

INSTITUTO TECNOLÓGICO DE AERONÁUTICA



Felipe da Conceição Guimarães

**ESTUDO E IMPLEMENTAÇÃO DE REDES
NEURAIS GERADORAS ADVERSÁRIAS**

Trabalho de Graduação
2019

Curso de Engenharia de Computação

Felipe da Conceição Guimarães

**ESTUDO E IMPLEMENTAÇÃO DE REDES
NEURAIS GERADORAS ADVERSARIAIS**

Orientador

Prof. Dr. Carlos Henrique Quartucci Forster (ITA)

ENGENHARIA DE COMPUTAÇÃO

**SÃO JOSÉ DOS CAMPOS
INSTITUTO TECNOLÓGICO DE AERONÁUTICA**

Dados Internacionais de Catalogação-na-Publicação (CIP)
Divisão de Informação e Documentação

Guimarães, Felipe da Conceição
Estudo e implementação de Redes Neurais Geradoras Adversariais / Felipe da Conceição Guimarães.
São José dos Campos, 2019.
39f.

Trabalho de Graduação – Curso de Engenharia de Computação– Instituto Tecnológico de Aeronáutica, 2019. Orientador: Prof. Dr. Carlos Henrique Quartucci Forster.

1. Aprendizado de Máquina. 2. Redes Neurais. 3. Redes Geradoras. I. Instituto Tecnológico de Aeronáutica. II. Título.

REFERÊNCIA BIBLIOGRÁFICA

GUIMARÃES, Felipe da Conceição. **Estudo e implementação de Redes Neurais Geradoras Adversariais**. 2019. 39f. Trabalho de Conclusão de Curso (Graduação) – Instituto Tecnológico de Aeronáutica, São José dos Campos.


CESSÃO DE DIREITOS

NOME DO AUTOR: Felipe da Conceição Guimarães

TÍTULO DO TRABALHO: Estudo e implementação de Redes Neurais Geradoras Adversariais.

TIPO DO TRABALHO/ANO: Trabalho de Conclusão de Curso (Graduação) / 2019

É concedida ao Instituto Tecnológico de Aeronáutica permissão para reproduzir cópias deste trabalho de graduação e para emprestar ou vender cópias somente para propósitos acadêmicos e científicos. O autor reserva outros direitos de publicação e nenhuma parte deste trabalho de graduação pode ser reproduzida sem a autorização do autor.



Felipe da Conceição Guimarães
Vila das Acácias, H8-B, 217
12228-461 – São José dos Campos – SP


ESTUDO E IMPLEMENTAÇÃO DE REDES NEURAIS GERADORAS ADVERSÁRIAS

Essa publicação foi aceita como Relatório Final de Trabalho de Graduação



Felipe da Conceição Guimarães

Autor



Carlos Henrique Quartucci Forster (ITA)

Orientador



Prof. Dr. Inaldo Capistrano Costa
Coordenador do Curso de Engenharia de Computação

São José dos Campos, 27 de novembro de 2019.

À minha vontade de não desistir

Agradecimentos

Agradeço a mim mesmo, por ter aguentado chegar até aqui. Só eu sei que o caminho foi sofrido e difícil, e o quanto precisei ser forte para aguentar até o final. Obrigado, Felipe.

Agradeço também a minha família. Meus avós e meus pais pelo apoio, amor e suporte que me deram durante todo esse tempo. Espero um dia poder devolver em dobro tudo que fizeram por mim.

Agradeço aos meus amigos do H8, que é a coisa mais positiva que levo desses últimos cinco anos, e que sem os quais o ITA não valeria nem um pouco a pena. Obrigado por me ajudarem a aguentar tantos dias de dor e sofrimento e me mostrarem que eu não estava sozinho. Só nós sabemos o quão desnecessariamente sofrida é essa jornada e vocês sempre me lembravam que existe um mundo lá fora do ITA e que uma hora isso iria acabar.

*“What’s the word for when it feels
inside your heart that everything
in the world is all right?”*

— RIN TEZUKA

Resumo

Este trabalho tem como objetivo realizar um estudo sobre um recente framework de redes neurais usado para geração de imagens, as *Generative Adversarial Networks*, ou GANs. Esse ramo de aplicação de redes neurais é bem diferente da aplicação comum onde se classifica entidades a partir de uma entrada.

Primeiramente discutiremos alguns fundamentos de redes neurais clássicas e seu funcionamento em problemas de classificação. Depois uma discussão teórica sobre as redes neurais convolucionais, frequentemente utilizadas em problemas de classificação de imagens. E por fim, uma discussão sobre as GANs.

Em seguida, faremos uma implementação de uma GAN para a geração de imagens a partir de um banco de dados e estudaremos o efeito de diferentes arquiteturas e parâmetros em seus resultados.

Abstract

This work has the objective of studying the framework of Generative Adversarial Networks, or GANs, used for image generation. This area of application of neural networks is different from the regular classification problems, where we need to predict a label for some input.

First, we will discuss some theories of regular neural networks and how they work in classification problems. After, a little about deep convolutional neural networks, mainly used in image classification problems. And lastly, a discussion about GANs.

Therefore, we will implement a GAN for creating images from a data set and will study the effects of different architectures and parameters on the results.

Lista de Figuras

FIGURA 2.1 – Modelo de rede neural com múltiplas camadas.	15
FIGURA 2.2 – Nessa imagem, podemos ver a diferença da quantidade de influência que um neurônio de uma camada anterior pode exercer na camada seguinte. Acima temos o exemplo de uma rede neural convolucional e abaixo o exemplo de uma rede neural simples. (GOODFELLOW <i>et al.</i> , 2016)	17
FIGURA 2.3 – Nessa imagem, temos em evidência no exemplo acima, todas as ligações que compartilham parâmetro, poupando memória e processamento. Abaixo, vemos que numa rede neural simples, um parâmetro pertence a apenas uma ligação. (GOODFELLOW <i>et al.</i> , 2016)	17
FIGURA 2.4 – Modelo representando o funcionamento do treinamento de uma GAN. A rede discriminadora recebe tanto imagens verdadeiras quanto imagens produzidas pela rede geradora a partir de um ruído aleatório, e é treinada para discriminar as imagens corretamente. (SILVA, 2017)	20
FIGURA 3.1 – Exemplo de imagens usadas para o treinamento das redes.	22
FIGURA 3.2 – Função de custo utilizada.	27
FIGURA 4.1 – Exemplos de imagens geradas pela rede neural final.	30
FIGURA 4.2 – Exemplos de imagens geradas pela rede neural final sem a tangente hiperbólica.	30
FIGURA 4.3 – Exemplos de imagens onde o discriminador não foi capaz de treinar o suficiente no número de épocas disponível devido à sua arquitetura.	31
FIGURA 4.4 – Exemplos de imagens produzidas pelas redes neurais densas, citadas em 3.2.1.1 e 3.2.2.1.	32

Sumário

1	INTRODUÇÃO	12
2	CONCEITOS TEÓRICOS	14
2.1	Redes Neurais	14
2.2	Redes Neurais Convolucionais	15
2.2.1	Convolução	16
2.2.2	Pooling	18
2.3	Generative Adversarial Nets	18
2.3.1	Modelo	18
2.4	Deep Convolutional Generative Adversarial Nets	20
3	IMPLEMENTAÇÃO	21
3.1	Objetivo e data set	21
3.1.1	Data set	21
3.1.2	Código de pre-processamento do dataset	21
3.1.3	Framework utilizado	22
3.2	Código implementado	23
3.2.1	Gerador	23
3.2.2	Discriminador	25
3.2.3	Função de custo	26
3.2.4	Treinamento	27
4	RESULTADOS	29
4.1	Imagens geradas	29

4.1.1	Versão final apresentada	30
4.1.2	Removendo a tangente hiperbólica como função de ativação	30
4.1.3	Casos em que o discriminador é fraco	31
4.1.4	Casos em que o gerador é fraco	31
4.1.5	Redes neurais sem convolução	31
5	CONCLUSÕES	33
	REFERÊNCIAS	34
	APÊNDICE A – CÓDIGO COMPLETO	36
A.1	Data set	36
A.2	Modelo	37

1 Introdução

Inteligência artificial é uma área da computação que estuda a inteligência demonstrada por máquinas ou computadores. Uma das técnicas mais populares de IA atualmente é a Rede Neural, que é uma técnica de aprendizado de máquinas desenvolvida desde o século passado, com seu primeiro modelo teórico tendo sido apresentado em 1943 pelos pesquisadores Warren McCulloch e Walter Pitts. O modelo se tornou muito popular com o passar dos anos devido a novos descobrimentos que foram cruciais para a possibilidade de aplicações práticas, como algoritmos de backpropagation e aplicações em diversas áreas do aprendizado supervisionado para classificações de imagens, sons e textos, mostrando a sua versatilidade para resolver diferentes problemas. Ela é uma técnica de aprendizado de máquina, ou Machine Learning, onde o sistema precisa aprender sobre o ambiente que influencia sua decisão.

Seu uso mais popular e simples é resolver problemas classificatórios ou discriminatórios. Nesse tipo de problema, o nosso modelo recebe no seu input um elemento, e a partir de suas características retorna como saída uma classe ou um conjunto de labels para esse elemento. Um exemplo muito popular é classificar um e-mail como spam ou não-spam a partir de seu conteúdo, como alguns servidores de email, como o Gmail, já fazem atualmente.

Outro exemplo é classificar imagens de números manuscritos, dizendo qual é o dígito que está na imagem. Tal exemplo é muito usado academicamente devido ao MNIST, que é um database com sessenta mil imagens classificadas e de mesma dimensão, facilitando o seu uso. Nesse problema, uma rede neural é treinada com imagens já classificadas de dígitos manuscritos, e depois é passada para sua entrada uma imagem de um dígito manuscrito e ela nos retorna o valor do dígito presente na imagem.

Porém, também podemos abordar a rede neural no sentido inverso, onde oferecemos para ela um conjunto de características, e ela nos retorna um elemento que se encaixa naquele conjunto. A ideia é ter uma rede neural que seja capaz de criar coisas (como músicas, texto e imagens) que se encaixem em uma determinada descrição. Para isso, faremos uso das GAN (generative adversarial networks), que foram primeiramente publicadas no artigo de um grupo de pesquisadores na universidade de Montreal (GOODFELLOW *et al.*, 2014).

Neste paper, é proposta uma solução para alguns problemas e dificuldades já enfrentados em outras tentativas de se usar redes neurais geradoras (como treinar a rede neural de um jeito eficiente por exemplo).

Nesse sistema, temos duas redes (uma que gera os elementos, e outra que julga se os elementos gerados são legítimos), num modelo de minimax, onde uma vai melhorando junto com a outra, ambas sendo adversárias. O objetivo desse trabalho é estudar esse framework e criar uma implementação para a geração de imagens do rostos de desenhos animados, assim como estudar os efeitos de diferentes arquiteturas no resultado e na eficiência da mesma.

2 Conceitos teóricos

2.1 Redes Neurais

As redes neurais artificiais são um modelo computacional inspirado, mas não idêntico, aos neurônios dos animais. Nesse modelo, a unidade básica e mais simples de processamento é um nó, também chamado de neurônio, que recebe como input a saída de outros neurônios e produz como output uma saída própria que pode ser consumida por outros neurônios ou pode ser a saída final do algoritmo.

Podemos organizar esses neurônios em camadas, obtendo as redes neurais em multi camadas, ou em inglês, as *feedforward neural networks*. O objetivo dessas redes é aproximar uma função f^* . Por exemplo, em um problema de classificação, podemos descrever uma função f^* com um mapeamento $y = f^*(x)$, onde mapeamos vários inputs x pra outputs y . Uma rede neural multicamadas vai ter um conjunto θ de pesos para as ligações entre seus neurônios, que um algoritmo de treinamento deve ajustar para fazer uma função $f(x)$ melhor se aproximar de $f^*(x)$ (GOODFELLOW *et al.*, 2016).

No caso das redes neurais em multi camadas, na prática cada camada é um conjunto de neurônios que recebe entradas da camada anterior e produz saídas pra próxima camada. Então, podemos dizer que a função f é uma composição das funções definidas por cada uma de suas camadas.

$$f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$$

Onde $f^{(i)}$ é a função definida pela camada i da nossa rede.

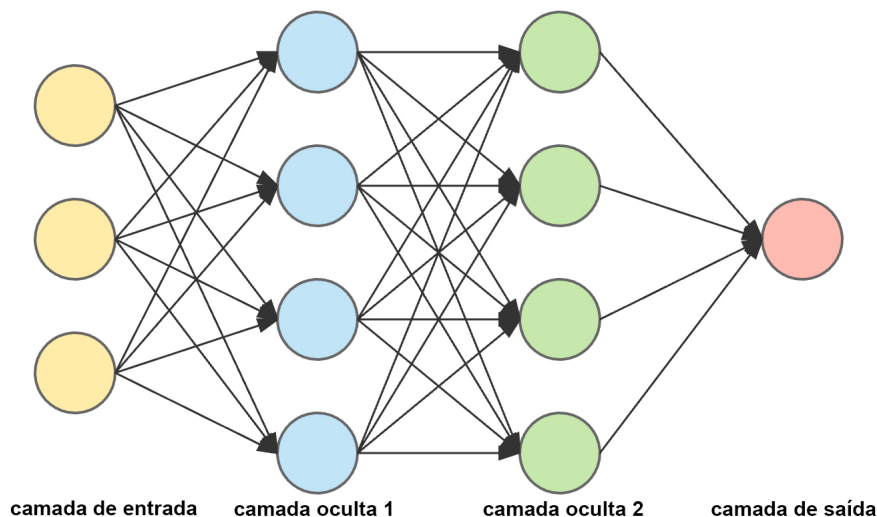


FIGURA 2.1 – Modelo de rede neural com múltiplas camadas.

O treinamento supervisionado de uma rede desse tipo ocorre a partir de um conjunto finito de pares (x, y) de entradas e saídas da função, e temos como objetivo fazer a função f se aproximar de f^* em todo o domínio possível da função. Com isso, definimos uma função de custo $J(\theta)$ para um conjunto de parâmetros θ . Essa função define o quão longe de f^* a função f está para o conjunto θ de parâmetros a partir dos valores de f para um conjunto x de entradas para os quais sabemos os resultados y da função f^* . Na prática, queremos minimizar o custo J , e para isso, é usado algoritmos de aprendizado a partir do gradiente da função de custo, procurando modificar os valores dos parâmetros na direção em que o gradiente decresce, também conhecido como *gradient descent*.

A escolha do algoritmo de *gradient descent* e da função de custo dependem do problema a ser resolvido, podendo influenciar tanto na velocidade de treinamento quanto no quão bom os resultados vão ser para um determinado caso. No decorrer desse trabalho, veremos os diferentes métodos que podemos usar para o objetivo específico desse projeto.

2.2 Redes Neurais Convolucionais

Problemas de classificações de imagens acabam tendo características que são invariantes à translação e rotação. Por exemplo, uma foto de uma pessoa transladada um pixel pra direita e rotacionada 5 graus continua sendo uma foto de uma pessoa. Nas entradas das redes neurais clássicas, isso se torna um problema já que cada neurônio da camada de entrada é invariante em sua posição. As redes neurais convolucionais são um tipo de redes neurais que se mostraram extremamente eficiente nesse tipo de problema. Seu nome

vem do fato de ela aplicar uma convolução em vez de uma multiplicação de matrizes no cálculo do resultado de uma camada da rede. Na prática um nó de uma camada não se conecta com todos os da seguinte, como numa rede neural convencional.

2.2.1 Convolução

A operação de convolução de funções f e g , na matemática, pode ser definida como:

$$s(t) = \int f(a)g(t-a)da = (f * g)(t)$$

Em redes neurais, o primeiro argumento (a função x) é o input, e a função g é chamada de **kernel**, e o output (a função s) é chamado de mapa de características, ou *feature map*.

Na prática, a convolução feita numa rede neural convolucional é discreta, agindo sobre nosso número finito de neurônios.

$$s(t) = (f * g)(t) = \sum_{a=-\infty}^{\infty} f(a)g(t-a)$$

No caso de uma imagem, que é bidimensional, temos que a nossa convolução terá que iterar sobre os pixels da imagem e as ligações dos neurônios (kernels). Com isso, chamando a função g de I , representando a nossa imagem e a função f de K representando nosso kernel, temos:

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(i-m, j-n)K(m, n)$$

A equação acima nos diz que o kernel agirá sobre um intervalo de neurônios próximos e retirando deles o output para a próxima camada, de modo que neurônios muito distantes não tem influência sobre os outros, que é o princípio de interações esparsas que as redes neurais convolucionais nos provê.

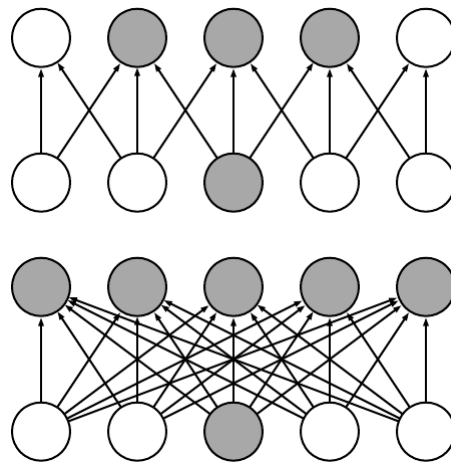


FIGURA 2.2 – Nessa imagem, podemos ver a diferença da quantidade de influência que um neurônio de uma camada anterior pode exercer na camada seguinte. Acima temos o exemplo de uma rede neural convolucional e abaixo o exemplo de uma rede neural simples. (GOODFELLOW *et al.*, 2016)

Uma outra vantagem são os parâmetros compartilhados, já que o kernel é único, o que aumenta a eficiência de treinamento já que temos que salvar menos informações. Na imagem seguinte, podemos ver a diferença entre uma rede convolucional, onde todos os parâmetros das ligações evidenciadas são o mesmo, e de uma rede neural tradicional.

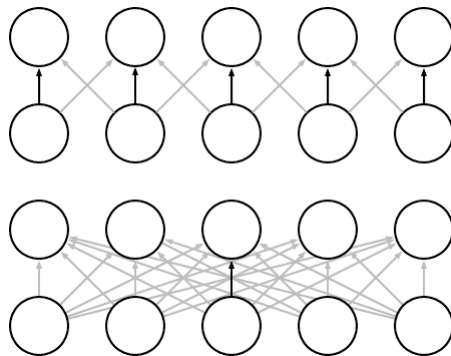


FIGURA 2.3 – Nessa imagem, temos em evidência no exemplo acima, todas as ligações que compartilham parâmetro, poupando memória e processamento. Abaixo, vemos que numa rede neural simples, um parâmetro pertence a apenas uma ligação. (GOODFELLOW *et al.*, 2016)

Além disso, temos a equivariância em relação a translação, visto que o mesmo kernel é utilizado nas diversas áreas da entrada para identificar as características importantes.

2.2.2 Pooling

Depois de passar pela fase de convolução e por uma função de ativação, a saída de uma camada de uma rede neural convolucional ainda precisa passar pela etapa de *pooling*. Nessa etapa, a função de pooling age sobre uma região do output, geralmente um retângulo, e nos dá uma informação sobre os outputs dentro desses retângulos. Por exemplo, o valor máximo ou a média das saídas na região.

A etapa de pooling seleciona as características mais relevantes e pode reduzir a dimensionalidade da entrada para a próxima camada. Consequentemente, ajuda a reduzir o overfitting. Além disso, ajuda a diminuir o número de neurônios na camada seguinte, reduzindo bastante o tempo de treinamento da rede.

2.3 Generative Adversarial Nets

Nas seções anteriores, discutimos um pouco sobre as redes neurais e suas aplicações em problemas de classificação. Nesse tipo de problema, como discutido em seções anteriores, se dá por tentar se aproximar de uma função f^* que classifica entradas em categorias. Porém, por muito tempo se estudou maneiras de se resolver o problema contrário, onde queremos produzir um entrada fictícia x a partir de uma classificação y .

O framework das Generative Adversarial Nets, ou GANs, foi proposto num artigo de um time liderado por Ian Goodfellow na universidade de Montreal. O principal problema abordado e usado como motivação para a defesa do potencial desse framework foram as tentativas anteriores de se usar redes neurais como modelos geradores. Alguns exemplos são as funções erro intratáveis matematicamente e necessidade de usar cadeias de Markov. No modelo das GANs, só precisaremos usar técnicas comuns usadas em sistemas classificatórios como gradient descent, backpropagation (GOODFELLOW *et al.*, 2014).

2.3.1 Modelo

O modelo define dois agentes: um gerador e um discriminador. Nos referimos ao gerador pela função G e o discriminador pela função D .

$$G(z, \theta_g) \text{ e } D(x, \theta_d)$$

Onde z é uma entrada dada para a rede neural geradora e x é uma saída de G . $D(x)$ representa a probabilidade de x ter saído do conjunto de dados real, usado para treinar este sistema. Nós treinamos D para classificar corretamente tanto os elementos que vieram do

conjunto de dados real quanto os elementos produzidos por G , dizendo que os do primeiro conjunto são verdadeiros (mais próximos de 1) e os do segundo são falsos (mais próximos de 0). Paralelamente, nós treinamos G para produzir o efeito contrário, minimizando a probabilidade de D dizer que seus outputs são falsos.

Em outras palavras, G e D jogam um jogo minimax em torno da seguinte função V :

$$\min_G \max_D V(G, D) = \mathbb{E}(x_{\text{in data}})[\log(D(x))] + \mathbb{E}(z_{\text{in input}})[\log(1 - D(G(z)))] \quad (2.1)$$

O equilíbrio do processo de treinamento é fundamental para se gerar um resultado satisfatório. Otimizar G demais em relação a D pode acabar gerando overfitting onde G se aproveita das vulnerabilidades de D para gerar seus elementos de apenas um modo, tirando a variedade do resultado. Por outro lado, ter um D muito bom em relação a G faz com que G não tenha informações suficientes de pra qual direção ir no seu treinamento. Por isso, no algoritmo usado, intercalamos em cada loop da iteração em treinar D e treinar G .

Em cada loop, intercalamos entre selecionar um conjunto de m entradas z para G e m elementos x do nosso conjunto de dados. E atualizamos o D a partir do gradiente da função V . Depois, selecionamos um conjunto de m entradas z novamente e atualizamos o G a partir do gradiente novamente.

Este sistema apresenta como desvantagens a necessidade de D e G estarem sempre sincronizados e de não ter uma representação explícita da função de probabilidade de G como acontece em alguns sistemas anteriores que tentavam criar redes neurais geradoras. Porém, a sua simplicidade matemática, usando apenas técnicas comuns de deep learning, já usadas anteriormente em sistemas classificatórios, é uma de suas grandes vantagens.

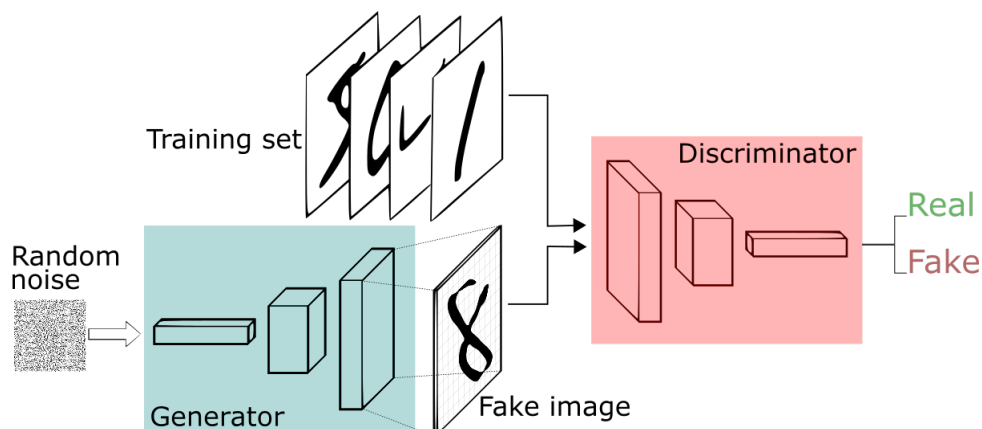


FIGURA 2.4 – Modelo representando o funcionamento do treinamento de uma GAN. A rede discriminadora recebe tanto imagens verdadeiras quanto imagens produzidas pela rede geradora a partir de um ruído aleatório, e é treinada para discriminar as imagens corretamente. (SILVA, 2017)

2.4 Deep Convolutional Generative Adversarial Nets

Com o avanço dos estudos das GANs, alguns problemas do modelo apresentado pelo grupo de Ian Goodfellow foram atacados por outros pesquisadores. Um dos principais problemas dos resultados do primeiro trabalho para quando aplicados em imagens foi gerar resultados borrados, chuviscados e um pouco incompreensíveis. A pesquisadora Emily Denton, junto com a Facebook AI, usou um método para tentar gerar imagens mais nítidas, aplicando um método chamado Laplacian Pyramid (DENTON *et al.*, 2015).

Em seu paper, observam-se resultados realmente mais nítidos, mas ainda há o problema de aparecerem um pouco tremidas qualitativamente. Dado esse problema, uma solução que surgiu posteriormente foram as DCGANs, ou deep convolutional GANs. Esta ideia aplica modelos de redes neurais convolucionais já usados em problemas classificatórios de imagens, e tenta aplicar um framework de GANs para gerar imagens.

Essa ideia foi publicada pelo artigo de Alec Radford e Luke Metz, Indico research, junto com mais um pesquisador do Facebook AI. Nele, eles propõem uma maneira de se arquitetar tanto G quando D num sistema de GANs usando redes neurais convolucionais. Suas recomendações incluem trocar as pooling layers de G e D por strided convolutional layers para D e fractional-strided convolutional layers para G . Usar ReLU como função de ativação dos neurônios de ambas as redes de G exceto a última e LeakyReLU para todas as camadas de D (RADFORD *et al.*, 2016).

3 Implementação

3.1 Objetivo e data set

3.1.1 Data set

O objetivo do trabalho é implementar uma GAN que crie rostos de personagens de desenhos animados japoneses, ou animes, a partir de um conjunto de dados. Para isso, primeiro foi obtido um conjunto de dados com rostos de vários personagens, as imagens foram escaladas para ficar no formato de 96 por 96 pixels, e por fim convertidas para preto e branco.

O banco de dados utilizado foi o (ANIME..., 2019). Esse conjunto possui mais de 11.000 rostos de personagens de animes para serem usados durante o treinamento. Depois do pre-processamento, em que redimensionamos e descolorimos a imagem, nossas imagens ficam prontas para serem usadas no treinamento da rede.

3.1.2 Código de pre-processamento do dataset

O código de obtenção e pre-processamento das imagens do data-set foi feito utilizando a linguagem Python. O código completo estará em anexo.

```
IMG_SIZE = 96
DATADIR = "./animeface-character-dataset/thumb"

training_data = []

def create_training_data():
    for root, dirs, files in os.walk(DATADIR):
        for img in files:
            try:
                img_array = cv2.imread(os.path.join(root, img), cv2.IMREAD_GRAYSCALE)
                new_array = cv2.resize(img_array, (IMG_SIZE, IMG_SIZE))
```

```
        training_data.append(new_array)
    except Exception as e:
        pass

create_training_data()

random.shuffle(training_data)

training_data = training_data[:1000]
```

Com esse código, pegamos um conjunto aleatório de 1000 imagens para usar como treinamento da rede. Na utilização no treinamento da rede, as entradas são normalizadas, de modo que o valor de cada pixel fique entre -1 e 1. Assim como usamos a tangente hiperbólica na saída da rede geradora para manter o mesmo padrão. A normalização da imagem é feita pelo seguinte código.

```
training_data = training_data / 255.0
batch = training_data[batch_size*i:batch_size*i + batch_size]
imagens_batch = batch.reshape((batch_size, 9216))
imagens_batch = imagens_batch * 2 - 1
```



FIGURA 3.1 – Exemplo de imagens usadas para o treinamento das redes.

3.1.3 Framework utilizado

Para a implementação da rede, foi escolhido o framework Tensor Flow, atualmente desenvolvido pela empresa Google e facilita bastante a implementação de diversas arquiteturas de redes neurais (ABADI *et al.*, 2015).

No código, vamos definir as redes do gerador e do discriminador. Além disso, parâmetros dos algoritmos de treinamento e técnicas utilizadas baseadas nos artigos de referência para esse trabalho. No decorrer da explicação de cada parte do código final, será discutido versões anteriores de escolha de algoritmos e arquitetura e os impactos das mudanças.

Todas as versões foram treinadas por até 500 épocas, onde cada época é um ciclo de treinamento de cada rede com um batch de 100 imagens reais e 100 imagens geradas pelo gerador.

3.2 Código implementado

3.2.1 Gerador

O código do gerador é uma rede neural que recebe como entrada um array de números aleatórios, também chamado de ruído. E sua saída é uma imagem de 96 por 96 pixels.

3.2.1.1 Versão 1

Numa primeira versão do código do projeto, foi implementada uma rede neural simples, sem camadas de convolução.

```
def gerador(ruido, reuse = None):
    with tf.variable_scope('gerador', reuse = reuse):
        camada_oculta1 = tf.nn.relu(tf.layers.dense(inputs = ruido, units
            =1280))
        camada_oculta2 = tf.nn.relu(tf.layers.dense(inputs=camada_oculta1,
            units=1280))
        camada_saida = tf.layers.dense(inputs = camada_oculta2, units =
            9216, activation = tf.nn.tanh)
        return camada_saida
```

Aqui usamos as funções de ativação ReLU nas camadas ocultas e tangente hiperbólica na camada de saída. O treinamento foi efetuado corretamente, porém esse modelo não apresentou resultados satisfatórios, produzindo imagens ruidosas depois de finalizado o treinamento.

3.2.1.2 Versão 2

A segunda versão já é uma DCGAN, com duas camadas de convolução transpostas para o upsampling do ruído para a imagem final. A primeira camada é uma camada densa

que recebe o ruído e produz como saída a entrada pra primeira camada de convolução. Depois de três camadas de convolução, a saída é gerada com a função de ativação tangente hiperbólica. Nas camadas de convolução ocultas, foi aplicada a função de ativação ReLU e Leaky ReLU, obtendo melhores resultados com a segunda e também batch normalization, que ajuda a manter os valores estáveis nos neurônios (CHINTALA *et al.*, 2016).

```
def gerador(ruido, reuse = None):
    with tf.variable_scope('gerador', reuse = reuse):
        camada_oculta1 = tf.layers.dense(inputs = ruido, units = 24*24*64)
        normalizacao_oculta1 = tf.layers.batch_normalization(inputs =
            camada_oculta1)
        saida_oculta1 = tf.nn.leaky_relu(normalizacao_oculta1)

        saida_oculta_2d_1 = tf.reshape(saida_oculta1, [-1, 24, 24, 64])

        deconvolucao1 = tf.layers.conv2d_transpose(inputs =
            saida_oculta_2d_1, filters = 32, strides = 1, kernel_size = [5,
            5], padding = 'same')

        normalizacao1 = tf.layers.batch_normalization(inputs =
            deconvolucao1)

        saida_convolucao1 = tf.nn.leaky_relu(normalizacao1)

        deconvolucao2 = tf.layers.conv2d_transpose(inputs =
            saida_convolucao1, filters = 16, strides = 2, kernel_size = [5,
            5], padding = 'same')

        normalizacao2 = tf.layers.batch_normalization(inputs =
            deconvolucao2)

        saida_convolucao2 = tf.nn.leaky_relu(normalizacao2)

        camada_saida = tf.layers.conv2d_transpose(inputs =
            saida_convolucao2, filters = 1, strides = 2, kernel_size = [5,
            5], padding = 'same')

        result = tf.nn.tanh(camada_saida)

    return tf.reshape(result, [-1, 9216])
```

As convoluções usam um kernel quadrado de lado 5, não mostrando muita variação de resultado com kernel menor.

3.2.2 Discriminador

O código do discriminador recebe como input uma imagem de 96 por 96 pixels e tem como output uma única saída que diz a probabilidade da imagem ter saído do data set, ou seja, de não ter sido gerada pelo gerador. Com isso, no treinamento, usamos que o valor esperado para uma entrada de uma imagem que veio do data set é 1, e o de uma imagem gerada pelo gerador a partir de um ruído é 0.

3.2.2.1 Versão 1

Como mostrada no caso anterior, temos uma versão inicial simples de uma rede neural densa de como o discriminador deveria funcionar.

```
def discriminador(X, reuse = None):  
    with tf.variable_scope('discriminador', reuse = reuse):  
        camada_oculta1 = tf.nn.relu(tf.layers.dense(inputs = X, units =  
            1280))  
        camada_oculta2 = tf.nn.relu(tf.layers.dense(inputs = camada_oculta1  
            , units = 1280))  
        logits = tf.layers.dense(camada_oculta2, units=1)  
        return logits
```

Como discutido na seção anterior, essa arquitetura não funciona muito bem, gerando imagens bem ruidosas.

3.2.2.2 Versão 2

Na versão final do código, usamos uma rede neural convolucional, como é comumente usada em problemas de classificação de imagens, já que na prática o discriminador é um classificador binário de imagens. Para isso, foi usada uma arquitetura de rede com duas camadas de convolução, uma quantidade parecida de camadas em relação ao gerador. Além disso, a fase de pooling ajuda no aumento da velocidade de treinamento da rede, mostrando resultados melhores com average pooling em relação a max pooling, como alertado por (CHINTALA *et al.*, 2016). O dropout também auxilia no treinamento, e usamos leaky ReLU de função de ativação nas camadas intermediárias e sigmoid na camada de saída.

```
def discriminador(X, reuse = None):  
    with tf.variable_scope('discriminador', reuse = reuse):  
        entrada = tf.reshape(X, [-1,96,96,1])
```

```
convolucao1 = tf.layers.conv2d(inputs = entrada, filters = 32,
                                kernel_size = [5,5], activation = tf.nn.leaky_relu, padding = '
                                same')

pooling1 = tf.layers.max_pooling2d(inputs = convolucao1, pool_size
                                    = [2,2], strides = 2)

convolucao2 = tf.layers.conv2d(inputs = pooling1, filters = 64,
                                kernel_size=[5,5], activation=tf.nn.leaky_relu, padding = 'same')

pooling2 = tf.layers.max_pooling2d(inputs=convolucao2, pool_size =
                                    [2,2], strides = 2)

flattening = tf.reshape(pooling2, [-1, 24*24*64])

densa = tf.layers.dense(inputs = flattening, units = 1024,
                        activation = tf.nn.leaky_relu)

dropout = tf.layers.dropout(inputs=densa, rate=0.2, training = True
                             )

logits = tf.layers.dense(inputs = dropout, units=1)
return logits
```

Essa arquitetura de rede neural funciona muito melhor que a versão 1, porém é mais custosa de treinar. Além disso, modificações nos parâmetros criam alterações qualitativas nos resultados, mostrando que é um modelo sensível e suscetível à experimentação, além do tempo demorado de espera para o treinamento da mesma.

3.2.3 Função de custo

Outra decisão importante na arquitetura foi escolher a função de custo e o algoritmo de treinamento para as redes neurais. Como vimos na equação 2.1, os agentes gerador e discriminador tentam maximizar e minimizar essa função. Na prática, usamos a sigmoid cross entropy como função de custo para cada um deles. Calculamos o erro produzido pelo discriminador a partir da entrada real e a partir da entrada gerada pelo gerador.

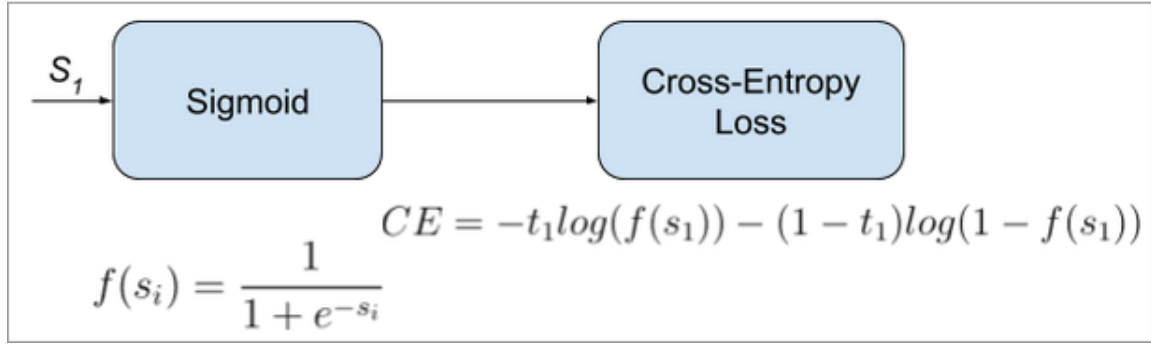


FIGURA 3.2 – Função de custo utilizada.

Para o erro do discriminador, o erro é a soma dos dois erros citados. E para o erro do gerador, é calculado o erro do discriminador com as imagens geradas, mas usando como resposta esperada o valor 1 para todas as saídas.

```

erro_discriminador_real = tf.reduce_mean(tf.nn.
    sigmoid_cross_entropy_with_logits(logits = logits_imagens_reais , labels
    = tf.ones_like(logits_imagens_reais)*(0.9)))
erro_discriminador_ruido = tf.reduce_mean(tf.nn.
    sigmoid_cross_entropy_with_logits(logits = logits_imagens_ruido , labels
    = tf.zeros_like(logits_imagens_ruido)))

erro_discriminador = erro_discriminador_real + erro_discriminador_ruido

erro_gerador = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
    logits = logits_imagens_ruido , labels = tf.ones_like(
    logits_imagens_ruido)))

```

3.2.4 Treinamento

Para o treinamento, o mais recomendado é o Adam Optimizer para ambos gerador e discriminador, apesar de ainda existir espaço para a taxa de aprendizado sugerida pro algoritmo (CHINTALA *et al.*, 2016). Esse algoritmo possui uma taxa de aprendizado adaptativa, que é um dos parâmetros que mais afetam a performance do modelo (GOODFELLOW *et al.*, 2016). Nas nossa implementação, testamos com a taxa padrão de 0.001 e com a de 0.0002 sugerida em (RADFORD *et al.*, 2016).

```

treinamento_discriminador = tf.train.AdamOptimizer(learning_rate=0.001).
    minimize(erro_discriminador , var_list = variaveis_discriminador)

```

```
treinamento_gerador = tf.train.AdamOptimizer(learning_rate=0.001).minimize(  
    erro_gerador, var_list=variaveis_gerador)
```

4 Resultados

A arquitetura final da GAN foi mostrada no capítulo anterior. Os resultados serão apresentados nesse capítulo.

Durante o treinamento, muitos modelos de rede foram testadas, variando tanto arquiteturas quanto função de ativações utilizadas em cada etapa. A arquitetura final apresentada anteriormente, que pode ser vista na íntegra em A.2 foi a que apresentou os melhores resultados qualitativos, apesar deles terem serem longe do ideal.

A rede do discriminador foi feita com poucas camadas devido ao fato de uma rede mais forte ter apresentado estagnação no treinamento, com o erro do gerador atingindo valores muito altos e sem conseguir produzir imagens satisfatórias.

O contrário ocorreu mais frequentemente. Com mudanças de alguns poucos detalhes, como a substituição da função de ativação do gerador de ReLU para Leaky ReLU fez com que o gerador ficasse muito a frente do discriminados, fazendo o erro do gerador estabilizar muito rapidamente em valores baixos sem gerar imagens muito diferentes.

Além disso, o alto tempo necessário para o treinamento faz com que as mudanças acabem não sendo muito frequentes, precisando de bastante experimentação e seguindo as instruções dos artigos de referência (CHINTALA *et al.*, 2016).

4.1 Imagens geradas

Houveram duas versões do algoritmo que geraram resultados satisfatórios. A versão final apresentada, e a mesma sem a tangente hiperbólica como função da ativação.

4.1.1 Versão final apresentada

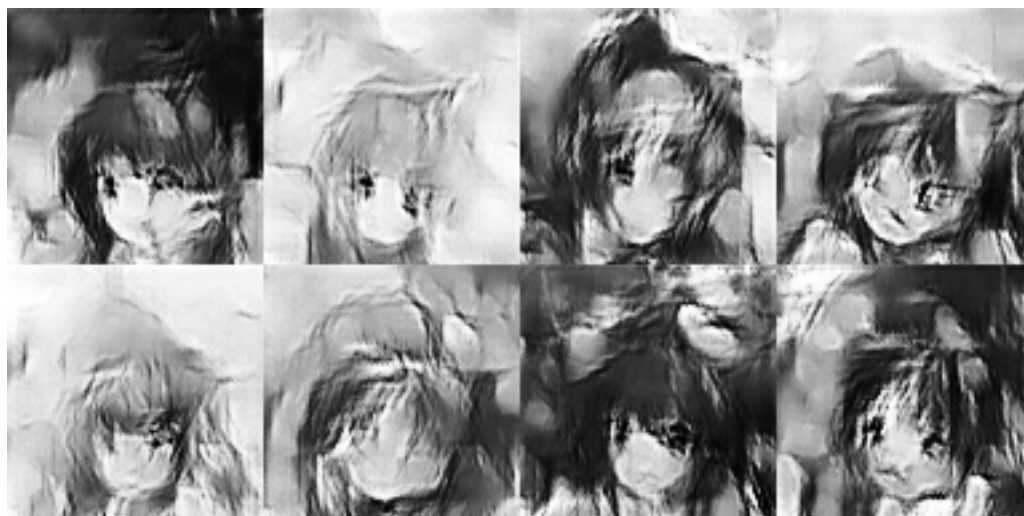


FIGURA 4.1 – Exemplos de imagens geradas pela rede neural final.

Nessa versão, as imagens ficaram mais genéricas e melhor definidas que as outras, mostrando o melhor resultado qualitativo. Abaixo mostraremos alguns exemplos notáveis que deram errado.

4.1.2 Removendo a tangente hiperbólica como função de ativação



FIGURA 4.2 – Exemplos de imagens geradas pela rede neural final sem a tangente hiperbólica.

Nessa versão, os resultados foram satisfatórios mas um pouco mais borrados que os anteriores visto que não se tinha a função de ativação final da tangente hiperbólica para normalizar as imagens.

4.1.3 Casos em que o discriminador é fraco

Alguns algoritmos testados fizeram com que o discriminador não conseguisse aprender muito bem de maneira rápida, fazendo com que o erro do gerador fosse muito baixo logo no começo, e o mesmo não produzisse imagens boas.

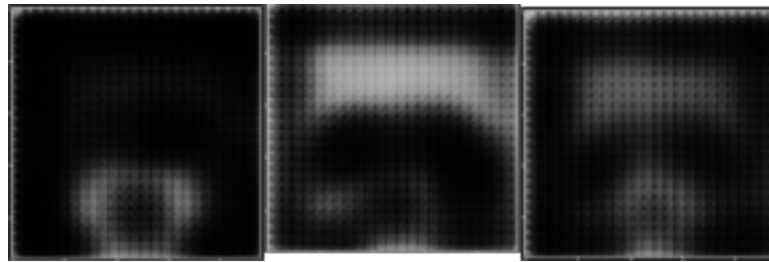


FIGURA 4.3 – Exemplos de imagens onde o discriminador não foi capaz de treinar o suficiente no número de épocas disponível devido à sua arquitetura.

Um exemplo de onde isso aconteceu, foi a utilização de average pooling no lugar de max pooling no discriminador, assim como a não utilização do leaky ReLU atrasou o treinamento do mesmo.

4.1.4 Casos em que o gerador é fraco

Em GANs onde o gerador é fraco, o mesmo não conseguia progredir, pois não tinha um direcionamento do discriminador do que estava fazendo de errado. Com isso, o mesmo gerava apenas ruído durante todo o treinamento. Isso ocorreu ao adicionar mais uma camada de convolução no discriminador nos testes executados.

4.1.5 Redes neurais sem convolução

A critério de comparação qualitativa, também destaco os resultados obtidos com redes neurais densas simples na resolução do mesmo problema.

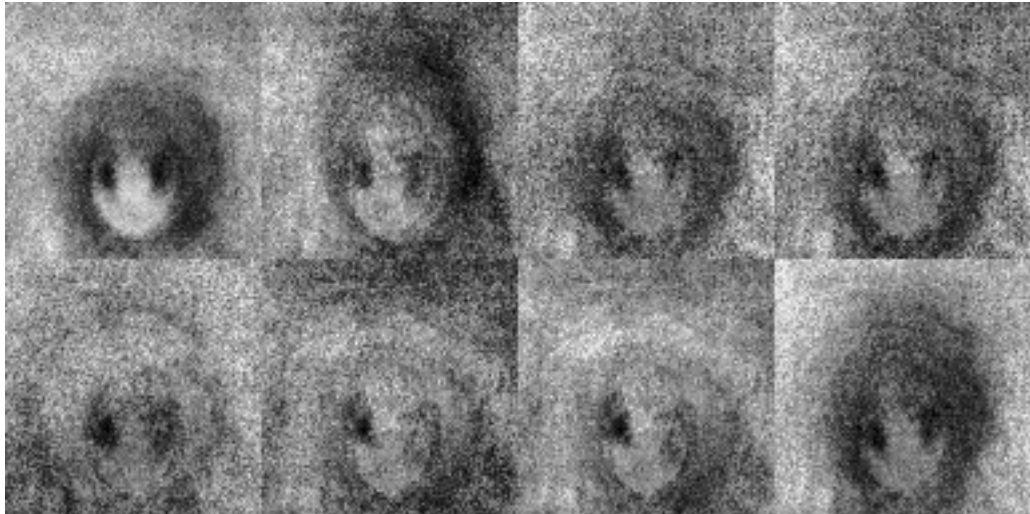


FIGURA 4.4 – Exemplos de imagens produzidas pelas redes neurais densas, citadas em 3.2.1.1 e 3.2.2.1.

Percebe-se que uma GAN feita com redes neurais densas ainda consegue se aproximar um pouco de um formato de um rosto. Porém, os resultados produzidos são borrados e ruidosos, não sendo qualitativamente bons. Além disso, podemos ver claramente um caso de overfitting devido a proximidade das imagens produzidas pela rede numa avaliação qualitativa.

5 Conclusões

Finalmente, após o extenso trabalho realizado, podemos concluir as diversas dificuldades de se treinar uma rede neural adversarial. Percebemos que pequenas variações na arquitetura da rede influenciam bastante no resultado e também no tempo de treinamento da rede.

As GANs são bem fáceis de serem implementadas, já que são constituídas de redes neurais comuns, sendo uma alternativa muito versátil em relação às já existentes. Mas o tempo de treinamento e o processamento necessário para treinar duas redes neurais convolucionais ao mesmo tempo pode ser muito custoso.

No capítulo 2, vimos todo o desenvolvimento teórico necessário para se chegar na formulação do framework das GANs, e vimos que uma das suas desvantagens se dá pelo fato de não ser uma aproximação exata da função que mapeia as entradas nas saídas. E no capítulo 3, vimos o desenvolvimento da implementação do código em que abordamos as recomendações de arquitetura dadas pelos artigos de referência e também vimos os efeitos práticos de cada escolha na nossa implementação.

Por fim, vimos nos resultados que as imagens qualitativamente mais satisfatórias geradas ainda não se aproximam tanto do ideal. Tal resultado pode ser devido à falta de tempo para treinar a rede, apesar de terem sido usadas pelo menos 500 épocas de treinamento em cada arquitetura, e também à problemas com a arquitetura no geral, visto que pequenas variações nos parâmetros provocaram grande influência no resultado final, gerando imagens não satisfatórias tanto por ter um discriminador muito fraco quanto por ter um gerador que não consegue progredir por ter um discriminador muito forte.

O desenvolvimento de GANs ainda está sendo muito estudado nos dias atuais, com algumas implementações muito boas em uso comercial, mas seu desenvolvimento ainda é refém de um elevado poder de processamento e suscetível a falhas devido ao design da arquitetura da rede implementada.

Referências

ABADI, M.; AGARWAL, A.; BARHAM, P.; BREVDO, E.; CHEN, Z.; CITRO, C.; CORRADO, G. S.; DAVIS, A.; DEAN, J.; DEVIN, M.; GHEMAWAT, S.; GOODFELLOW, I.; HARP, A.; IRVING, G.; ISARD, M.; JIA, Y.; JOZEFOWICZ, R.; KAISER, L.; KUDLUR, M.; LEVENBERG, J.; MANÉ, D.; MONGA, R.; MOORE, S.; MURRAY, D.; OLAH, C.; SCHUSTER, M.; SHLENS, J.; STEINER, B.; SUTSKEVER, I.; TALWAR, K.; TUCKER, P.; VANHOUCKE, V.; VASUDEVAN, V.; VIÉGAS, F.; VINYALS, O.; WARDEN, P.; WATTENBERG, M.; WICKE, M.; YU, Y.; ZHENG, X. **TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems**. 2015. Software available from tensorflow.org. Disponível em: <<https://www.tensorflow.org/>>.

ANIME faces character dataset. 2019. Disponível em:
<<http://www.nurs.or.jp/~nagadomi/animeface-character-dataset/>>.

CHINTALA, S.; DENTON, E.; ARJOVSKY, M.; MATHIEU, M. **How to Train a GAN? Tips and tricks to make GANs work**. 2016. Disponível em:
<<https://github.com/soumith/ganhacks>>.

DENTON, E. L.; CHINTALA, S.; SZLAM, a.; FERGUS, R. Deep generative image models using a laplacian pyramid of adversarial networks. In: CORTES, C.; LAWRENCE, N. D.; LEE, D. D.; SUGIYAMA, M.; GARNETT, R. (Ed.). **Advances in Neural Information Processing Systems 28**. Curran Associates, Inc., 2015. p. 1486–1494. Disponível em: <<http://papers.nips.cc/paper/5773-deep-generative-image-models-using-a-laplacian-pyramid-of-adversarial-networks.pdf>>.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. [S.l.]: MIT Press, 2016. <http://www.deeplearningbook.org>.

GOODFELLOW, I.; POUGET-ABADIE, J.; MIRZA, M.; XU, B.; WARDE-FARLEY, D.; OZAIR, S.; COURVILLE, A.; BENGIO, Y. Generative adversarial nets. In: GHAHRAMANI, Z.; WELLING, M.; CORTES, C.; LAWRENCE, N. D.; WEINBERGER, K. Q. (Ed.). **Advances in Neural Information Processing Systems 27**. Curran Associates, Inc., 2014. p. 2672–2680. Disponível em:
<<http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf>>.

RADFORD, A.; METZ, L.; CHINTALA, S. Unsupervised representation learning with deep convolutional generative adversarial networks. 2016. Disponível em:
<<https://arxiv.org/abs/1511.06434>>.

SILVA, T. **A Short Introduction to Generative Adversarial Networks**. 2017.
Disponível em: <<https://sthalles.github.io/intro-to-gans/>>.

Apêndice A - Código completo

A.1 Data set

```
import numpy as np
import matplotlib.pyplot as plt
import os
import cv2
import random
import pickle

IMG_SIZE = 96
DATADIR = "./animeface-character-dataset/thumb"

training_data = []

def create_training_data():
    # Itera em todos os arquivos do data set
    for root, dirs, files in os.walk(DATADIR):
        for img in files:
            try:
                img_array = cv2.imread(os.path.join(root, img), cv2.
                    IMREAD_GRAYSCALE)
                new_array = cv2.resize(img_array, (IMG_SIZE, IMG_SIZE))
                training_data.append(new_array)
            except Exception as e:
                pass

create_training_data()

# Reordena aleatoriamente as imagens
random.shuffle(training_data)

# Seleciona um conjunto a ser usado para o treinamento
training_data = training_data[:1000]
training_data = np.array(training_data).reshape(-1, IMG_SIZE, IMG_SIZE, 1)
```

```
# Salva o arquivo a ser lido no treinamento da rede
pickle_out = open("data_set1.pickle", "wb")
pickle.dump(training_data, pickle_out)
pickle_out.close()
```

A.2 Modelo

```
import tensorflow as tf
tf.reset_default_graph()
import pickle
import numpy as np

# Leitura do arquivo gerado pelo código acima
training_data = pickle.load(open("data_set1.pickle", "rb"))

# Normalizacao das imagens
training_data = training_data/255.0

ruido_ph = tf.placeholder(tf.float32, [None, 100])

def gerador(ruido, reuse = None):
    # Rede geradora de imagens. Recebe um ruído aleatório como entrada.
    # Tem uma camada oculta densa, e tres camadas de deconvolucao,
    # todas com funcao de ativacao relu, com excessao da ultima que usa
    # tangente hiperbolica
    # Batch normalization eh aplicada em todas as camadas para preservar a
    # dimensao dos valores
    with tf.variable_scope('gerador', reuse = reuse):
        camada_oculta1 = tf.layers.dense(inputs = ruido, units = 24*24*64)
        normalizacao_oculta1 = tf.layers.batch_normalization(inputs =
            camada_oculta1)
        saida_oculta1 = tf.nn.relu(normalizacao_oculta1)

        saida_oculta_2d_1 = tf.reshape(saida_oculta1, [-1, 24, 24, 64])

        deconvolucao1 = tf.layers.conv2d_transpose(inputs =
            saida_oculta_2d_1, filters = 32, strides = 1, kernel_size = [5,
            5], padding = 'same')

        normalizacao1 = tf.layers.batch_normalization(inputs =
            deconvolucao1)
```

```
saida_convolucao1 = tf.nn.relu(normalizacao1)

deconvolucao2 = tf.layers.conv2d_transpose(inputs =
    saida_convolucao1, filters = 16, strides = 2, kernel_size = [5,
    5], padding = 'same')

normalizacao2 = tf.layers.batch_normalization(inputs =
    deconvolucao2)

saida_convolucao2 = tf.nn.relu(normalizacao2)

camada_saida = tf.layers.conv2d_transpose(inputs =
    saida_convolucao2, filters = 1, strides = 2, kernel_size = [5,
    5], padding = 'same')

result = tf.nn.tanh(camada_saida)

return tf.reshape(result, [-1, 9216])

imagens_reais_ph = tf.placeholder(tf.float32, [None, 9216])

def discriminador(X, reuse = None):
    #Codigo da rede discriminadora. Duas camadas de convolucao e uma camada
    densa.
    #Max pooling e utilizada no downsampling e leaky relu de funcao de
    ativacao.
    with tf.variable_scope('discriminador', reuse = reuse):
        entrada = tf.reshape(X, [-1,96,96,1])

        convolucao1 = tf.layers.conv2d(inputs = entrada, filters = 32,
            kernel_size = [5,5], activation = tf.nn.leaky_relu,
            padding = 'same')

        pooling1 = tf.layers.max_pooling2d(inputs = convolucao1, pool_size
            = [2,2], strides = 2)

        convolucao2 = tf.layers.conv2d(inputs = pooling1, filters = 64,
            kernel_size=[5,5], activation=tf.nn.leaky_relu,
            padding = 'same')

        pooling2 = tf.layers.max_pooling2d(inputs=convolucao2, pool_size =
            [2,2], strides = 2)

        flattening = tf.reshape(pooling2, [-1, 24*24*64])

        densa = tf.layers.dense(inputs = flattening, units = 1024,
            activation = tf.nn.leaky_relu)
```

```
        dropout = tf.layers.dropout(inputs=densa, rate=0.2, training = True
        )

        logits = tf.layers.dense(inputs = dropout, units=1)
        return logits

logits_imagens_reais = discriminador(imagens_reais_ph)
logits_imagens_ruido = discriminador(gerador(ruido_ph), reuse = True)

# Erros do discriminador e gerador, usando sigmoid cross entropy.

erro_discriminador_real = tf.reduce_mean(tf.nn.
    sigmoid_cross_entropy_with_logits(logits = logits_imagens_reais, labels
    = tf.ones_like(logits_imagens_reais)*(0.9)))

erro_discriminador_ruido = tf.reduce_mean(tf.nn.
    sigmoid_cross_entropy_with_logits(logits = logits_imagens_ruido, labels
    = tf.zeros_like(logits_imagens_ruido)))

erro_discriminador = erro_discriminador_real + erro_discriminador_ruido

erro_gerador = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
    logits = logits_imagens_ruido, labels = tf.ones_like(
    logits_imagens_ruido)))

variaveis = tf.trainable_variables()

variaveis_discriminador = [v for v in variaveis if 'discriminador' in v.
    name]
variaveis_gerador = [v for v in variaveis if 'gerador' in v.name]

# Algoritmo de treinamento usado foi o Adam Optimizer, como recomendado
pela literatura de referencia.
treinamento_discriminador = tf.train.AdamOptimizer(learning_rate=0.001).
    minimize(erro_discriminador, var_list = variaveis_discriminador)
treinamento_gerador = tf.train.AdamOptimizer(learning_rate=0.001).minimize(
    erro_gerador, var_list=variaveis_gerador)

batch_size = 100
amostras_teste = []

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
```



```
saver = tf.train.Saver()

# Loop principal de treinamento
for epoca in range(5000):
    numero_batches = len(training_data) // batch_size
    for i in range(numero_batches):
        batch = training_data[batch_size*i:batch_size*i + batch_size]
        imagens_batch = batch.reshape((batch_size, 9216))
        imagens_batch = imagens_batch * 2 - 1

        batch_ruido = np.random.uniform(-1, 1, size=(batch_size, 100))

        _, custod = sess.run([treinamento_discriminador,
                              erro_discriminador], feed_dict = {imagens_reais_ph:
                              imagens_batch, ruido_ph: batch_ruido})

        _, custog = sess.run([treinamento_gerador, erro_gerador],
                              feed_dict = {ruido_ph: batch_ruido})

    print('epoca:_' + str(epoca + 1) + '_erro_D:_' + str(custod) + '_\n'
          'erro_G:_' + str(custog))

    if ((epoca+1)%500 == 0):
        # A cada 500 epocas, salva o modelo
        saver.save(sess, 'model_' + str(epoca))
```

FOLHA DE REGISTRO DO DOCUMENTO

1. CLASSIFICAÇÃO/TIPO TC	2. DATA 14 de novembro de 2019	3. DOCUMENTO N DCTA/ITA/TC-073/2019	4. N DE PÁGINAS 40
5. TÍTULO E SUBTÍTULO: Estudo e implementação de Redes Neurais Geradoras Adversariais			
6. AUTOR(ES): Felipe da Conceição Guimarães			
7. INSTITUIÇÃO(ÕES)/ÓRGÃO(S) INTERNO(S)/DIVISÃO(ÕES): Instituto Tecnológico de Aeronáutica – ITA			
8. PALAVRAS-CHAVE SUGERIDAS PELO AUTOR: Aprendizado de máquina, Redes neurais geradoras.			
9. PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO: Aprendizagem (inteligência artificial); Redes neurais; Classificação de imagens; Estruturas (processamento de dados); Computação.			
10. APRESENTAÇÃO: (X) Nacional () Internacional ITA, São José dos Campos. Curso de Graduação em Engenharia de Computação. Orientador: Prof. Dr. Carlos Henrique Quartucci Forster. Publicado em 2019.			
11. RESUMO: Este trabalho tem como objetivo realizar um estudo sobre um recente framework de redes neurais usado para geração de imagens, as <i>Generative Adversarial Networks</i> , ou GANs. Esse ramo de aplicação de redes neurais é bem diferente da aplicação comum onde se classifica entidades a partir de uma entrada. Primeiramente discutiremos alguns fundamentos de redes neurais clássicas e seu funcionamento em problemas de classificação. Depois uma discussão teórica sobre as redes neurais convolucionais, frequentemente utilizadas em problemas de classificação de imagens. E por fim, uma discussão sobre as GANs. Em seguida, faremos uma implementação de uma GAN para a geração de imagens a partir de um banco de dados e estudaremos o efeito de diferentes arquiteturas e parâmetros em seus resultados.			
12. GRAU DE SIGILO: (X) OSTENSIVO () RESERVADO () SECRETO			