(G)} = logD(x) + log(1 - D(G(z))) \tag{7}$$

Contudo, relembrando a função custo da discriminadora, o primeiro termo logarítmico diz respeito apenas a rede discriminadora, pois envolve as imagens reais e não as imagens falsas geradas, portanto é irrelevante para a rede geradora. Dessa forma, a função custo da geradora pode ter seu objetivo alterado, ao invés de usar *t* com valor 0 para imagens falsas, pode-se usar *t* com valor 1 e $p(t) = D(G(z))$, deixando de ser um jogo de soma-zero já que a soma de seus custos não é mais zero. Assim:

$$J^{(G)} = -t\ log\ p(t) = -logD(G(z)) \tag{8}$$

Finalmente, deseja-se o valor esperado de todos dados possíveis, ou seja, sua esperança, obtendo o custo médio de cada rede:

$$J^{(D)} = -\{E[\ logD(x)] + E[\ log(1 - D(G(z)))]\} \tag{9}$$

$$J^{(G)} = -E[logD(G(z))] \tag{10}$$

Uma vez explicados os princípios de funcionamento de uma GAN, será apresentado a seguir o funcionamento das redes neurais convolucionais (CNN, do inglês *Convolutional Neural Network*).

4.6. Convolutional Neural Networks (CNNs)

As CNNs são redes neurais que realizam processamento de imagens, recebendo uma imagem de entrada, alterando os pesos e bias para os diferentes aspectos da imagem e sendo capazes de diferenciá-los. Sua arquitetura inclui camadas de convoluções e de *pooling*[21] que realizam, basicamente, multiplicações

com os pixels das imagens de forma a reduzir seu tamanho, mantendo seus aspectos mais importantes para o aprendizado da rede.

Uma camada de convolução consiste em um *kernel* (filtro) de um certo tamanho dimensional e que possui um número de peso em cada elemento. O filtro é deslizado na imagem de acordo com o número de *strides* (passos) e, para cada passo, ele realiza a soma dos pixels da imagem naquela região após cada um ser multiplicado com o peso correspondente no filtro. Cada imagem gerada após passar pelo filtro é denominada *feature map*. Pode-se observar esse comportamento nas figuras 7 e 8.

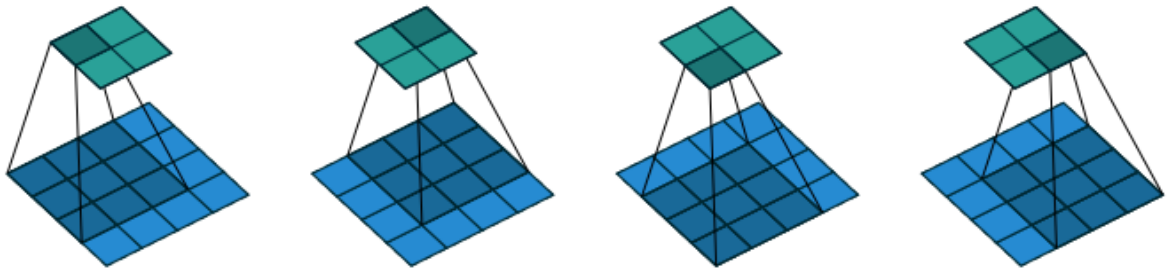

Figura 7 - Convolução com *kernel* de dimensão 3 x 3, *stride* unitário e entrada de tamanho 4 x 4[6].

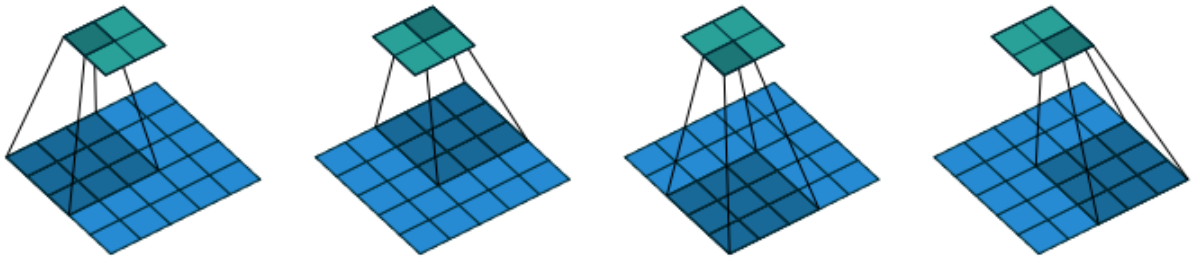

Figura 8 - Convolução com *kernel* de dimensão 3 x 3, *stride* de tamanho 2 (horizontal e vertical) e entrada de tamanho 5 x 5[6].

Uma camada de *pooling* reduz o tamanho da imagem, usualmente, calculando o valor máximo ou médio dos pixels de diferentes regiões desta e funciona de modo similar à convolução, através de uma janela que desliza na imagem e realiza as operações de *pooling*, reduzindo a capacidade computacional necessária para processar a imagem. A diferença básica entre *pooling* e convolução é que o *pooling* realiza funções diferentes da combinação linear do *kernel* em uma convolução.

Assim, a arquitetura de uma CNN é composta por camadas de convoluções e *pooling* intercaladas e sua quantidade pode variar dependendo da complexidade das entradas na rede. Após a rede reduzir a entrada o suficiente e obter seus aspectos mais importantes, os dados são redimensionados e passados para uma rede neural regular para realizar a classificação.

Uma vez explicados os princípios de funcionamento de uma CNN, será apresentado a seguir uma arquitetura em particular das GANs, denominada rede adversária generativa convolucional profunda (DCGAN, do inglês *Deep Convolutional Generative Adversarial Network*), a qual faz uso de convoluções assim como as CNNs.

4.7. Deep Convolutional Generative Adversarial Networks (DCGANs)

O tipo de GAN que foi utilizada neste trabalho é a DCGAN[22], que através da utilização de convoluções sem *pooling*, são capazes de gerar imagens de alta qualidade e resolução, e de forma mais estável em comparação com GANs mais simples.

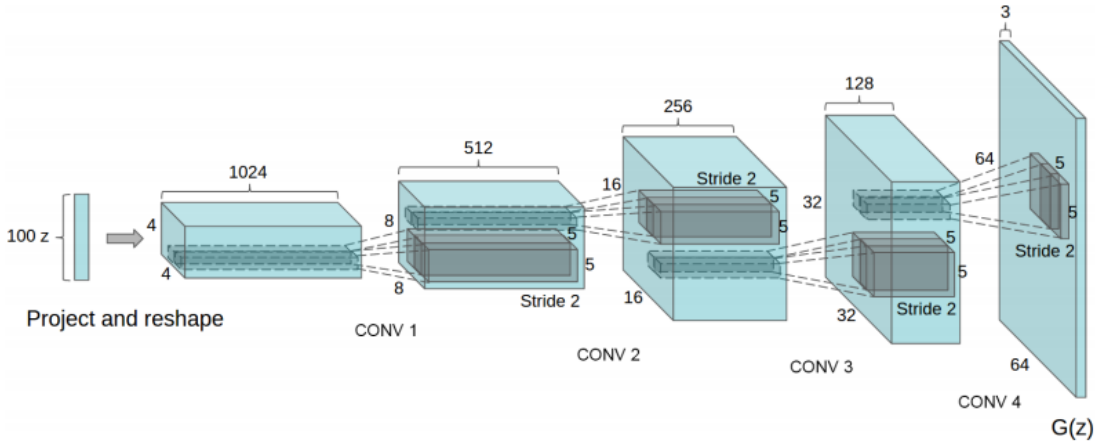

Figura 9 - Estrutura de um gerador em uma DCGAN[23].

Na figura 9 pode-se observar a estrutura de uma rede geradora em uma DCGAN. O gerador inicia com um vetor de distribuição uniforme *z* de dimensão 100 e remodela esse vetor em um objeto tridimensional, o qual passa por uma série de convoluções transpostas a fim de aumentar o tamanho da imagem, até formar uma imagem pixelada de tamanho 64 x 64 e com 3 canais de cor.

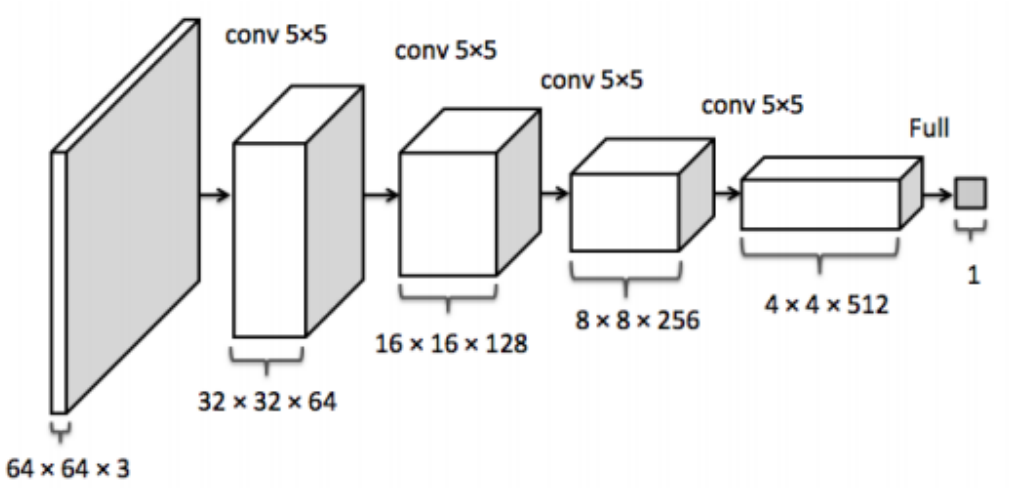

Figura 10 - Estrutura de um discriminador em uma DCGAN[23].

Na figura 10 pode-se observar a estrutura de uma rede discriminadora em uma DCGAN. O discriminador inicia com uma imagem de tamanho 64 x 64 pixels e com 3 canais de cor, passa a imagem por uma série de convoluções até obter a classificação.

Com uma convolução regular, você consegue adquirir uma imagem do mesmo tamanho ou menor, porém a rede geradora aumenta o tamanho da imagem ao longo do processo como pode ser observado na figura 9. Isso é feito através de convoluções transpostas, que resulta em uma imagem maior que a imagem de entrada[6].

Seguem algumas características da implementação da DCGAN:

- *All-Convolutional Network*: ao invés de utilizar as técnicas *pooling* e convoluções intercaladas no processamento das imagens, a rede utiliza apenas convoluções, reduzindo o número de multiplicações e dependendo apenas do número de *strides* da convolução para fazer um *downsample* ou *upsample* da imagem.
- *Adam Optimizer*[24][25]: algoritmo adaptativo do tipo gradiente descendente que busca adaptar a taxa de aprendizagem (*learning rate*) da rede utilizando estimativas do primeiro e segundo momentos do gradiente.
- *Leaky Rectified Linear Unit* (Leaky ReLU): ao utilizar o ReLU como função de ativação na rede geradora é recomendável utilizar o Leaky ReLU para a

discriminadora a fim de resolver o problema de “neurônios mortos”, quando o valor de saída é 0 e o gradiente também fica 0, não havendo mais mudança, pois não há mais alguma “descida”. Pode-se observar o comportamento dessas funções na figura 11 a seguir:

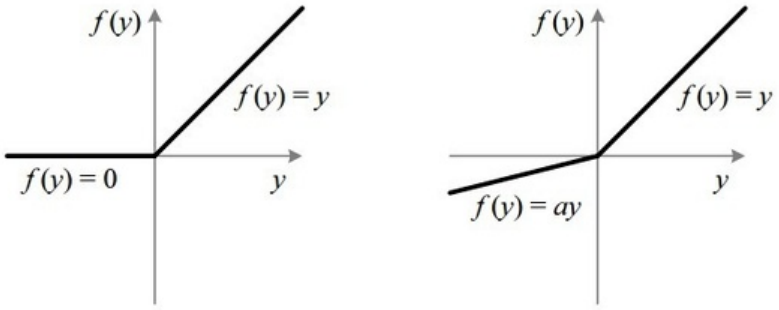

Figura 11 - ReLU na esquerda e Leaky ReLU na direita[26].

- *Batch Normalization*[27]: normalização dos dados em cada camada da rede neural. Ocorre após a transformação linear dos dados (multiplicação pelo peso e soma com o bias) e antes de passar pela função de ativação na camada. A normalização realiza operações com os dados de saída em cada camada utilizando a média do lote (*batch*) e seu desvio padrão, a fim de acelerar a taxa de aprendizagem da rede.

5. RESULTADOS

Para obtenção dos resultados a serem mostrados, foram utilizadas 42 mil imagens do conjunto de dados MNIST, retiradas do arquivo “train.csv” no site “https://www.kaggle.com/c/digit-recognizer/data”. As imagens foram divididas em 656 lotes de tamanho 64 e duas épocas de treinamento foram realizadas. Cada imagem representa um dígito de 0 a 9 e possui um tamanho de 28 x 28 pixels, totalizando 784 colunas de dados e mais uma coluna rótulo com o valor do dígito.

A rede DCGAN utilizada possui as seguintes características, primeiramente para a rede discriminadora:

1. Uma camada convolucional com 1 *feature map* de entrada, 2 *feature maps* de saída, tamanho do *kernel* 5 x 5, stride de valor 2 e função de ativação Leaky ReLU. Assim, os pesos ficam organizados em dimensões 5 x 5 x 1 x 2 e a entrada e saída da camada ficam com tamanhos 28 x 28 x 1 e 14 x 14 x 2 respectivamente.
2. Uma camada convolucional com 2 *feature maps* de entrada, 64 *feature maps* de saída, tamanho de *kernel* 5 x 5, stride de valor 2 e função de ativação Leaky ReLU. Assim, os pesos ficam organizados em dimensões 5 x 5 x 2 x 64 e a entrada e saída da camada ficam com tamanhos 14 x 14 x 2 e 7 x 7 x 64 respectivamente.
3. Uma camada densa (onde todos os neurônios recebem entrada de todos os neurônios na camada anterior) com tamanho de entrada 784, tamanho de saída 1024 para o bias e função de ativação Leaky ReLU. Os pesos são organizados em uma matriz de tamanho 784 x 1024, sendo que 784 remete ao número de pixels das imagens MNIST: 28 x 28.
4. Uma camada densa com tamanho de entrada 1024, tamanho de saída 1 para o bias e uma função de ativação f(x) = x. Os pesos são organizados em uma matriz de tamanho 1024 x 1.

Agora para a rede geradora:

1. Espaço latente configurado como um vetor de tamanho 100.

2. Uma camada densa com tamanho de entrada 100, tamanho de saída 1024 para o bias e uma função de ativação ReLU. Os pesos são organizados em uma matriz de tamanho 100 x 1024.
3. Uma camada densa com tamanho de entrada 1024, tamanho de saída 6272 (*128 * 7 * 7*, sendo o valor 7 encontrado através da divisão da dimensão da imagem 28 pelos strides de valor 2 nas duas camadas convolucionais, então *28 / (2 * 2)*) para o bias e uma função de ativação ReLU. Os pesos são organizados em uma matriz de tamanho 1024 x 6272.
4. Uma camada convolucional transposta com 128 *feature maps* de entrada, 128 *feature maps* de saída, tamanho de filtro 5 x 5, stride de valor 2 e função de ativação ReLU. Assim, os pesos ficam organizados em dimensões 5 x 5 x 128 x 128 e o formato da saída da camada é configurado com dimensões 14 x 14 x 128.
5. Uma camada convolucional transposta com 128 *feature maps* de entrada, 1 *feature map* de saída, tamanho de filtro 5 x 5, stride de valor 2 e função de ativação sigmoidal. Assim, os pesos ficam organizados em dimensões 5 x 5 x 1 x 128 e o formato da saída da camada é configurado com dimensões 28 x 28 x 1, que é justamente o tamanho de uma imagem na base de dados MNIST.

As camadas convolucionais são formadas pela função “tf.nn.conv2d” do TensorFlow, as camadas densas são formadas pelo processo de multiplicação matricial entre a entrada e os pesos e depois a soma com o bias, e, por fim, as camadas convolucionais transpostas são formadas pela função “tf.nn.conv2d_transpose” do TensorFlow. A função “tf.nn.bias_add” também do TensorFlow, é utilizada para somar o bias com o resultado da convolução.

5.1. Base de dados MNIST

Uma imagem de um lote com 64 amostras geradas é salva a cada 50 iterações, mostrando a evolução da GAN em reproduzir os dígitos ao longo do total de 1312 iterações (656 lotes x 2 épocas de treinamento). Além disso, foi feito um gráfico da

variação dos custos das redes geradora e discriminadora ao longo da execução do algoritmo. As figuras 12 a 18 mostram os resultados obtidos ao longo das iterações.

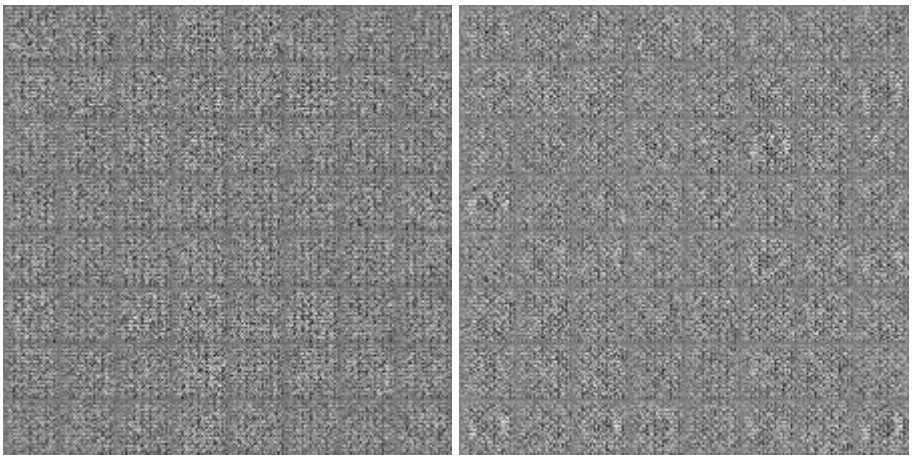

Figura 12 - 64 amostras após 50 e 100 iterações.

Observando a figura 12, nota-se que a rede geradora ainda não conseguiu gerar alguma amostra semelhante a um dígito e nenhuma imagem aparenta ter alguma forma clara.

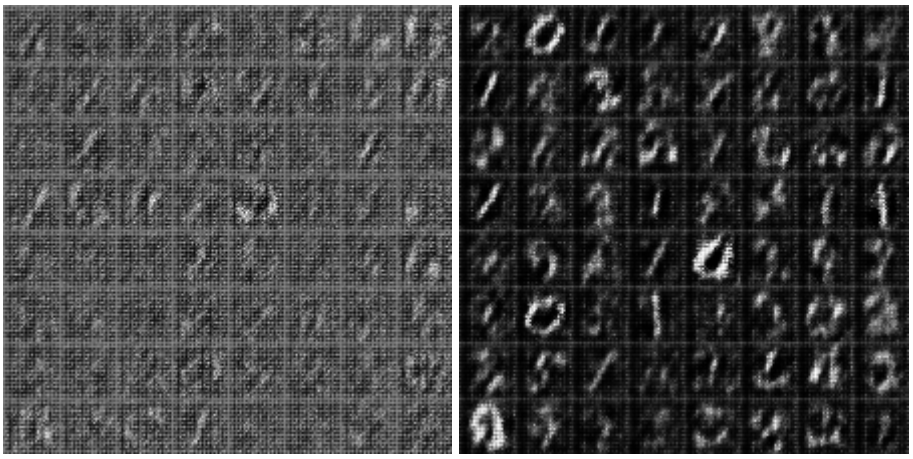

Figura 13 - 64 amostras após 150 e 200 iterações.

Observando a figura 13, mais especificamente as imagens na iteração 200, nota-se que a rede geradora está produzindo formas bem mais claras, levemente semelhantes a dígitos. Também é possível notar uma leve melhora nas imagens da iteração 150 em comparação com as da figura 7.

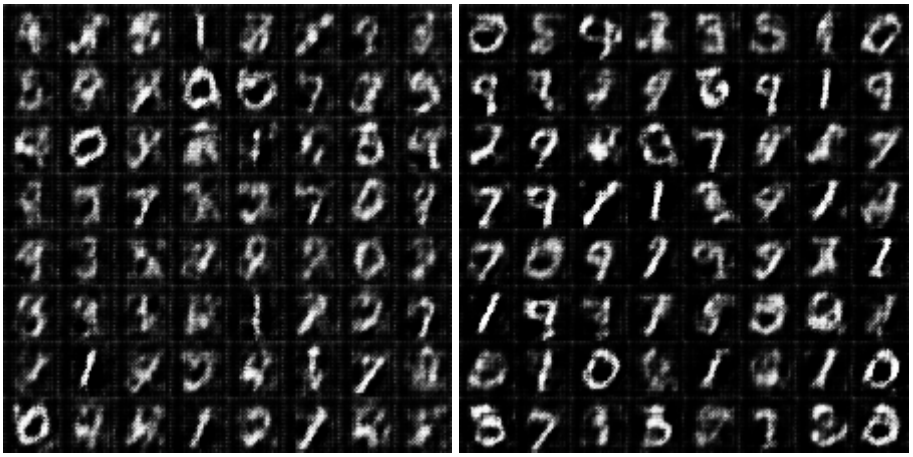

Figura 14 - 64 amostras após 250 e 300 iterações.

Observando a figura 14, nota-se dígitos bem mais definidos e distinguíveis.

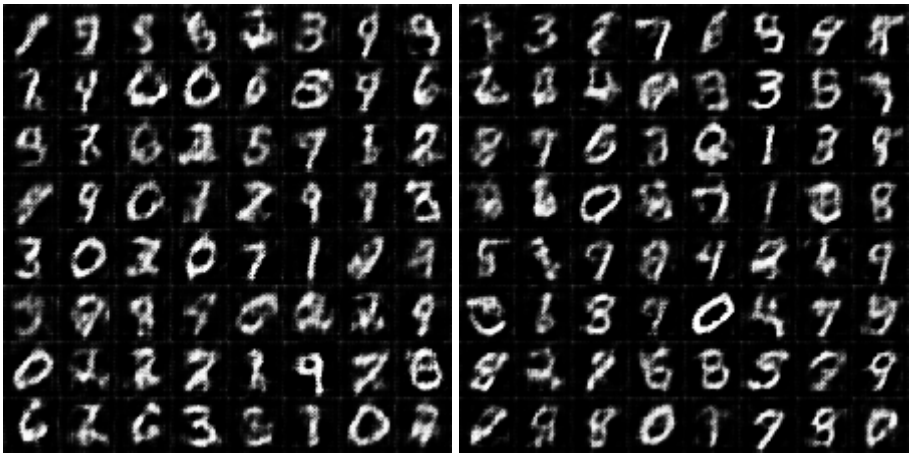

Figura 15 - 64 amostras após 350 e 400 iterações.

Observando a figura 15, os dígitos não aparentam ter alguma diferença notável em relação aos dígitos da iteração 300.

Figura 16 - 64 amostras após 650 e 700 iterações.

Observando a figura 16, nota-se uma leve melhora na resolução dos dígitos gerados, com algumas poucas imagens possuindo formas não reconhecíveis.

Figura 17 - 64 amostras após 1250 e 1300 iterações.

Observando a figura 17, as últimas iterações apresentam uma leve melhora na resolução dos dígitos gerados em comparação à figura 11.

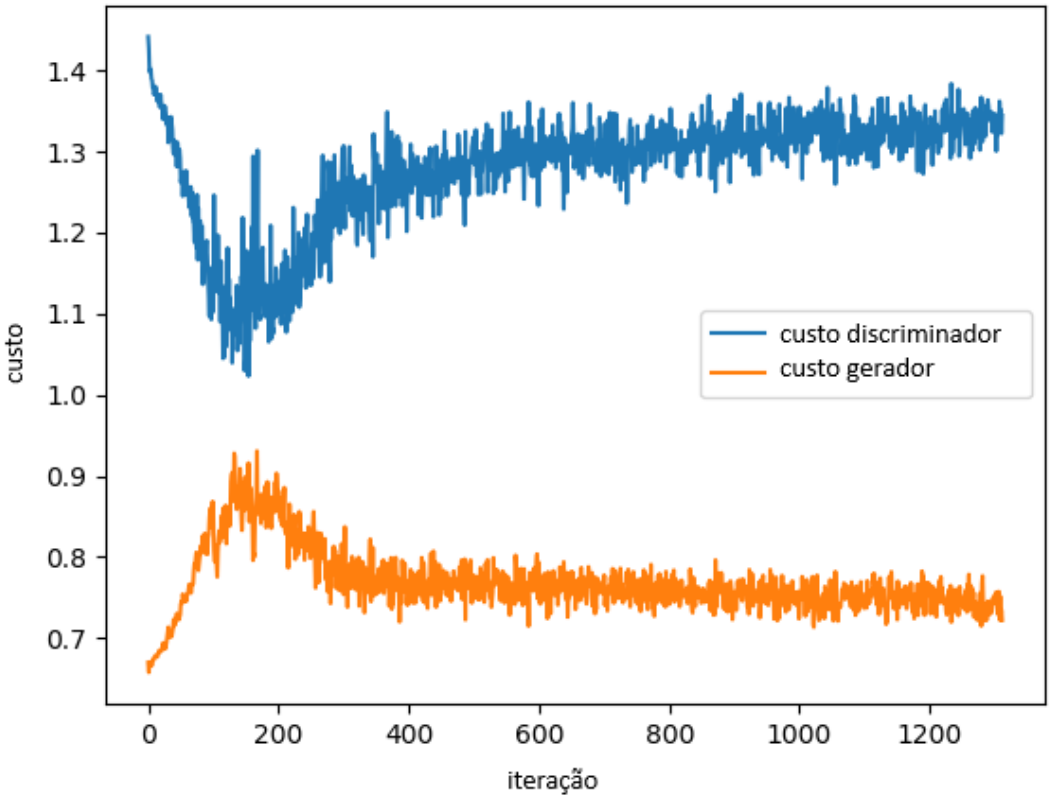

Figura 18 - Evolução dos custos das redes geradora e discriminadora ao longo das 1312 iterações.

Analisando as imagens obtidas das iterações e a evolução dos custos das redes na figura 18, lembrando que o objetivo de cada rede é minimizar seu custo, pode-se observar que da primeira iteração até a iteração 150, aproximadamente, houve uma grande variação nos custos. O gerador estava gerando imagens ruins e o discriminador, por conta disso, estava conseguindo distinguir bem as imagens “reais” e “falsas”, tendo seu custo bem reduzido, e, analogamente, o gerador tendo seu custo aumentado.

Contudo, na iteração 200 o gerador começa a gerar imagens com formas mais claras, isso mostra um pico no gráfico de custo do gerador e um vale no gráfico de custo do discriminador. A partir daí, a rede discriminadora vai ter mais dificuldade em classificar as imagens e seu custo vai aumentar, enquanto a rede geradora vai gerar dígitos cada vez mais próximos dos reais e terá seu custo reduzido.

Na iteração 300, os dígitos gerados estão mais bem definidos e o discriminador apresenta maior dificuldade em classificar o que é “real” e o que é “falso”, tendo seu

custo aumentado a partir da iteração 200 até a iteração 350 aproximadamente onde começa a estabilizar. De forma análoga, a rede geradora teve seu custo reduzido a partir da iteração 200 até a iteração 350 aproximadamente.

A partir da iteração 400, as imagens apresentaram melhoras leves na resolução, assim, podendo observar o custo do discriminador com uma leve inclinação ascendente e o custo do gerador com uma leve inclinação descendente até fim das iterações.

5.2. Base de dados MNIST com imagens cortadas em posições aleatórias

Para obtenção dos resultados a seguir, foram utilizadas as mesmas imagens da base de dados MNIST, porém um quadrado de dimensão 6 x 6 pixels é cortado em uma posição aleatória em cada imagem de forma a dificultar a geração de dígitos corretos pela GAN como mostrado na figura 19.

Esse teste tem como objetivo verificar se, mesmo com os dígitos cortados, a GAN é capaz de gerar imagens corretas de dígitos e comparar os resultados desse teste com o teste realizado anteriormente sem os dígitos cortados. As figuras 20 a 26 mostram os resultados obtidos ao longo das iterações.

Figura 19 - Exemplos de imagens da base de dados MNIST com um quadrado de tamanho 6 x 6 pixels cortado em uma posição aleatória em cada imagem.

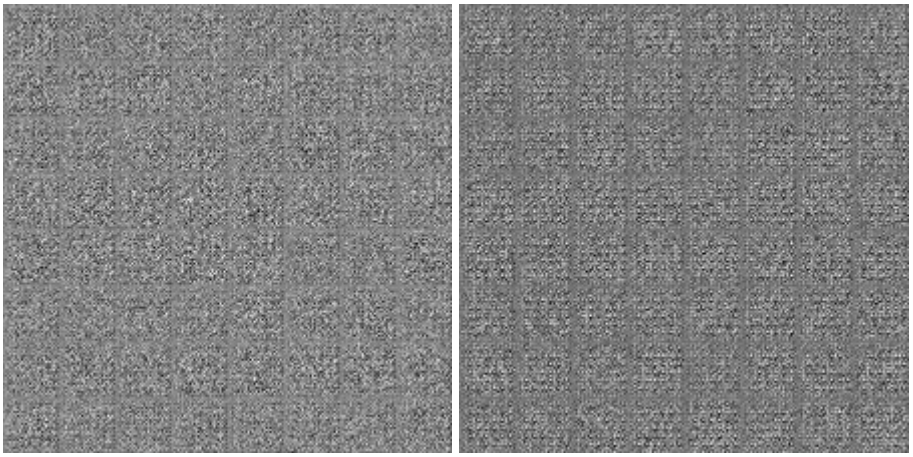

Figura 20 - 64 amostras após 50 e 100 iterações.

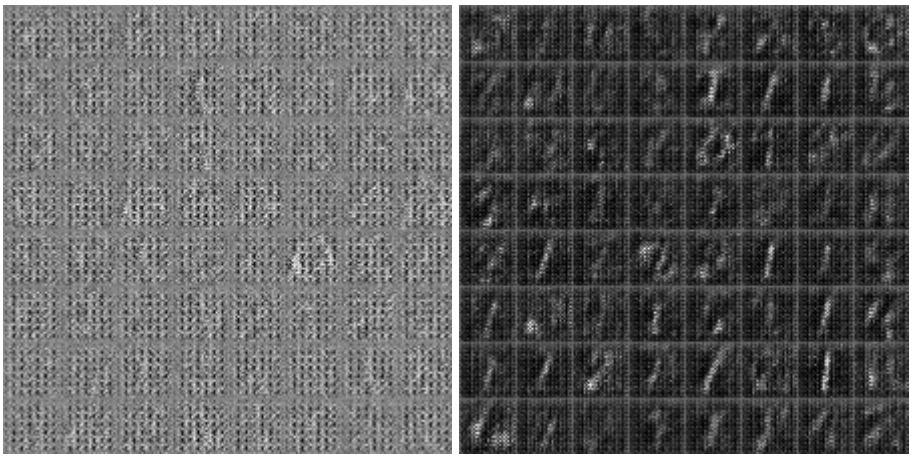

Figura 21 - 64 amostras após 150 e 200 iterações.

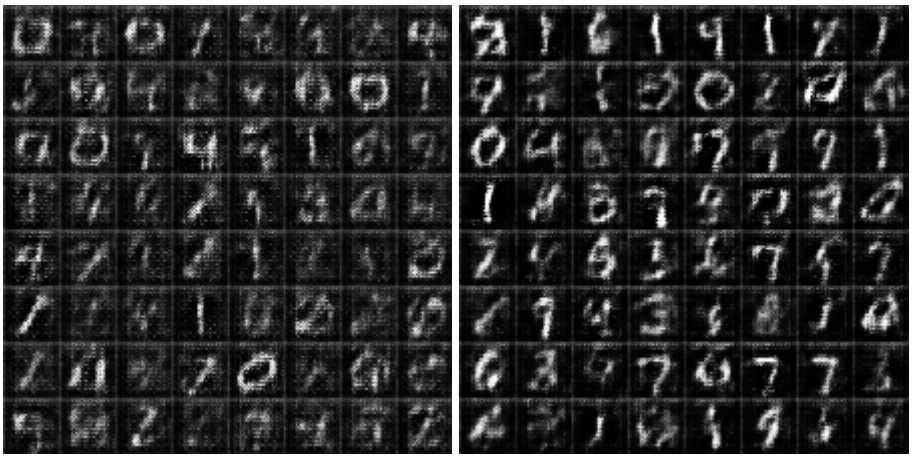

Figura 22 - 64 amostras após 250 e 300 iterações.

Figura 23 - 64 amostras após 350 e 400 iterações.

Figura 24 - 64 amostras após 650 e 700 iterações.

Figura 25 - 64 amostras após 1250 e 1300 iterações.

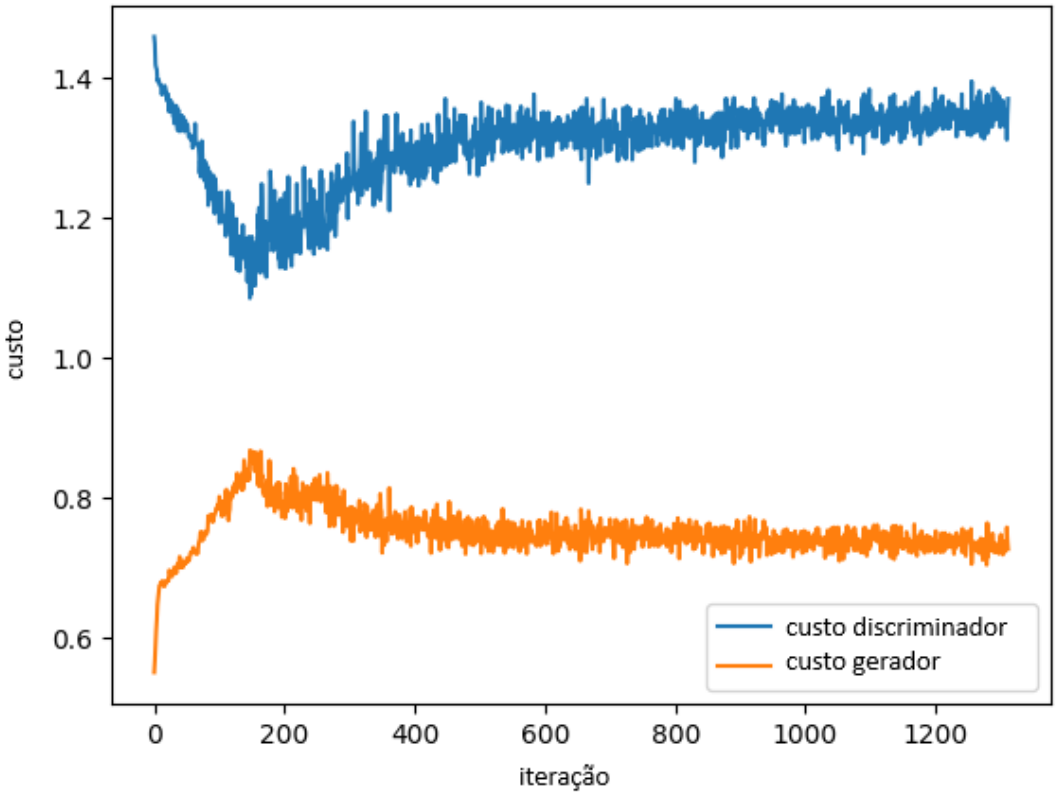

Figura 26 - Evolução dos custos das redes geradora e discriminadora ao longo das 1312 iterações com imagens cortadas dos dígitos MNIST.

Comparando os resultados de ambos os testes, utilizando imagens completas e imagens cortadas em posições aleatórias, é possível notar um comportamento bem semelhante na geração de imagens e a evolução dos custos das redes geradora e discriminadora ao longo das iterações. O custo da geradora e da discriminadora atingiram um máximo e um mínimo respectivamente próximo a iteração de número 200, onde a geradora começou a gerar imagens mais claras. Desde então, os custos tiveram alterações significativas até aproximadamente a iteração 400, gerando dígitos cada vez mais nítidos, e, após isso, houveram alterações leves nos custos até o final das iterações.

Apesar das semelhanças, é possível observar nos resultados do teste com imagens cortadas uma quantidade mais significativa de dígitos com alguma deformação ou falha do que no teste anterior. Contudo, apesar de receber imagens incompletas, a GAN foi capaz de gerar imagens de números inteligíveis.

5.3. Base de dados MNIST com imagens cortadas no centro e com expansão gradual do corte

Observando o sucesso na geração de dígitos inteligíveis no teste anterior, dessa vez cada imagem será cortada no centro e serão realizados testes para diferentes tamanhos de corte, começando por 4 x 4 pixels, 6 x 6 pixels, 8 x 8 pixels e assim por diante até a GAN ser incapaz de gerar dígitos inteligíveis. Pode-se observar exemplos dos cortes graduais na figura 27 abaixo:

Figura 27 - Imagens da base de dados MNIST com um corte no centro de tamanhos(da esquerda para a direita, respectivamente): 4x4 pixels, 6x6 pixels, 8x8 pixels, 10x10 pixels, 12x12 pixels, 14x14 pixels, 16x16 pixels e 18x18 pixels.

5.3.1. Corte 4 x 4 pixels

Figura 28 - 64 amostras após 250 e 300 iterações.

Figura 29 - 64 amostras após 1250 e 1300 iterações.

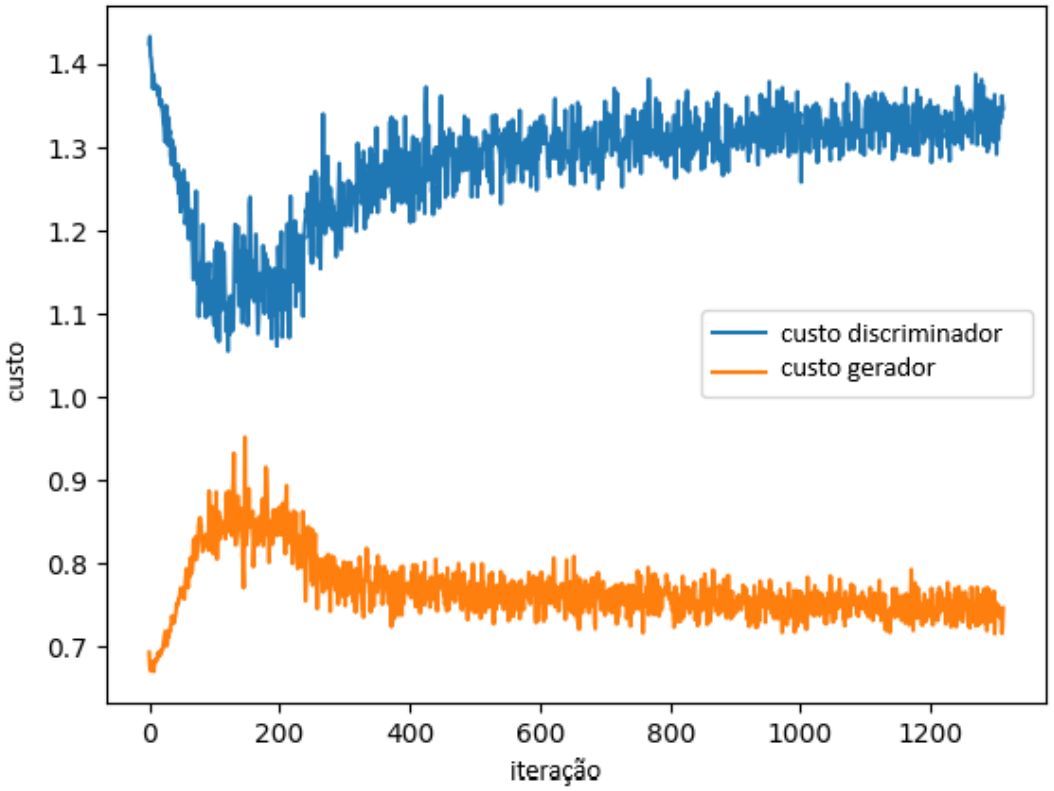

Figura 30 - Evolução dos custos das redes geradora e discriminadora ao longo das 1312 iterações para cortes de tamanho 4 x 4 pixels.

É possível notar que o gráfico de custo manteve-se semelhante aos testes anteriores e a GAN foi capaz de gerar números inteligíveis.

5.3.2. Corte 6 x 6 pixels

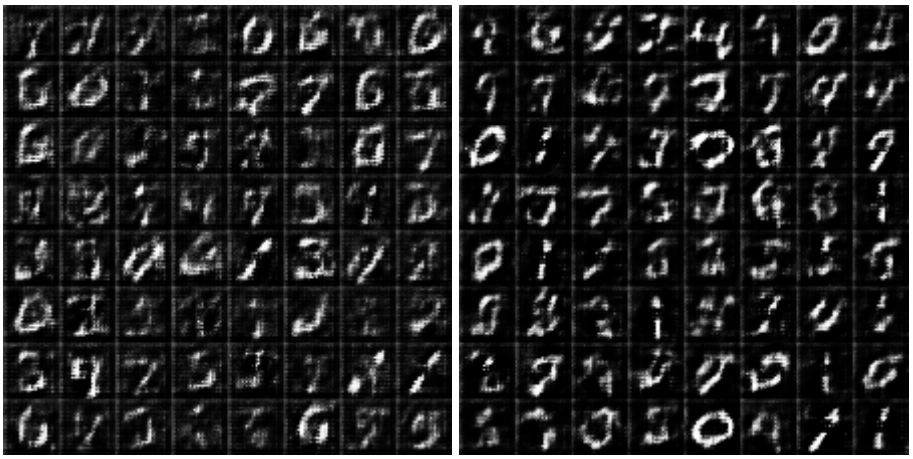

Figura 31 - 64 amostras após 250 e 300 iterações.

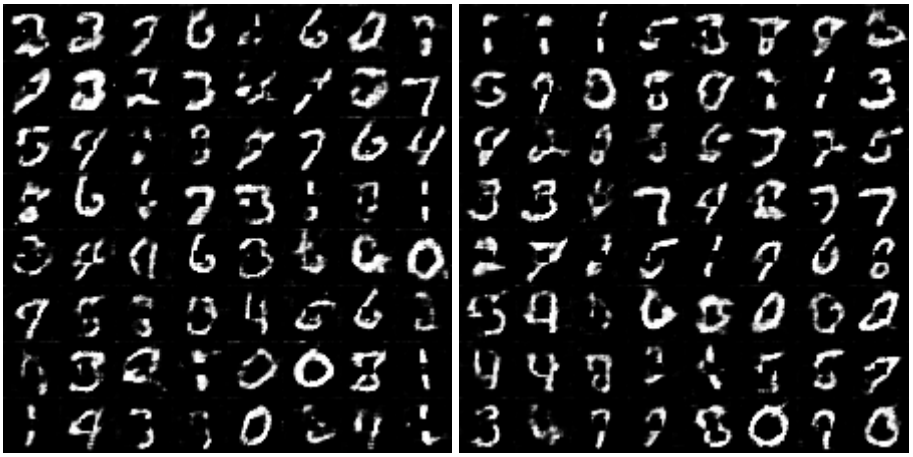

Figura 32 - 64 amostras após 1250 e 1300 iterações.

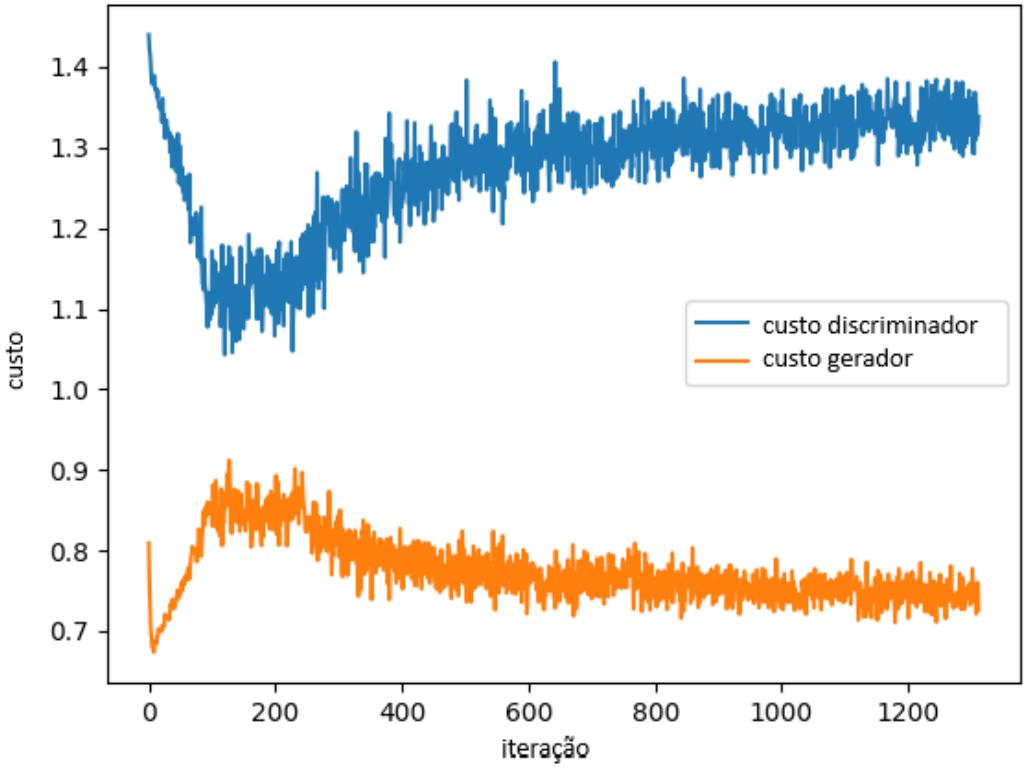

Figura 33 - Evolução dos custos das redes geradora e discriminadora ao longo das 1312 iterações para cortes de tamanho 6 x 6 pixels.

Boa parte dos dígitos ainda são inteligíveis, porém o corte no centro já está dificultando significativamente a leitura das imagens geradas pela GAN.

5.3.3. Corte 8 x 8 pixels

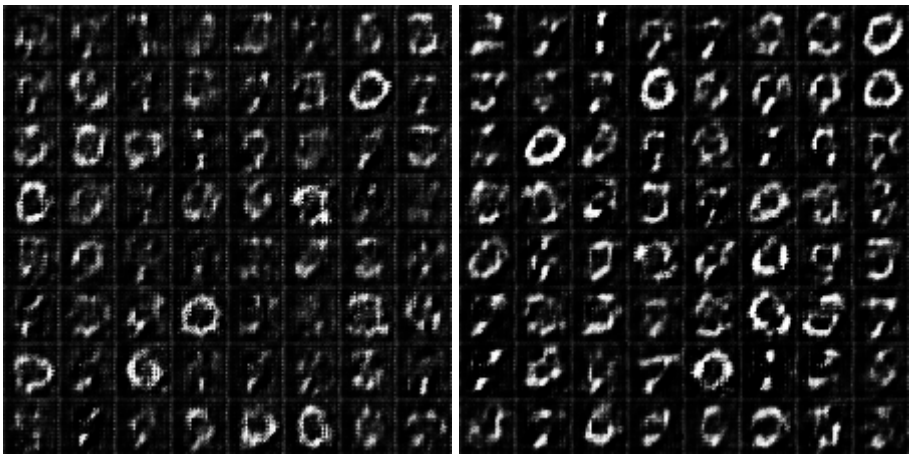

Figura 34 - 64 amostras após 250 e 300 iterações.

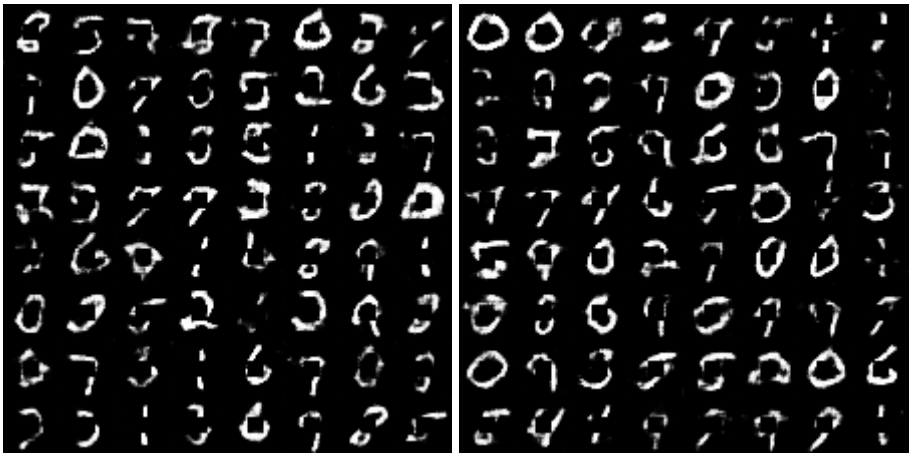

Figura 35 - 64 amostras após 1250 e 1300 iterações.

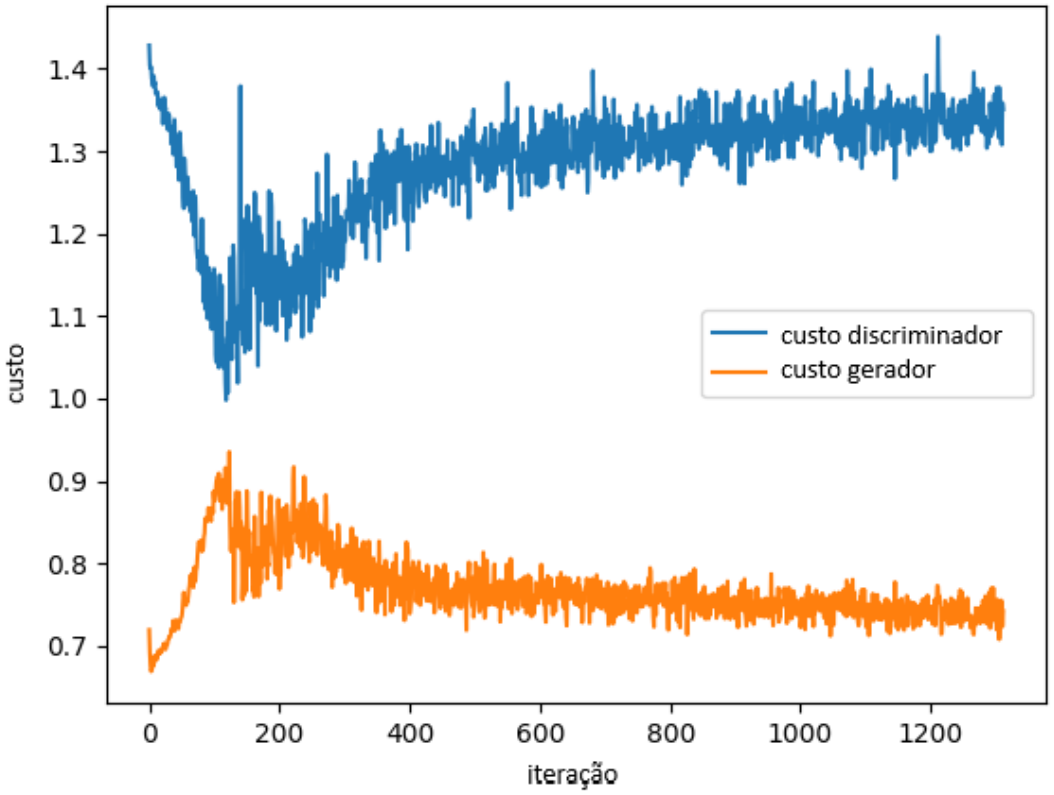

Figura 36 - Evolução dos custos das redes geradora e discriminadora ao longo das 1312 iterações para cortes de tamanho 8 x 8 pixels.

Ainda é possível identificar os dígitos gerados, porém é necessário um esforço maior em relação ao caso anterior de corte de tamanho de 6 x 6 pixels.

5.3.4. Corte 10 x 10 pixels

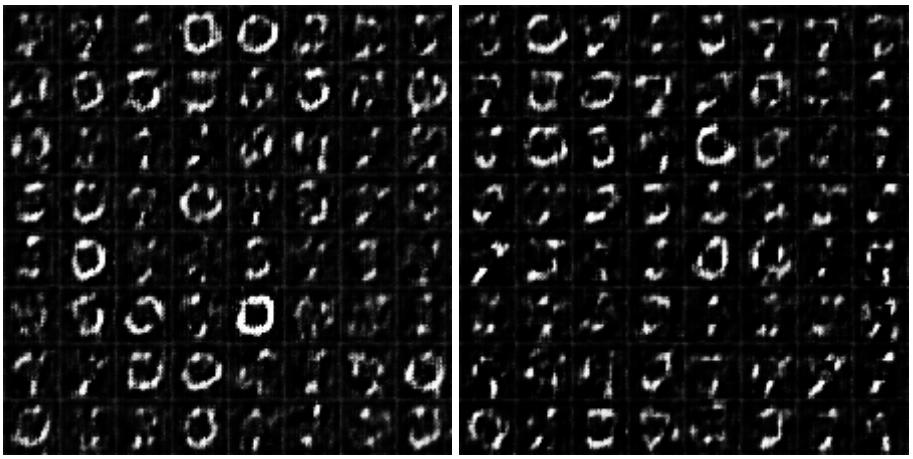

Figura 37 - 64 amostras após 250 e 300 iterações.

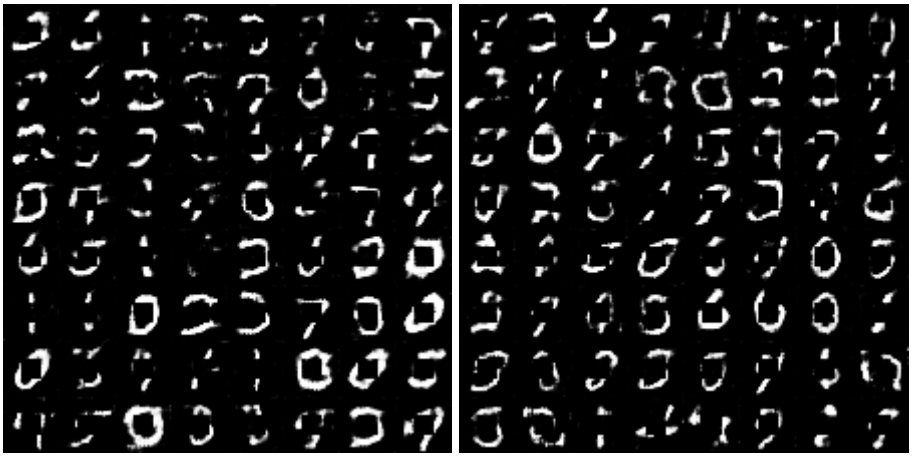

Figura 38 - 64 amostras após 1250 e 1300 iterações.

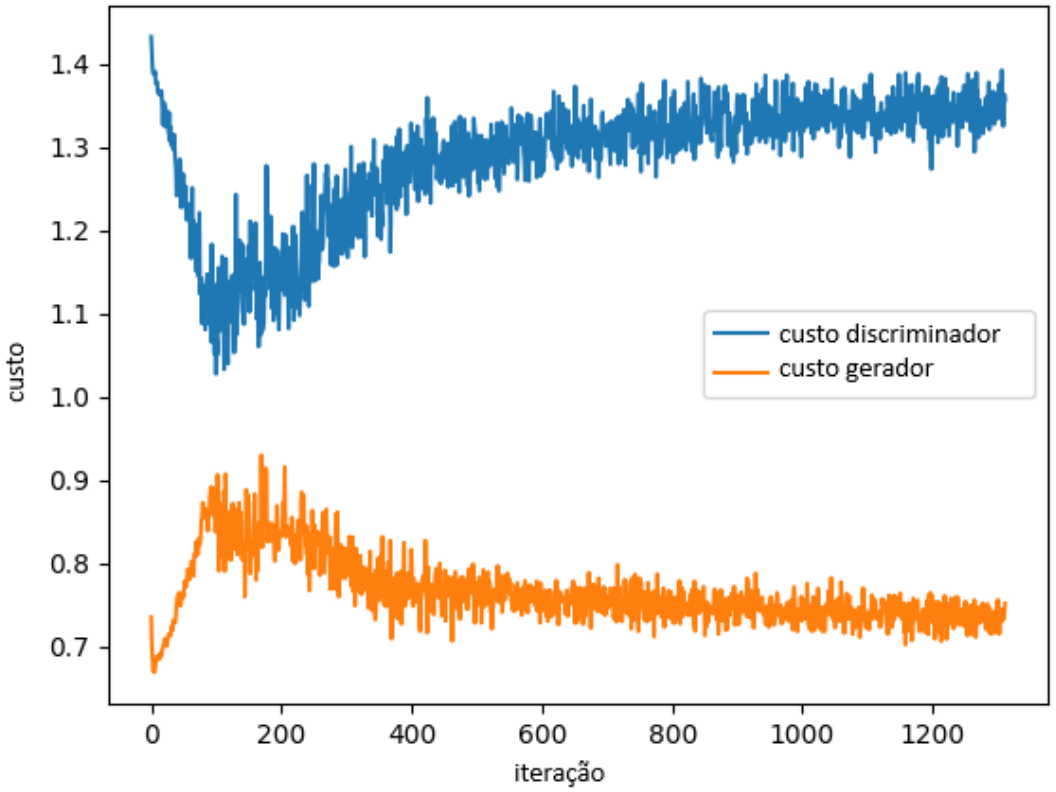

Figura 39 - Evolução dos custos das redes geradora e discriminadora ao longo das 1312 iterações para cortes de tamanho 10 x 10 pixels.

Conforme o aumento do corte nas imagens, a GAN foi gerando imagens cada vez menos inteligíveis com a presença do quadrado preto no meio como era de se esperar. Ainda é possível identificar alguns dígitos, porém a leitura é ainda mais difícil do que no caso do corte de 8 x 8 pixels.

5.3.5. Corte 12 x 12 pixels

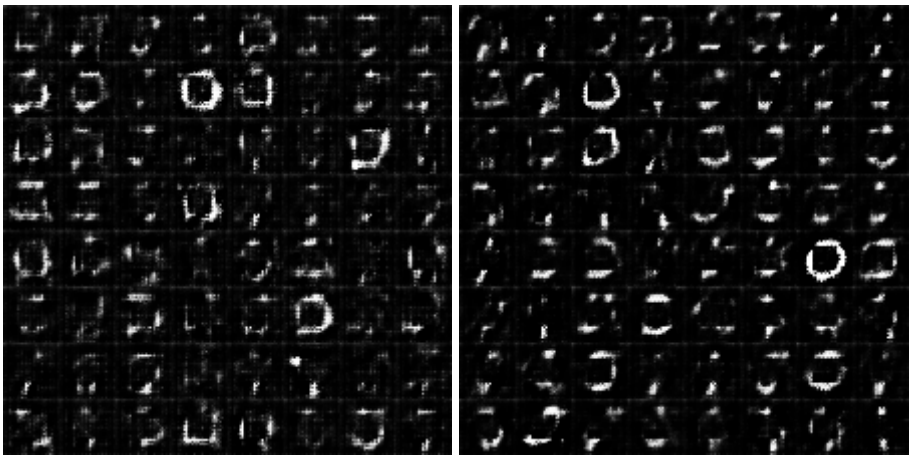

Figura 40 - 64 amostras após 250 e 300 iterações.

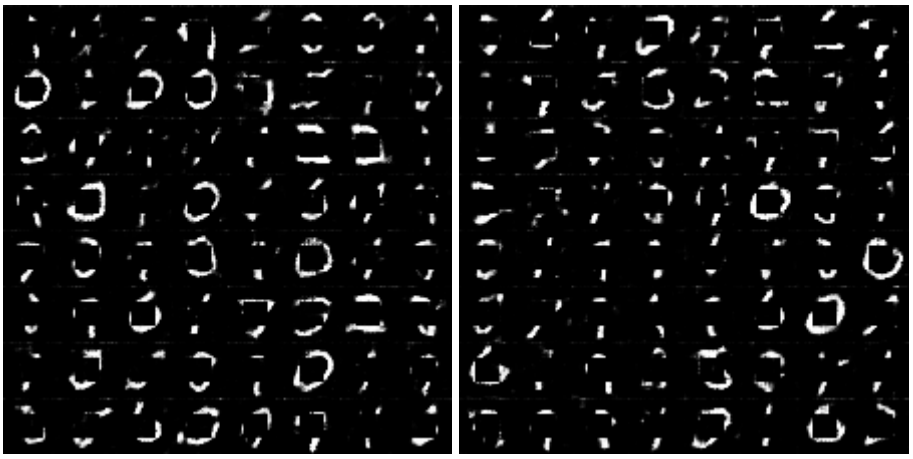

Figura 41 - 64 amostras após 1250 e 1300 iterações.

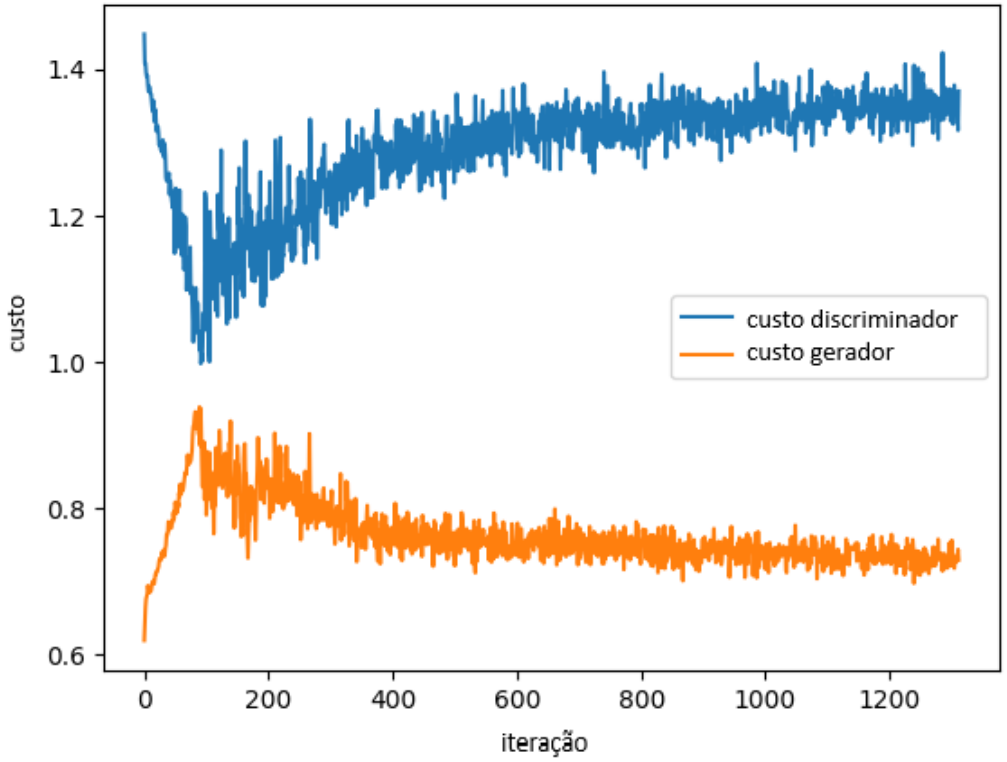

Figura 42 - Evolução dos custos das redes geradora e discriminadora ao longo das 1312 iterações para cortes de tamanho 12 x 12 pixels.

Os dígitos gerados agora são escassamente inteligíveis, porém é possível notar como o gráfico de custos da GAN manteve-se semelhante em todos os testes, o que é de se esperar devido a pouca quantidade de detalhes e pixels que cada imagem possui e o corte apresentar pouca alteração na informação da imagem. O problema é que a GAN gerou dígitos que não fazem sentido para leitura humana.

5.4. Geração de dígitos a partir de imagens incompletas

Após verificar como os cortes nas imagens base influenciam a geração das imagens pela GAN, foi testada a capacidade dela de produzir dígitos inteligíveis e inteiros para outras situações com imagens incompletas: quando uma porção significativa é retirada de cada imagem, porém a porção é retirada de uma posição aleatória para cada uma e quando uma porcentagem das imagens estão incompletas e outra a porcentagem está inteira.

Nessas duas situações é possível que a GAN obtenha informações que estão faltando nas regiões de algumas imagens através de outras imagens que não estão cortadas naquela região, um processo similar à técnica “Copy-and-Paste” de Inpainting onde a idéia é procurar por imagens semelhantes com a imagem incompleta e preencher a parte ausente “colando” pedaços das imagens semelhantes.

5.4.1. Cortes de 12 x 12 pixels em posições aleatórias

Removendo porções de tamanho 12 x 12 pixels de cada imagem, em um local aleatório para cada uma, foram obtidos os seguintes resultados:

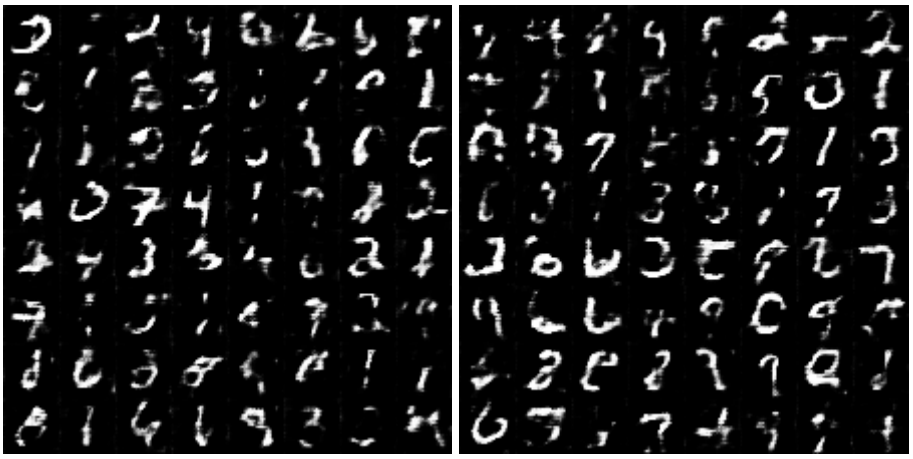

Figura 43 - 64 amostras após 1150 e 1200 iterações.

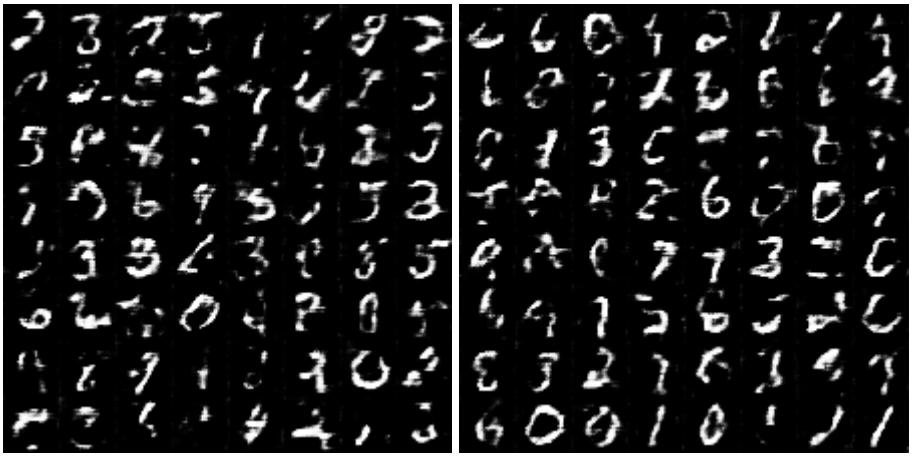

Figura 44 - 64 amostras após 1250 e 1300 iterações.

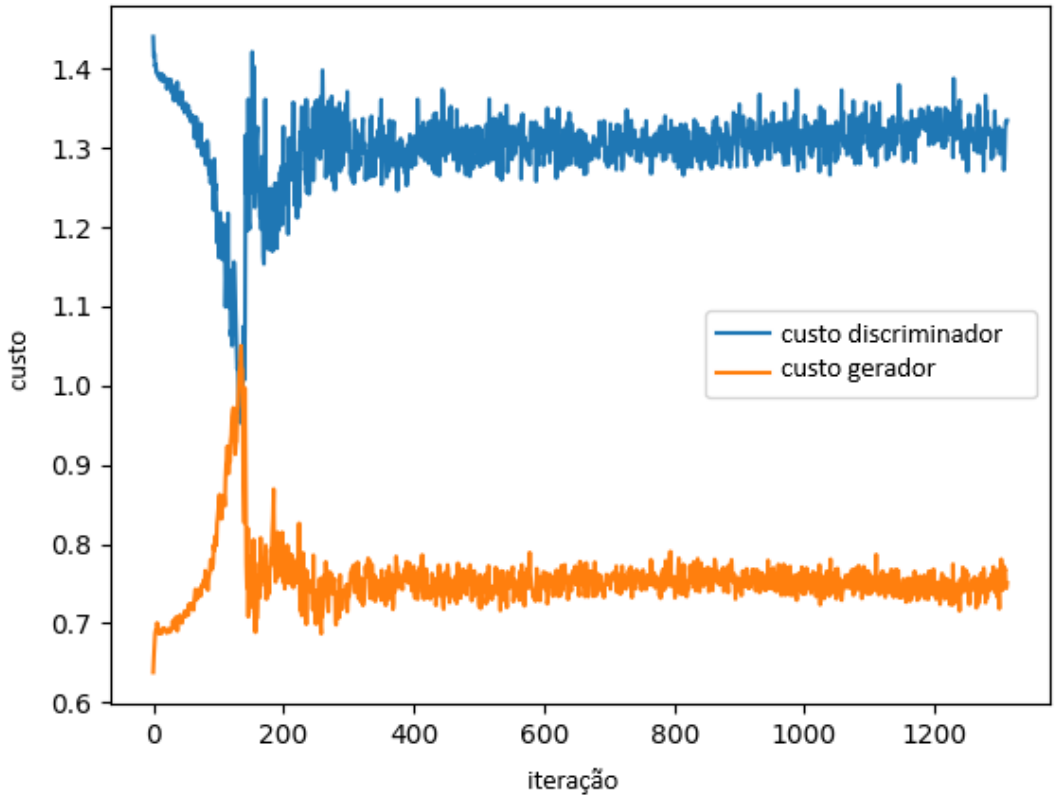

Figura 45 - Evolução dos custos das redes geradora e discriminadora ao longo das 1312 iterações para cortes de tamanho 12 x 12 pixels em posições aleatórias.

Observa-se que, apesar de grande parte dos dígitos gerados apresentarem deformação e cortes, é possível identificar aproximadamente metade deles visualmente.

5.4.2. Cortes de 12 x 12 pixels no centro de 50% das imagens

A parte central de cada imagem da base de dados MNIST é a mais rica em informações para identificação dos dígitos. Assim, os cortes foram aplicados no centro das imagens, porém em apenas uma porção delas, começando por 50%, ou seja, 21 mil imagens com cortes e 21 mil sem cortes, obtendo os resultados abaixo:

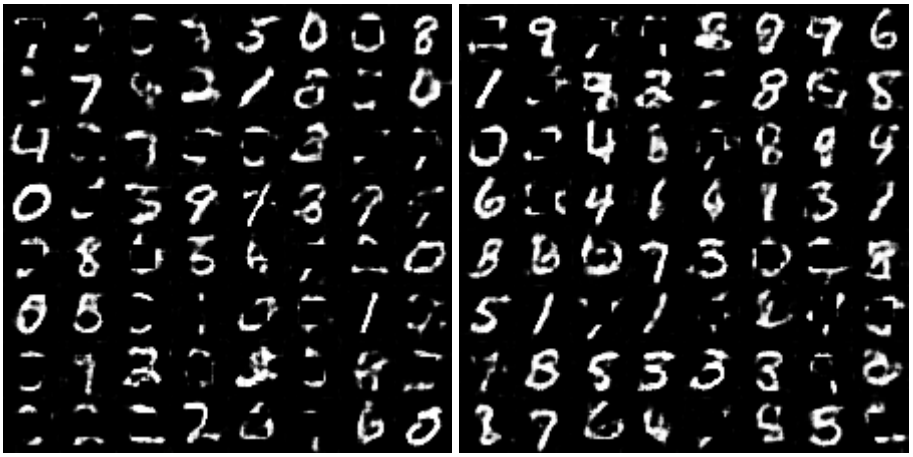

Figura 46 - 64 amostras após 1250 e 1300 iterações.

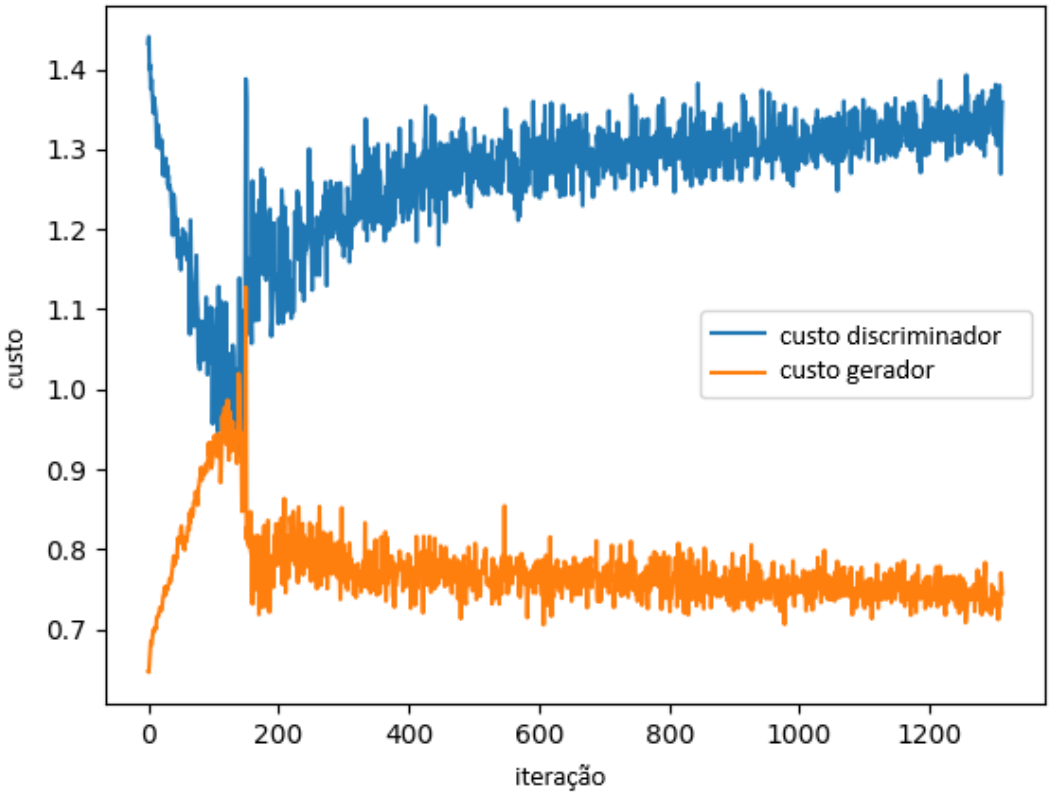

Figura 47 - Evolução dos custos das redes geradora e discriminadora ao longo das 1312 iterações para cortes de tamanho 12 x 12 pixels em 50% da base de dados.

A GAN gerou dígitos com deformações e com cortes no centro, porém também foi capaz de gerar uma boa quantidade de dígitos inteligíveis e bem definidos.

5.4.3. Cortes de 12 x 12 pixels no centro de 75% das imagens

Agora os resultados a serem mostrados foram de um teste onde 75% das imagens da base dados MNIST tiveram seus centros cortados, ou seja, 31.500 imagens com cortes e 10.500 imagens sem cortes:

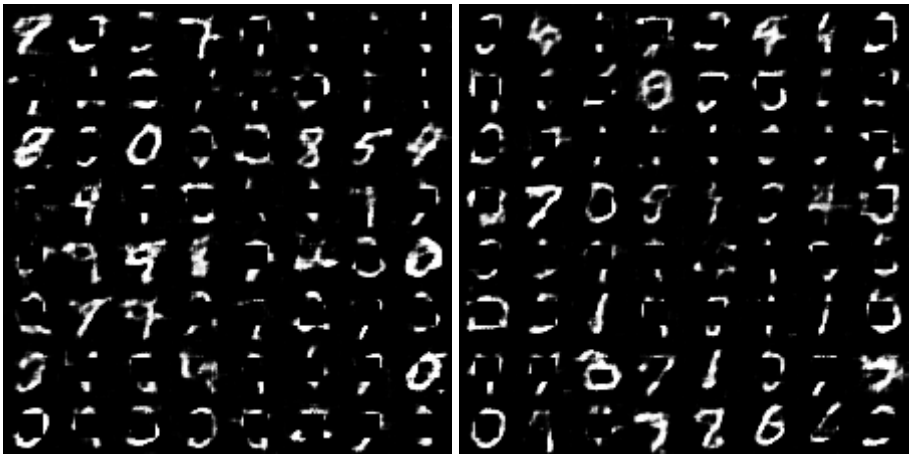

Figura 48 - 64 amostras após 1250 e 1300 iterações.

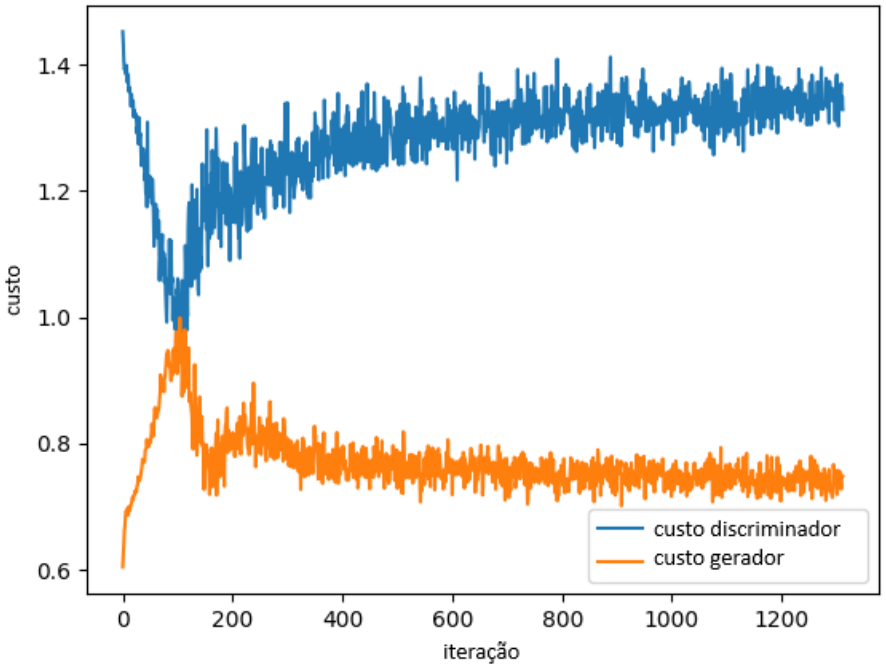

Figura 49 - Evolução dos custos das redes geradora e discriminadora ao longo das 1312 iterações para cortes de tamanho 12 x 12 pixels em 75% da base de dados.

É possível notar uma presença bem maior de números deformados e cortados em relação ao teste com 50% da base de dados cortada, porém ainda há presença de dígitos inteligíveis e bem definidos.

5.4.4. Cortes de 12 x 12 pixels no centro de 90% das imagens

Agora os resultados a serem mostrados foram de um teste onde 90% das imagens da base dados MNIST tiveram seus centros cortados, ou seja, 37.800 imagens com cortes e 4.200 imagens sem cortes:

Figura 50 - 64 amostras após 1250 e 1300 iterações.

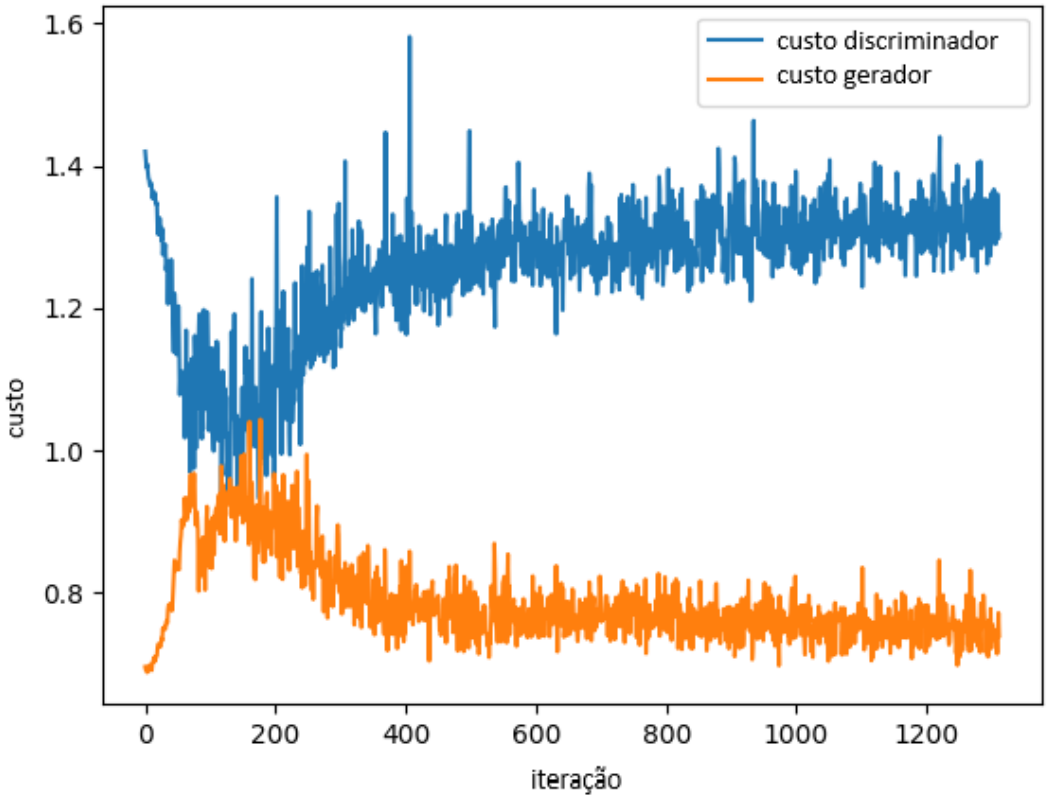

Figura 51 - Evolução dos custos das redes geradora e discriminadora ao longo das 1312 iterações para cortes de tamanho 12 x 12 pixels em 90% da base de dados.

Neste teste, já não é mais possível identificar dígitos gerados pela GAN, além do 0 e 1 em alguns casos e havendo confusão entre os dígitos 4 e 9.

6. CONSIDERAÇÕES FINAIS

Considerando todas as capacidades das GANs, podendo gerar uma nova realidade semelhante à nossa, aprimorar a qualidade do que já existe, automatizar trabalhos complexos e gerar conteúdos relevantes, e tudo isso feito por uma inteligência artificial é sem dúvida um avanço tecnológico importante e atrativo.

A combinação de uma rede geradora com uma rede discriminadora mostrou-se eficaz na geração de imagens bem semelhantes às reais e a evolução dos seus custos mostrou fielmente uma competição entre redes querendo otimizar coisas opostas. Apesar dos resultados não terem mostrado uma ótima restauração das imagens, existem outros estudos com GANs que mostram excelentes resultados da aplicação de *inpainting*. Além disso, com este trabalho, pode-se aprofundar conhecimentos na linguagem Python, *deep learning* e GANs.

6.1. Perspectivas para trabalhos futuros

A partir deste trabalho, pode-se realizar aplicações de GANs para geração de imagens coloridas a partir de outros conjuntos de dados mais complexos como o CIFAR-10 e de rostos humanos. Além disso, pode-se construir redes neurais aplicando técnicas de *inpainting* mencionadas, como codificadores de contexto e MSNPS de forma a realizar restauração de imagens.

7. REFERÊNCIAS

[1] ANYOHA, R. The History of Artificial Intelligence. 2017. Disponível em: <http://sitn.hms.harvard.edu/flash/2017/history-artificial-intelligence/> Acesso em: 26 de abr. de 2020.

[2] KAUFMAN, D. Artificial Intelligence Comes to Hollywood. Disponível em: <http://www.studiodaily.com/2017/04/artificial-intelligence-comes-hollywood/> Acesso em:26 de abr. de 2020.

[3] VINCENT, J. How three french students used borrowed code to put the first ai portrait in christie's. 2018. Disponível em: <https://www.theverge.com/2018/10/23/18013190/ai-art-portrait-auction-christies-belamy-obvious-robbie-barrat-gans> Acesso em: 26 de abr. de 2020.

[4] LI, C. 10 Papers You Must Read for Deep Image Inpainting. Disponível em: <https://towardsdatascience.com/10-papers-you-must-read-for-deep-image-inpainting-2e41c589ced0> Acesso em: 29 de mar. de 2021.

[5] GOODFELLOW, I. et al. Generative Adversarial Nets. Disponível em: <https://arxiv.org/pdf/1406.2661.pdf> Acesso em: 26 de abr. de 2020.

[6] DUMOULIN, V. VISIN, F. A guide to convolution arithmetic for deep learning. Disponível em: <https://arxiv.org/pdf/1603.07285.pdf> Acesso em: 31 de mar. de 2021.

[7] KARRAS, T. AILA, T. LAINE, S. LEHTINEN, J. Progressive Growing of GANs for Improved Quality, Stability and Variation. Disponível em: <https://arxiv.org/pdf/1710.10196.pdf> Acesso em: 01 de ago. de 2020.

[8] ZHU, J. et al. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. Disponível em: <https://arxiv.org/pdf/1703.10593v6.pdf> Acesso em: 01 de ago. de 2020.

[9] KARRAS, T. AILA, T. LAINE, S. A Style-Based Generator Architecture for Generative Adversarial Networks. Disponível em: <https://arxiv.org/pdf/1812.04948.pdf> Acesso em: 01 de ago. de 2020.

[10] LEDIG, C. et al. Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network. Disponível em: <https://arxiv.org/pdf/1609.04802.pdf> Acesso em: 26 de abr. de 2020.

[11] HE, K. et al. Deep Residual Learning for Image Recognition. Disponível em: <https://arxiv.org/pdf/1512.03385.pdf> Acesso em: 30 de mar. de 2021.

[12] PATHAK, D. et al. Context Encoders: Feature Learning by Inpainting. Disponível em: <https://arxiv.org/pdf/1604.07379.pdf> Acesso em: 29 de mar. de 2021.

[13] YANG, C. et al. High-Resolution Image Inpainting using Multi-Scale Neural Patch Synthesis. Disponível em: <https://arxiv.org/pdf/1611.09969.pdf> Acesso em: 29 de mar. de 2021.

[14] GATYS, L. A. et al. A Neural Algorithm of Artistic Style. Disponível em: <https://arxiv.org/pdf/1508.06576.pdf> Acesso em: 29 de mar. de 2021.

[15] IIZUKA, S. et al. Globally and Locally Consistent Image Completion. Disponível em: <http://iizuka.cs.tsukuba.ac.jp/projects/completion/data/completion_sig2017.pdf> Acesso em: 29 de mar. de 2021.

[16] DEMIR, U. UNAL, G. Patch-Based Image Inpainting with Generative Adversarial Networks. Disponível em: <https://arxiv.org/pdf/1803.07422.pdf> Acesso em: 29 de mar. de 2021.

[17] ISOLA, P. et al. Image-to-Image Translation with Conditional Adversarial Networks. Disponível em: <https://arxiv.org/pdf/1611.07004.pdf> Acesso em: 30 de mar. de 2021.

[18] The CIFAR-10 dataset. Disponível em: <https://www.cs.toronto.edu/~kriz/cifar.html> Acesso em: 26 de abr. de 2020.

[19] BROWNLEE, J. A Gentle Introduction to Generative Adversarial Networks (GANs). 2019. Disponível em: <https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/> Acesso em: 26 de abr. de 2020.

[20] GODOY, D. Understanding binary cross-entropy / log loss: a visual explanation. Disponível em: <https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a> Acesso em: 30 de mar. de 2021.

[21] Short Introduction to Convolutions and Pooling: Deep Learning 101! 2018. Disponível em: <https://medium.com/analytics-vidhya/deep-learning-methods-1700548a3093> Acesso em: 26 de abr. de 2020.

[22] RADFORD, A. METZ, L. CHINTALA, S. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. 2016. Disponível em: <https://arxiv.org/pdf/1511.06434.pdf> Acesso em: 26 de abr. de 2020.

[23] AMOS, B. Image Completion with Deep Learning in TensorFlow. 2016. Disponível em: <https://bamos.github.io/2016/08/09/deep-completion/#step-1-interpreting-images-as-samples-from-a-probability-distribution> Acesso em: 26 de abr. de 2020.

[24] BUSHAEV, V. Adam — latest trends in deep learning optimization. 2018. Disponível em: <https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c> Acesso em: 26 de abr. de 2020.

[25] KINGMA, D. P. BA, J. L. Adam: A Method for Stochastic Optimization. 2017. Disponível em: <https://arxiv.org/pdf/1412.6980.pdf> Acesso em: 26 de abr. de 2020.

[26] SHAMRAI, D. What is Leaky ReLU?. 2019. Disponível em: <https://www.quora.com/What-is-leaky-ReLU> Acesso em: 26 de abr. de 2020.

[27] IOFFE, S. SZEGEDY, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. 2015. Disponível em: <https://arxiv.org/pdf/1502.03167v3.pdf> Acesso em: 26 de abr. de 2020.

[28] Lazy Programmer Inc. Deep Learning: GANs and Variational Autoencoders. Disponível em: <https://www.udemy.com/course/deep-learning-gans-and-variational-autoencoders/> Acesso em: 26 de abr. de 2020.

8. APÊNDICE

8.1. Código DCGAN utilizando conjunto de dados MNIST

O código abaixo foi desenvolvido pela Lazy Programmer Inc. no curso “Deep Learning: GANs and Variational Autoencoders[28]”:

```
from __future__ import print_function, division
from builtins import range, input
# Note: you may need to update your version of future

import os
import scipy as sp
import numpy as np
import pandas as pd
import tensorflow as tf # need to install a tensorflow
version 1.x
import matplotlib.pyplot as plt
from scipy.misc import imread # need to install a scipy
version lower than 1.2.0
from sklearn.utils import shuffle
from datetime import datetime

# some constants
LEARNING_RATE = 0.0002
BETA1 = 0.5
BATCH_SIZE = 64
EPOCHS = 2
SAVE_SAMPLE_PERIOD = 50


# make dir to save samples
if not os.path.exists('samples'):
  os.mkdir('samples')

# Leaky Rectified Linear Unit function
def lrelu(x, alpha=0.2):
  return tf.maximum(alpha*x, x)

# get MNIST images from "../large_files/train.csv"
def get_mnist(limit=None):
  if not os.path.exists('../large_files'):
    print("You must create a folder called large_files
adjacent to the class folder first.")
```

```
  if not os.path.exists('../large_files/train.csv'):
    print("Looks like you haven't downloaded the data or it's
not in the right spot.")
    print("Please get train.csv from
https://www.kaggle.com/c/digit-recognizer")
    print("and place it in the large_files folder.")

  print("Reading in and transforming data...")
  df = pd.read_csv('../large_files/train.csv')
  data = df.values
  # np.random.shuffle(data)
  X = data[:, 1:] / 255.0 # data is from 0..255
  Y = data[:, 0]
  X, Y = shuffle(X, Y)
  if limit is not None:
    X, Y = X[:limit], Y[:limit]
  return X, Y

def scale_image(im):
  # scale to (-1, +1)
  return (im / 255.0)*2 - 1

def files2images(filenames):
  return [scale_image(imread(fn)) for fn in filenames]

class ConvLayer:
  def __init__(self, name, mi, mo, apply_batch_norm,
filtersz=5, stride=2, f=tf.nn.relu):
    # mi = input feature map size
    # mo = output feature map size
    # self.W =
tf.Variable(0.02*tf.random_normal(shape=(filtersz, filtersz,
mi, mo)))
    # self.b = tf.Variable(np.zeros(mo, dtype=np.float32))
    self.W = tf.get_variable(
      "W_%s" % name,
      shape=(filtersz, filtersz, mi, mo),
      # initializer=tf.contrib.layers.xavier_initializer(),

initializer=tf.truncated_normal_initializer(stddev=0.02),
    )
    self.b = tf.get_variable(
      "b_%s" % name,
      shape=(mo,),
      initializer=tf.zeros_initializer(),
    )
    self.name = name
    self.f = f
```

```
    self.stride = stride
    self.apply_batch_norm = apply_batch_norm
    self.params = [self.W, self.b]

  def forward(self, X, reuse, is_training):
    # print("**************** reuse:", reuse)
    conv_out = tf.nn.conv2d(
      X,
      self.W,
      strides=[1, self.stride, self.stride, 1],
      padding='SAME'
    )
    conv_out = tf.nn.bias_add(conv_out, self.b)

    # apply batch normalization
    if self.apply_batch_norm:
      conv_out = tf.contrib.layers.batch_norm(
        conv_out,
        decay=0.9,
        updates_collections=None,
        epsilon=1e-5,
        scale=True,
        is_training=is_training,
        reuse=reuse,
        scope=self.name,
      )
    return self.f(conv_out)


class FractionallyStridedConvLayer:
  def __init__(self, name, mi, mo, output_shape,
apply_batch_norm, filtersz=5, stride=2, f=tf.nn.relu):
    # mi = input feature map size
    # mo = output feature map size
    # NOTE!!! shape is specified in the OPPOSITE way from
regular conv
    # self.W =
tf.Variable(0.02*tf.random_normal(shape=(filtersz, filtersz,
mo, mi)))
    # self.b = tf.Variable(np.zeros(mo, dtype=np.float32))
    self.W = tf.get_variable(
      "W_%s" % name,
      shape=(filtersz, filtersz, mo, mi),
      # initializer=tf.contrib.layers.xavier_initializer(),
      initializer=tf.random_normal_initializer(stddev=0.02),
    )
    self.b = tf.get_variable(
      "b_%s" % name,
```

```
      shape=(mo,),
      initializer=tf.zeros_initializer(),
    )
    self.f = f
    self.stride = stride
    self.name = name
    self.output_shape = output_shape
    self.apply_batch_norm = apply_batch_norm
    self.params = [self.W, self.b]

  def forward(self, X, reuse, is_training):
    conv_out = tf.nn.conv2d_transpose(
      value=X,
      filter=self.W,
      output_shape=self.output_shape,
      strides=[1, self.stride, self.stride, 1],
    )
    conv_out = tf.nn.bias_add(conv_out, self.b)

    # apply batch normalization
    if self.apply_batch_norm:
      conv_out = tf.contrib.layers.batch_norm(
        conv_out,
        decay=0.9,
        updates_collections=None,
        epsilon=1e-5,
        scale=True,
        is_training=is_training,
        reuse=reuse,
        scope=self.name,
      )

    return self.f(conv_out)


class DenseLayer(object):
  def __init__(self, name, M1, M2, apply_batch_norm,
f=tf.nn.relu):
    self.W = tf.get_variable(
      "W_%s" % name,
      shape=(M1, M2),
      initializer=tf.random_normal_initializer(stddev=0.02),
    )
    self.b = tf.get_variable(
      "b_%s" % name,
      shape=(M2,),
      initializer=tf.zeros_initializer(),
    )
```

```
    self.f = f
    self.name = name
    self.apply_batch_norm = apply_batch_norm
    self.params = [self.W, self.b]

  def forward(self, X, reuse, is_training):
    a = tf.matmul(X, self.W) + self.b

    # apply batch normalization
    if self.apply_batch_norm:
      a = tf.contrib.layers.batch_norm(
        a,
        decay=0.9,
        updates_collections=None,
        epsilon=1e-5,
        scale=True,
        is_training=is_training,
        reuse=reuse,
        scope=self.name,
      )
    return self.f(a)


class DCGAN:
  def __init__(self, img_length, num_colors, d_sizes,
g_sizes):

    # save for later
    self.img_length = img_length
    self.num_colors = num_colors
    self.latent_dims = g_sizes['z']

    # define the input data
    self.X = tf.placeholder(
      tf.float32,
      shape=(None, img_length, img_length, num_colors),
      name='X'
    )
    self.Z = tf.placeholder(
      tf.float32,
      shape=(None, self.latent_dims),
      name='Z'
    )

    # note: by making batch_sz a placeholder, we can specify
a variable
    # number of samples in the FS-conv operation where we are
required
```

```
    # to pass in output_shape
    # we need only pass in the batch size via feed_dict
    self.batch_sz = tf.placeholder(tf.int32, shape=(),
name='batch_sz')

    # build the discriminator
    logits = self.build_discriminator(self.X, d_sizes)

    # build generator
    self.sample_images = self.build_generator(self.Z,
g_sizes)

    # get sample logits
    with tf.variable_scope("discriminator") as scope:
      scope.reuse_variables()
      sample_logits = self.d_forward(self.sample_images,
True)

    # get sample images for test time (batch norm is
different)
    with tf.variable_scope("generator") as scope:
      scope.reuse_variables()
      self.sample_images_test = self.g_forward(
        self.Z, reuse=True, is_training=False
      )

    # build costs
    self.d_cost_real =
tf.nn.sigmoid_cross_entropy_with_logits(
      logits=logits,
      labels=tf.ones_like(logits)
    )
    self.d_cost_fake =
tf.nn.sigmoid_cross_entropy_with_logits(
      logits=sample_logits,
      labels=tf.zeros_like(sample_logits)
    )
    self.d_cost = tf.reduce_mean(self.d_cost_real) +
tf.reduce_mean(self.d_cost_fake)
    self.g_cost = tf.reduce_mean(
      tf.nn.sigmoid_cross_entropy_with_logits(
        logits=sample_logits,
        labels=tf.ones_like(sample_logits)
      )
    )
    real_predictions = tf.cast(logits > 0, tf.float32)
    fake_predictions = tf.cast(sample_logits < 0, tf.float32)
    num_predictions = 2.0*BATCH_SIZE
```

```
    num_correct = tf.reduce_sum(real_predictions) +
tf.reduce_sum(fake_predictions)
    self.d_accuracy = num_correct / num_predictions


    # optimizers
    self.d_params = [t for t in tf.trainable_variables() if
t.name.startswith('d')]
    self.g_params = [t for t in tf.trainable_variables() if
t.name.startswith('g')]

    self.d_train_op = tf.train.AdamOptimizer(
      LEARNING_RATE, beta1=BETA1
    ).minimize(
      self.d_cost, var_list=self.d_params
    )
    self.g_train_op = tf.train.AdamOptimizer(
      LEARNING_RATE, beta1=BETA1
    ).minimize(
      self.g_cost, var_list=self.g_params
    )

    # show_all_variables()
    # exit()

    # set up session and variables for later
    self.init_op = tf.global_variables_initializer()
    self.sess = tf.InteractiveSession()
    self.sess.run(self.init_op)


  def build_discriminator(self, X, d_sizes):
    with tf.variable_scope("discriminator") as scope:

      # build conv layers
      self.d_convlayers = []
      mi = self.num_colors
      dim = self.img_length
      count = 0
      for mo, filtersz, stride, apply_batch_norm in
d_sizes['conv_layers']:
        # make up a name - used for get_variable
        name = "convlayer_%s" % count
        count += 1

        layer = ConvLayer(name, mi, mo, apply_batch_norm,
filtersz, stride, lrelu)
        self.d_convlayers.append(layer)
```

```
        mi = mo
        print("dim:", dim)
        dim = int(np.ceil(float(dim) / stride))


      mi = mi * dim * dim
      # build dense layers
      self.d_denselayers = []
      for mo, apply_batch_norm in d_sizes['dense_layers']:
        name = "denselayer_%s" % count
        count += 1

        layer = DenseLayer(name, mi, mo, apply_batch_norm,
lrelu)
        mi = mo
        self.d_denselayers.append(layer)


      # final logistic layer
      name = "denselayer_%s" % count
      self.d_finallayer = DenseLayer(name, mi, 1, False,
lambda x: x)

      # get the logits
      logits = self.d_forward(X)

      # build the cost later
      return logits


  def d_forward(self, X, reuse=None, is_training=True):
    # encapsulate this because we use it twice
    output = X
    for layer in self.d_convlayers:
      output = layer.forward(output, reuse, is_training)
    output = tf.contrib.layers.flatten(output)
    for layer in self.d_denselayers:
      output = layer.forward(output, reuse, is_training)
    logits = self.d_finallayer.forward(output, reuse,
is_training)
    return logits


  def build_generator(self, Z, g_sizes):
    with tf.variable_scope("generator") as scope:

      # determine the size of the data at each step
      dims = [self.img_length]
```

```
      dim = self.img_length
      for _, _, stride, _ in
reversed(g_sizes['conv_layers']):
        dim = int(np.ceil(float(dim) / stride))
        dims.append(dim)

      # note: dims is actually backwards
      # the first layer of the generator is actually last
      # so let's reverse it
      dims = list(reversed(dims))
      print("dims:", dims)
      self.g_dims = dims


      # dense layers
      mi = self.latent_dims
      self.g_denselayers = []
      count = 0
      for mo, apply_batch_norm in g_sizes['dense_layers']:
        name = "g_denselayer_%s" % count
        count += 1

        layer = DenseLayer(name, mi, mo, apply_batch_norm)
        self.g_denselayers.append(layer)
        mi = mo

      # final dense layer
      mo = g_sizes['projection'] * dims[0] * dims[0]
      name = "g_denselayer_%s" % count
      layer = DenseLayer(name, mi, mo, not
g_sizes['bn_after_project'])
      self.g_denselayers.append(layer)


      # fs-conv layers
      mi = g_sizes['projection']
      self.g_convlayers = []

      # output may use tanh or sigmoid
      num_relus = len(g_sizes['conv_layers']) - 1
      activation_functions = [tf.nn.relu]*num_relus +
[g_sizes['output_activation']]

      for i in range(len(g_sizes['conv_layers'])):
        name = "fs_convlayer_%s" % i
        mo, filtersz, stride, apply_batch_norm =
g_sizes['conv_layers'][i]
        f = activation_functions[i]
```

```
        output_shape = [self.batch_sz, dims[i+1], dims[i+1],
mo]
        print("mi:", mi, "mo:", mo, "outp shape:",
output_shape)
        layer = FractionallyStridedConvLayer(
          name, mi, mo, output_shape, apply_batch_norm,
filtersz, stride, f
        )
        self.g_convlayers.append(layer)
        mi = mo

      # get the output
      self.g_sizes = g_sizes
      return self.g_forward(Z)


  def g_forward(self, Z, reuse=None, is_training=True):
    # dense layers
    output = Z
    for layer in self.g_denselayers:
      output = layer.forward(output, reuse, is_training)

    # project and reshape
    output = tf.reshape(
      output,
      [-1, self.g_dims[0], self.g_dims[0],
self.g_sizes['projection']],
    )

    # apply batch norm
    if self.g_sizes['bn_after_project']:
      output = tf.contrib.layers.batch_norm(
        output,
        decay=0.9,
        updates_collections=None,
        epsilon=1e-5,
        scale=True,
        is_training=is_training,
        reuse=reuse,
        scope='bn_after_project'
      )

    # pass through fs-conv layers
    for layer in self.g_convlayers:
      output = layer.forward(output, reuse, is_training)

    return output
```

```
  def fit(self, X):
    d_costs = []
    g_costs = []

    N = len(X)
    n_batches = N // BATCH_SIZE
    total_iters = 0
    for i in range(EPOCHS):
      print("epoch:", i)
      np.random.shuffle(X)
      for j in range(n_batches):
        t0 = datetime.now()

         batch = X[j*BATCH_SIZE:(j+1)*BATCH_SIZE]

        Z = np.random.uniform(-1, 1, size=(BATCH_SIZE,
self.latent_dims))

        # train the discriminator
        _, d_cost, d_acc = self.sess.run(
          (self.d_train_op, self.d_cost, self.d_accuracy),
          feed_dict={self.X: batch, self.Z: Z, self.batch_sz:
BATCH_SIZE},
        )
        d_costs.append(d_cost)

        # train the generator
        _, g_cost1 = self.sess.run(
          (self.g_train_op, self.g_cost),
          feed_dict={self.Z: Z, self.batch_sz: BATCH_SIZE},
        )
        # g_costs.append(g_cost1)
        _, g_cost2 = self.sess.run(
          (self.g_train_op, self.g_cost),
          feed_dict={self.Z: Z, self.batch_sz: BATCH_SIZE},
        )
        g_costs.append((g_cost1 + g_cost2)/2) # just use the
avg

        print("  batch: %d/%d  -  dt: %s - d_acc: %.2f" %
(j+1, n_batches, datetime.now() - t0, d_acc))


        # save samples periodically
        total_iters += 1
        if total_iters % SAVE_SAMPLE_PERIOD == 0:
          print("saving a sample...")
```

```
          samples = self.sample(64) # shape is (64, D, D,
color)

          # for convenience
          d = self.img_length

          if samples.shape[-1] == 1:
            # if color == 1, we want a 2-D image (N x N)
            samples = samples.reshape(64, d, d)
            flat_image = np.empty((8*d, 8*d))

            k = 0
            for i in range(8):
              for j in range(8):
                flat_image[i*d:(i+1)*d, j*d:(j+1)*d] =
samples[k].reshape(d, d)
                k += 1

            # plt.imshow(flat_image, cmap='gray')
          else:
            # if color == 3, we want a 3-D image (N x N x 3)
            flat_image = np.empty((8*d, 8*d, 3))
            k = 0
            for i in range(8):
              for j in range(8):
                flat_image[i*d:(i+1)*d, j*d:(j+1)*d] =
samples[k]
                k += 1
            # plt.imshow(flat_image)

          # plt.savefig('samples/samples_at_iter_%d.png' %
total_iters)
          sp.misc.imsave(
            'samples/samples_at_iter_%d.png' % total_iters,
            flat_image,
          )

    # save a plot of the costs
    plt.clf()
    plt.plot(d_costs, label='discriminator cost')
    plt.plot(g_costs, label='generator cost')
    plt.legend()
    plt.savefig('cost_vs_iteration.png')

  def sample(self, n):
    Z = np.random.uniform(-1, 1, size=(n, self.latent_dims))
    samples = self.sess.run(self.sample_images_test,
feed_dict={self.Z: Z, self.batch_sz: n})
```

```
    return samples

def get_mnist(limit=None):
  if not os.path.exists('../large_files'):
    print("You must create a folder called large_files
adjacent to the class folder first.")
  if not os.path.exists('../large_files/train.csv'):
    print("Looks like you haven't downloaded the data or it's
not in the right spot.")
    print("Please get train.csv from
https://www.kaggle.com/c/digit-recognizer")
    print("and place it in the large_files folder.")

  print("Reading in and transforming data...")
  df = pd.read_csv('../large_files/train.csv')
  data = df.values
  # np.random.shuffle(data)
  X = data[:, 1:] / 255.0 # data is from 0..255
  Y = data[:, 0]
  X, Y = shuffle(X, Y)
  if limit is not None:
    X, Y = X[:limit], Y[:limit]
  return X, Y

def mnist():
  X, Y = get_mnist()
  X = X.reshape(len(X), 28, 28, 1)
  dim = X.shape[1]
  colors = X.shape[-1]

  # for mnist
  d_sizes = {
    'conv_layers': [(2, 5, 2, False), (64, 5, 2, True)],
    'dense_layers': [(1024, True)],
  }
  g_sizes = {
    'z': 100,
    'projection': 128,
    'bn_after_project': False,
    'conv_layers': [(128, 5, 2, True), (colors, 5, 2,
False)],
    'dense_layers': [(1024, True)],
    'output_activation': tf.sigmoid,
  }


  # setup gan
  # note: assume square images, so only need 1 dim
```

```
  gan = DCGAN(dim, colors, d_sizes, g_sizes)
  gan.fit(X)
  # samples = gan.sample(1) # just making sure it works

  # since training will take a considerable
  # amount of time, let's just save some
  # samples to disk rather than plotting now


if __name__ == '__main__':
  mnist()
```