UNIVERSIDADE DE SÃO PAULO
Instituto de Matemática e Estatística
Bacharelado em Matemática Aplicada e Computacional

Diego Barbosa Marques

Rede Adversária Generativa Convolucional Profunda Aplicada a Exames de Raio-x do Tórax

Orientadora: Profª. Drª. Florencia Leonardi

São Paulo
2021

Resumo

Este trabalho visa apresentar um estudo de caso no uso de Redes Adversárias Generativas aplicadas a imagens médicas, mais especificamente em exames de raio-x do tórax. O fundamento para tal tema é a crescente procura por modelos capazes de predizerem a presença de uma doença em exames médicos. Uma das dificuldades enfrentadas pela área é a insuficiência de dados para o treinamento de modelos assertivos, em decorrência da alta sensibilidade dos dados de pacientes médicos. Este trabalho, então, apresenta possíveis soluções para o problema utilizando uma Rede Adversária Generativa Convolucional Profunda (DCGAN), capaz de gerar imagens fidedignas de exames de raio-x do tórax, servindo como complementação dos dados (Data Augmentation). O estudo é apresentado quebrando em partes cada processo de uma DCGAN, especificando e detalhando os fundamentos de Redes Neurais Artificiais (ANN), Redes Neurais Convolucionais(CNN) e Redes Adversárias Generativas (GAN).

Palavras-chave: Aprendizado profundo, Perceptron, Rede Neural Artificial, Convolução, Modelos Geradores, GAN, DCGAN.

Abstract

This work aims to present a case study on the use of Generative Adversarial Networks applied to medical images, more specifically in chest x-ray exams. The foundation for this theme is a growing demand for models to predict the presence of a disease in medical examinations. One of the difficulties faced by the area is the lack of data for training assertive models, due to the high sensitivity of data from medical patients. The work, then, presents possible solutions to the problem using a Deep Convolutional Generative Adversarial Network (DCGAN), capable of generating reliable images of chest x-ray exams, serving as Data Augmentation. The study is breaking into parts each process of a DCGAN, specifying and detailing the fundamentals of Artificial Neural Networks (ANN), Convolutional Neural Networks (CNN) and Generative Adversarial Networks (GAN).

Keywords: Deep Learning, Perceptron, Artificial Neural Network, Convolution, Generative Models, GAN, DCGAN.

Contents

1 Introdução **7**
1.1 Aplicação da Inteligência Artificial na Medicina 7
1.2 Objetivo 7

2 Rede Neural Artificial **8**
2.1 Definição 8
2.2 Perceptron 9
2.3 Modelo Linear 9
2.4 Multilayer Perceptron 10
2.5 Função de Ativação 11
2.5.1 Função Sigmoid 11
2.5.2 Função Tanh 11
2.5.3 Função ReLU 12
2.5.4 Função Leaky ReLU 13
2.6 Etapa de Feedforward 13
2.7 Função de Custo 15
2.7.1 Entropia Cruzada Binária(BCE) 15
2.8 Gradiente Descendente 16
2.8.1 Gradiente Descendente Estocástico 17
2.8.2 RMSProp 18
2.8.3 Adam 19
2.9 Backpropagation 21
2.10 Inicialização Aleatória 23
2.11 Regularização 24
2.11.1 Dropout 24
2.12 Normalização 25
2.12.1 Batch Normalization 25

3 Rede Neural Convolucional **27**
3.1 Convolução 28
3.1.1 Filtro 29
3.1.2 Padding 30
3.1.3 Stride 31
3.1.4 Pooling 31
3.2 Convolução sobre Volume 32
3.3 Convolução Transposta 33
3.4 LeNet-5 33

4 Redes Adversárias Generativas **35**
4.1 Redes Adversárias 35
4.2 Propriedades do Modelo 37

4.3 Redes Adversárias Generativas Convolucionais Profundas 38
4.3.1 Características da Rede 38

5 Métricas de Avaliação 40
5.1 Extração de Características 41
5.2 Inception-v3 41
5.3 Fréchet Inception Score (FID) 42

6 Experimento 43
6.1 Arquitetura DCGAN 43
6.1.1 Discriminador 43
6.1.2 Gerador 44
6.2 Etapas de treinamento 44
6.2.1 Treinando o Discriminador - Etapas (1) e (2) 44
6.2.2 Treinando o Gerador - Etapa (3) 47
6.3 Base de Dados 50
6.4 Resultados 51
6.4.1 Distância FID 52

7 Conclusão 55

1 Introdução

1.1 Aplicação da Inteligência Artificial na Medicina

A Inteligência Artificial, com o avanço das técnicas de aprendizado de máquina e aprendizagem profunda, tem proporcionado grandes benefícios para a área da medicina e saúde. Tarefas complexas estão sendo simplificadas por modelos que automatizam processos, desde prevenção até detecção de doenças. Um dos avanços mais significativos está presente nos modelos de classificação de imagens; No contexto médico, hoje é possível criar modelos que detectam um câncer em uma tomografia ou uma simples pneumonia num exame de raio-x. As aplicações são extensas, e a cada dia novas técnicas e novos modelos são desenvolvidos com o propósito de auxiliar e tornar a medicina mais eficaz.

Um dos grandes desafios, porém, é a complexidade em estruturar bases de dados, em particular pela alta sensibilidade que os dados médicos necessitam. Este entrave tem chamado atenção para os modelos geradores, aqueles que não tem o propósito de classificar uma amostra, mas gerar elas (ou algo próximo delas). Isso permite que dados de difícil acesso, como nos trabalhos com imagens de lesões do fígado[9] e imagens vasculares da retinas[23], possam ser gerados com o propósito de servir como entrada para um possível modelo de classificação. As Redes Adversárias Generativas, apresentadas por Ian Goodfellow[11] em 2015, são as mais promissora entre as redes generativas, apesar do curto tempo de existência, tendo uma vasta gama de aplicações na área médica.

1.2 Objetivo

Este trabalho visa demonstrar uma possível aplicação das Redes Adversarias Generativas(GAN) na área médica, ao gerar imagens de raio-x do tórax, afim de alimentar uma base de dados para classificação. Nos capítulos 2 e 3 serão apresentados os conceitos, respectivamente, de Rede Neural Artificial (ANN) e Rede Neural Convolucional (CNN). No capítulo 4 serão apresentadas as definições de uma GAN e de uma DCGAN, este último servindo como gerador do experimento final. O capítulo 5 mostra as métricas de avaliação e os desafios e dificuldades dos modelos GAN. Ao final serão apresentados os resultados dos experimentos, assim como possíveis sugestões para aprimorar o trabalho. O trabalho em questão não tem como objetivo criar um gerador a nível comercial, devido ao alto custo computacional e financeiro de se criar imagens de alta resolução, mas sim apresentar e exemplificar as técnicas utilizadas na área.

2 Rede Neural Artificial

2.1 Definição

A primeira ideia de uma Rede Neural foi introduzida, em 1943, pelo neuropsicólogo Warren McChulloch e o matemático Walter Pitts[24], no trabalho "A Logical Calculus of Ideas Immanent in Nervous Activity". Estudando os neurônios dos animais, os dois apresentaram um simplificado modelo computacional de como estes funcionam. Nos anos seguintes, que sucederam trabalhos como o do Perceptron na década de 50 e 60 por Rosenblatt[29], houve uma grande esperança de que essas arquiteturas de Redes Neurais pudessem ter muitas aplicações na área da computação e da inteligencia artificial. Porém, sua alta complexidade matemática, o custo computacional excessivo e a necessidade de um alto fluxo de dados para a época fizeram os trabalhos na área cessarem por algum período. Durante esse tempo, outros modelos como o SVM(Support Vector Machine) se popularizaram e tomaram conta da atenção dos estatísticos e matemáticos. O que trouxe a atenção de volta às Redes Neurais foram o desenvolvimento de computadores mais potentes e as propagandas futurísticas que tais arquiteturas despertam no imaginário. Além disso, questões técnicas que mostravam ser um empecilho para a área foram se revelando cada vez menos nebulosas e mais fáceis de serem resolvidas, tornando as redes neurais mais eficientes que outros modelos populares.

Podemos pensar numa Rede Neural Artificial usando como exemplo um neurônio real, do nosso próprio cérebro. Nele há um núcleo, onde a informação é armazenada; um axion, por onde a informação flui; e um dendrito, por onde o neurônio recebe a informação de outros neurônios. De forma simples, cada neurônio transmite suas informações para um ou mais neurônios em que está conectado, fluindo as informações para todo o cérebro. Partindo da estrutura simples de um neurônio, podemos conectar diversos outros entre si, com o objetivo de computar as mais complexas tarefas que um cérebro pode executar.

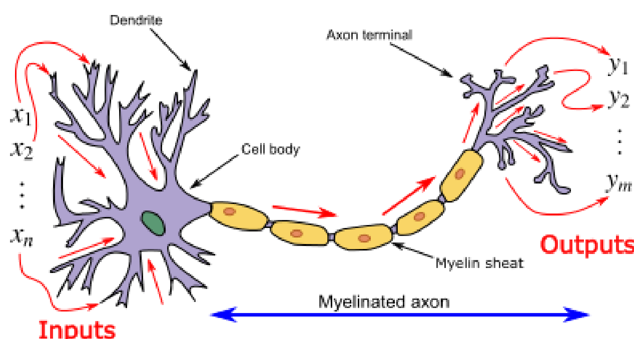

Figura 1: Neurônio de um cérebro.

Fonte: significados.com.br/neuronios/

Inicialmente, McCulloch e Pitts[24] propuseram um modelo simples de um neurônio biológico, que computa respostas binárias, partindo de decisões simples com poucos neurônios conectados, dada uma regra lógica.

2.2 Perceptron

O perceptron, desenvolvida na Década de 50 e 60 por Frank Rosenblatt[29], é uma arquitetura de Rede Neural que mais se aproxima das redes atuais, mais robusta que a anterior, dando o impulso necessário para o desenvolvimento da área. Nela um conjunto de entradas binárias $\{x_1, x_2, ..., x_n\}$ produz uma única saída binária.

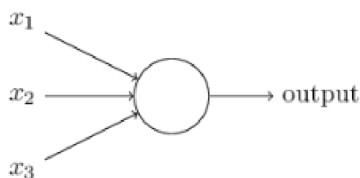

Figura 2: Perceptron.
Fonte: Nielsen [26]

Como regra, Rosemblatt[29] propôs o uso de pesos(ou parâmetros) reais $\{w_1, w_2, ..., w_n\}$. Partindo delas é calculada, então, a soma do produto entre as entradas e os pesos(ou seu produto vetorial), para assim estabelecer sua saída, baseado num valor de corte(threshold):

$$\text{saída} \ = \begin{cases} 0 & \text{se } \sum_j w_j x_j \leq \ \text{threshold} \\ 1 & \text{se } \sum_j w_j x_j > \ \text{threshold} \end{cases}$$

Podemos estender a definição do Perceptron e colocar o threshold do outro lado da equação, transformando-o no que é conhecido como viés(ou bias):

$$\text{saída} \ = \begin{cases} 0 & \text{se } \sum_j w_j x_j + b \leq 0 \\ 1 & \text{se } \sum_j w_j x_j + b > 0 \end{cases}$$

Apesar de tomar decisões complexas, o perceptron é limitado, dado que nossa saída será sempre binária, o que não abrange a maioria dos problemas de aprendizado profundo. Para isso versões mais modernas de uma Rede Neural Artificial permitem o uso de funções não-lineares para moldar esses problemas, seja ele um classificador de imagem ou uma simples regressão logística, computando valores reais.

2.3 Modelo Linear

O modelo mais básico de uma Rede Neural é aquele que tem como base um modelo linear. Sua estrutura já foi apresentada no perceptron, porém aqui não há uma condição, a entrada será o próprio produto vetorial entre X e W, somado ao viés b.

$$z = \sum w_i x_i + b$$

Assim como no perceptron, w_i são os pesos associados a cada variável explicativa x_i e b é

o viés do modelo. Podemos chamar cada x_i como uma característica (feature) e os pesos w_i e viés b como os parâmetros do modelo. Nosso $\hat{y}$ será z. O objetivo principal desse modelo é minimizar os erros, afim de aproximar z de nosso y real, utilizando uma função de Custo:

$$C(w,b) = \frac{1}{N}\sum_{i=1}^{N}(z_i - y_i)^2$$

O grande problema desse tipo de modelo é a não-linearidade dos problemas enfrentados numa tarefa de classificação. Em um problema de classificação binário de imagens, sabemos que a distribuição não pode ser linear, logo não teremos estimadores apropriados usando uma forma linear.

2.4 Multilayer Perceptron

Em um Multilayer Perceptron estendemos a definição de um Perceptron. Ela é composta por 3 camadas básicas: **Camada de Entrada** $x \in \mathbb{R}^n$; camadas intermediárias chamadas de **Camadas Ocultas**; e a **Camada de Saída** $\hat{y}$. Cada elemento de uma camada l se conecta inteiramente com os elementos da camada $l+1$, até computar a **Camada de Saída** L. No caso de um classificador binário, a **Camada de Saída** será $\hat{y} \in [0,1]$. O objetivo deste tipo de rede é computar $\hat{y} = f(x;\theta)$ de forma a "aprender" valores do parâmetro θ que melhor aproximam y da função f. É também dado o nome de Feedforward pois as informações fluem entre as camadas da rede do começo ao fim. Existem outros tipos de redes que não as Feedforward, como as utilizadas nas Redes Neurais Recorrentes(RNN), porém não está no escopo deste trabalho.

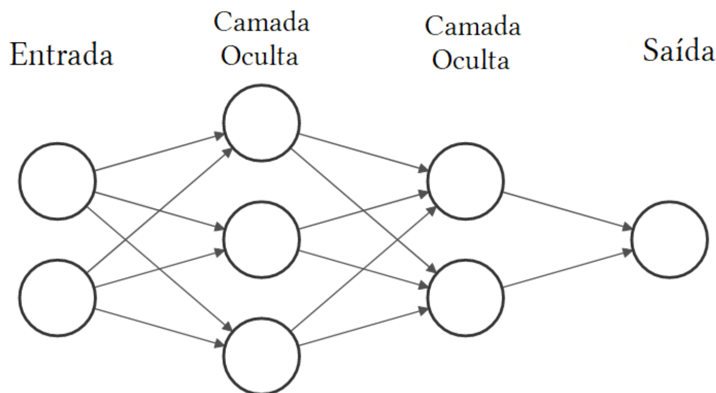

Figura 3: Estrutura de uma Rede Neural.

Definida a **Entrada** x e a **Saída** z, a **Camada Oculta** será definida como um modelo linear z, porém com a aplicação de uma **Função de Ativação**.

2.5 Função de Ativação

Como comentado anteriormente, um problema de classificação não apresenta uma forma linear, portanto é preciso alguma função capaz de transformar o modelo em não-linear. As Rede Neurais fazem uso das chamadas Funções de Ativação:

$$a = g(z) = g(\sum w_i x_i + b)$$

Há inúmeras funções de ativação que podem ser usadas em uma Rede Neural, variando se são usadas em uma **Camada Oculta** ou **Camada de Saída**.

2.5.1 Função Sigmoid

A função Sigmoid, usada em uma camada de saída, nos dá um valor que pode ser interpretado como uma probabilidade $\hat{y} = P(y = 1 \mid x)$, apesar de não ser uma função de densidade, dado $x \in \mathbb{R}^n$ e $y \in \{0, 1\}$. A função é definida como:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Quando z é consideravelmente grande, $\sigma(z) \to 1$, e quando z é consideravelmente pequeno, $\sigma(z) \to 0$.

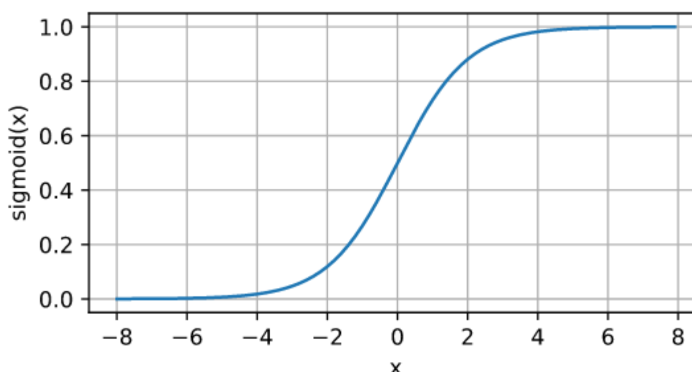

Figura 4: Função Sigmoid.
Fonte: Wikipedia.

2.5.2 Função Tanh

A função Tanh, assim como a Sigmoid, também é usada em uma camada de saída, porém y está definido entre -1 e 1:

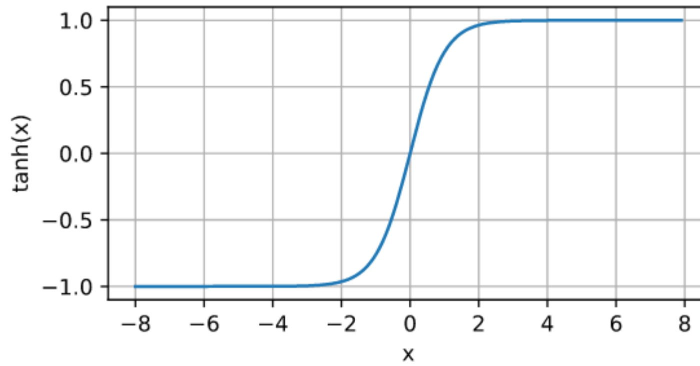

Figura 5: Função Tanh.
Fonte: Wikipedia.

Em uma camada na Rede Neural:

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

2.5.3 Função ReLU

A função ReLU(Rectified Linear Unit), comumente usada em camadas intermediárias, é definida como:

$$ReLU(z) = max(0, z)$$

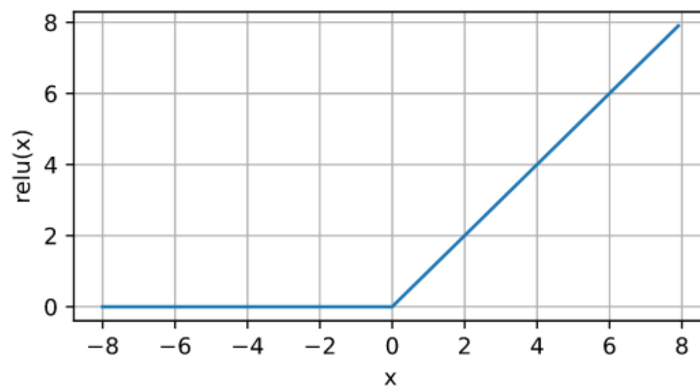

Figura 6: Função ReLU.
Fonte: Wikipedia.

A intuição por trás da função é "desligar" elementos em uma camada que parecem não ter importância para a rede, dado que $z < 0$. Apesar de não apresentar uma derivada em 0, ainda assim é possível utilizar o ReLU como uma função de ativação, pois dificilmente o gradiente da rede será 0.

2.5.4 Função Leaky ReLU

Leaky ReLU é uma variação das funções Rectifiers, definida como:

$$LeakyReLU(z) = max(\alpha z, z), \alpha \in \mathbb{R}$$

O hiperparâmetro α deve ser otimizado durante o treino da Rede Neural, porém é recomendado um valor entre 10^{-2} e 0, sendo este último caso o próprio ReLU.

2.6 Etapa de Feedforward

A etapa de Feedforward consiste em aplicar as funções de ativação em cada etapa linear entre uma camada e outra. Para isso haverá uma matriz de pesos $W^{[l]}$ e um vetor de viés $b^{[l]}$ entre cada camada $l = 1, ..., L$. Analogamente a um neurônio real, podemos comparar cada elemento de uma camada com um neurônio do cérebro:

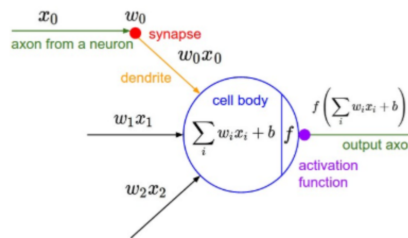

Figura 7: Detalhamento de um elemento da Rede Neural.
Fonte: Roffo[28].

As informações que chegam ao dendrito, as entradas X, são multiplicados aos respectivos pesos W e somados ao viés b. A área em azul da imagem será o núcleo do neurônio, onde todas as operações são feitas e as funções de ativação $g(z)$ são aplicadas. A seguir, é apresentada uma arquitetura geral de uma Rede Neural Artificial, tomando como base e generalizando um neurônio(ou elemento) como o da figura 7.

É preciso fazer observações quanto aos subscritos:

$[l]$: l_{th} camada da rede

(m) : m_{th} exemplo da base de treino

n_x : número de features da base de treino

n_l : número de elementos da l_{th} camada

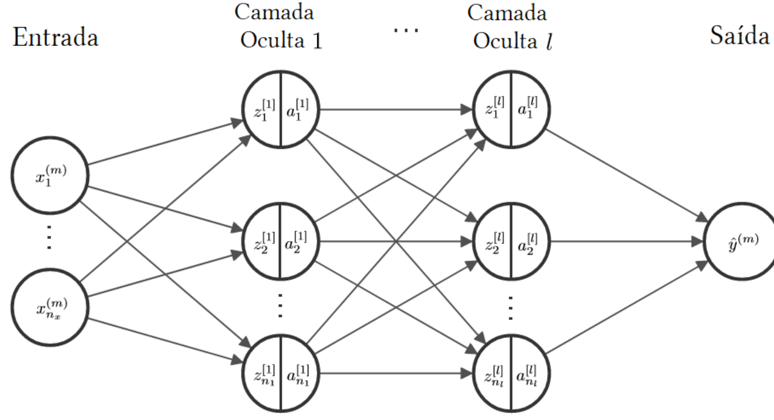

Figura 8: Representação dos elementos de uma Rede Neural.

Na primeira etapa do processo X será a entrada da primeira camada. É feito, então, o produto entre $W^{[1]}$ e X, somado ao viés $b^{[1]}$, resultando em $Z^{[1]}$. Este servirá como entrada para a primeira função de ativação $g(z)$, resultando em $A^{[1]}$(Mais a frente, neste trabalho, será mostrado que é possível e recomendado, logo a cada etapa desta, adicionar regularizações como Dropout, porém continuaremos a explicação do processo de Feedforward sem essas ferramentas). Em seguida, $A^{[1]}$ será nossa entrada para a segundo camada oculta, efetuando o produto entre $W^{[2]}$ e $A^{[1]}$, somado ao viés $b^{[2]}$(perceba que aqui $A^{[1]}$ fará o mesmo papel que X fez na etapa passada). O $Z^{[2]}$ resultante da operação anterior será aplicado em outra função de ativação(podendo ser outra função diferente), resultando em $A^{[2]}$. O processo continua semelhante até a última camada oculta l, obtendo $Z^{[l]}$ e logo em seguida $A^{[l]}$. Por final, $A^{[l]}$ será o valor estimado de $\hat{Y}$.

O processo descrito acima segue a seguinte ordem:

$$\begin{cases} Z^{[1]} = W^{[1]}X + b^{[1]} \\ A^{[1]} = g(Z^{[1]}) \end{cases} \tag{1}$$

$$\begin{cases} Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]} \\ A^{[2]} = g(Z^{[2]}) \end{cases} \tag{2}$$

$$\vdots$$

$$\begin{cases} Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]} \\ A^{[l]} = g(Z^{[l]}) \end{cases} \tag{3}$$

Finalmente calculando:

$$\hat{Y} = A^{[l]}$$

Em que:

$$
\begin{aligned}
&W^{[l]} \in \mathbb{R}^{n_l \times n_{l-1}} \\
&b^{[l]} \in \mathbb{R}^{n_l} \\
&X \in \mathbb{R}^{n_x \times m} \\
&\hat{Y} \in \mathbb{R}^{m}
\end{aligned}
$$

No caso de um classificar binário, $\hat{Y} \in [0,1]^m$.

Fazer o produto entre os inputs e os pesos W, somado-se aos viés b, resulta que todos os elementos de cada camada l estão inteiramente conectados com o camada seguinte $l+1$, por isso a denominação comumente utilizada de camadas inteiramente conectadas(fully-connected layers). Os pesos $W^{[l]}$ terão dimensão $(n_l \times n_{l-1})$, e os vieses $b^{[l]}$ terão dimensão n_l. Camadas com muitos elementos podem impactar significantemente no custo computacional e não necessariamente aumentam a eficiência das redes. É preciso, empiricamente, testar diferentes tamanhos de camadas(quantidade de elementos), assim como a profundidade da rede(quantidade de camadas ocultadas). Computacionalmente é recomendado utilizar camadas de tamanho múltiplos de 2, como $64, 128, \ldots$, afim de aproveitar com mais eficiência a memória a ser reservada pelo computador durante o treinamento. É bastante recomendado usar ReLU como função de ativação das camadas intermediárias.

A etapa seguinte é avaliar o quão próximo $\hat{Y}$ está do valor real. No caso de um $Y \in \mathbb{R}$, é preciso calcular o erro quadrático entre a estimativa e o valor real. Porém, este trabalho tem apenas interesse no cenário em que Y é binário. Assim, nossa última camada terá como função de ativação uma função Sigmoid, afim de computar um valor $\hat{Y} \in [0,1]$. Como explicado anteriormente, o valor de probabilidade $\hat{Y}$ estimado nos indicará se o verdadeiro valor Y está próximo de 0 ou de 1. A função de ativação Tanh também será utilizada em uma camada final em determinada etapa da rede utilizada neste trabalho, porém nosso classificador terá sempre uma função Sigmoid como saída.

2.7 Função de Custo

A função de Custo tem como objetivo dizer quão próximo $\hat{Y}$, ou $f(x;\theta)$, está de Y. Diferentemente de uma Regressão Linear, em que $Y \in \mathbb{R}$, em um problema de classificação binário teremos $Y \in \{0,1\}$, logo a função de Custo deverá ser diferente.

2.7.1 Entropia Cruzada Binária(BCE)

A função de Perda de um classificador binário pode ser definida como:

$$L(\hat{y}^{(i)}, y^{(i)}) = -[y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})]$$

Logo, a função de Custo Entropia Cruzada Binária(BCE) é definida como:

$$C(W, b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m} \sum_{i=1}^{m} [y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})]$$

Há dois casos, dado um determinado valor de $y^{(i)}$:

- Quando $y^{(i)} = 0$,

  $$L(\hat{y}^{(i)}, y^{(i)}) = -\log(1 - \hat{y}^{(i)})$$

  Caso $\hat{y}^{(i)} \to 0$, ou seja, se aproxime do verdadeiro valor de $y^{(i)} = 0$, a função de Perda será mínima em 0. Caso $\hat{y}^{(i)}$ esteja mais distante de 0, $L(\hat{y}^{(i)}, y^{(i)})$ irá explodir.

- Quando $y^{(i)} = 1$:

  $$L(\hat{y}^{(i)}, y^{(i)}) = -\log(\hat{y}^{(i)})$$

  Caso $\hat{y}^{(i)} \to 1$, ou seja, se aproxima do verdadeiro valor de $y^{(i)} = 1$, a função de Perda será mínima em 0. Caso $\hat{y}^{(i)}$ esteja mais distante de 1, $L(\hat{y}^{(i)}, y^{(i)})$ irá explodir.

A função de entropia(BCE) nos permite encontrar os parâmetros W e b que melhor aproximam $\hat{Y}$ de Y. Para isso, é preciso utilizar algoritmos de otimização capazes de minimizar a função $C(W, b)$.

2.8 Gradiente Descendente

Tendo definida a função de Custo, é preciso minimiza-lá com o objetivo de encontrar os melhores parâmetros W e b. O Gradiente Descendente, algoritmo base para otimizar a função de Custo de uma Rede Neural, tem a seguinte definição:

$$v \to v' = v - \eta \nabla C(v)$$

Em que η é a taxa de aprendizagem(learning rate) e $\nabla C(v)$ é o gradiente de $C(v)$ em relação a v. Subtraindo o gradiente de v, o algoritmo visa atualizar v' de tal forma a aproximá-lo de seu mínimo global. A intuição é "andar" na direção contrária ao gradiente. Em uma Rede Neural, o otimizador é definido como:

$$W^{[l]} \to W^{[l]\prime} = W^{[l]} - \eta \frac{\partial C}{\partial W^{[l]}}$$
$$b^{[l]} \to b^{[l]\prime} = b^{[l]} - \eta \frac{\partial C}{\partial b^{[l]}}$$

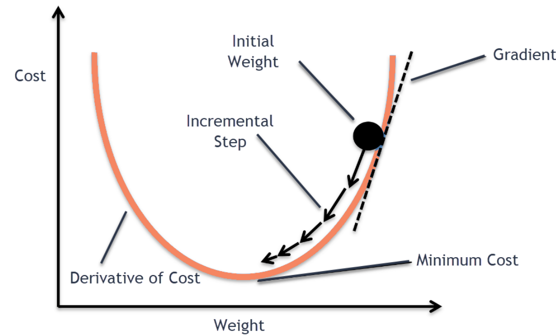

Figura 9: Gradiente Descendente.
Fonte: blog.clairvoyantsoft.com.

Dado que o método de Gradiente Descendente utiliza um batch(subdivisão da base de dados) de treinamento inteiro, seu custo computacional será muito alto e muitas vezes ineficiente. Para solucionar este problema, existem extensões do otimizador, como o Gradiente Descendente Estocástico, RMSprop e Adam.

2.8.1 Gradiente Descendente Estocástico

O Gradiente Descendente Estocástico(SGD) tem grande importância na aceleração da otimização de parâmetros em uma função objetivo para encontrar um máximo ou mínimo, caso a função seja diferenciável. Dado que muitas destas funções são estocásticas, principalmente por conterem a soma de subfunções de diferentes observações de uma base de dados, o Gradiente Descendente é mais eficiente tomando observações individuais. No caso de um Gradiente Descendente simples, tomamos a amostra inteira para otimizar a função; com o SGD, tomamos apenas uma observação por vez, aleatoriamente, assumindo que a base de treino apresenta uma distribuição similar e é redundante. Essa diferença já é um grande avanço, já que o custo computacional do Gradiente Descendente aumenta consideravelmente quando se aumenta o tamanho da base de dados. Sua história está diretamente ligada aos grandes sucessos nas descobertas e criações na área de aprendizagem profunda, já que uma das grandes dificuldades na utilização das Redes Neurais é justamente o custo computacional.

Sua grande vantagem é a velocidade com que o otimizador converge para o mínimo global. Sua desvantagem é a grande quantidade de ruído e a possibilidade de divergir, dado que é um Processo Estocástico.

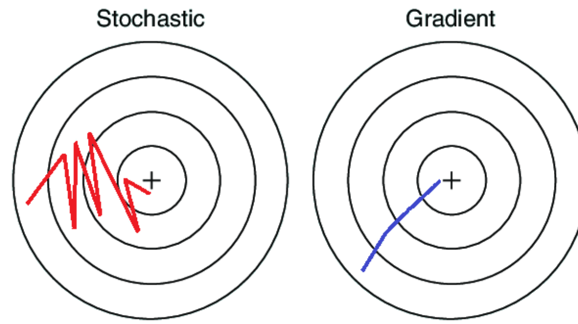

Figura 10: Gradiente Descendente Estocástico.
Fonte: Carpenter et al.[4].

2.8.2 RMSProp

Muitas técnicas foram testadas ou desenvolvidas para otimizar o algoritmo de Gradiente Descendente, que como mostrado anteriormente, apresenta muitos problemas, considerando que é preciso trabalhar com um alto volume de dados. Uma das ferramentas mais importantes utilizadas e fundamental na área de aprendizagem profunda é o uso de momentos. O momento tem como objetivo resolver o problema de uma taxa de aprendizagem constante e evitar que o algoritmo demore para convergir ou divirja, ao mesmo tempo estabilizando o treino e acelerando processo. A intuição da sua aplicação no algoritmo de Gradiente Descendente é maximizar os passos na direção correta ao ponto ótimo e minimizar os passos na direção errada. É possível pensar no processo de Médias Móveis como uma suavização do caminho a ser percorrido pelo Gradiente Descendente.

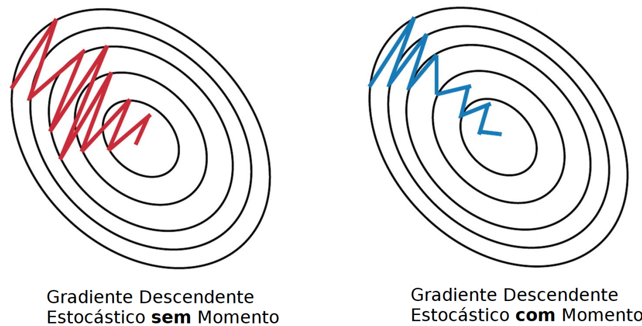

Figura 11: Comparação entre SGD sem e com momento.
Fonte: eloquentarduino.github.io

O RMSprop, proposto por Geoff Hinton[33], resolve o problema da taxa de aprendizagem η constante, aplicando algo análogo a ideia de momentos. Ele foi desenvolvido empiricamente em uma das aulas que Hinton estava preparando para seu curso e logo depois foi amplamente adotado na área de aprendizado profundo. Primeiramente definimos a etapa do cálculo das Médias Móveis, porém com a diferença de calcular o quadrado de cada elemento do gradiente dos parâmetros:

$$S_{\partial W^{[l]}} = \beta S_{\partial W^{[l-1]}} + (1-\beta)[\partial W^{[l]}]^2$$

$$S_{\partial b^{[l]}} = \beta S_{\partial b^{[l-1]}} + (1-\beta)[\partial b^{[l]}]^2,$$

em que:

$$\partial W^{[l]} = \frac{\partial C}{\partial W^{[l]}},\ \partial b^{[l]} = \frac{\partial C}{\partial b^{[l]}}$$

Assim como a taxa de aprendizagem, β será um hiperparâmetro a ser otimizado durante o treinamento. É comumente usado $\beta = 0.9$ como valor inicial. Logo após calcular S, é efetuada a etapa de atualização dos parâmetros, porém novamente Hinton[33] tomou uma abordagem diferente em relação ao padrão utilizado. Sua ideia foi subtrair, dos parâmetros, seus respectivos gradientes, porém com estes divididos pela raiz quadrada de S. Quanto maior o valor de S, menor será o passo dado pelo algoritmo, evitando uma possível divergência; e quanto menor o valor de S, maior será o passado dado, acelerando o processo na direção correta. Por fim é possível atualizar os parâmetros:

$$W^{[l]} = W^{[l]} - \eta \frac{\partial W^{[l]}}{\sqrt{S_{\partial W^{[l]}}}}$$

$$b^{[l]} = b^{[l]} - \eta \frac{\partial b^{[l]}}{\sqrt{S_{\partial b^{[l]}}}}$$

2.8.3 Adam

O otimizador Adaptive Moment Estimation(Adam), proposto por Diederik Kingma e Jimmy Ba[18], faz uso do método dos momentos e do RMSProp de forma a acelerar a otimização, aplicado a um mini-batch. Sendo baseado no SGD e nos momentos, o algoritmo Adam tem se mostrado o mais avançado e eficiente algoritmo de otimização. Ele é inicializado sendo primeiramente calculado o primeiro momento, como mostrado anteriormente, em que β_1 é um hiperparâmetro:

$$V_{\partial W^{[l]}} = \beta_1 S_{\partial W^{[l]}} + (1 - \beta_1)\partial W^{[l]}$$

$$V_{\partial b^{[l]}} = \beta_1 S_{\partial b^{[l]}} + (1 - \beta_1)\partial b^{[l]}$$

A intuição aqui continua a mesma, acelerar ou desacelerar os passos durante o processo. Logo em seguida será utilizado o RMSProp com médias móveis para calcular o segundo momento, em que β_2 será outro hiperparâmetro:

$$S_{\partial W^{[l]}} = \beta_2 S_{\partial W^{[l-1]}} + (1 - \beta_2)[\partial W^{[l]}]^2$$

$$S_{\partial b^{[l]}} = \beta_2 S_{\partial b^{[l-1]}} + (1 - \beta_2)[\partial b^{[l]}]^2$$

Uma correção de viés é aplicada em V e S, pois os mesmos são inicializados com o valor 0. Sem uma correção, teríamos um viés em torno de valores pequenos. A correção é feita como:

$$V^{\text{correct}}_{\partial W^{[l]}} = \frac{V_{\partial W^{[l]}}}{(1 - \beta_1^t)},\ V^{\text{correct}}_{\partial b^{[l]}} = \frac{V_{\partial b^{[l]}}}{(1 - \beta_1^t)}$$

$$S^{\text{correct}}_{\partial W^{[l]}} = \frac{S_{\partial W^{[l]}}}{(1 - \beta_2^t)},\ S^{\text{correct}}_{\partial b^{[l]}} = \frac{S_{\partial b^{[l]}}}{(1 - \beta_2^t)}$$

Atualizando os parâmetros:

$$W^{[l]} = W^{[l]} - \eta \frac{V^{\text{correct}}_{\partial W^{[l]}}}{\sqrt{S^{\text{correct}}_{\partial W^{[l]}}} + \epsilon}$$

$$b^{[l]} = b^{[l]} - \eta \frac{V^{\text{correct}}_{\partial b^{[l]}}}{\sqrt{S^{\text{correct}}_{\partial b^{[l]}}} + \epsilon},$$

em que t é o número de iterações no mini-batch e ϵ um termo de erro para não ocasionar uma divisão por zero. Os hiperparâmetros podem ser testados e otimizados, mas Diederik Kingma e Jimmy Ba[18] concluem que os hiperparâmetros são mais eficentes e servem para

uma variedade de casos e problemas quando $\beta_1 = 0.9$, $\beta_2 = 0.999$ e $\epsilon = 10^{-8}$. A taxa de aprendizagem η ainda deve ser testada e otimizada.

2.9 Backpropagation

Como é preciso calcular os gradientes dos parâmetros em relação à função de Custo, é necessário um algoritmo capaz de suportar os diversos tamanhos que uma Rede Neural possa ter. Algoritmos tradicionais de diferenciação, como o de diferenças finitas, não são bem otimizados para o problema, devido ao grande número de parâmetros da rede. Como solução, é possível fazer uso do Backpropagation, proposto por Rumelhard et al.[30] em 1986, em que o cálculo do gradiente dos parâmetros de camadas finais é armazenado para se calcular o gradiente dos parâmetros de camadas anteriores, utilizando a regra da cadeia.

Nosso objetivo principal é calcular $\frac{\partial C}{\partial w_j^{[l]}}$ e $\frac{\partial C}{\partial b^{[l]}}$ para obter a atualização dos parâmetros $w_j^{[l]}$ e $b^{[l]}$, respectivamente, sendo $w_j^{[l]}$ o peso do j_{th} elemento da l_{th} camada. Considerando $[L]$ a última camada da rede, podemos obter esse valor através da regra da cadeia:

$$\frac{\partial C}{\partial w_j^{[L]}} = \frac{\partial C}{\partial a_j^{[L]}} \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} \frac{\partial z_j^{[L]}}{\partial w_j^{[L]}}$$

$$\frac{\partial C}{\partial b^{[L]}} = \frac{\partial C}{\partial a_j^{[L]}} \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} \frac{\partial z_j^{[L]}}{\partial b^{[L]}}$$

Agora consideremos δ^l como sendo o erro da l_{th} camada, ou seja, a variação no gradiente da função de Perda em relação aos parâmetros:

$$\delta^L = \frac{\partial C}{\partial a_j^{[L]}} \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}}$$

Sendo

$$z_j^{[L]} = w_j^{[L]} a_j^{[L-1]} + b^{[L]},$$

temos:

$$\frac{\partial z_j^{[L]}}{\partial w_j^{[L]}} = a_j^{[L-1]}, \frac{\partial z_j^{[L]}}{\partial b^{[L]}} = 1$$

Logo, podemos computar a atualização de $w_j^{[L]}$ e $b^{[L]}$ como:

$$\frac{\partial C}{\partial w_j^{[L]}} = \delta^L a_j^{[L-1]}$$

$$\frac{\partial C}{\partial b^{[L]}} = \delta^L$$

Podemos generalizar para qualquer $l = 1, ..., L$ dando continuidade à regra da cadeia e utilizando o erro δ^l no calculo. Achando $\frac{\partial L}{\partial w_j^{[l-1]}}$:

$$\frac{\partial C}{\partial w_j^{[l-1]}} = \frac{\partial C}{\partial a_j^{[l]}} \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial a_j^{[l-1]}} \frac{\partial a_j^{[l-1]}}{\partial z_j^{[l-1]}} \frac{\partial z_j^{[l-1]}}{\partial w_j^{[l-1]}}$$

$$\frac{\partial C}{\partial w_j^{[l-1]}} = \delta^l \frac{\partial a_j^{[l-1]}}{\partial z_j^{[l-1]}} a_j^{[l-1]}$$

Achando $\frac{\partial L}{\partial b^{[l-1]}}$:

$$\frac{\partial C}{\partial b^{[l-1]}} = \frac{\partial C}{\partial a_j^{[l]}} \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial a_j^{[l-1]}} \frac{\partial a_j^{[l-1]}}{\partial z_j^{[l-1]}} \frac{\partial z_j^{[l-1]}}{\partial w_j^{[l-1]}}$$

$$\frac{\partial C}{\partial b^{[l-1]}} = \delta^l \frac{\partial a_j^{[l-1]}}{\partial z_j^{[l-1]}} a_j^{[l-1]}$$

Dado que $\frac{\partial a_j^{[l-1]}}{\partial z_j^{[l-1]}} = w_j^{[l]}$, então:

$$\frac{\partial C}{\partial w_j^{[L-1]}} = \delta^L w_j^{[L]} a_j^{[L-1]}$$

$$\frac{\partial C}{\partial b^{[L-1]}} = \delta^L w_j^{[L]} a_j^{[L-1]}$$

Tendo calculado a variação do peso $w_j^{[l]}$, podemos estender o calculo para todo o conjunto de parâmetros $W^{[l]}$. Para isso é utilizado o operador $\odot$, definido como o produto Hadamard, em que cada elemento de uma matriz é multiplicada pelo respectivo elemento da outra matriz. Essa vetorização é extremamente útil quando queremos computar de uma só vez os gradientes para todo o conjunto de parâmetros, economizando tempo e custo de processamento. Dado $g^{[l]}$ a função de ativação na l_{th} camada, primeiro é preciso calcular o erro na l_{th} camada, como feito anteriormente. Generalizando é possível calcular δ^l sabendo o valor de δ^{l+1}:

$$\delta^L = \frac{\partial C}{\partial A^{[L]}} \odot g'^{[L]}(Z^{[L]})$$

$$\delta^l = ((W^{[l+1]})^T \delta^{l+1}) \odot g'^{[l]}(Z^{[l]})$$

Logo em seguida é calculado o gradiente da função de Custo C em relação a $W^{[l]}$ e $b^{[l]}$:

$$\frac{\partial C}{\partial W^{[l]}} = A^{[l-1]} \delta^l$$

$$\frac{\partial C}{\partial b^{[l]}} = \delta^l$$

2.10 Inicialização Aleatória

Por fazer uso de um algoritmo de otimização como o Gradiente Descendente, não é possível inicializar todos os parâmetros como 0, pois, no final, todos eles irão convergir para o mesmo valor, assim como inicializar de forma simétrica também não será correto. É preciso, então, iniciar cada peso de forma aleatória, mais especificamente gerando aleatoriamente valores de uma distribuição Normal. Empiricamente, o método que melhor inicializa os parâmetros é a Inicialização Xavier[10], que pode ser definida como:

$$W^{[l]} \sim \mathcal{N}\left(0, \frac{1}{n^{[l-1]}}\right)$$

Inicializando os parâmetros desta forma há um menor risco da rede divergir ou gerar parâmetros semelhantes, estagnando o treinamento.

2.11 Regularização

A etapa de regularização é de extrema importância durante um treinamento. Dado o ruído que pode se originar durante o treino, tanto pela limitação do tamanho da amostra, quanto pela não-linearidade do problema, é importante amenizar o peso que esses ruídos trazem para dentro do modelo. É preciso também evitar o overfitting, ou seja, que o modelo se torne enviesado com a base de treino e performe mal na base de teste e validação. Para isso, muitas técnicas já conhecidas foram testadas e utilizadas nas arquiteturas de Redes Neurais, como penalizações $L2$ e $L1$, ou até mesmo parando o treino quando este começar a performar mal no teste. Com uma capacidade computacional ilimitada, seria possível obter a média das previsões de cada possível conjunto de parâmetros e calcular sua probabilidade a posteriori, dado a base de treino. Isso seria possível numa rede pequena e com uma base de treino também pequena, mas inviável quando se aumenta uma das duas coisas.

2.11.1 Dropout

Dado o problema de regularizar, Geoffrey Hinton et al.[31] desenvolveram a camada de Dropout, em que, aleatoriamente, a cada etapa do treino, elementos de uma camada são desligados afim de variar a arquitetura da rede e evitar que certos elementos tenham um peso muito maior que outros, evitando consideravelmente o overfitting. Dado uma probabilidade p pré-estabelecida, cada elemento de uma camada tem probabilidade p de ser desligado durante uma época do treinamento.

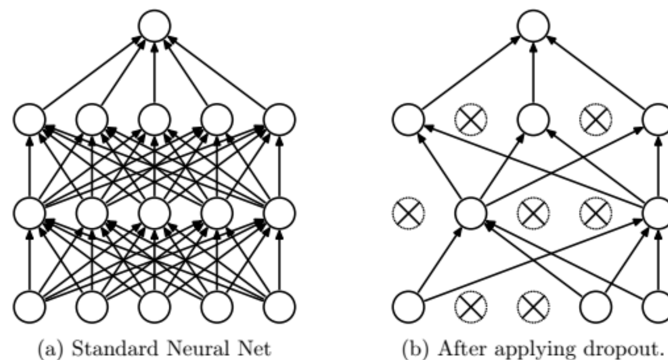

Figura 12: Rede com e sem Dropout.
Fonte: Hinton et al.[31]

O Dropout é comumente utilizada nas Redes Neurais, sendo uma camada quase indispensável nas arquiteturas atuais.

2.12 Normalização

A normalização das variáveis independentes X é uma etapa importante da modelagem, pois torna o processo de otimização mais eficiente. É preciso calcular a média μ e a variância σ^2 da distribuição de X:

$$\mu = \frac{1}{m}\sum x$$
$$\sigma^2 = \frac{1}{m}\sum (x-\mu)^2$$

Para normalizar as variáveis:

$$X = \frac{X-\mu}{\sqrt{\sigma^2}}$$

Porém, quando estamos tratando de Redes Neurais, podemos tratar cada camada oculta também como a entrada da camada seguinte, tendo elas também uma distribuição e parâmetros a serem otimizados. Analogamente, assim como é importante normalizar X, também pode ser importante normalizar a saída $Z^{[l]}$, que servirá de entrada para a camada seguinte $l+1$. E há um problema quando não normalizamos cada camada de uma Rede Neural. Dado que a distribuição dos dados em cada uma das camadas muda durante o treino, isso desacelera o tempo de treinamento e dificulta na escolha de uma taxa de aprendizagem e na forma como os parâmetros devem ser inicializados.

2.12.1 Batch Normalization

Pensando neste problema, Serge Ioff e Christian Szegedy[15] desenvolveram o que é chamado de Batch Normalization. Quando mudamos os parâmetros de uma camada, haverá uma influência considerável nos parâmetros das camadas seguintes. Como a distribuição das camadas muda constantemente, isso pode estender o tempo de treinamento e causar, até mesmo, problemas de overfitting. Os autores propõem normalizar as camadas para cada mini-batch de treino. O Batch Normalization permite que seja usada uma taxa de aprendizagem menos restritiva e inicializações mais fáceis de serem implementadas. Apesar do Dropout ainda ser utilizado em conjunto com o Batch Normalization, este também ajuda como um regularizador. No trabalho original os autores fazem a normalização em $Z^{[l]}$, porém também é possível fazer o mesmo após a função de ativação, ou seja, em $A^{[l]}$. Como primeira etapa, é calculada a média μ e a variância σ^2 de $Z^{[l]}$:

$$\mu = \frac{1}{m}\sum_i z_i^{[l]}$$
$$\sigma^2 = \frac{1}{m}\sum_i (z_i^{[l]} - \mu)^2$$

Para, em seguida, normalizar as variáveis, sendo ϵ uma constante para evitar divisão por zero:

$$Z_{\mathrm{n}}^{[l]} = \frac{Z^{[l]} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

Podemos ainda parametrizar a normalização de $Z_{\mathrm{n}}^{[l]}$, permitindo com que a própria normalização seja uma etapa a ser otimizada. São utilizados os hiperparâmetros γ e β:

$$\widetilde{Z}^{[l]} = \gamma Z_{\mathrm{n}}^{[l]} + \beta,$$

Batch Normalization, portanto, pode ser tratado como uma camada da Rede Neural, que se encaixa anteriormente à camada que queremos normalizar. Com a camada normalizada, a etapa seguinte ao Batch Normalization será o uso da função de ativação que servirá de entrada para a camada seguinte:

$$A^{[l]} = g(\widetilde{Z}^{[l]})$$

3 Rede Neural Convolucional

Quando tratamos de dados não estruturados, como imagens, o uso de uma simples Rede Neural pode ser computacionalmente custoso e ineficiente. Se considerarmos cada pixel de uma imagem como uma característica(feature) x_i, nossa rede irá crescer consideravelmente quanto maior for a imagem. Sendo a rede inteiramente conectada, o tamanho das matrizes de pesos seriam exageradamente grandes.

Para exemplificar melhor, uma imagem é representada no computador por uma matriz(imagem preto e branco) ou um conjunto de matrizes(imagem colorida). Um conjunto de matrizes também pode ser chamado de tensor. Cada matriz do tensor, no caso de uma imagem colorida, representa uma camada de cor; cada elemento da matriz representa um pixel que pode variar de 0(mais escuro ou preto) até 255(mais claro ou branco). No caso de uma imagem colorida(RGB), um único pixel pode representar 256^3 cores diferentes. Sendo n_h a altura da imagem e n_w a largura, uma imagem com 1 camada(preto e branco) terá $n_h * n_w$ pixels(elementos, ou features). Em uma imagem RGB(colorida), com $n_c = 3$, teremos $n_h * n_w * 3$ pixels ou features.

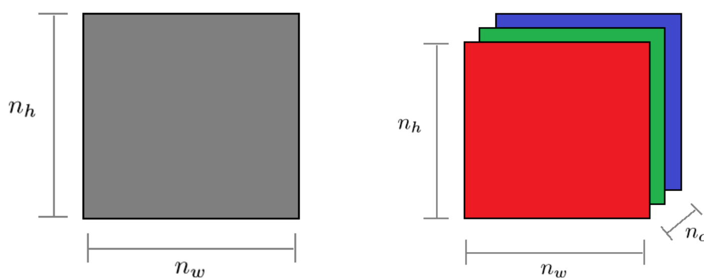

Figura 13: Dimensão de uma imagem Cinza e RGB.

Se considerarmos uma imagem em alta resolução 1280x720, conectada a uma camada da rede com 128 elementos, apenas na primeira camada oculta teríamos $1280 * 720 * 3 * 128 \approx 3.54 * 10^8$ parâmetros, inviável computacionalmente para qualquer algoritmo de otimização. Há também problemas com relação à estrutura de uma imagem: em uma rede feedforward não é possível detectar elementos básicos como rotação, tamanho ou distorções geométricas, ignorando até mesmo pixels(ou conjuntos) que são altamente correlacionados com outros. Além disso, há um problema de redundância com os parâmetros, em que muitas vezes parâmetros diferentes captam uma mesma informação. Como solução a esses problemas, Yann LeCun[21] propôs um modelo de Rede Neural Convolucional, em que são aplicadas convoluções e filtros na imagem como uma camada da rede. Esta solução diminui consideravelmente o número de parâmetros e leva em conta todas as características que um conjunto de pixels(ou features) possa ter. LeCun se baseou no sistema visual dos gatos, estudado por Hubel e Wisel[14] na década de 60.

A ideia geral é mais simples do que se possa apresentar. A parte central, que dá nome à Rede Neural Convolucional, é a operação de convolução. Ela, através de filtros, tem a

função de simplificar e, ao mesmo tempo, captar as características de uma imagem ou de um conjunto de imagens. Os filtros, que podem ser pré-estabelecidos ou parâmetros da rede, farão o papel de "procurar" dentro da imagem essas características. Essa operação diminui a quantidade de parâmetros necessários e encurta consideravelmente o tamanho da rede, diminuindo o tempo de processamento e modelando o problema com uma eficiência muito maior. Outras ferramentas, como padding, stride e pooling, são utilizadas para sanar problemas que a operação de convolução pode causar, como redução de dimensão e limitação da profundidade da rede. Por último, a convolução transposta, que terá grande importância nas GANs, funciona basicamente como uma convolução reversa, em que partindo de uma dimensão menor, é possível aumentar a dimensão de uma matriz (ou imagem). A seguir são apresentados os detalhes de cada peça da Rede Neural Convolucional.

3.1 Convolução

Convolução é uma operador linear, representada por $*$, que, no contexto de uma matriz, soma o produto dos elementos de duas matrizes distintas em deslocamento. Dado uma imagem I de tamanho n_hxn_w e uma matriz F de tamanho f_hxf_w, uma operação de Convolução se dá:

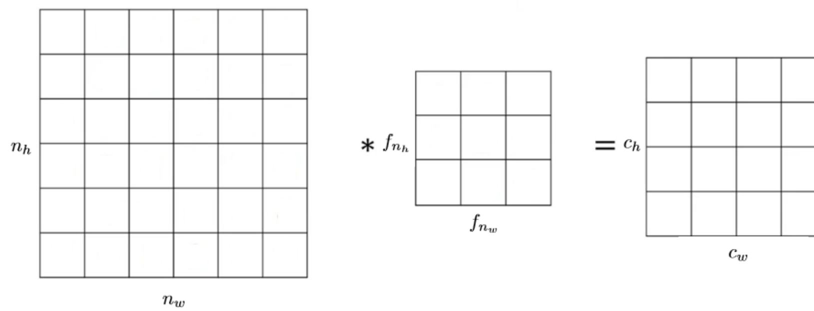

Figura 14: Representação de uma operação de Convolução.

Em que C é uma matriz de tamanho,

$$c_h = n_h - f_h + 1$$
$$c_w = n_w - f_w + 1$$

Os valores de C serão:

$$C = I * F$$
$$C_{m,k} = \sum_{j=m}^{(f_w+m-1)} \sum_{i=k}^{(f_h+k-1)} I_{i,j} F_{i,j},$$
$$m = \{1, \ldots, c_h\}, k = \{1, \ldots, c_w\}$$

3.1.1 Filtro

Podemos considerar um filtro F como sendo uma matriz capaz de detectar estruturas em uma imagem, como bordas verticais e horizontais. Como exemplo, podemos detectar bordas verticais com o seguinte filtro:

$$F = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

Dado uma imagem $I_{4\text{x}6}$:

$$I = \begin{bmatrix} a & a & a & 0 & 0 & 0 \\ a & a & a & 0 & 0 & 0 \\ a & a & a & 0 & 0 & 0 \\ a & a & a & 0 & 0 & 0 \end{bmatrix}$$

A convolução entre I e F terá como resultado:

$$I * F = \begin{bmatrix} 0 & 3a & 3a & 0 \\ 0 & 3a & 3a & 0 \end{bmatrix}$$

Visualmente, há uma linha vertical na matriz I, entre as colunas 3 e 4. Aplicar o filtro F resultará numa matriz em que as colunas do centro terão valores não negativos, indicando que essa linha vertical realmente está no centro de I. Isso posteriormente será usado pela rede como uma característica particular da imagem.

Aplicando o mesmo filtro $3x3$ a uma operação de convolução em uma imagem de tamanho $200x300$, resultará em outra imagem $198x298$:

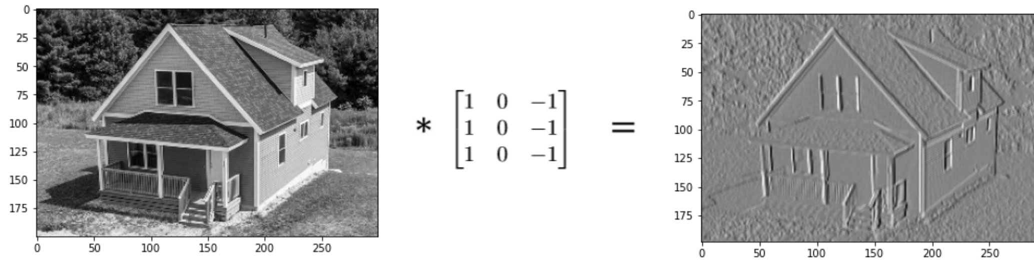

Figura 15: Representação visual da aplicação de um Filtro na Convolução.

Perceba como no exemplo anterior foi possível detectar bordas/linhas verticais na casa. A rede poderá identificar que uma determinada combinação de linhas verticas será uma característica de uma casa. Para detectar bordas horizontais, precisamos apenas transpor o filtro F. A principal ideia por trás de uma Rede Neural Convolucional, por tanto, é aprender características de uma imagem através de filtros passados como um parâmetro do modelo. A rede, então, irá aprender a detectar as estruturas da imagem ao minimizar a função de Custo e atualizar os parâmetros dos filtros. Ao não passarmos um filtro em específico há a vantagem da rede aprender filtros diversificados, que servirão para detectar diferentes características na imagem. Isso torna o processo mais simples, rápido e robusto.

3.1.2 Padding

Há dois problemas em aplicar filtros em uma imagem: redução do tamanho e perda de informação. Ao reduzir o tamanho, não será possível criar uma rede profunda. Perder informação também será um problema nos cantos da imagem. Isso pode ser facilmente resolvido adicionando linhas e colunas de valor 0 nos cantos, processo chamado de Padding.

Ao aplicar um Padding de tamanho p a uma imagem $I_{n_h \text{x} n_w}$, a nova imagem terá dimensão $(n_h + 2p)\text{x}(n_w + 2p)$

$$\begin{bmatrix} I \end{bmatrix}_{n_h \text{x} n_w} = \begin{bmatrix} 0 & \dots & 0 \\ \vdots & I_{n_h \text{x} n_w} & \vdots \\ 0 & \dots & 0 \end{bmatrix}_{(n_h+2p)\text{x}(n_w+2p)}$$

Ao se aplicar uma convolução $I * F$, C terá dimensão:

$$c_h = n_h + 2p - f_{n_h} + 1$$
$$c_w = n_w + 2p - f_{n_w} + 1$$

Há dois casos comumente usados ao se aplicar Padding: **Same convolution**, onde p é escolhido de tal forma que $n_h = c_h$ e $n_w = c_w$; e **Valid convolution**, em que $p = 0$.

3.1.3 Stride

Stride é uma extensão da convolução, em que a operação se dá pulando s linhas e colunas durante a operação. Quando $s = 1$, temos uma convolução sem Stride.

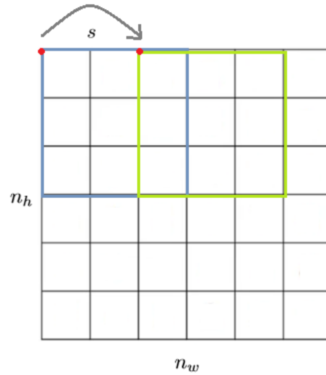

Figura 16: Representação de um Stride $s = 2$.

Dado uma imagem $I_{n_h x n_w}$, um filtro F de tamanho $f_h x f_w$, um padding p e um stride s, a operação de convolução $I * F = C_{c_h x c_w}$ terá dimensão:

$$c_h = \lfloor \frac{n_h + 2p - f_h + 1}{s} \rfloor$$
$$c_w = \lfloor \frac{n_w + 2p - f_w + 1}{s} \rfloor$$

3.1.4 Pooling

O Pooling tem como objetivo reduzir a dimensão dos dados(downsampling), simplificando sua estrutura e cortando informações a mais. Ela é apresentada mais comumento como Max Pooling e Average Pooling. Dado uma imagem I, será utilizado um kernel(ou subdivisão da imagem) para fazer a operação de Pooling. A diferença entre Max e Average se dá dentro da submatriz, onde o primeiro tomará o valor máximo do kernel e o segundo fará a média dos valores. No exemplo a seguir, utilizando uma imagem I_{4x4} e um kernel K_{2x2}, a imagem I será subdividida em 4 partes de mesma dimensão do kernel K, para enfim ser aplicada a operação escolhida (Max ou Average):

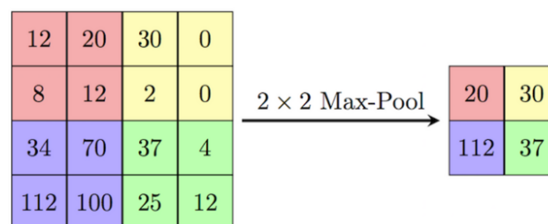

Figura 17: Exemplo de um Max Pooling.
Fonte: computersciencewiki.

A imagem I sofreu uma redução de informação, ou um downsampling, passando para metade do tamanho. As dimensões da matriz final irão variar apenas em relação ao tamanho do kernel K.

3.2 Convolução sobre Volume

Tratando de modelos com imagens em 3 camadas e arquiteturas mais complexas, é comum fazermos convoluções não apenas em matrizes, mas também em tensores (conjunto de matrizes). A convolução sobre Volume, ou convolução 2D, geralmente é feita fixando no filtro a profundidade do tensor a ser convolucionado, variando apenas a largura e altura. O tensor I terá dimensão n_hxn_wxn_c, enquanto o filtro F terá dimensão f_hxf_wxn_c. A operação é feita da mesma forma, multiplicando os elementos de I e F e depois somando o resultado; As operações de Padding, Stride e Pooling se mantêm validas. Exemplo de uma convolução com apenas 1 filtro:

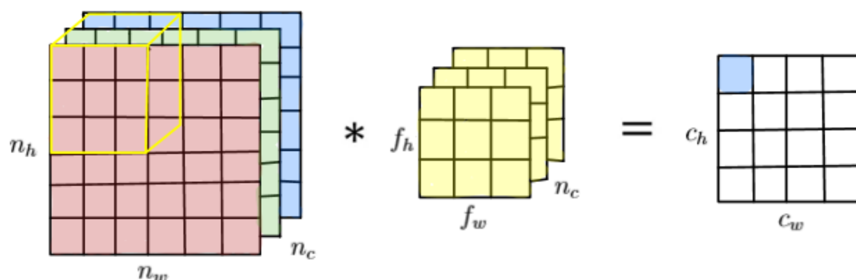

Figura 18: Convolução sobre Volume (Conv2d).
Fonte: guandi1995.github.io

Caso a camada seja construída com n filtros, n convoluções serão aplicadas, tendo como saída uma pilha contendo os resultados das convoluções em forma de tensor, de dimensão c_hxc_wxn:

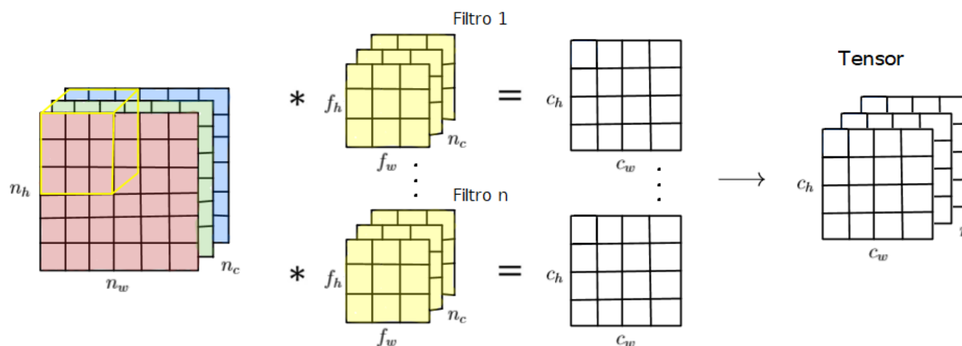

Figura 19: Convolução sobre Volume (Conv2d) gerando um Tensor.
Fonte: guandi1995.github.io

3.3 Convolução Transposta

Convolução Transposta é uma técnica em que se faz upsampling da matriz, ou seja, aumenta a dimensão da mesma. Ela é de suma importância ao se criar Redes Adversárias Generativas (GANs), em que partimos de um vetor de variáveis aleatórias de tamanho definido e chegamos em uma imagem (matriz ou tensor). Assim como na convolução padrão, a convolução transposta aplica filtros como parâmetros. O exemplo a seguir mostra uma convolução transposta entre uma matriz I de tamanho $2x2$ e um filtro F de tamanho $2x2$, resultando numa matriz de tamanho $3x3$.

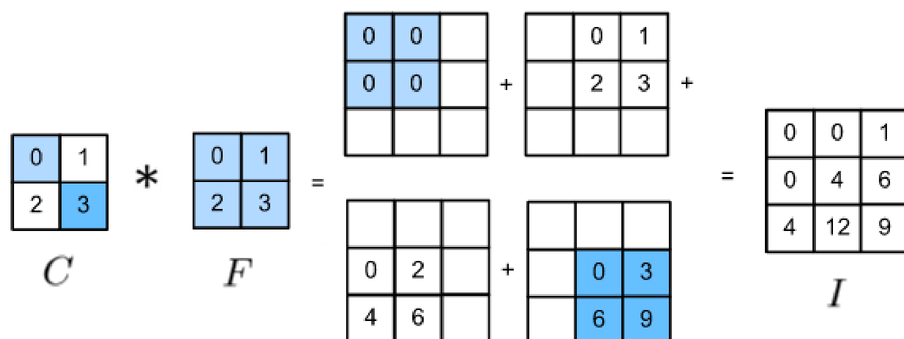

Figura 20: Exemplo de uma Convolução Transposta.

Cada elemento da matriz C é multiplicado por todos os elementos da matriz F, tendo o resultado somado posteriormente. Para se encontrar o tamanho da matriz final I é preciso apenas inverter a fórmula para encontrar o valor de C na Convolução. Sendo C de tamanho c_hxc_w, e F de tamanho f_hxf_w, teremos I de tamanho n_hxn_w:

$$n_h = c_h + f_h - 1$$
$$n_w = c_w - f_w - 1$$

Na convolução transposta a operação sobre volume também se mantém válida.

3.4 LeNet-5

Yann LeCun[21] propôs a LeNet-5, uma das primeiras Redes Neurais Convolucionais, capaz de reconhecer dígitos e letras manuscritas em preto e branco. A rede composta de 6 camadas ocultas: A camada $C1$ é uma convolução contendo 6 filtros de tamanho $5x5$ e $p = 2$; $S2$ é uma camada com 6 Average Poolings de tamanho $2x2$ e $s = 2$; $C3$ outra convolução com 16 filtros também de tamanho $5x5$, porém sem padding; seguido por mais uma camada de 16 Average Poolings $S4$ de tamanho $2x2$, com $s = 2$. Ao final, $C5$ e $F6$ são camadas inteiramente conectadas. Por fim, uma Activation Function é utilizada para computar a

saída, assim como uma função de Custo é utilizada para atualizar os parâmetros da rede.

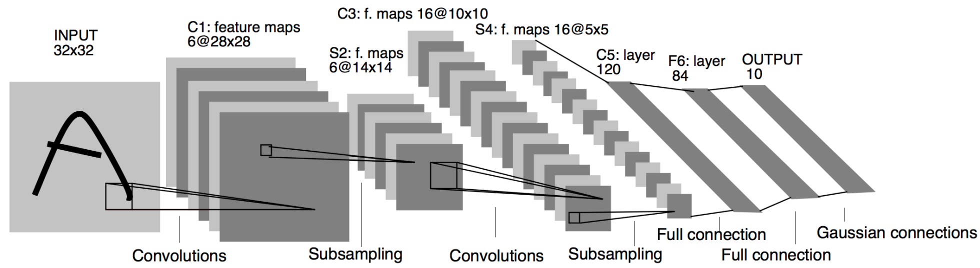

Figura 21: Arquitetura LeNet-5.
Fonte: Yann LeCun[21]

No caso do experimento de LeCun, a saída é um vetor de tamanho 10, contendo a probabilidade da imagem em questão ser um dos 10 dígitos numéricos. No lugar de uma Função Sigmoid, é utilizada uma Softmax, que pode computar valores de probabilidades para todas as classes presentes na base de dados.

4 Redes Adversárias Generativas

As GANs(Redes Adversárias Generativas), apresentadas por Ian Goodfellow[11] para estimar modelos generativos por meio de um processo adversário, são constituídas de duas Redes Neurais simultâneas: uma rede generativa G, que captura a distribuição dos dados, e uma rede discriminativa D, que estima a probabilidade de uma amostra vir da base de treino ao invés de G. O modelo G é treinado maximizando a probabilidade de D cometer um erro. Esse tipo de rede é um jogo minimax de dois jogadores.

Grande parte dos modelos tradicionais de aprendizado profundo se referem a modelos discriminativos, em que características(features) são processadas até gerar uma classe de saída(ou um valor de probabilidade). Grande parte do sucesso destas redes se dá pelo uso de backpropagation e métodos de regularização. Porém, modelos generativos possuem um menor impacto na área de aprendizado profundo devido à alta complexidade para se computar as probabilidade da distribuição dos dados, tendo exemplos como as Deep Boltzmann Machines(DBM), estimadas por MCMC(Markov Chain Monte Carlo).

Em uma GAN o modelo generativo proposto é colocado contra um adversário: um modelo discriminativo D que aprende a determinar se uma amostra vem da distribuição original ou do modelo G. A competição entre os dois faz com que os modelos aprendam um com outro: D tentando classificar as amostras geradas e G tentando gerar amostras mais próximas da distribuição dos dados. Uma amostra G é gerada alimentando um multilayer perceptron com ruído aleatório ξ, em que sua saída será a entrada de outro multilayer perceptron D. Esse tipo de abordagem permite o uso de backpropagation e métodos de regularização já definidos em uma Rede Neural Artificial, sem precisar fazer inferência ou usar cadeia de Markov.

$$
\begin{aligned}
D &: X, \theta_d \rightarrow \hat{Y}, P(Y|X) \\
G &: \overset{\text{ruído}}{\xi}, \theta_g \rightarrow \hat{X}, P(X|Y)
\end{aligned}
$$

4.1 Redes Adversárias

Dada uma base de dados X, é preciso definir o gerador G e o discriminador D:

- Definimos $G(\xi; \theta_g)$ como sendo uma função diferenciável representada por um multilayer perceptron de parâmetros θ_g, sendo ξ um ruído com distribuição de probabilidade $p_\xi(\xi)$, que servirá de entrada para gerar a distribuição de probabilidade p_g.

- Definimos $D(x; \theta_d)$ como sendo um multilayer perceptron que resulta em um escalar(classe ou probabilidade). $D(x)$ representa a probabilidade de x vir da amostra original ao invés de p_g. D é treinado para maximizar a probabilidade de acertar a classe da amostra original e da amostra gerada G, enquanto G é treinado para minimizar $log(1-D(G(\xi)))$,

em que $D(G(\xi))$ é a probabilidade de $G(\xi)$ ser da distribuição real, portanto queremos minimizar a chance de $\hat{y}$, ou $D(G(\xi))$, ser da classe 0 (falsa).

Sumarizando, podemos definir o treinamento de D e G como sendo um jogo de minimax entre dois jogadores com uma função de valor $V(G, D)$:

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

É possível, e também mais indicado, treinar G maximizando $\log(D(G(\xi)))$, evitando que D aprenda muito rápido a rejeitar dados gerados pela distribuição de G, caso o mesmo seja mal definido e muito diferente da distribuição real. O processo de treinamento é feito primeiro treinando D, depois mantendo os parâmetros θ_d fixos a fim de treinar os parâmetros θ_g. Ian[11] propõe um algoritmo com minibatch SGD com k etapas:

Algorithm 1 Rede Adversária Generativa

1: **for** número de iterações de treino **do**
2:     **for** k etapas **do**
3:         Dado uma amostra de m ruídos $\{\xi^{(1)}, \ldots, \xi^{(m)}\}$ de uma distribuição $p_g(\xi)$
4:         Dado uma amostra de m exemplos $\{x^{(1)}, \ldots, x^{(m)}\}$ de uma distribuição $p_{data}(x)$
5:         Atualizar D com SGD:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} [\log D(x^{(i)}) + \log(1 - D(G(\xi^{(i)})))]$$

6:     **end for**
7:     Dado uma amostra de m ruídos $\{\xi^{(1)}, \ldots, \xi^{(m)}\}$ de uma distribuição $p_g(\xi)$
8:     Atualizar G com SGD:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log(1 - D(G(\xi^{(i)})))$$

9: **end for**

Uma forma simplificada de entender o modelo é olhando o fluxograma do mesmo. No treinamento de D(Figura 20) é gerado, através de G, batchs $\hat{X}_g$ de classe "$falso$" com uma distribuição p_g de um ruído ξ em (1). Depois, como entrada, é usado X_d de classe "$real$" com distribuição p_{data} e $\hat{X}_g$(gerado em (1)) para computar a saída $\hat{Y}$ em (2). Através da função de Custo BCE os parâmetros θ_d de D são atualizados em (3). Esta etapa é treinada de forma a fazer o discriminador D aprender quais são imagens da distribuição original e quais são da distribuição de G.

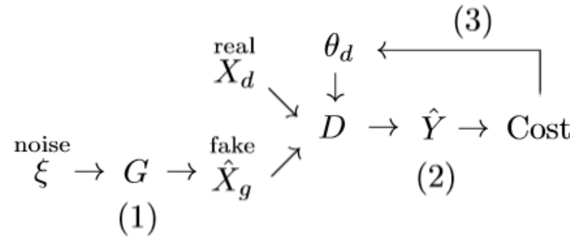

Figura 22: Fluxograma do treinamento do Discriminador.

No treinamento de G(Figura 21) é gerado um batch $\hat{X}_g$ de classe "*real*" com uma distribuição p_g de um ruído ξ em (1). Depois, como entrada, é usado apenas $\hat{X}_g$ para computar a saída $\hat{Y}$ em (2). Através da Função de Custo BCE, os parâmetros θ_g de G são atualizados em (3). O intuito de passar $\hat{X}_g$ como "*real*" é enganar o discriminador D, de tal forma que, caso ele "entenda" que $\hat{X}_g$ não está próximo da distribuição p_{data}, ou seja, classifique como "*falso*" os dados, o otimizador atualize os parâmetros θ_g; Caso as distribuições estejam próximas, há poucas mudanças a serem feitas nos parâmetros.

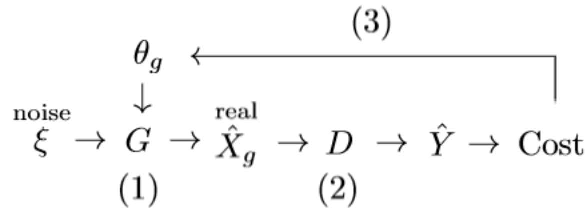

Figura 23: Fluxograma do treinamento do Gerador.

É preciso tomar cuidado com o treinamento dos dois modelos, pois, caso um deles convirja muito rápido, irá "vencer" o oponente. O principal objetivo não é ter vencedores, mas sim os dois melhorarem e se balancearem mutuamente. Se o discriminador for ruim e convergir muito rápido, não saberá identificar as imagens corretas, ocasionando do gerador não convergir. Esta é uma das grandes dificuldades das arquiteturas GANs, pois não há uma métrica bem definida para saber quando os modelos conseguiram convergir ou não e quando estão em balanceamento.

4.2 Propriedades do Modelo

Ian[11] prova que podemos chegar em $p_g = p_{data}$. Primeiro deve-se considerar o discriminador D dado qualquer gerador G:

- Proposição 1(Ótimo Global): Para um G fixo, o discriminador ótimo D será:

$$D_G^*(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)}$$

Para qualquer $(a, b) \in \mathbb{R}^2$, a função $y \to a\log(y) + b\log(1-y)$ será máximo em $[0,1]$ em $\frac{a}{a+b}$.

- Proposição 2(Convergência do Algoritmo): Se G e D forem bem definidos e a cada etapa do algoritmo o discriminador consegue encontrar o ponto ótimo dado G, e p_g é atualizado de forma a melhorar o critério

$$\mathbf{E}_{x\sim p_{data}(x)}[\log D^*(x)] + \mathbf{E}_{x\sim p_\xi(\xi)}[\log(1 - D^*_G(x))],$$

então p_g converge para p_{data}.

4.3 Redes Adversárias Generativas Convolucionais Profundas

Já foi demonstrado que uma Rede Neural Feedforward simples não é o suficiente para classificar imagens complexas. O mesmo se aplica às GANs, em que dificilmente o gerador G irá conseguir atingir um bom resultado, com um discriminador D igualmente ineficaz. A solução mais óbvia é incorporar as Redes Neurais Convolucionais às GANs propostas por Ian Goodfellow. Introduzida em 2016 por Alec Radford, Luke Metz e Soumith Chintala[27], a DCGAN(Rede Adversária Generativa Convolucional Profunda) utiliza Redes Neurais Convolucionais na arquitetura da GAN, sendo capaz de estabilizar o treino e atingir resultados mais complexos para imagens de maior qualidade.

4.3.1 Características da Rede

O discriminador D apresenta uma arquitetura comum de uma CNN, em que são feitas convoluções contendo stride, padding e pooling seguido por funções de ativações LeakyReLU, uma camada de Flatten e por fim uma camada Sigmoid. A função de Custo será a Entropia Cruzada Binária(BCE).

Figura 24: Discriminador de uma DCGAN.
Fonte: Alec Radford et al.[27]

O gerador G originalmente é iniciado com um ruído aleatório ξ de dimensão 100. Logo em seguida é redimensionado para servir como entrada de Convoluções Transpostas com stride, contendo camadas LeakyReLu e Batch Normalization intermediários entre cada etapa, até chegar na dimensão desejada.

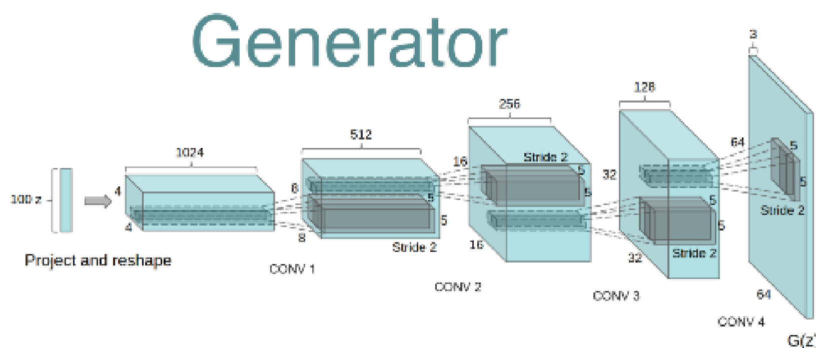

Figura 25: Gerador de uma DCGAN.
Fonte: Alec Radford et al.[27]

O treinamento se dá pelo mesmo modo que uma GAN simples é treinada, alternando a atualização dos parâmetros de D e G separadamente.

Arquiteturas anteriores já tentavam, de alguma forma, incorporar CNNs à GAN, mas sem sucesso. Os autores originais da DCGAN atribuem o sucesso da rede principalmente à camada de Batch Normalization, que conseguiu estabilizar o treino e balancear o discriminador e o gerador. O grande desafio da arquitetura é o alto custo computacional, principalmente quando há uma dificuldade em saber quando os modelos convergem. A seguir são mostradas amostras de um exemplo de uma DCGAN treinada na base MNIST[22](base de dados de dígitos escritos a mão). Em poucas épocas a rede consegue gerar imagens muito semelhantes às da base original:

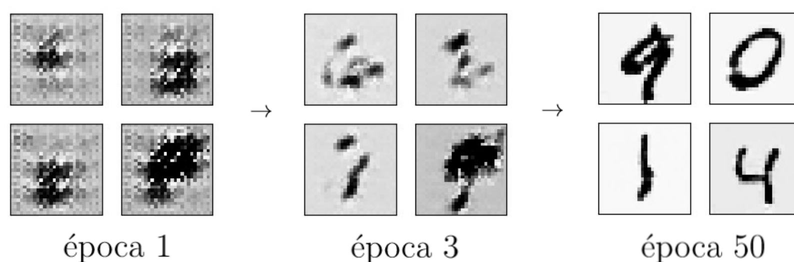

Figura 26: Resultados de uma GAN em diferentes épocas.

Na primeira época de treinamento o gerador apenas gera borrões que apresentam resquícios das imagens reais, mas que são facilmente diferenciadas pelo discriminador. Na época 3 em diante o gerador começa a aprender e atualizar seus parâmetros na direção correta da distribuição real e, em 50 épocas, já é possível ver uma grande semelhança entre amostras geradas e amostras reais.

5 Métricas de Avaliação

Um dos grandes desafios das arquiteturas GANs são as métricas de avaliação. Diferente de um classificador, não existe um modelo, algoritmo ou métrica capaz de avaliar diretamente a qualidade das imagens. Também não é possível usar o próprio discriminador para avaliar as imagens geradas.

Em uma imagem, há duas métricas qualitativas que é preciso avaliar:

- **Fidelidade**: qualidade das imagens geradas;
- **Diversidade**: variação das imagens geradas.

É preciso transformar essas duas métricas em quantitativas, afim de se calcular as distâncias entre duas imagens. É possível gerar imagens fieis à distribuição original, mas se o modelo gera apenas a mesma imagem, ou um número limitado e pequeno de imagens, não temos um gerador de fato. O mesmo se aplica à diversidade, um gerador que cria imagens diversas, mas de baixa qualidade, não terá utilidade. Otimizando as métricas é possível testar se as distribuições de cada amostra de imagens são próximas ou não da realidade e se são bem diversificadas.

Há duas formas possíveis de se calcular uma distância entre duas imagens:

- **Distância entre pixels**: duas imagens são subtraídas uma da outra e sua diferença somadas. Método simples, porém ineficaz quanto maior a complexidade das imagens;
- **Distância entre Características(features)**: leva em conta, inicialmente, as características que uma imagem pode apresentar, para só depois utilizar cada subgrupo de características para calcular a distância.

Calcular a distância por pixels é extremamente ineficaz, pois, em imagens complexas, como a de um cachorro ou gato, qualquer diferença mínima entre pixels fará a distância aumentar ou diminuir consideravelmente, além de, evidentemente, não levar em conta características importantes (como um nariz ou uma boca) que definem um animal. É preciso, então, utilizar características capazes de identificar detalhes da classe que está sendo treinada/gerada. Como comentado, não é possível utilizar as características detectadas pelo próprio discriminador, pois o mesmo estará enviesado pelo contato com o gerador e com a base de treino.

5.1 Extração de Características

Uma rede complexa como uma CNN consegue extrair inúmeras características de uma imagem ou objeto, sejam elas as singularidades de um cachorro, ou o formato de um carro. Tendo as características de uma imagem já conhecida de uma classe X, podemos compara-lá com as de outra imagem que acreditamos ser da mesma classe X. Como não é possível utilizar a rede do discriminador da nossa GAN, o desafio então está em encontrar(ou treinar) uma rede externa que consiga extrair tais características a fim de compararmos imagens reais e geradas.

No capítulo das CNNs vimos que toda a camada de pooling será condensada em uma camada totalmente conectada, com o propósito de gerarmos um vetor de probabilidades para cada classe treinada(camada de Saída). É preciso, portanto, excluir a camada totalmente conectada e de saída da rede, permanecendo apenas a camada de pooling final(que será a nova camada de saída), pois ela é a que melhor representa as características de uma imagem em toda a rede. Para essa tarefa é comumente utilizada a Rede Inception-v3.

5.2 Inception-v3

Criada pelo Google[32], a Inception-v3 foi treinada para o desafio de classificação ImageNet[6], que contém uma base de dados com 14 milhões de imagens e 1000 classes. Obtendo um dos melhores resultados do desafio, a rede conta com extensas camadas e milhões de parâmetros.

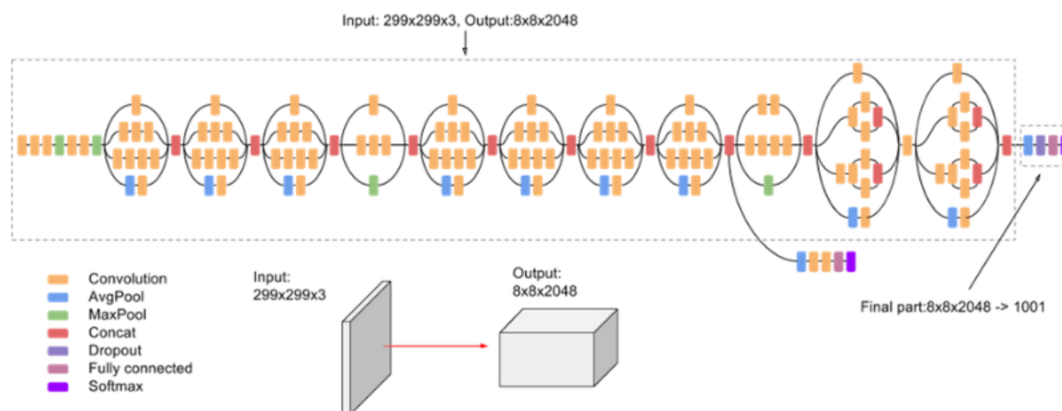

Figura 27: Arquitetura da Inception-v3.

Fonte: cloud-google.

O modelo apresenta uma entrada de dimensão 299x299x3 e uma saída de dimensão 8x8x2048. A saída se conecta a uma camada inteiramente conectada, que terminará num vetor de tamanho 1001 através de uma função de ativação Softmax(análogo a uma Sigmoid, mas para mais de 2 classes). Nosso interesse será apenas na camada de saída, que será simplificada em um vetor de tamanho 2048 através de um Average Pooling.

5.3 Fréchet Inception Score (FID)

A Fréchet Inception Distance, ou FID, é uma métrica de avaliação que mede a distância entre curvas ou compara dois conjuntos de distribuições. Dada uma amostra de imagens com features X e Y, extraídas pela Rede Inception-v3, a distância FID é calculada como:

$$FID = \|\mu_X - \mu_Y\|^2 + \mathrm{Tr}\left(\Sigma_X + \Sigma_Y - 2\sqrt{\Sigma_X \Sigma_Y}\right),$$

em que μ_X e μ_Y são as médias de X e Y, respectivamente; Σ_X e Σ_Y as covariâncias. O operador $Tr(A)$ é a soma da diagonal de uma matriz A. Assumindo que X e Y apresentam uma distribuição Normal Multivariada, é possível calcular a FID entre um conjunto de imagens Reais e Falsas.

Tendo um conjunto N de imagens falsas geradas pela DCGAN, teremos uma matriz de features X de tamanho Nx2048, que poderá ser comparada com outra matriz Y de mesmo tamanho, contendo features de amostras de imagens reais. Quanto menor a distância FID, mais próximas as duas distribuições estão uma da outra.

6 Experimento

Os parâmetros da arquitetura DCGAN foram selecionados empiricamente, sendo testadas diversas combinações e avaliando, tanto qualitativamente as imagens geradas, quanto quantitativamente, usando as distâncias FID. A arquitetura apresentada a seguir não tem como intuito ser a melhor possível, nem a mais eficiente, mas, sendo computacionalmente leve e simples, gerar imagens diversificadas e fidedignas, como foi definido no capítulo de Métricas de Avaliação.

6.1 Arquitetura DCGAN

A arquitetura utilizada no modelo é simplificada e computacionalmente leve, devido às limitações computacionais de hardware. Para isso foram utilizadas poucas camadas tanto no gerador como no discriminador:

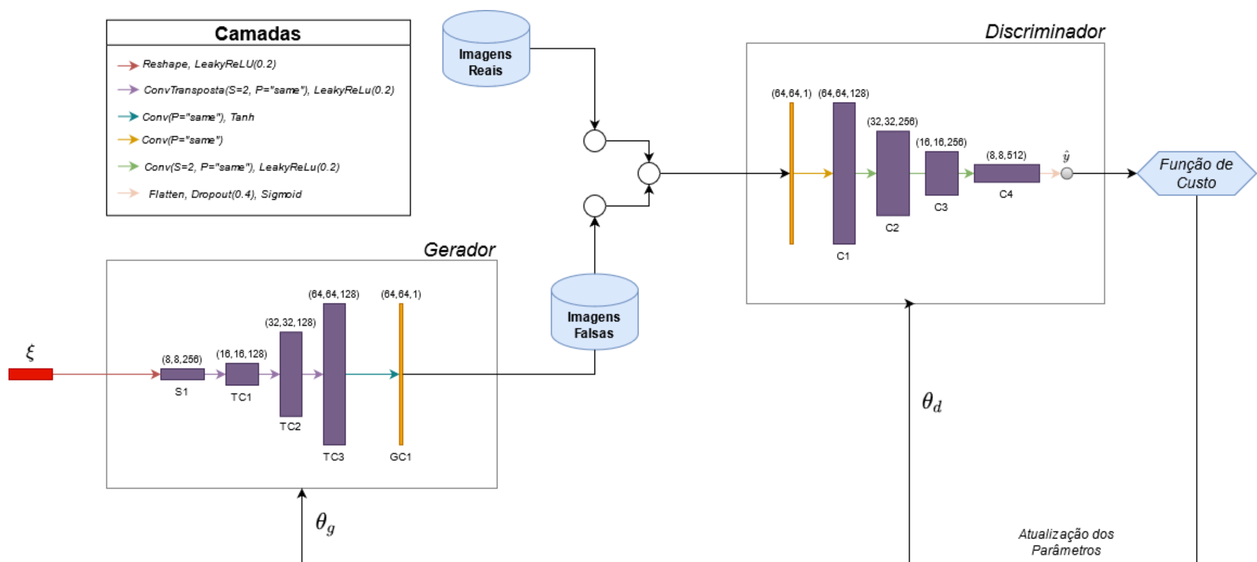

Figura 28: Esquematização da arquitetura DCGAN do experimento.

6.1.1 Discriminador

O discriminador apresenta 4 camadas de convolução, seguidas por camadas totalmente conectadas. Foi utilizada a função de ativação LeakyReLU em sequência às convoluções. Logo após a quarta convolução é aplicado um Flatten para distribuir a saída em um único vetor. Em seguida é aplicada uma camada Dropout como regularização. Por fim, uma função Sigmoid é utilizada para computador um valor $\hat{y} \in [0, 1]$, que servirá para inferir se a imagem passada ao modelo é real ou falsa. A entrada da rede é um tensor Nx64x64x1, em que N representa o tamanho do batch de treinamento. Uma observação importante é a ausência da camada de Batch Normalization. Ela não foi utilizada pois, nas imagens de treino, se obtiveram resultados semelhantes em arquiteturas com e sem ela. Sua retirada diminui o custo computacional do processo.

6.1.2 Gerador

O gerador apresenta 1 camada totalmente conectada, seguida por 3 camadas de convolução transposta e 1 camada de convolução. A camada totalmente conectada apresenta tamanho 16384, seguida por uma ativação LeakyReLU. Logo em seguida um redimensionamento(Reshape) é feito, transformando em um tensor de tamanho 8x8x256. Três convoluções transpostas são feitas seguidas também por uma LeakyReLU. Por fim, uma convolução é feita se aplicando uma função Tanh como ativação final. Como saída teremos um tensor Nx64x64x1, de mesma dimensão da entrada do discriminador D. Aqui, assim como no discriminador, as camadas de Batch Normalization foram suprimidas.

6.2 Etapas de treinamento

Para explicar a estimação do modelo, podemos analisar o passo a passo das etapas de treinamento para apenas uma observação. O processo é feito com um loop em 3 etapas:

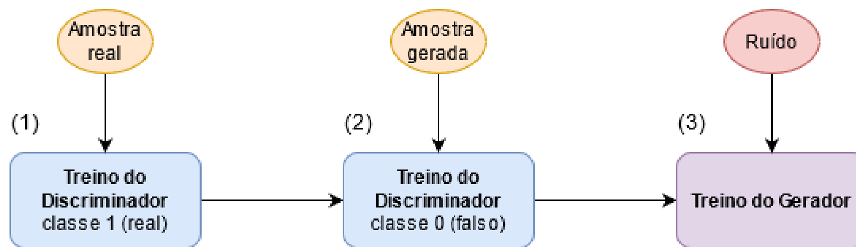

Figura 29: Etapas de treino da DCGAN.

Na etapa (1) o discriminador D é treinado apenas em imagens reais (classe 1); Na etapa (2) o mesmo processo é feito, mas tendo como entrada imagens falsas (classe 0), geradas pelo gerador G; Por fim, na etapa (3), é treinado o gerador G conectando as duas redes uma a outra, tendo como entrada o ruído ξ.

6.2.1 Treinando o Discriminador - Etapas (1) e (2)

As duas etapas se diferenciam apenas na observação utilizada como entrada. Na etapa (1) é utilizada uma imagem X_{real}, enquanto na etapa (2) é utilizado $G(\xi)$. Tomando como exemplo uma observação X_{real}, a rede começa em $C1$ com uma convolução sobre volume com 128 filtros de tamanho 3x3x1 e padding="same" (a saída terá mesma altura e largura da observação inicial). Cada filtro resultará em uma matriz de tamanho 64x64x1, que em conjunto resultarão num tensor de tamanho 64x64x128. O processo é seguido de uma função de ativação Leaky ReLU de parâmetro 0.2. Na imagem a seguir, o quadrado em azul representa 1 dos 128 filtros aplicados, a caixa da esquerda (Imagem) representa a entrada, e a caixa da direita ($C1$) a saída final depois de aplicado todos os filtros:

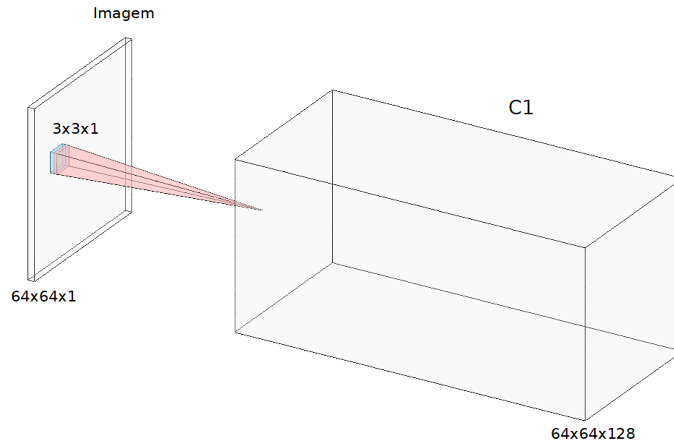

Figura 30: Convolução C1.

Na convolução da camada $C2$ teremos 256 filtros com stride=2, que resultará num tensor 32x32x256, tendo cada filtro dimensão 3x3x128. Importante ressaltar que cada convolução irá gerar uma matriz 32x32x1, que serão empilhadas, sendo o processo feito para cada um dos 256 filtros. Novamente aqui e nas imagens subsequentes a parte em azul será a representação de um filtro:

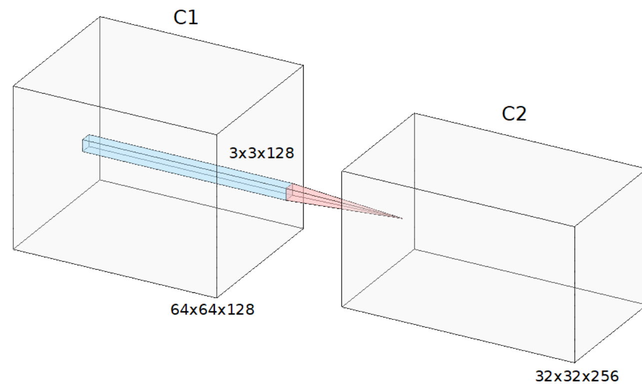

Figura 31: Convolução C2.

As convoluções das camadas $C3$ e $C4$ são feitas de forma análoga, $C3$ com 256 filtros de tamanho 3x3x256 e $C4$ com 512 filtros de tamanho 3x3x512, as duas convoluções com stride=2 e seguidas por uma LeakyReLU de parâmetro 0.2.

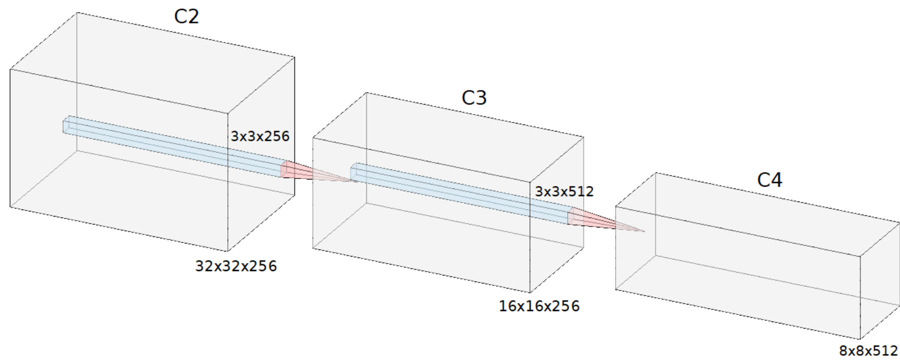

Figura 32: Convoluções C3 e C4.

Por fim, é aplicada uma operação de Flatten (transformação de um tensor em um vetor) e Dropout, resultando em uma camada de tamanho $8*8*512 = 32769$ totalmente conectada por uma função de ativação Sigmoid:

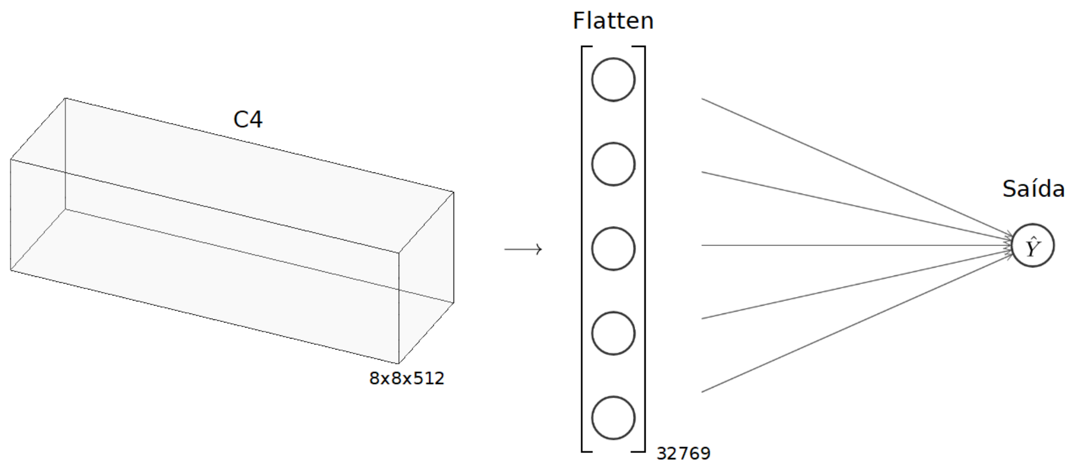

Figura 33: Camadas finais do Discriminador.

A quantidade de parâmetros entre cada camada depende do tamanho do filtro e da quantidade de filtros aplicados na convolução. Tendo n_f filtros de tamanho n_hxn_wxn_c, teremos $(n_h * n_w * n_c * n_f)$ pesos w e (n_f) vieses b.

Um resumo do discriminador D é apresentado a seguir:

Camada	Dimensão de Saída	# Parâmetros
Imagem	64x64x1	-
C1	64x64x128	1280
LeakyReLU	64x64x128	-
C2	32x32x256	295168
LeakyReLU	32x32x256	-
C3	16x16x256	590080
LeakyReLU	16x16x256	-
C4	8x8x512	1180160
LeakyReLU	8x8x512	-
Flatten	32768	-
Dropout	32768	-
Sigmoid	1	32769

Tabela 1: Arquitetura do Discriminador.

Teremos pouco mais de 2 milhões de parâmetros na rede do discriminador, atualizados no Backpropagation por uma função de Custo BCE. Na etapa 1 a função de Custo receberá uma imagem de classe 1 (real), na etapa 2 uma saída $G(\xi)$ de classe 0 (falso). Um resumo do processo de treinamento das etapas 1 e 2 é mostrado a seguir:

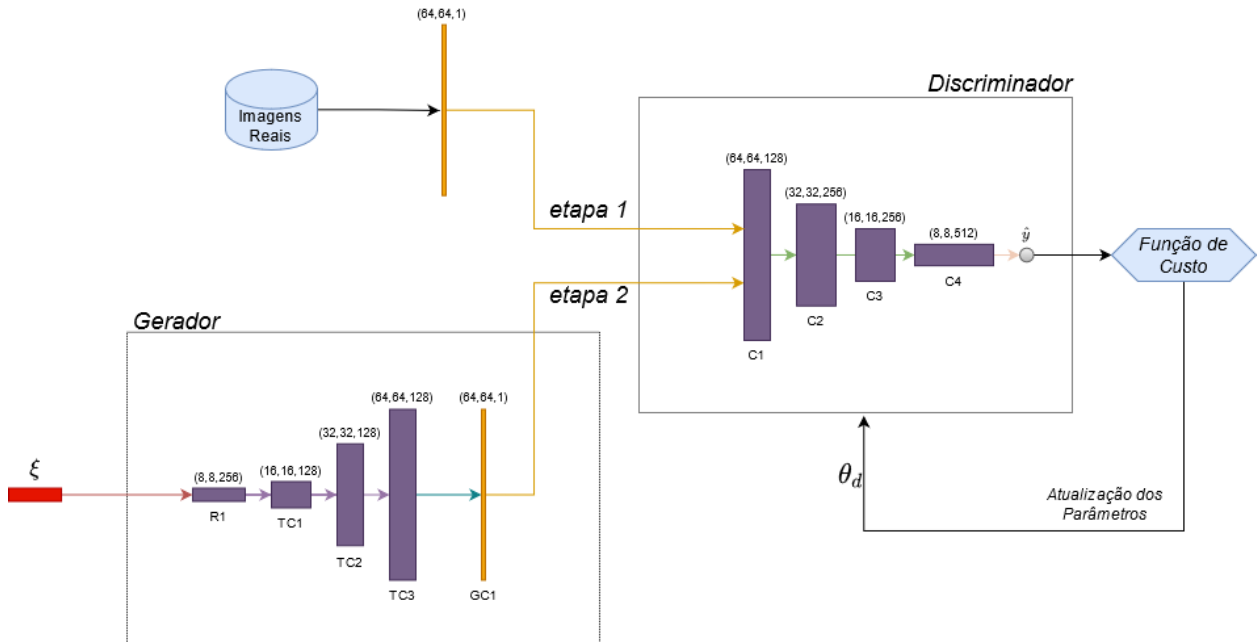

Figura 34: Diagrama das etapas 1 e 2.

6.2.2 Treinando o Gerador - Etapa (3)

Nesta etapa é utilizado o discriminador D, treinado nas etapas (1) e (2), porém com seus parâmetros $\theta_d(w, b)$ mantidos fixos, não sendo alterado no Backpropagation. Apenas

os parâmetros $\theta_g(w, b)$ do gerador G serão atualizados. Isso é feito conectando a rede de G com a rede de D, o primeiro gerando imagens falsas $G(\xi)$ e o segundo estimando $D(G(\xi))$ (classificação). As imagens falsas serão passadas para D como pertencentes a classe 1 (real), com o intuito de enganar o discriminador. Caso D tenha um bom desempenho, o gerador G estará gerando boas imagens e poucas mudanças precisam ser feitas aos parâmetros de G; Caso tenha um mal desempenho, os parâmetros $\theta_g(w, b)$ de G serão atualizados de acordo com a função de Perda BCE.

Inicialmente será gerado um ruído ξ de tamanho 100 de uma distribuição Normal. Em seguida será feito o redimensionamento do ruído para um tamanho 16384 através de uma camada $D1$ totalmente conectada, convertido em um tensor de tamanho 8x8x256 em $R1$:

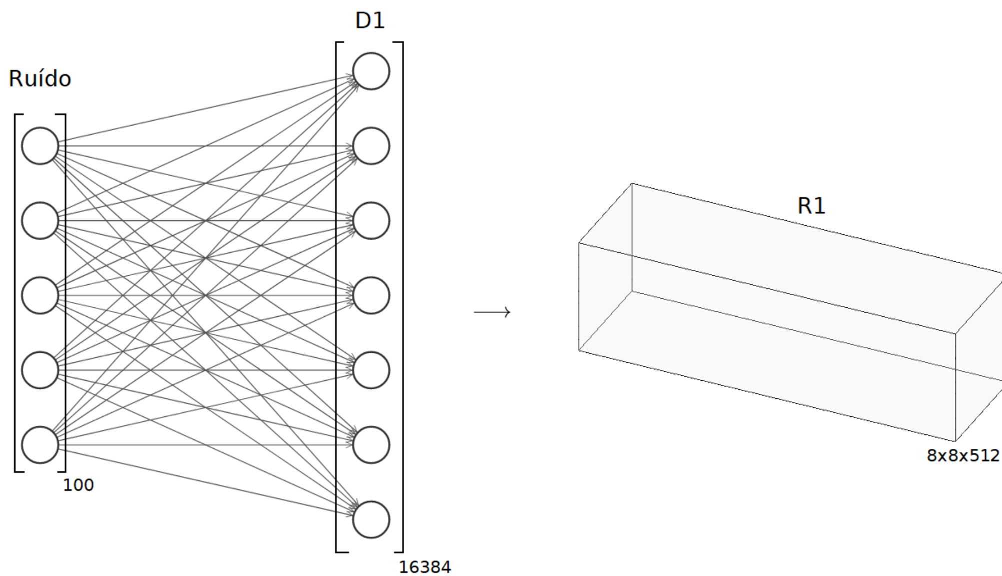

Figura 35: Redimensionamento S1.

A seguir são feitas 3 convoluções transpostas, $TC1$, $TC2$ e $TC3$, com 128 filtros cada, padding="same" e stride=2. $TC1$ possui filtros de tamanho 4x4x256; $TC2$ e $TC3$ possuem filtros de tamanho 4x4x128. Todas as camadas são seguidas por uma função de ativação LeakyReLU de parâmetro 0.2. A camada $R1$ de dimensão 8x8x256, após as operações de convolução transposta, resultará num tensor 64x64x128:

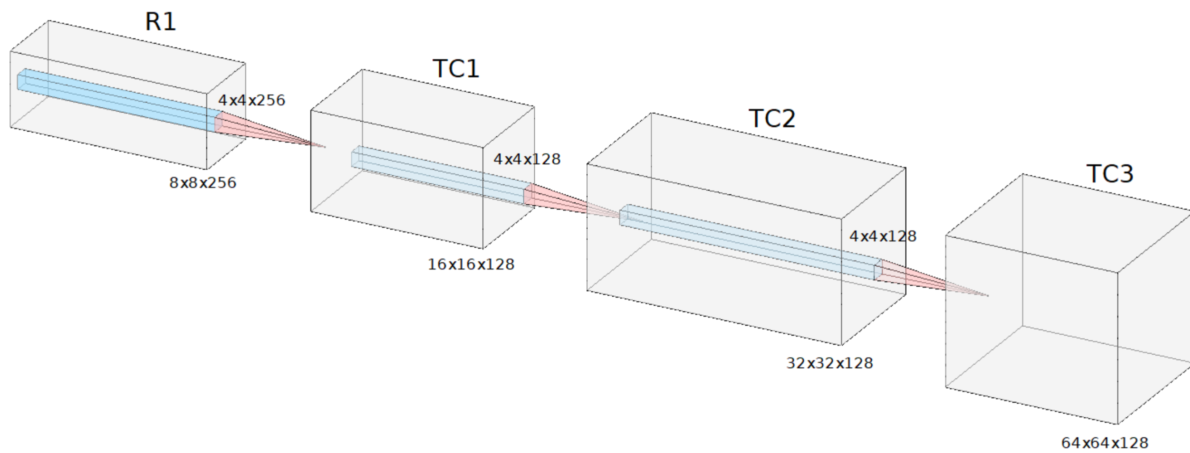

Figura 36: Convoluções Transpostas TC1, TC2 e TC3.

Por fim, será aplicada uma convolução em $TC3$, com 1 filtro de dimensão 3x3x128, padding="same" e função de ativação Tanh, formando a camada $GC1$, uma matriz de dimensão 64x64x1:

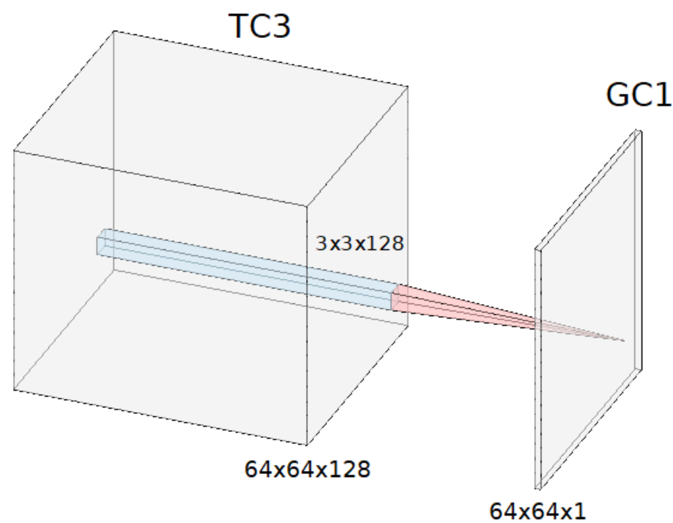

Figura 37: Convolução GC1.

A camada $GC1$ será a entrada para o discriminador D. Como comentado anteriormente, nesta parte $GC1$ terá classe 1 (real), com o intuito de enganar o discriminador.

Um resumo do gerador G é apresentado a seguir:

Camada	Dimensão de Saída	# Parâmetros
Ruído	100	-
D1	16384	1654784
LeakyReLU	16384	-
R1	8x8x256	-
TC1	16x16x128	524416
LeakyReLU	16x16x128	-
TC2	32x32x128	262272
LeakyReLU	32x32x128	-
TC3	64x64x128	262272
LeakyReLU	64x64x128	-
GC1	64x64x1	1153

Tabela 2: Arquitetura do Gerador.

Teremos algo próximo de 2.7 milhões de parâmetros na rede do gerador, atualizados por uma função de Custo BCE no Backpropagation. Um resumo do processo de treinamento da etapa 3 é mostrado a seguir:

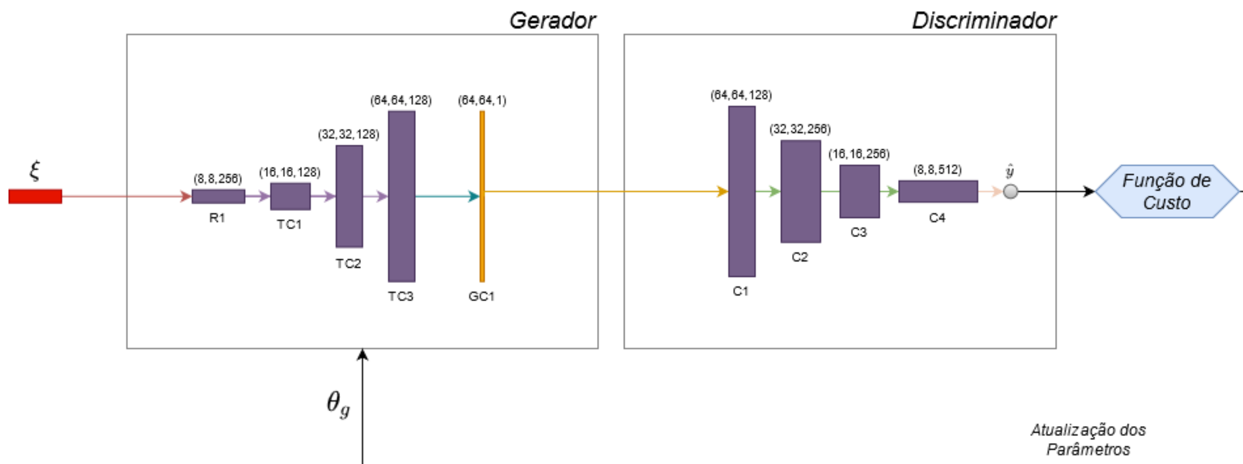

Figura 38: Diagrama da etapa 3.

6.3 Base de Dados

A base de dados contém 5856 imagens de raio-x do tórax, proveniente de crianças entre 1 e 5 anos de idade. As imagens foram selecionadas no Guangzhou Women and Children's Medical Center, em Guangzhou, China[17].

A base está divida em 2 categorias: NORMAL, de pessoas saudáveis e que não apresentam nenhum tipo de infecção no pulmão; e PNEUMONIA, de pessoas com infecções pulmonares. A base foi dividida em treino e teste da seguinte forma:

Classe	# treino	# teste
NORMAL	1341	234
PNEUMONIA	3875	390
TOTAL	5216	624

Tabela 3: Base de dados.

Todas as imagens foram redimensionadas para o tamanho 64x64x1.

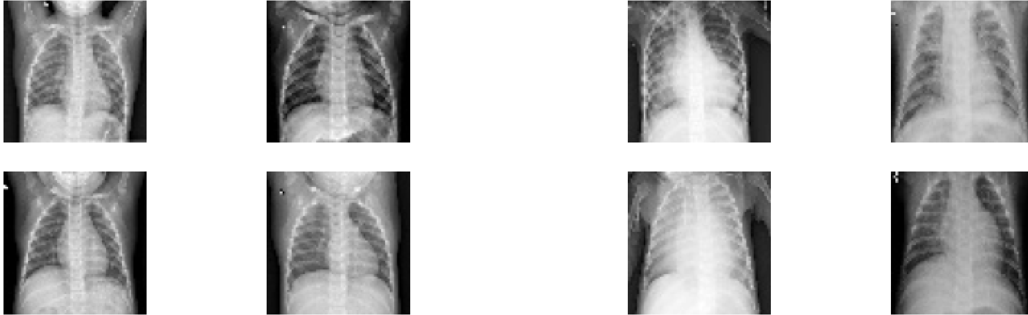

Figura 39: Raio-x NORMAL

Figura 40: Raio-x PNEUMONIA

Fonte: Mendeley Data[17]

É possível observar que as imagens da classe PNEUMONIA apresentam uma área mais branca na região do pulmão, o que não é possível observar na classe NORMAL. Isso é uma característica fundamental a ser detectada pelo discriminador e gerador durante o treino.

6.4 Resultados

Foram treinadas duas GANs, uma para cada classe (NORMAL e PNEUMONIA). Foram utilizados 10 batchs para a classe NORMAL e 30 batchs para a classe PNEUMONIA. Ambos foram treinadas em 200 épocas.

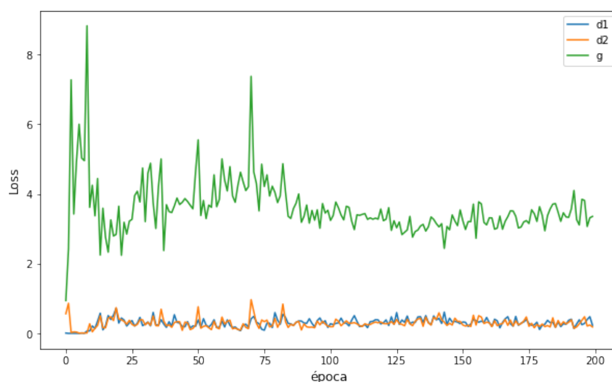

Figura 41: Loss do Discriminador e Gerador para a classe NORMAL.

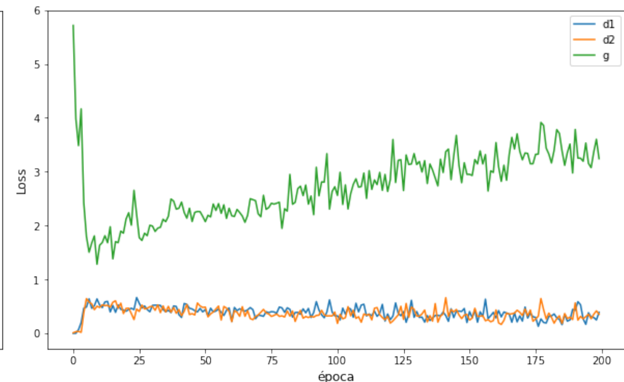

Figura 42: Loss do Discriminador e Gerador para a classe PNEUMONIA.

No gráfico: g será o Loss do gerador; $d1$ o Loss do discriminador em imagens falsas; $d2$ o Loss do discriminador em imagens reais. O gráfico do Loss não nos dá informações tão precisas para dizer se o gerador conseguiu aprender com eficiência a gerar a distribuição real, apenas nos diz que o treino prosseguiu de forma estável. Por isso é importante olhar como as imagens estão sendo geradas, tanto em relação à fidelidade como em diversidade, além de analisar a distância FID entre as distribuições reais e geradas. Olhando diretamente numa amostra de imagens geradas, em épocas distintas, é possível observar que houve uma evolução no aprendizado do gerador:

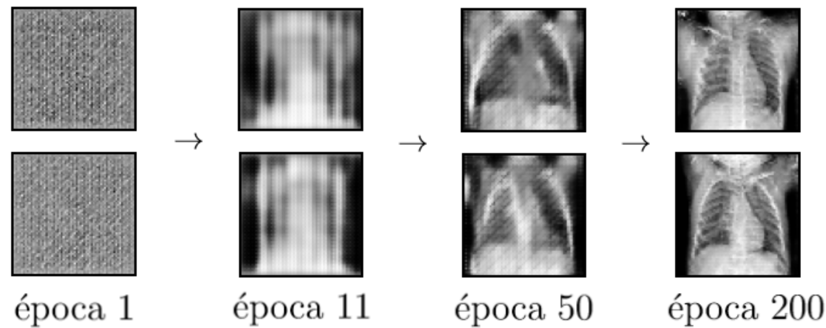

Figura 43: Resultados da DCGAN em diferentes épocas para a classe NORMAL.

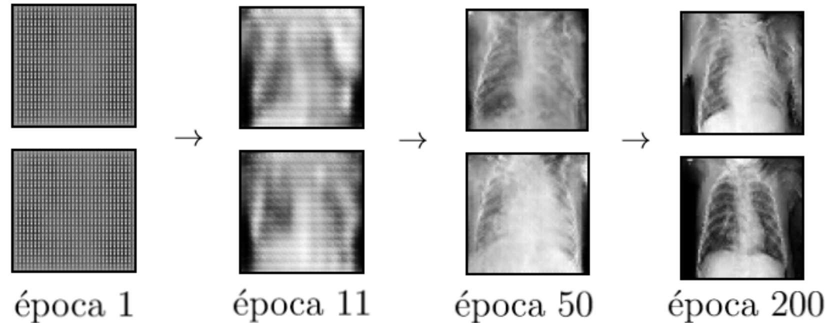

Figura 44: Resultados da DCGAN em diferentes épocas para a classe PNEUMONIA.

6.4.1 Distância FID

Para o cáculo do FID foram utilizadas amostras da base de teste. Foram feitas 100 iterações comparando 100 imagens reais e 100 imagens geradas pela DCGAN, em que, em cada interação, amostras aleatórias foram selecionadas da base real e novas imagens falsas foram geradas. A nomenclatura AxB denota a comparação entre o conjunto A e B. As abreviações são definidas como:

RN: imagens Reais NORMAL

RP: imagens Reais PNEUMONIA

FN: imagens Falsas NORMAL

FP: imagens Falsas PNEUMONIA

Os 4 conjuntos foram comparados com o propósito de observar a semelhança entre as distribuições. É de se esperar que conjuntos comparados entre si mesmo apresentem um valor próximo de zero, por isso foram excluídos da análise.

A expectativa é de que conjuntos de mesma classe apresentem distâncias mais baixas comparado a conjuntos de classes diferentes. Imagens NORMAL reais e falsas (RNxFN) devem apresentar distância consideravelmente mais baixas comparado a um par de conjuntos de imagens NORMAL e PNEUMONIA, assim como imagens PNEUMONIA real e falsas (RPxFP) também devem apresentar distâncias baixas. A seguir é apresentado o boxplot das 100 distâncias calculadas em cada iteração, separadas por par de conjuntos. Em **azul** estão pares de mesma classe e em **vermelho** pares de classes diferentes:

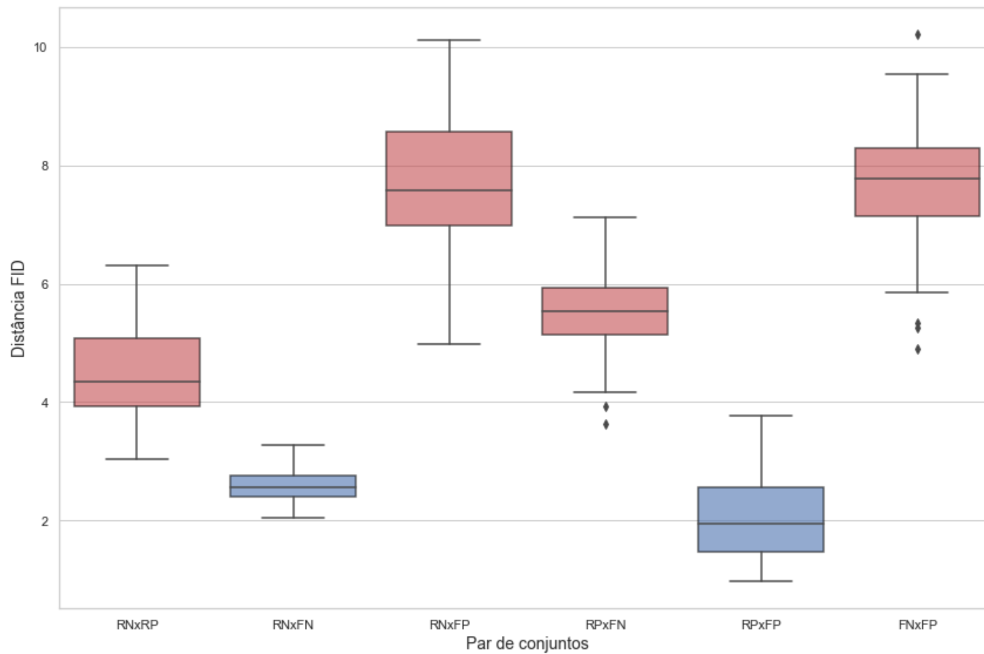

Figura 45: Boxplot das distâncias FID para cada par de conjuntos.

A seguir a tabela descritiva da métrica FID:

	mean	std	min	25%	50%	75%	max
RNxRP	4.467217	0.745554	3.045637	3.935836	4.339930	5.070940	6.319213
RNxFN	2.595408	0.254912	2.039646	2.411774	2.567080	2.756945	3.272817
RNxFP	7.697779	1.130095	4.978535	6.984023	7.582666	8.565713	10.119095
RPxFN	5.486598	0.623257	3.634400	5.141299	5.530263	5.933036	7.116484
RPxFP	2.038317	0.700336	0.973648	1.461451	1.942160	2.568950	3.763087
FNxFP	7.698875	0.946177	4.893967	7.150980	7.777950	8.291801	10.217985

É possível observar que a hipótese de distâncias mais baixas para pares de mesma classe se mostrou verdadeira. Os pares de maior interesse (RNxFN) e (RPxFP) apresentam média e mediana abaixo de 2.6. É provável que, com um tempo maior de treino, o modelo apresentaria melhores métricas, dado que o gerador parece ter convergido com sucesso para a distribuição da base Real em apenas 200 épocas. Pode-se concluir, por tanto, que a rede DCGAN proposta foi capaz de gerar imagens muito próximas à distribuição da amostra real.

7 Conclusão

Com poucos recursos e uma arquitetura simples foi possível gerar imagens diversificadas e fidedignas em relação às imagens reais. Como demonstrado utilizando a métrica FID, a distribuição p_g convergiu para a distribuição p_{data}. Apesar disso, ainda é difícil concluir o grau de precisão dessa aproximação. Um dos maiores desafios em relação a falta de informação foi saber se a rede realmente estava convergindo durante a etapa de treino, sendo necessário testar empiricamente cada configuração do modelo. Entretanto, o experimento demonstrou o poder das GANs como um gerador de imagens, sendo um substituto ideal para os modelos antigos, mais caros computacionalmente e de difícil convergência.

Na área médica já é possível ver trabalhos desenvolvidos e em desenvolvimento que utilizam as GANs como ferramentas fundamentais para popular a base de dados com mais informações e criar modelos de predição mais assertivos, evitando casos de overfitting por desbalanceamento e falta de dados. Podemos citar trabalhos como a geração de imagens de células embrionárias[7] até o redimensionamento (reshape) para alta resolução de imagens de ressonância magnética do cérebro[5]. Fora da área médica também há uma grande aplicação na computação gráfica, tanto transformando e adicionando novas características a imagens e vídeos, como modificando a estrutura dos dados (qualidade da imagem, dimensão, quadros por segundo, etc). Novas técnicas, como a WGAN e StyleGAN, complementam as arquiteturas DCGAN, possuindo um grau de complexidade maior e gerando imagens com uma maior qualidade e com maior precisão de convergência. É esperado que as arquiteturas GAN se popularizem cada vez mais com o passar do tempo e novas ferramentas possam ser criadas com seu auxílio.

Referências

[1] Bishop, C. M. Neural networks and their applications. *Review of Scientific Instruments 65*, 6 (1994), 1803–1832.

[2] Borji, A. Pros and cons of gan evaluation measures, 2018.

[3] Brownlee, J. *Generative Adversarial Networks with Python: Deep Learning Generative Models for Image Synthesis and Image Translation.* Machine Learning Mastery, 2019.

[4] Carpenter, K., Cohen, D., Jarrell, J., and Huang, X. Deep learning and virtual drug screening. *Future Medicinal Chemistry 10* (10 2018).

[5] Chen, Y., Xie, Y., Zhou, Z., Shi, F., Christodoulou, A. G., and Li, D. Brain MRI super resolution using 3d deep densely connected neural networks. *CoRR abs/1801.02728* (2018).

[6] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition* (2009), Ieee, pp. 248–255.

[7] Dirvanauskas, D., Maskeliūnas, R., Raudonis, V., Damaševičius, R., and Scherer, R. Sensors (Basel)HEMIGEN: Human Embryo Image Generator Based on Generative Adversarial Networks. *Sensors (Basel) 19*, 16 (Aug 2019).

[8] Dumagpi, J. K., and Jeong, Y.-J. Evaluating gan-based image augmentation for threat detection in large-scale xray security images. *Applied Sciences 11*, 1 (2021).

[9] Frid-Adar, M., Diamant, I., Klang, E., Amitai, M., Goldberger, J., and Greenspan, H. Gan-based synthetic medical image augmentation for increased CNN performance in liver lesion classification. *CoRR abs/1803.01229* (2018).

[10] Glorot, X., and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics* (2010).

[11] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial networks. *Advances in Neural Information Processing Systems 3* (06 2014).

[12] Goodfellow, I. J., Bengio, Y., and Courville, A. *Deep Learning.* MIT Press, Cambridge, MA, USA, 2016. http://www.deeplearningbook.org.

[13] Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., and Hochreiter, S. Gans trained by a two time-scale update rule converge to a local nash equilibrium, 2018.

[14] Hubel, D., and Wiesel, T. Receptive fields, binocular interaction, and functional architecture in the cat's visual cortex. *Journal of Physiology 160* (1962), 106–154.

[15] Ioffe, S., and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[16] Izbicki, R., and dos Santos, T. M. *Aprendizado de máquina: uma abordagem estatística.* 2020.

[17] Kermany, Daniel; Zhang, K. G. M. Large dataset of labeled optical coherence tomography (oct) and chest x-ray images, 2018.

[18] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization, 2017.

[19] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems* (2012), F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25, Curran Associates, Inc.

[20] LeCun, Y., Bengio, Y., and Hinton, G. Deep learning. *Nature 521*, 7553 (2015), 436–444.

[21] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. 306–351.

[22] LeCun, Y., and Cortes, C. MNIST handwritten digit database.

[23] Mahapatra, D., and Bozorgtabar, B. Retinal vasculature segmentation using local saliency maps and generative adversarial networks for image super resolution. *CoRR abs/1710.04783* (2017).

[24] Mcculloch, W., and Pitts, W. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics 5* (1943), 127–147.

[25] Mehlig, B. Machine learning with neural networks, 2021.

[26] Nielse, M. *Neural Networks and Deep Learning.* 2019.

[27] Radford, A., Metz, L., and Chintala, S. Unsupervised representation learning with deep convolutional generative adversarial networks, 2016.

[28] Roffo, G. Ranking to learn and learning to rank: On the role of ranking in pattern recognition applications.

[29] Rosenblatt, F. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review 65 6* (1958), 386–408.

[30] Rumelhart, D., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *Nature 323* (1986), 533–536.

[31] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research 15*, 56 (2014), 1929–1958.

[32] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions, 2014.

[33] Tieleman, T., and Hinton, G. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[34] Zhang, A., Lipton, Z. C., Li, M., and Smola, A. J. *Dive into Deep Learning.* 2020. https://d2l.ai.