

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA CIVIL E
AMBIENTAL

DETECÇÃO DE FISSURAS EM CONCRETO USANDO
DEEP LEARNING

TÚLIO DE ARAÚJO VIEIRA

ORIENTADOR: LENILDO SANTOS DA SILVA

MONOGRAFIA DE PROJETO FINAL 2 EM ENGENHARIA
CIVIL

BRASÍLIA/DF: DEZEMBRO/2020

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA

DETECÇÃO DE FISSURAS EM CONCRETO USANDO DEEP LEARNING

Resumo

A detecção de fissuras por algoritmos de computador é algo muito desejado, pois é o primeiro passo para possibilitar a extração de informações chave sobre patologias e manutenibilidade de edificações de maneira automática. Tradicionalmente, essa detecção tem sido feita com técnicas de processamento de imagens, utilizando os operadores de Sobel e Canny.

Contudo, o advento algoritmos de *Deep Learning* para visão computacional e seus bons resultados em diversas aplicações trazem grandes promessas para a detecção automática de patologias de construções. A grande vantagem desses algoritmos é o aprendizado automático das regiões danificadas com base no conjunto de dados fornecido.

Objetiva-se, primeiramente, apresentar uma revisão bibliográfica, explicando o que são redes neurais, seus tipos, e como funcionam. Em seguida, treina-se uma rede neural convolucional de classificação para a detecção de fissuras em concreto, usando um banco de dados *open-source*. Obteve-se boas métricas de treino, validação e teste, apesar de a boa performance estar limitada a situações semelhantes às retratadas pelos dados usados.

Por fim, desenvolve-se um aplicativo *web* para a aplicação do modelo no navegador. Os resultados mostram o grande potencial desse novo paradigma, que pode ser expandido, no futuro, para a detecção de outras patologias em construções.

Abstract

The detection of cracks by computer algorithms is something very desired, as it is the first step to enable the extraction of key insights about pathologies and maintenance of buildings in an automatic way. Traditionally, this detection has been done with image processing techniques, using the Sobel and Canny operators.

However, the advent of Deep Learning algorithms for computer vision and their good results in several applications bring great promises for the automatic detection of construction pathologies. The great advantage of these algorithms is the automatic learning of the damaged regions based on the dataset provided.

Firstly, this paper aims to present a bibliographic review, explaining what neural networks are, their types, and how they work. Then, a classification convolutional neural network is trained for the detection of cracks in concrete, using an open-source dataset. Good training, validation and test metrics were obtained, although good performance was limited to situations similar to those portrayed by the data used.

Finally, a web application for the application of the model in the browser is developed. The results show the great potential of this new paradigm, which can be expanded, in the future, for the detection of other pathologies in constructions.

Palavras-Chave: neural; fissura; deep; machine; learning; concreto.

Key-words: neural; crack; deep; machine; learning; concrete.

**DEPARTAMENTO DE ENGENHARIA CIVIL E
AMBIENTAL**

**DETECÇÃO DE FISSURAS EM CONCRETO USANDO
DEEP LEARNING**

TÚLIO DE ARAÚJO VIEIRA

**MONOGRAFIA DE PROJETO FINAL SUBMETIDO AO DEPARTAMENTO DE ENGENHARIA
CIVIL E AMBIENTAL DA UNIVERSIDADE DE BRASÍLIA COMO PARTE DOS REQUISITOS
NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM ENGENHARIA CIVIL.**

APROVADA POR:

**Lenildo Santos da Silva, PhD (ENC-FT-UnB)
(ORIENTADOR)**

**Evangelos Dimitrios Christakou, Dr. (ENC-FT-UnB)
(EXAMINADOR INTERNO)**

**Márcio Augusto Roma Buzar, Dr. (FAU-UnB)
(EXAMINADOR EXTERNO)**

**Marco Aurélio Souza Bessa, Dr. (UniCEUB)
(EXAMINADOR EXTERNO)**

DATA: BRASÍLIA/DF, ____ de dezembro de 2020

HORÁRIO: _____

SUMÁRIO

1.	INTRODUÇÃO	7
2.	OBJETIVOS	7
2.1.	OBJETIVO GERAL	7
2.2.	OBJETIVOS ESPECÍFICOS	8
3.	REVISÃO BIBLIOGRÁFICA	8
3.1.	REDES NEURAIS CONVENCIONAIS.....	8
3.1.1.	Perceptron.....	9
3.1.2.	Perceptron Multi-Camada (MLP).....	15
3.2.	REDES CONVOLUCIONAIS	20
3.2.1.	Camada Convolucional	21
3.2.2.	Camada de Subamostragem (<i>Pooling</i>)	26
3.2.3.	Camada Densa	27
3.3.	TREINANDO REDES NEURAIS	28
3.3.1.	<i>Underfitting</i> e <i>Overfitting</i>	29
3.3.2.	Dados	29
4.	METODOLOGIA	31
4.1.	FERRAMENTAS	32
4.2.	BANCO DE DADOS	32
4.3.	ARQUITETURA DO MODELO	33
4.4.	PRÉ-PROCESSAMENTO	35
4.5.	PARÂMETROS DE TREINAMENTO	36
5.	RESULTADOS.....	36
6.	DESENVOLVIMENTO DE APLICATIVO.....	38
7.	LIMITAÇÕES	39
8.	CONCLUSÃO	43
9.	SUGESTÕES PARA TRABALHOS FUTUROS	44
10.	ANEXOS.....	45
10.1.	Código para divisão dos conjuntos de treino, validação e teste	45
10.2.	Código para pré-processamento de dados	47
10.3.	Código para definição de arquitetura do modelo	48
10.4.	Código para treinamento do modelo	49
11.	REFERÊNCIAS BIBLIOGRÁFICAS	50

LISTA DE TABELAS

Tabela 1: Algoritmo de aprendizado do perceptron	13
Tabela 2: Dimensões das camadas da rede neural.....	34
Tabela 3: Hiperparâmetros de aprendizado	36

LISTA DE FIGURAS

Figura 1 – Comparação entre neurônio e perceptron: (a) Modelo de Neurônio Humano; (b) Modelo Matemático de um Perceptron	9
Figura 2 – Gráfico de estudantes aceitos (azul) e rejeitados (vermelho).....	10
Figura 3 – Arquitetura do perceptron	10
Figura 4 – Reta de classificação com parâmetros aleatórios	11
Figura 5 – Gráfico da função <i>softmax</i> (Pasquier, 2018).....	13
Figura 6 – Evolução da posição da linha de classificação em cada época (<i>epoch</i>).....	14
Figura 7 – Evolução do erro em cada época (<i>epoch</i>)	14
Figura 8 – Perceptron Multi-Camada	15
Figura 9 – Esquema de uma rede neural.....	15
Figura 10 – Esquema simplificado de rede neural	16
Figura 11 – Esquema de uma rede neural de múltiplas classificações	16
Figura 12 – Esquema do algoritmo de classificação (<i>feedforward</i>)	17
Figura 13 – Esquema do algoritmo de aprendizado (<i>backpropagation</i>)	18
Figura 14 – Classificação com múltiplas classes sem <i>softmax</i>	19
Figura 15 – Classificação com múltiplas classes usando <i>softmax</i>	20
Figura 16 – Organização de camadas em uma rede convolucional (arquitetura da rede VGG-16).....	21
Figura 17 – Representação de uma imagem como um tensor de três dimensões.....	21
Figura 18 – Janela de convolução	22
Figura 19 – Passo igual a 1: o filtro move um pixel para cada conexão	22
Figura 20 – Funcionamento do filtro	23
Figura 21 – Conexão local com compartilhamento de parâmetros	25
Figura 22 – Detecção de padrões usando filtros de convolução.....	25
Figura 23 – Resultado da aplicação de diferentes filtros na convolução.....	26
Figura 24 – <i>Max pooling</i>	27
Figura 25 – Esquema de uma rede convolucional.....	28
Figura 26 – Complexidade de informações detectadas ao longo da rede.....	28
Figura 27 – <i>Underfitting</i> e <i>Overfitting</i>	29
Figura 28 – Detecção de fissura usando rede neural convolucional.....	31
Figura 29 – Metodologia do trabalho	32
Figura 30 – Amostras do banco de dados de Özgenel (2018)	33
Figura 31 – Estrutura do banco de dados	33
Figura 32 – Arquitetura da rede convolucional	34
Figura 33 – Fluxograma do pré-processamento de imagens	35
Figura 34 – Evolução da acurácia de treino e validação	36
Figura 35 – Predições feitas em amostra do conjunto de teste	37
Figura 36 – Operação da janela deslizante*	38
Figura 37 – Demonstração do aplicativo desenvolvido: configurações	38
Figura 38 – Demonstração do aplicativo desenvolvido: resultado.....	39
Figura 39 – Detecção de fissura em condições ideais	40
Figura 40 – Detecção de fissura em superfície muito manchada	41
Figura 41 – Falsos positivos devido a manchas	41
Figura 42 – Detecção de fissura em foto com baixo contraste entre fissura e superfície....	42
Figura 43 – Detecção de fissura em foto com baixa resolução	42
Figura 44 – Detecção com tamanhos de janela diferentes.....	43

1. INTRODUÇÃO

Inteligência Artificial (IA) é um campo que vem mudando drasticamente não só diversas áreas do conhecimento, mas também traz grandes expectativas em relação ao futuro das profissões. Enquanto existem projeções de grande crescimento de demanda por cientistas de dados, discute-se também a possível ameaça a trabalhos com mão-de-obra não qualificada, em que a IA pode oferecer uma alternativa de baixo custo.

Aprendizado Profundo (*Deep Learning*) é um subconjunto de Aprendizado de Máquina (*Machine Learning*), que é um campo dedicado ao estudo e desenvolvimento de máquinas que aprendem (Trask, 2019), a qual pode ser vista como uma etapa da IA. Também chamado de Rede Neural Profunda (*Deep Neural Network*), refere-se a Redes Neurais Artificiais (RNA) com várias camadas. Nas últimas décadas, foi considerado uma das ferramentas mais poderosas e se tornou muito popular na literatura, pois é capaz de lidar com uma grande quantidade de dados. O interesse em ter camadas ocultas mais profundas começou recentemente a superar o desempenho dos métodos clássicos em diferentes campos, especialmente no reconhecimento de padrões (Trask, 2019).

Redes Neurais são usadas em muitas áreas, como algoritmos de busca em sites de pesquisa, algoritmos de recomendações de conteúdo, carros autônomos, reconhecimento de fala (áudio), reconhecimento de linguagem natural (texto) e visão computacional (imagens). O uso em reconhecimento de imagens é feito principalmente com Redes Neurais Convolucionais.

Assim, o presente trabalho pretende aplicar Redes Neurais Convolucionais para detecção de fissuras em estruturas em concreto por meio do processamento de imagens, sobretudo as obtidas com drones.

A detecção de fissuras por inspeção visual pode ser um processo muito trabalhoso, dependendo do número de fissuras e da dificuldade de acesso, além de depender bastante da subjetividade do observador. Assim, diversos métodos foram propostos para automatizar este processo, que consistem em técnicas de processamento de imagens. Porém, a implementação dessas técnicas é difícil quando existem condições adversas, como mudanças de iluminação e texturas diferentes (Cha *et al.*, 2017).

É nesse sentido que a utilização de redes neurais traz a expectativa de ser um método adequado em relação à estabilidade na detecção, mesmo considerando-se variações nas condições de aquisição das imagens, como iluminação, ângulo de aquisição, textura, dimensão das aberturas, entre outros. Esta expectativa se deve sobretudo à capacidade de aprender automaticamente as características relevantes para a detecção de fissuras, considerando que existam condições adversas nos dados de aprendizado.

2. OBJETIVOS

2.1. OBJETIVO GERAL

Implementar uma Rede Neural Convolucional capaz de detectar fissuras em peças estruturais em concreto a partir de um banco de dados *open-source*. Utilizar a arquitetura de Cha *et al.* (2017), tendo como ambiente de desenvolvimento o *Jupyter Notebook*, usando o pacote *keras* e linguagem *python*.

2.2. OBJETIVOS ESPECÍFICOS

- Implementar, treinar e ajustar uma rede neural para detecção de fissuras;
- Utilizar processamento de redes neurais convolucionais na GPU do computador;
- Utilizar processamento de redes neurais convolucionais em nuvem;
- Analisar resultados de treino, validação e teste;
- Aplicar o modelo em produção e definir os seus limites.

3. REVISÃO BIBLIOGRÁFICA

Na pesquisa feita sobre trabalhos relacionados a detecção de fissuras encontraram-se diversos artigos, que abordaram o tema com diferentes metodologias.

Trabalhos mais antigos trouxeram propostas de detecção por meio de técnicas de processamento de imagens. Dentre eles, pode-se citar o trabalho de Pereira (2015) que propôs a utilização dos operadores de Sobel e Canny para detecção de fissuras, e o trabalho de Melo Júnior (2016), que realizou mais experimentações com essa técnica. Porém, cada imagem de fissura requer a otimização de certos parâmetros de detecção, além de que muitos ruídos podem ser detectados dependendo das condições da imagem, fazendo-se necessário a utilização de métodos de remoção de ruído nessas ocasiões.

Os trabalhos mais recentes, por sua vez, trazem a proposta de detecção de fissuras por meio de redes neurais convolucionais, que são a arquitetura mais adequada para visão computacional. A utilização dessas redes traz a vantagem de que as características relevantes para a detecção de fissuras podem ser aprendidas automaticamente, ainda que existam condições adversas nos dados de aprendizado. Dentre esses trabalhos, pode-se citar o trabalho de Cha *et al.* (2017), que buscou a detecção usando redes convolucionais com classificação.

Porém as redes convolucionais são capazes de gerar não só outputs de classificação de imagens (se a imagem é de um gato ou cachorro, por exemplo), mas também de informar localização e detecção de objetos (retângulo onde se encontra o gato na imagem) e segmentação semântica (pixels que pertencem ao gato). Nesses dois últimos tipos de classificação, são usadas redes totalmente convolucionais, ou seja, sem camadas densas.

Assim, existem dezenas de trabalhos recentes (entre 2018 e 2019) que propõem a detecção de fissuras usando redes totalmente convolucionais com segmentação semântica, dentre os quais pode-se citar Dung (2018), Zhang *et al.* (2016), Zhang *et al.* (2017) e Zhang *et al.* (2019). Essa técnica recente tem representado grande avanço no campo de detecção de patologias em Concreto armado, asfalto, entre outros, pois é capaz de definir uma região de contorno da fissura irregular.

3.1. REDES NEURAI CONVENCIONAIS

As Redes Neurais Convencionais são formadas por múltiplos perceptrons organizados em múltiplas camadas, formando um modelo computacional capaz de reconhecer padrões com base nos dados de aprendizado.

Cada ponto de dado possui informações de input, que são números medidos para a resolução de um problema. Esses números podem ser a temperatura do dia, o preço de uma ação no dia de ontem, ou o índice de acerto de um jogador de baseball (Trask, 2019).

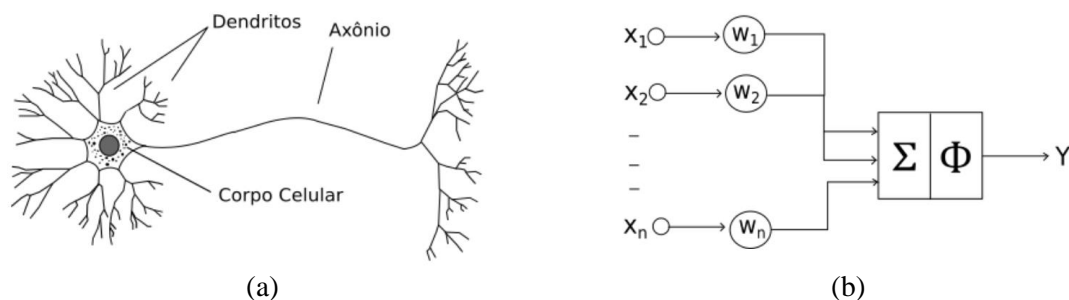
Com base nessas informações, a rede neural tenta fazer uma predição, como “dado a temperatura de hoje, há uma probabilidade de 0% de as pessoas usarem roupas de banho”, ou “dado o índice de acerto de um jogador de baseball, ele tem a probabilidade de 30% de fazer um *home run*”, ou “dado o preço da ação de ontem, o preço de hoje é 101,52” (Trask, 2019).

Para cada predição, a rede neural faz uma comparação com o resultado real, que constitui o rótulo ou “gabarito” de cada conjunto de informações de input. Com base nessa comparação, a rede neural se adapta para realizar previsões cada vez mais acuradas.

3.1.1. Perceptron

O perceptron é a unidade mais simples de uma rede neural. Se compararmos uma rede neural ao cérebro humano, os perceptrons seriam os neurônios. Ambos os elementos funcionam da mesma forma: eles recebem uma série de inputs e produzem um único output (Figura 1).

Figura 1 – Comparação entre neurônio e perceptron: (a) Modelo de Neurônio Humano; (b) Modelo Matemático de um Perceptron

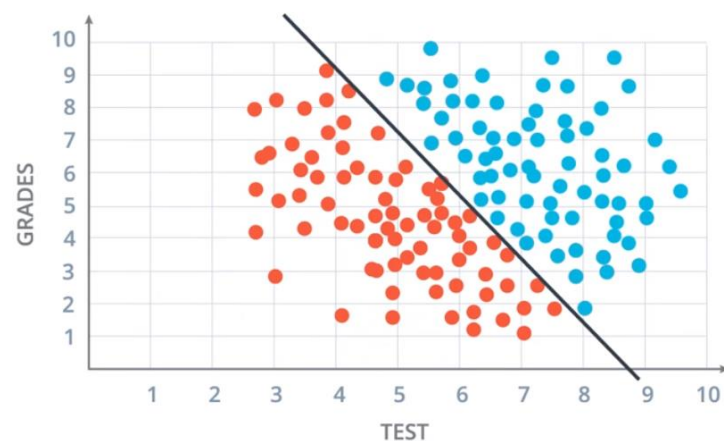


Fonte: Maia (2019)

As redes neurais, não importa o quão complexa seja a tarefa que realizem, sempre resolvem problemas de classificação. Para ilustrar o funcionamento do perceptron, será apresentado um simples problema de classificação:

Imagine que em uma universidade a aceitação do aluno é resultado de uma relação entre as notas do curso (*grades*) e a nota do exame de admissão (*test*). Todos os estudantes são classificados em duas classes (aceito, em azul, ou rejeitado, em vermelho), e os resultados são plotados em um gráfico 2d (Figura 2).

Figura 2 – Gráfico de estudantes aceitos (azul) e rejeitados (vermelho)



Fonte: Udacity (2017)

O gráfico acima, portanto, apresenta um conjunto de dados de estudantes que foram aprovados ou rejeitados com base nas suas notas do curso e do teste de admissão. A resolução do problema consiste em encontrar a linha que separa os estudantes entre os aceitos e rejeitados. Essa linha constituirá um modelo capaz de prever a aceitação de novos estudantes no futuro.

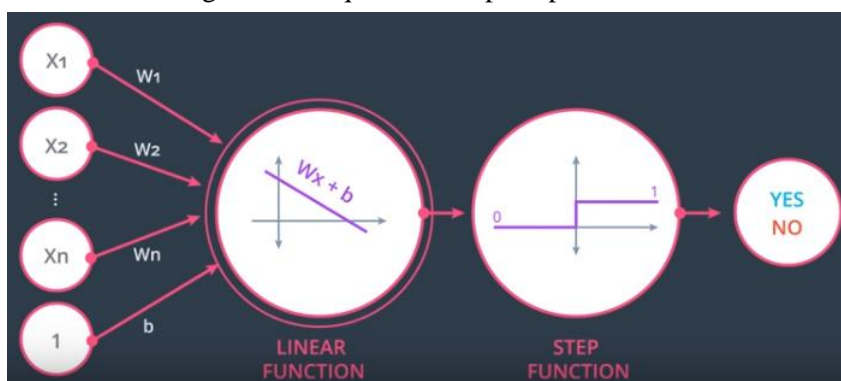
3.1.1.1. Algoritmo do perceptron

Dado uma série de inputs (x_1, x_2, \dots, x_n), são atribuídos uma série de pesos (w_1, w_2, \dots, w_n) e um termo independente (b). O output, ou previsão (\hat{y}) do perceptron é calculado da seguinte forma:

$$z = \sum_{i=1}^n w_i x_i + b \quad (1)$$

$$\hat{y} = f(z) \quad (2)$$

Figura 3 – Arquitetura do perceptron



Fonte: Udacity (2017)

Na Figura 3, a função passo (*step function*) serve para controlar o valor do output do perceptron. Essa função, chamada de função de ativação, é definida da seguinte forma:

$$f(z) = \begin{cases} 1, & \text{se } z > 0 \\ 0, & \text{se } z \leq 0 \end{cases} \quad (3)$$

Dessa forma, outputs positivos classificam o estudante com o valor 1, que significa aprovação. Outputs menores ou igual a zero classificam o estudante com o valor 0, que significa rejeição. A função passo não é a única função de ativação usada nos perceptrons, e mais tarde serão apresentadas outras funções de ativação.

Voltando ao exemplo, tem-se que o perceptron será responsável por traçar uma reta do tipo:

$$w_1x_1 + w_2x_2 + b = 0 \quad (4)$$

Em que:

- x_1 e x_2 são as variáveis nota e teste, respectivamente;
- w_1 e w_2 são os pesos;
- b é o termo independente (*bias*).

Essa equação pode ser reescrita na forma vetorial:

$$W \cdot X + b = 0 \quad (5)$$

Em que:

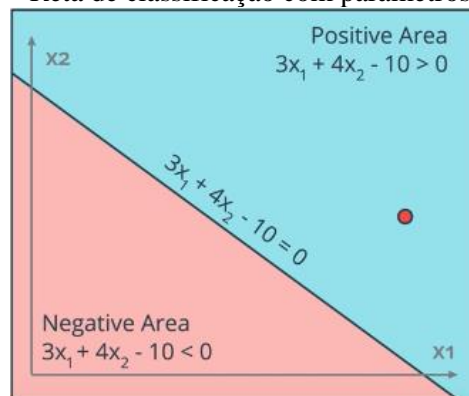
- $X = [x_1 \ x_2]$;
- $W = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$.

A classificação fica da seguinte forma:

- Ponto azul (aluno aceito): $WX + b \geq 0$;
- Ponto vermelho (aluno rejeitado): $WX + b < 0$.

Atribuindo valores aleatórios aos parâmetros, digamos $w_1 = 3$, $w_2 = 4$ e $b = -10$, resulta na seguinte reta (Figura 4):

Figura 4 – Reta de classificação com parâmetros aleatórios



Fonte: Udacity (2017)

Conforme pode-se notar, os parâmetros usados resultaram em uma classificação errada para um dos pontos, pois tem-se um ponto vermelho (aluno rejeitado) na região azul definida pela linha.

O objetivo, portanto, é mudar a posição da linha de forma que a classificação se torne correta. Para isso, usa-se algum algoritmo de aprendizado para a rede neural, que irá atualizar os pesos para fornecerem classificações cada vez melhores. Este algoritmo é chamado de retropropagação (*backpropagation*).

3.1.1.2. Gradiente de descida (*gradient descent*)

Nesta seção será apresentado um algoritmo de aprendizado da rede neural. Com base no erro obtido na saída da rede, atualizam-se os parâmetros (pesos e *bias*), de forma a melhorar a previsão da rede.

O erro E pode ser calculado de várias maneiras diferentes. Dentre elas, a mais intuitiva é a diferença entre a previsão (\hat{y}) e o rótulo (y). Para evitar valores negativos e simplificar as contas, usa-se a metade da diferença ao quadrado:

$$E = \frac{1}{2}(y_i - \hat{y}_i)^2 \quad (6)$$

Dessa forma, o objetivo do aprendizado da rede é minimizar o valor da função de erro. Isso é obtido com o gradiente da função de erro em relação aos pesos, que são os parâmetros a serem atualizados. O gradiente fornece a direção de maior crescimento da função E : tomando o sinal contrário do gradiente, tem-se a direção de maior decréscimo da função, o que garante que os pesos sejam atualizados de forma a gerarem melhores previsões. Por isso, esse gradiente é chamado de gradiente de descida (*gradient descent*). Assim, tem-se a seguinte fórmula para atualização dos pesos:

$$w_i \leftarrow w_i - \alpha \frac{\partial E}{\partial w_i} \quad (7)$$

O parâmetro α é o passo tomado na correção da rede neural, chamado de taxa de aprendizado (*learning rate*). Tipicamente, adotam-se valores de α entre 0,01 e 0,1, e seu efeito no aprendizado será discutido mais adiante. Se aplicarmos as equações ao perceptron, temos:

$$\begin{aligned} \frac{\partial E}{\partial w_i} &= \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_i} = -(y - \hat{y}) \cdot f'(z) \cdot x_i \\ \frac{\partial E}{\partial b} &= \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b} = -(y - \hat{y}) \cdot f'(z) \\ \delta &= (y - \hat{y}) \cdot f'(z) \\ w_i &\leftarrow w_i + \alpha \cdot \delta \cdot x_i \\ b &\leftarrow b + \alpha \cdot \delta \end{aligned} \quad (8)$$

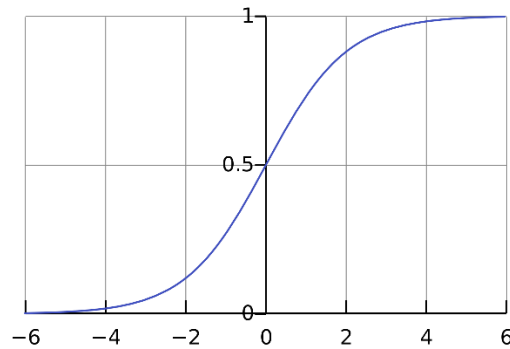
Nas equações acima, δ é chamado de termo de erro (*error term*).

Assim, tem-se uma fórmula para a atualização dos pesos, mas resta um problema: a atualização dos pesos depende da derivada da função de ativação do perceptron ($f'(z)$). Porém, a derivada da função passo é sempre zero, o que inviabiliza a utilização dessa função no algoritmo. Faz-se necessário a utilização de uma outra função de ativação, que seja contínua e diferenciável.

Uma função tipicamente usada neste tipo de rede é a função sigmoide ($\sigma(z)$), definida por:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad , \quad \sigma'(z) = \sigma(z) \cdot (1 - \sigma(z)) \quad (9)$$

Figura 5 – Gráfico da função *softmax* (Pasquier, 2018)



Conforme pode-se notar, a função sigmoide é diferenciável em todo o seu domínio, tem uma derivada simples e resulta em valores entre 0 e 1 (

Figura 5). A classificação, portanto, fica da seguinte forma:

- $\sigma(z) > 0,5 \rightarrow$ Aluno aprovado
- $\sigma(z) \leq 0,5 \rightarrow$ Aluno rejeitado

Essa classificação é chamada de entropia cruzada binária (*binary cross-entropy*), que demonstra uma grande vantagem: a partir de agora as classificações, mesmo que estejam certas, geram um valor de erro. Isso garante que todos os pontos gerem um termo de erro e, conseqüentemente, uma atualização para os pesos, o que torna o algoritmo de aprendizado muito mais eficiente.

Assim, o algoritmo de aprendizado do perceptron fica da seguinte forma (Tabela 1):

Tabela 1: Algoritmo de aprendizado do perceptron
Inicializar com pesos aleatórios: w_1, w_2, b
Repetir e vezes:
Para todos os pontos (x_1, x_2) , cada um com um rótulo e previsão (y e \hat{y}):
$z = w_1x_1 + w_2x_2 + b$
$\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z))$
$\delta = (y - \hat{y}) \cdot \sigma'(z)$
$w_1 \leftarrow w_1 + \alpha \cdot \delta \cdot x_1$
$w_2 \leftarrow w_2 + \alpha \cdot \delta \cdot x_2$
$b \leftarrow b + \alpha \cdot \delta$

O resultado da aplicação deste algoritmo é apresentado nas figuras 6 e 7, a seguir. Pode-se notar que a linha de classificação começa em uma posição aleatória e, a cada iteração (época) do algoritmo, move-se mais perto da posição ótima. Ao mesmo tempo, o valor da função de erro diminui até chegar em um valor mínimo.

Figura 6 – Evolução da posição da linha de classificação em cada época (*epoch*)

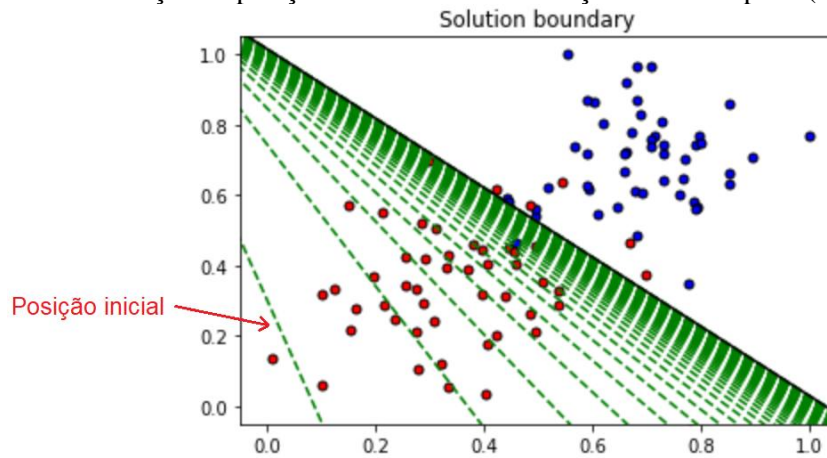
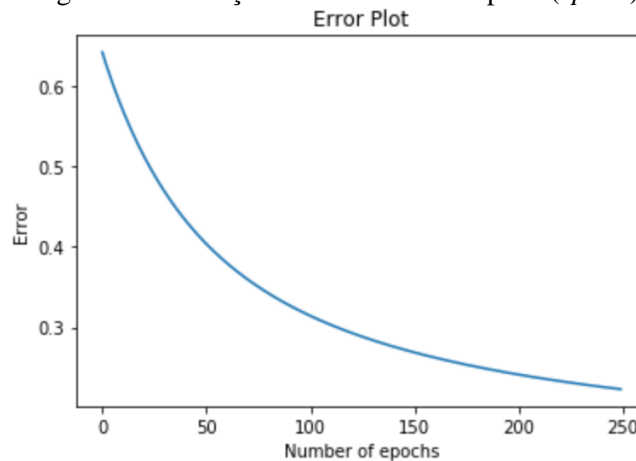


Figura 7 – Evolução do erro em cada época (*epoch*)



Por fim, é importante discutir o efeito da taxa de aprendizado α no algoritmo (ver conjunto de equações (8)).

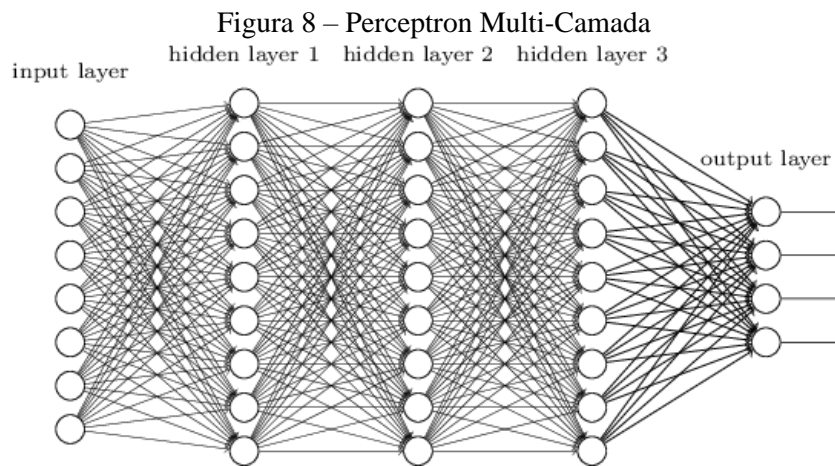
Em linhas gerais, uma taxa de aprendizado elevada permite que o modelo aprenda mais rapidamente, com o custo de chegar a um conjunto final de pesos abaixo do ideal. Uma taxa de aprendizado menor pode permitir que o modelo aprenda um conjunto de pesos mais ideal ou até globalmente ideal, mas pode levar um tempo significativamente maior para treinar (Brownlee, 2019a).

Em casos extremos, uma taxa de aprendizado muito grande resultará em atualizações de peso muito grandes e o desempenho do modelo (como sua perda no conjunto de dados de treinamento) oscilará nas épocas do treinamento. Diz-se que o desempenho oscilante é causado por pesos que divergem (são divergentes). Uma taxa de aprendizado muito pequena pode nunca convergir ou ficar presa em uma solução abaixo do ideal (Brownlee, 2019a).

Conforme mencionado anteriormente, o valor da taxa de aprendizado pode ser obtido por tentativa e erro, e tipicamente adotam-se valores entre 0,1 e 0,01 como tentativas iniciais. Uma outra alternativa é adotar uma taxa de aprendizado que varia em cada época conforme uma função, de forma a diminuir α à medida que a função de erro chega perto de seu valor mínimo (Udacity, 2017).

3.1.2. Perceptron Multi-Camada (MLP)

O Perceptron Multi-Camada, ou *Multi-Layer Perceptron (MLP)* consiste em conectar múltiplos perceptrons para gerar modelos de classificação mais complexos. Essa arquitetura, também chamada de Rede Neural, é capaz de gerar previsões com elevados índices de confiabilidade baseado no aprendizado por uma base de dados (Figura 8).



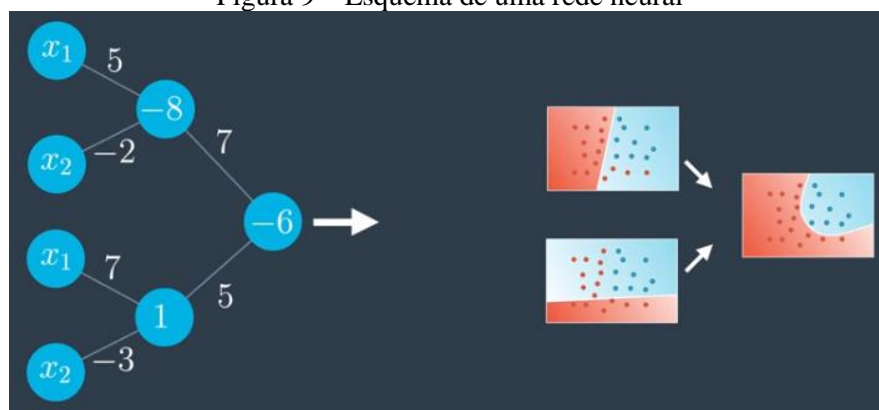
Fonte: Hochuli (2016)

3.1.2.1. Modelos não lineares

No 3.1.1, foi apresentado que o perceptron é capaz de gerar fronteiras lineares para a classificação de dados. Porém, surge a pergunta: como resolver problemas em que é necessária uma fronteira não linear? Por exemplo, e se a fronteira entre os alunos aprovados e rejeitados não fosse uma reta, e sim uma curva?

A resposta é combinar dois ou mais perceptrons. Essa combinação é feita com uma soma ponderada, usando uma função de ativação no final para obter um valor entre 0 e 1. Ou seja, coloca-se mais um perceptron no final da arquitetura para combinar os perceptrons que recebem o input. Isso resulta na organização apresentada na Figura 9, a seguir.

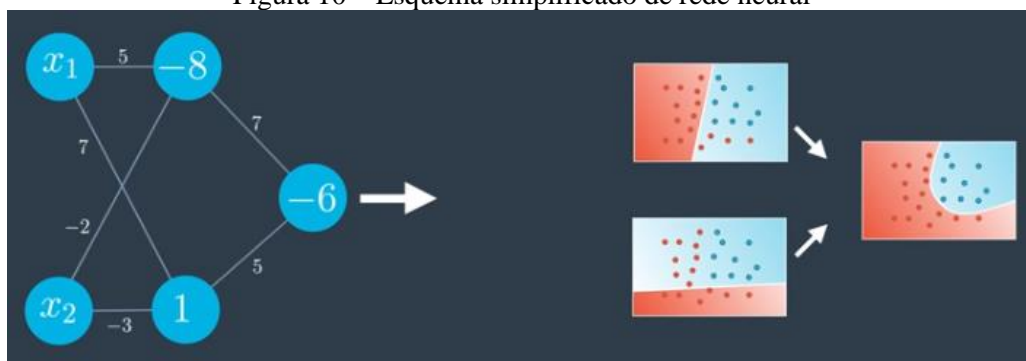
Figura 9 – Esquema de uma rede neural



Fonte: Udacity (2017)

Usando a notação convencional de redes neurais, tem-se a representação apresentada na Figura 10, a qual apresenta o modelo de rede neural e o resultado obtidos com uma combinação de funções lineares (Perceptrons).

Figura 10 – Esquema simplificado de rede neural

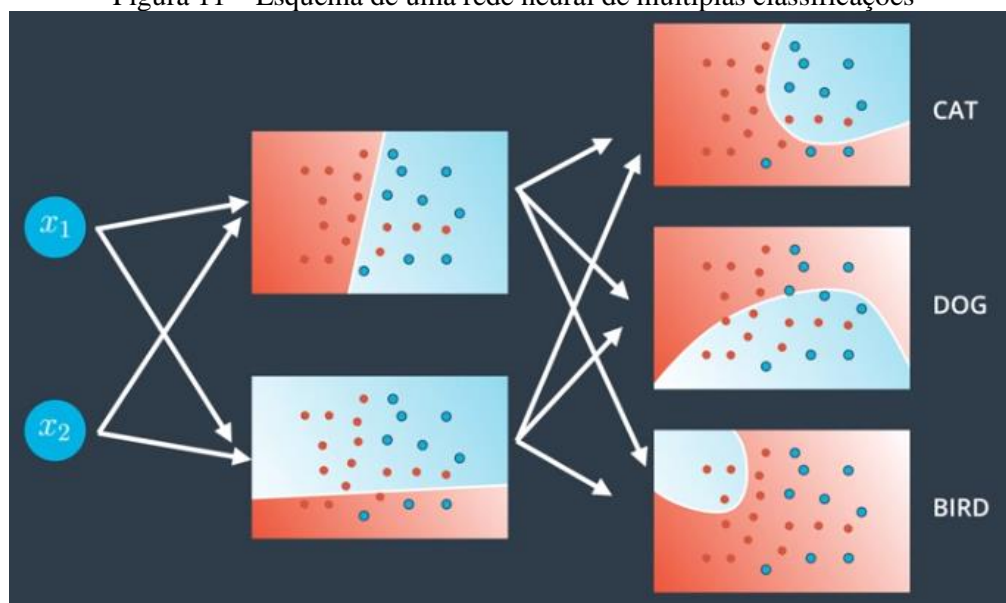


Fonte: Udacity (2017)

Essa combinação de perceptrons é chamada de Perceptron Multi-Camada (MLP), ou Rede Neural Profunda. A primeira camada é denotada de camada de entrada (*input layer*), as camadas intermediárias são chamadas de camadas ocultas (*hidden layers*), e a camada final é a camada de saída (*output layer*).

Além disso, a última camada da rede neural pode ser organizada de forma a gerar múltiplos outputs (Figura 11). Essas ideias serão discutidas nos itens seguintes.

Figura 11 – Esquema de uma rede neural de múltiplas classificações

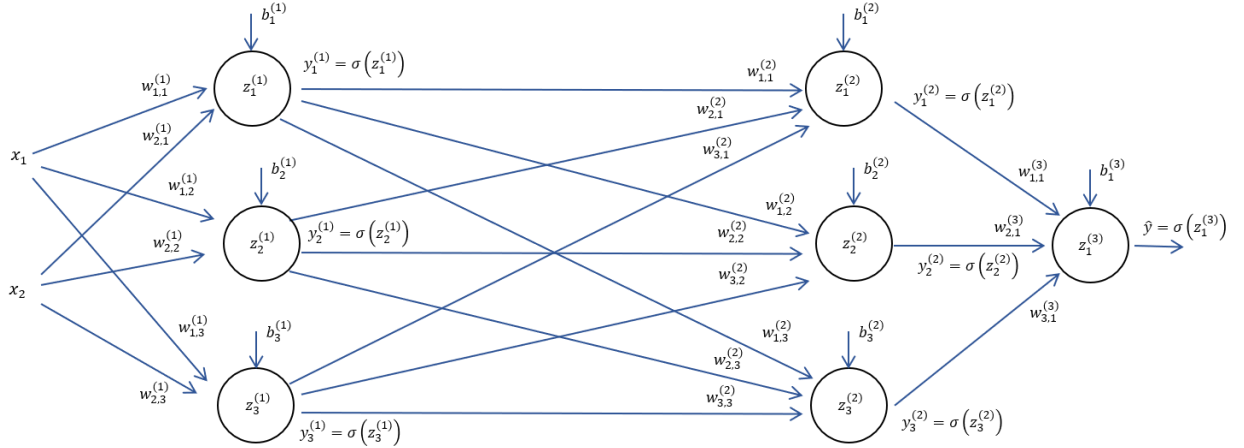


Fonte: Udacity (2017)

3.1.2.2. Arquitetura Geral

A Figura 12, a seguir, mostra uma Rede Neural com três camadas: uma camada de input, com 3 perceptrons, uma camada oculta, com 3 perceptrons, e uma camada de output, com 1 perceptron. Essa arquitetura é capaz de gerar um modelo altamente não linear, pois as fronteiras lineares da primeira camada oculta são combinadas na segunda camada, cujos outputs são combinados na última camada.

Figura 12 – Esquema do algoritmo de classificação (*feedforward*)



Conforme pode-se notar na Figura 12, os índices sobrescritos indicam a qual camada os parâmetros pertencem. Os números subscritos, por sua vez, indicam a qual perceptron os parâmetros pertencem. Assim, o peso $w_{i,j}^{(k)}$ pertence à camada k e faz a conexão entre o perceptron i da camada anterior (camada $k - 1$) e o perceptron j da camada k . Na última camada, o output final é sinalizado com um acento circunflexo (\hat{y}_j), enquanto os rótulos são sinalizados sem sobrescrito (y_j).

O algoritmo de classificação da rede neural, portanto, é resultado do processamento do input na primeira camada, cujo output é usado como input na camada seguinte, e assim por diante, até a geração de um conjunto de outputs da rede (\hat{y}_j). Esse algoritmo é chamado de *feedforward*.

Porém, o uso de pesos aleatórios resultará em previsões aleatórias, fazendo-se necessário um algoritmo de aprendizado para ajustar os pesos de todas as camadas e melhorar as previsões. Esse algoritmo, chamado de retropropagação (*backpropagation*) é o caso geral do algoritmo usado para o aprendizado de apenas um perceptron (item 3.1.1.2). Mais uma vez, define-se uma função de erro e utiliza-se o gradiente descendente para atingir o mínimo desta função: a única diferença é que esse algoritmo será usado em múltiplas camadas.

A seguir, será apresentada uma explicação matemática dos algoritmos de classificação (*feedforward*) e de aprendizado (*backpropagation*) para a Rede Neural Profunda.

3.1.2.3. Feedforward

Conforme a Figura 12, tem-se, em uma camada qualquer:

$$y_j^{(k)} = \sigma(z_j^{(k)}) \quad (10)$$

$$z_j^{(k)} = \begin{cases} \sum_{i=1}^n (w_{i,j}^{(k)} \cdot x_i) + b_j^{(k)}, & \text{para } k = 1 \text{ (primeira camada)} \\ \sum_{i=1}^n (w_{i,j}^{(k)} \cdot y_i^{(k-1)}) + b_j^{(k)}, & \text{para } k > 1 \end{cases} \quad (11)$$

Em notação matricial:

$$W^{(k)} = \begin{bmatrix} w_{1,1}^{(k)} & \dots & w_{1,m}^{(k)} \\ \vdots & \ddots & \vdots \\ w_{n,1}^{(k)} & \dots & w_{n,m}^{(k)} \end{bmatrix} \quad (12)$$

$$B^{(k)} = [b_1^{(k)} \quad \dots \quad b_m^{(k)}] \quad (13)$$

$$Z^{(k)} = [z_1^{(k)} \quad \dots \quad z_m^{(k)}] \quad (14)$$

$$X = [x_1 \quad \dots \quad x_n] \quad (15)$$

Em que n e m referem-se ao número de perceptrons nas camadas $k - 1$ e k , respectivamente. Portanto,

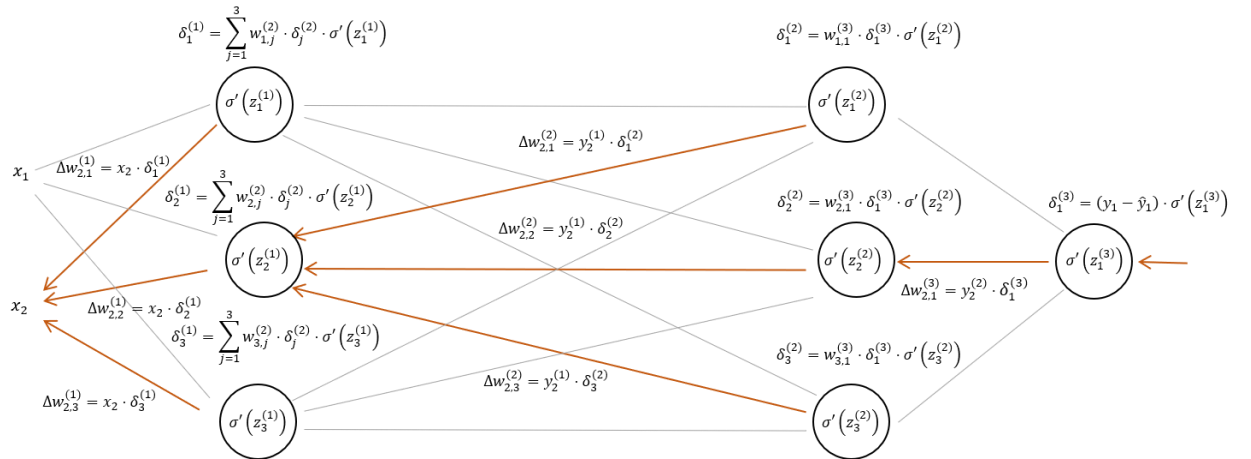
$$Z^{(k)} = \begin{cases} X \cdot W^{(k)} + B^{(k)}, & \text{para } k = 1 \text{ (primeira camada)} \\ Y^{(k-1)} \cdot W^{(k)} + B^{(k)}, & \text{para } k > 1 \end{cases} \quad (16)$$

$$Y^{(k)} = \sigma(Z^{(k)}) = [y_1^{(k)} \quad \dots \quad y_m^{(k)}] \quad (17)$$

3.1.2.4. Backpropagation

A retropropagação no perceptron multi-camada é feita calculado-se o gradiente da função de erro em relação a cada parâmetro (pesos e termo independente). Para a última camada da rede, usa-se a fórmula do δ apresentada no conjunto de equações (8). Nas camadas anteriores, a fórmula para atualização dos pesos vem da regra da cadeia, e equivale a alimentar a rede de trás para frente com o δ da última camada, multiplicando pela derivada da função de ativação em cada perceptron. O esquema da retropropagação está apresentado na Figura 13, a seguir.

Figura 13 – Esquema do algoritmo de aprendizado (*backpropagation*)



Assim, fórmula geral de atualização dos pesos é:

$$w_{i,j}^{(k)} = w_{i,j}^{(k)} + \alpha \cdot \Delta w_{i,j}^{(k)} \quad (18)$$

$$\Delta w_{i,j}^{(k)} = \begin{cases} x_i \cdot \delta_j^{(k)}, & \text{para } k = 1 \text{ (primeira camada)} \\ y_i^{(k-1)} \cdot \delta_j^{(k)}, & \text{para } k > 1 \end{cases} \quad (19)$$

$$\delta_i^{(k)} = \begin{cases} (y_i - \hat{y}_i) \cdot \sigma'(z_i^{(k)}) , & \text{para } k = l \text{ (última camada)} \\ \sum_{j=1}^m (w_{i,j}^{(k+1)} \cdot \delta_j^{(k+1)}) \cdot \sigma'(z_i^{(k)}) , & \text{para } k < l \end{cases} \quad (20)$$

Em que m é o número de perceptrons na camada $k + 1$.

Em notação matricial:

$$\delta^{(k)} = [\delta_1^{(k)} \quad \dots \quad \delta_n^{(k)}] \quad (21)$$

$$\Delta W^{(k)} = \begin{cases} X^T \cdot \delta^{(k)} , & \text{para } k = 1 \text{ (primeira camada)} \\ Y^{(k-1)T} \cdot \delta^{(k)} , & \text{para } k > 1 \end{cases} \quad (22)$$

$$\delta^{(k)} = \begin{cases} (Y - \hat{Y}) \odot \sigma'(Z^{(k)}) , & \text{para } k = l \text{ (última camada)} \\ (\delta^{(k+1)} \cdot W^{(k+1)T}) \odot \sigma'(Z^{(k)}) , & \text{para } k < l \end{cases} \quad (23)$$

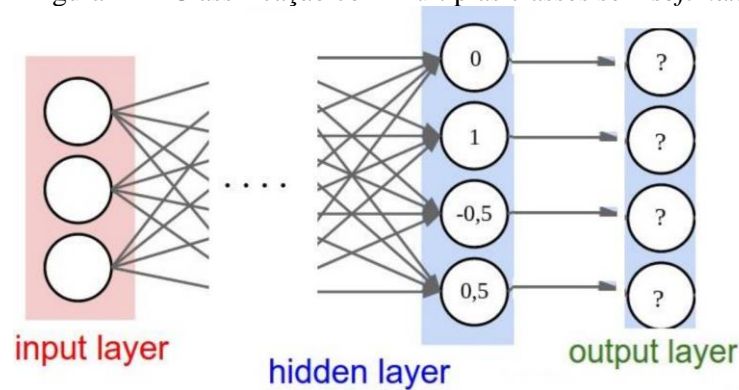
3.1.2.5. Softmax

Conforme apresentado no item 3.1.1.2, a implementação de uma função de ativação contínua e derivável resulta em uma classificação com valores entre 0 e 1. Essa classificação, portanto, pode ser considerada como a probabilidade que a rede atribui a classificação.

No exemplo de admissão dos alunos, isso significa que o output da rede consiste na probabilidade do aluno ser aprovado. Quando essa probabilidade é maior que 0,5, a rede considera que houve aprovação. Caso contrário, considera-se rejeição.

Porém, quando a rede gera múltiplas previsões, com múltiplos perceptrons na camada final, não é mais possível considerar os valores da saída como probabilidades, pois pode-se usar outras funções de ativação que gerem valores negativos (probabilidade só pode ser positiva) e a soma dos valores não necessariamente será igual a 1 (Figura 14). O que fazer nessa situação?

Figura 14 – Classificação com múltiplas classes sem *softmax*



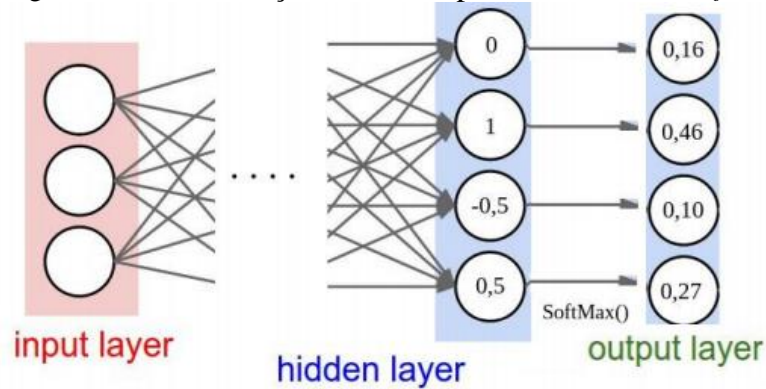
Fonte: Hochuli (2016)

Para resolver este problema, basta aplicar uma função que transforma todos os valores de entrada em números positivos cuja soma é 1, preservando as relações de magnitude dos inputs (inputs maiores devem resultar em probabilidades maiores). Essa função é chamada de *softmax*, e é amplamente usada em redes com classificação em múltiplas classes. Essa função é definida pela equação a seguir:

$$\text{softmax}(z_i) = y_i = \frac{e^{z_i}}{\sum_{j=1}^n e^{z_j}} \quad (24)$$

A Figura 15, a seguir, mostra a arquitetura final da rede. Nota-se que todos os valores são positivos com soma igual a 1, com o maior valor resultando na maior probabilidade, o segundo maior resultando na segunda maior probabilidade e assim por diante. A previsão da rede, portanto, é o maior valor da *softmax* (maior probabilidade).

Figura 15 – Classificação com múltiplas classes usando *softmax*



Fonte: Hochuli (2016)

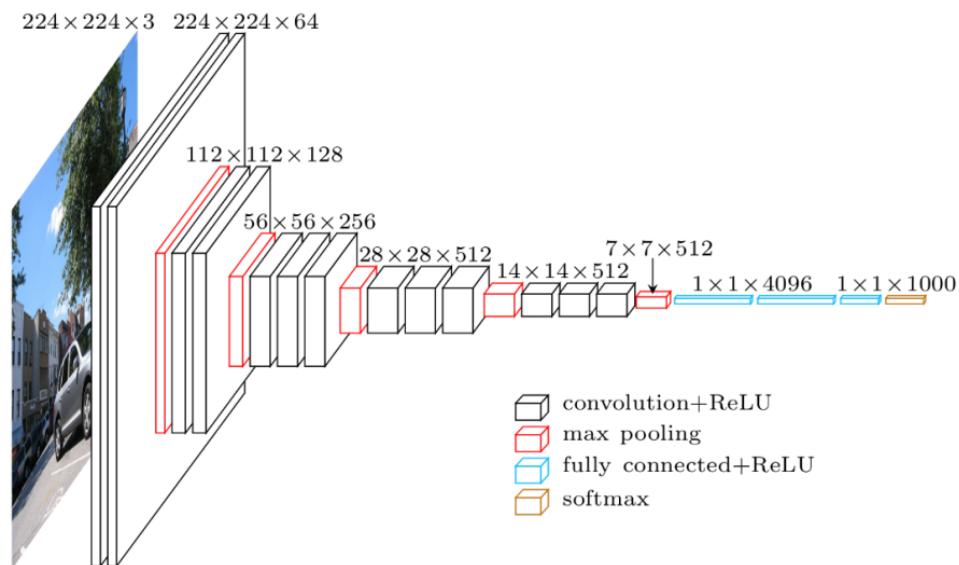
3.2. REDES CONVOLUCIONAIS

A aplicação de Redes Neurais Convolucionais (CNNs) tem trazido grandes resultados na área de reconhecimento de padrões, incluindo processamento de imagens e voz, controle de carros autônomos e detecção de câncer. Essas conquistas foram o fruto do desenvolvimento de novas arquiteturas de redes neurais, que são capazes de realizar tarefas que não eram possíveis com os modelos convencionais.

As redes convolucionais trabalham com os mesmos princípios de uma rede neural convencional: perceptrons são organizados em camadas, tomando diversos valores numéricos como input e gerando um ou mais outputs. Definem-se as funções de erro e de ativação, e o aprendizado se dá com a retropropagação. O que muda é a estrutura da rede: em vez de usar camadas totalmente conectadas, usam-se camadas convolucionais, que são conectadas localmente. Isso reduz drasticamente o número de parâmetros necessários para a aprendizagem (pesos), o que torna possível que as redes aprendam muito mais rápido. Intercaladas entre as camadas convolucionais, são usadas camadas de subamostragem (*pooling*) e, ao final da rede, são usadas camadas totalmente conectadas (camadas densas) para gerar o output final (Figura 16).

A seguir, apresenta-se o funcionamento dessas três camadas nas redes convolucionais. Esse funcionamento será explicado no contexto do reconhecimento de imagens, que é o foco do presente trabalho.

Figura 16 – Organização de camadas em uma rede convolucional (arquitetura da rede VGG-16)



Fonte: Jordan (2018a)

3.2.1. Camada Convolucional

As camadas convolucionais são o elemento constituinte mais importante das redes convolucionais. Essas camadas são responsáveis por executar a maior parcela do trabalho computacional.

A primeira camada convolucional toma como input uma imagem, que nada mais é do que um tensor de três dimensões (figura 17). Na altura e largura, encontram-se os valores dos pixels que compõem a imagem; na profundidade, organizam-se os três canais de cor (*RGB*, vermelho, verde e azul). Assim, já pode-se notar uma distinção importante das redes convolucionais: essas redes têm como input tensores de três dimensões, enquanto as redes convencionais tomam como input vetores de apenas uma dimensão.

Figura 17 – Representação de uma imagem como um tensor de três dimensões



Fonte: Udacity (2017)

Os parâmetros da camada convolucional são um conjunto de filtros. Cada filtro tem pequenas dimensões espaciais (largura e altura), mas a sua profundidade é a mesma do tensor de input.

Esse filtro escaneia a imagem ao longo de sua largura e altura. Em cada posição que o filtro percorre, multiplicam-se os valores do input pelos pesos correspondentes do filtro. Esses produtos são somados, adicionando-se um termo independente (*bias*), e aplica-se uma função de ativação ao final. Cada valor de output é organizado em uma matriz, que é

chamada de mapa de ativação do filtro. Cada mapa de ativação pode ser considerado como um canal de uma nova imagem formada na camada convolucional, que servirá de input para a camada seguinte.

O processo da convolução é apresentado nas figuras 18, 19 e 20. O input é uma figura de dimensões [5,5,3], e têm-se dois filtros (W_0 e W_1) de dimensões [3,3,3]. O output é um mapa de ativação com dimensões [3,3,2]. Para realizar a convolução, é necessário definir três hiperparâmetros importantes. São eles:

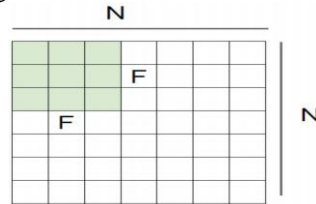
1. **Profundidade do output (*depth*):** A profundidade escolhida para o output define o número de filtros usados na convolução. Conforme a Figura 20, 2 filtros resultam em um mapa de ativação com profundidade igual a 2.
2. **Passo (*stride*):** Refere-se ao passo com o qual o filtro se move. Quando o passo é igual a 1, move-se o filtro um pixel por vez (Figura 19). Se o passo for igual a 2, assim como na Figura 20, o filtro pula dois pixels por vez. Quanto maior o passo, menor as dimensões espaciais do output. O passo é o mesmo tanto na direção horizontal quanto na vertical. Nota-se também que passos menores do que a dimensão do filtro criam sobreposições entre as diferentes posições do filtro.
3. **Preenchimento (*zero-padding*):** Na Figura 20, nota-se um preenchimento com zeros ao redor da matriz de input. Essa configuração ajuda a controlar as dimensões do output. Além disso, o preenchimento também previne uma possível perda de informações ao redor da borda da imagem.

As dimensões espaciais do output (O), sejam elas a altura ou a largura, podem ser calculadas em função das dimensões espaciais do input (N), do filtro (F), do passo (S) e da quantidade de preenchimento usada (P). Tem-se:

$$O = 1 + \frac{N + 2P - F}{S} \quad (25)$$

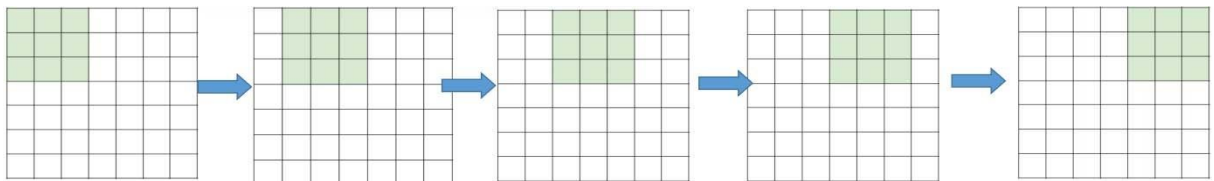
Nota-se que os hiperparâmetros devem ser escolhidos de forma que as dimensões do output sejam um número inteiro. Caso contrário, os hiperparâmetros são considerados inválidos.

Figura 18 – Janela de convolução



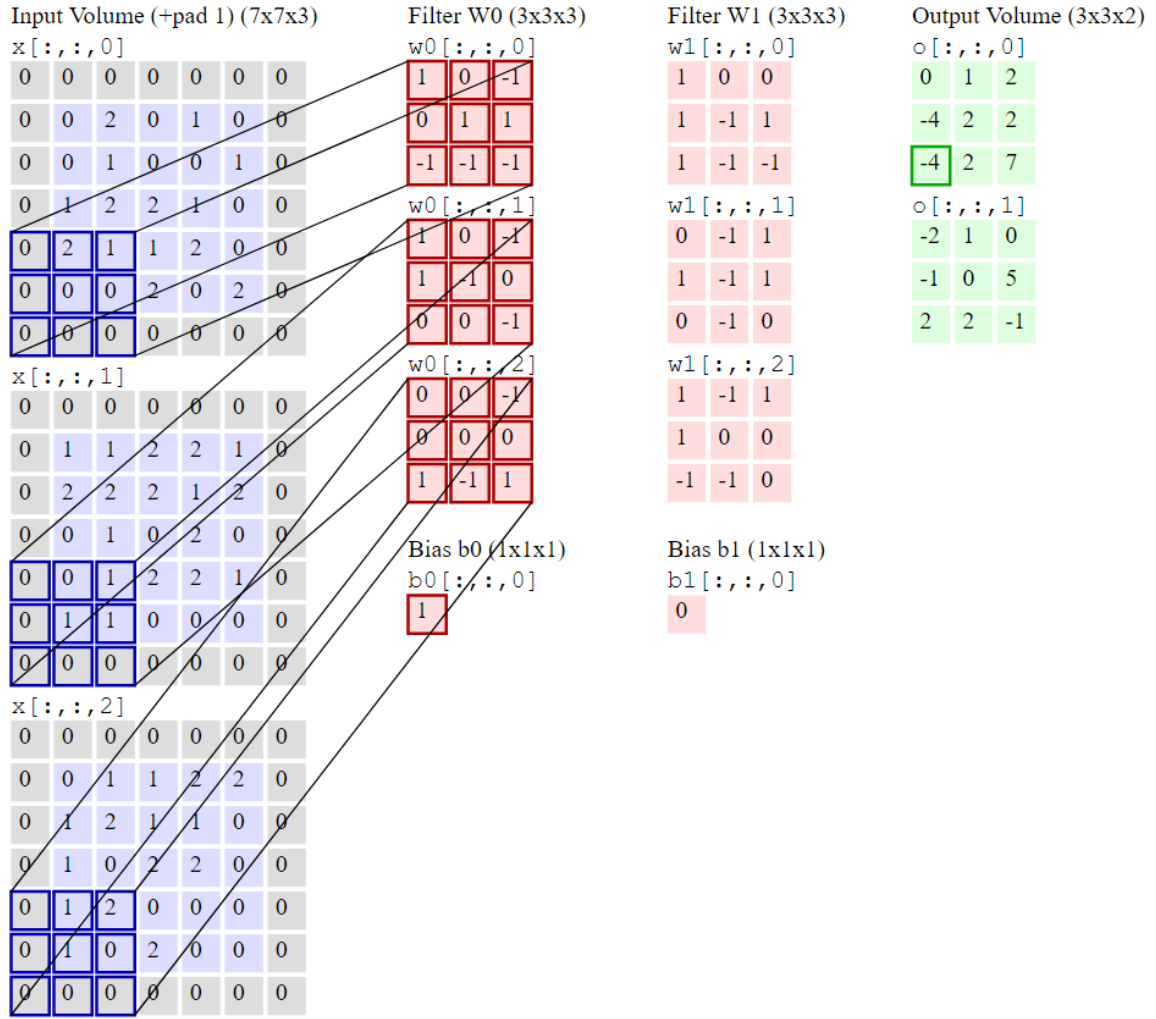
Fonte: Albawi *et al.* (2017)

Figura 19 – Passo igual a 1: o filtro move um pixel para cada conexão



Fonte: Albawi *et al.* (2017)

Figura 20 – Funcionamento do filtro



Fonte: CS231n (2015)

Matematicamente, o processo da convolução é descrito da seguinte forma:

$$G[m, n] = (f * h)[m, n] = \sum_j \sum_k f[j, k] \cdot h[m + j, n + k] + b$$

Para um caso geral, com passo maior que 1 e explicitando a profundidade do filtro, tem-se:

$$G[m, n] = (f * h)[m, n, l] = \sum_j \sum_k \sum_l f[j, k, l] \cdot h[s \cdot m + j, s \cdot n + k, l] + b \quad (26)$$

Em que:

- $G[m, n]$: resultado de cada elemento da matriz de output, sendo m e n os seus índices;
- $f[j, k, l]$: elemento do filtro, sendo j , k e l os índices da linha, coluna e profundidade, respectivamente;
- h : tensor de input, sendo j , k e l os índices da linha, coluna e profundidade do filtro, e m e n os índices da linha e coluna da matriz de output;
- s : passo;

- b : termo independente.

Ao final da convolução, aplica-se uma função de ativação ao tensor de output, gerando um mapa de ativação. Em redes convolucionais, é comum a utilização da função de ativação *ReLU* (*Rectified Linear Unit*). Trata-se de uma simples função que vem trazendo bons resultados para o aprendizado desse tipo de rede. Esta função é definida por:

$$ReLU(x) = \max(0, x)$$

$$\frac{\partial}{\partial x} ReLU(x) = \begin{cases} 1, & \text{se } x > 0 \\ 0, & \text{caso contrário} \end{cases} \quad (27)$$

O número de parâmetros de aprendizado (n) em uma camada de convolução é dado pela equação a seguir (28).

$$n = KFFD_{in} + K \quad (28)$$

Em que:

- K : número de filtros na camada convolucional
- F : altura e largura dos filtros convolucionais
- D_{in} : profundidade da camada anterior

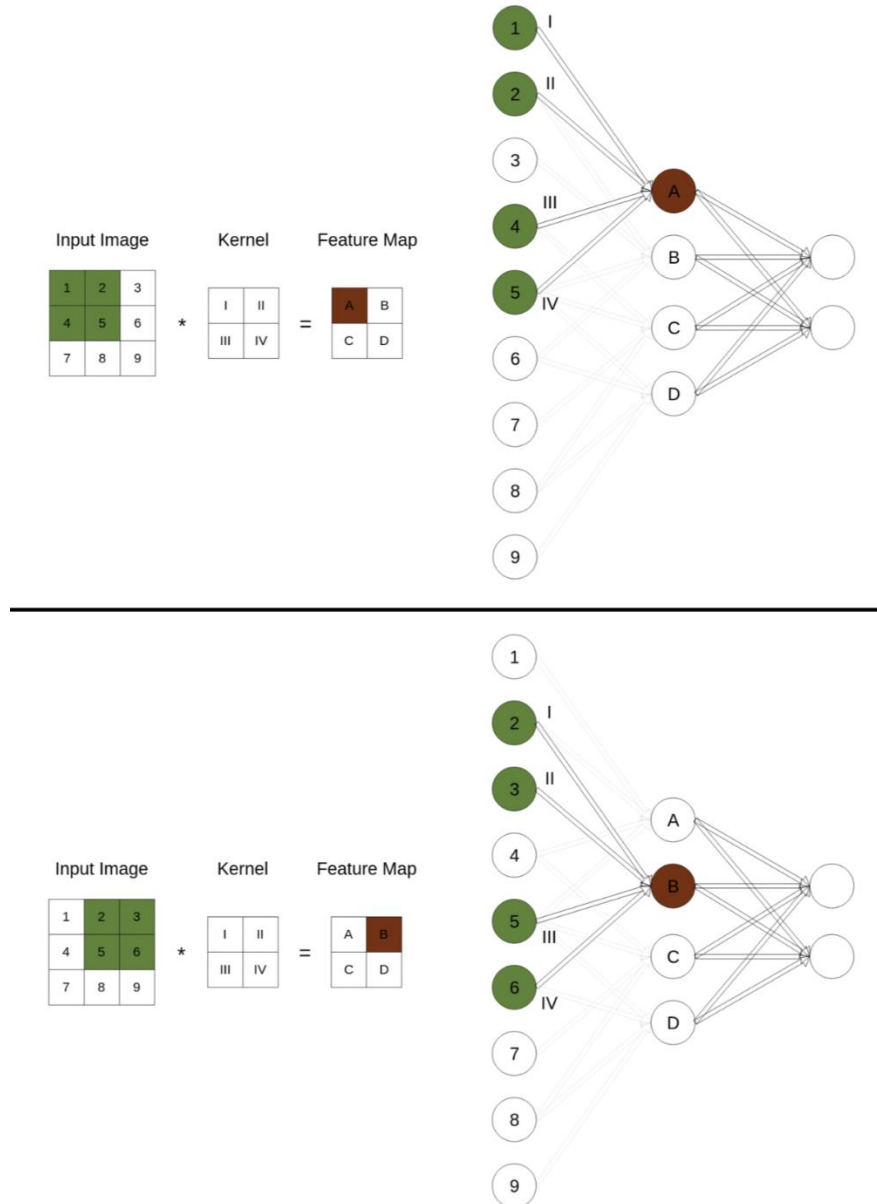
Assim, a equação (28) mostra o impacto da camada convolucional na redução de parâmetros. Por exemplo, se quiséssemos gerar um mapa de ativação de 32×32 a partir de uma imagem de dimensões $32 \times 32 \times 3$ usando uma camada totalmente conectada, seriam necessários $(32 \times 32) \times (32 \times 32 \times 3) + (32 \times 32) = 3.146.752$ parâmetros. Contudo, uma camada convolucional com um filtro de $5 \times 5 \times 3$ só necessita de $5 \times 5 \times 3 + 1 = 81$ parâmetros. Cada filtro pode ser representado como uma série de perceptrons que são conectados localmente e compartilham os mesmos pesos, conforme a Figura 21.

Para mostrar o poder da camada de convolução, a Figura 23, a seguir, mostra o resultado do processamento com diferentes filtros previamente definidos. Pode-se notar que os pesos dos filtros podem ser selecionados para aplicar certos efeitos na imagem, como deixar as arestas nítidas (*sharpen*) e desfoque (*blur*). É assim que esses tipos de efeitos são atingidos em *softwares* de edição de imagem, como o *Photoshop*.

Porém, as redes convolucionais usam os filtros para detectar certos padrões na imagem. Nas primeiras camadas, detectam-se simples padrões, como bordas (Figura 23, *edge detection*). Essa detecção se dá da seguinte forma: cada filtro pode ser considerado como uma pequena imagem, e o output desse filtro será máximo sempre que ele “escanear” regiões da imagem de input semelhantes à imagem do filtro (Figura 22). Em contrapartida, o output será mínimo sempre que o filtro escanear regiões muito distintas da imagem do filtro. A função de ativação *ReLU*, por sua vez, transformará em zero os menores valores (negativos) de output e deixará apenas os maiores valores, o que resultará em um mapa de ativação que sinaliza as regiões da imagem original em que o filtro detectou esse determinado padrão. O resultado dessa convolução pode ser notado nas figuras 22 e 23.

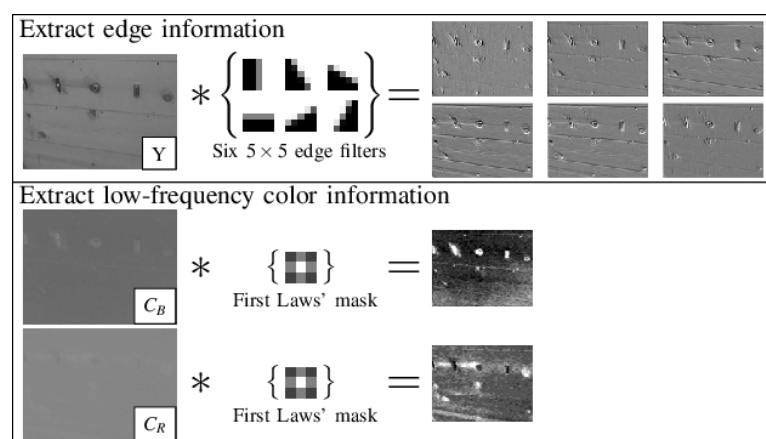
Por fim, deve-se notar que os pesos necessários para essa detecção são “aprendidos” pela rede neural com a retropropagação, sem a necessidade de serem definidos explicitamente.

Figura 21 – Conexão local com compartilhamento de parâmetros





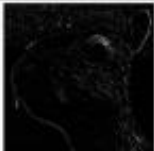




Fonte: Skalski (2019)

Figura 22 – Detecção de padrões usando filtros de convolução



Fonte: Sukhoy (2010)

Figura 23 – Resultado da aplicação de diferentes filtros na convolução

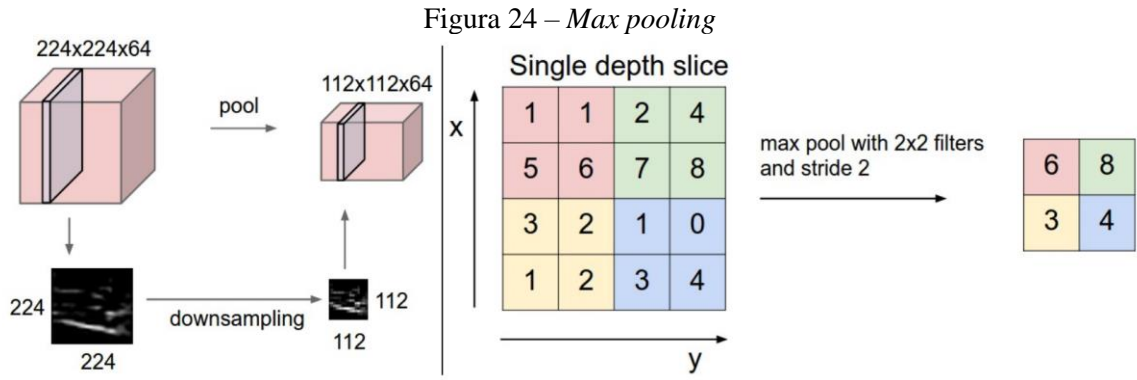
Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Fonte: Albawi *et al.* (2017)

3.2.2. Camada de Subamostragem (*Pooling*)

É usual a presença de camadas de subamostragem após as camadas convolucionais. A sua função é reduzir o tamanho espacial dos mapas de ativação e ao mesmo tempo manter os detalhes importantes. Assim, tem-se uma redução do número de parâmetros de aprendizado, o que acelera o processamento de treinamento e também controla o *overfitting*. A forma mais comum de subamostragem quando se trabalha com imagens é o *max pooling*, em que filtros percorrem o tensor de input e, em cada posição, seleccionam da janela de subamostragem o maior valor: todos esses valores são agrupados em uma nova matriz de output que tem dimensões espaciais menores do que a de input.

Tipicamente, são usados filtros de dimensão 2x2 aplicados com um passo de 2. Isso resulta em um output com altura e largura iguais à metade das dimensões espaciais de input, descartando 75% das ativações (Figura 24). Nota-se que a profundidade do tensor é preservada.



Fonte: CS231n (2015)

Para uma operação de *pooling* qualquer, as dimensões do output são dadas por:

$$L_2 = \frac{L_1 - F}{S + 1}, \quad A_2 = \frac{A_1 - F}{S + 1}, \quad D_2 = D_1 \quad (29)$$

Em que

- L_1 , A_1 e D_1 : largura, altura e profundidade do input;
- L_2 , A_2 e D_2 : largura, altura e profundidade do output;
- S : passo da janela de subamostragem.

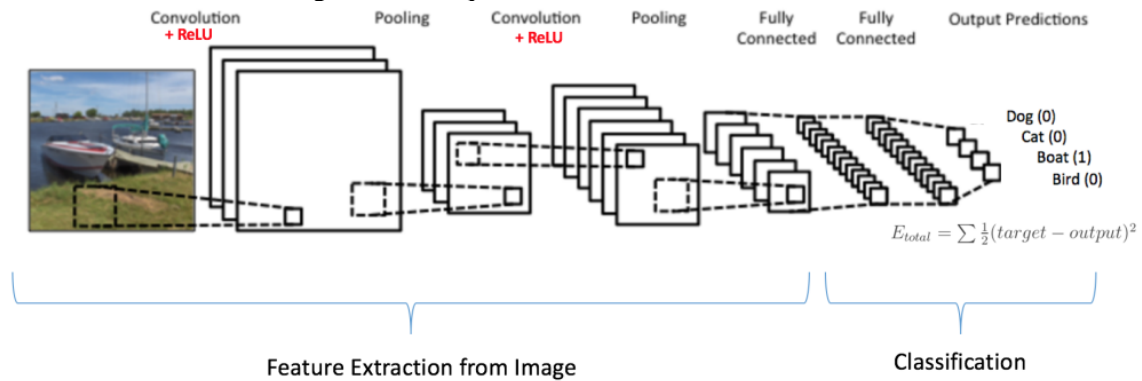
3.2.3. Camada Densa

Ao longo da rede convolucional, camadas convolucionais e de subamostragem são implementadas de forma a reduzir gradativamente as dimensões espaciais dos mapas de ativação (altura e largura), e ao mesmo tempo aumentar sua profundidade, usando um número cada vez maior de filtros.

No final da rede, resta um mapa com altura e largura pequenos, e profundidade bem maior do que a inicial. A esta camada, faz-se uma série de conexões densas, ou seja, acrescentam-se um conjunto de camadas totalmente conectadas. Essas camadas são as mesmas camadas usadas nas redes neurais convencionais, em que cada perceptron está conectado a todos os perceptrons da camada anterior. Essa arquitetura está esquematizada na Figura 25.

O motivo de acrescentar essas camadas densas é o fato de que as informações mais complexas da rede estão no seu final, e são muito importantes para a classificação final da imagem. Como essas informações não podem ser perdidas, faz-se necessário o uso de camadas totalmente conectadas.

Figura 25 – Esquema de uma rede convolucional

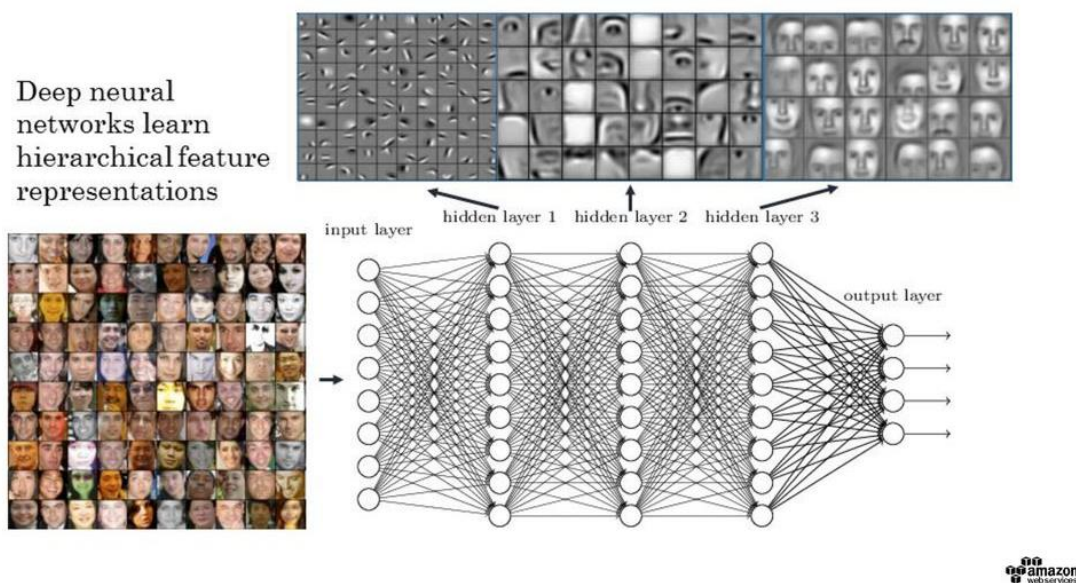


Fonte: Nehme (2018)

Em contrapartida, as informações mais simples da rede neural são detectadas no início da rede, como bordas e texturas.

A evolução da complexidade de informações detectadas ao longo da rede é apresentada na Figura 26, a seguir. As características mais complexas são detectadas a partir de um padrão de combinação das características mais simples. No exemplo de detecção de faces, identificam-se primeiro bordas, e depois características mais complexas, como nariz, boca e olho. A combinação dessas últimas torna possível a detecção de diferentes faces. Por fim, a classificação final (de quem é o rosto) se dá na camada final, totalmente conectada.

Figura 26 – Complexidade de informações detectadas ao longo da rede CNN Feature Detection



Fonte: Vahid (2017)

3.3. TREINANDO REDES NEURAIIS

Em última análise, o principal motivo para se implementar uma rede neural é a capacidade de generalização, ou seja, a capacidade do modelo em gerar bons resultados em dados que não foram usados para o aprendizado.

Assim, existem dois problemas que podem acontecer no processo de treinamento de uma rede neural. São eles: *underfitting* e *overfitting*. Para evitar esses problemas, existem diversas

medidas que podem ser tomadas. Dentre elas, destacam-se as escolhas feitas na etapa de tratamento de dados, que são fundamentais. Essas questões serão discutidas em mais detalhes nos itens a seguir.

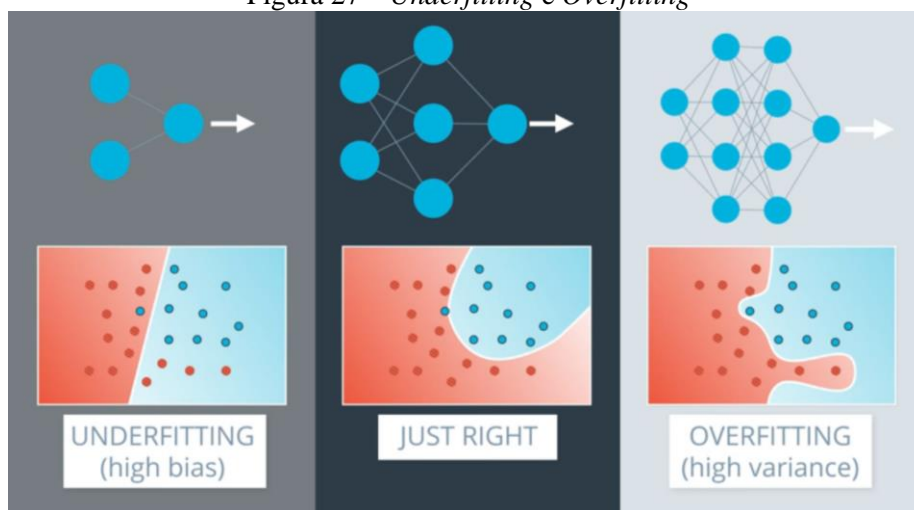
3.3.1. *Underfitting e Overfitting*

Underfitting refere-se a um modelo que não é capaz de modelar bem os dados de treinamento. Isso implica em um modelo que produz elevados erros de treinamento (Brownlee, 2016).

Overfitting, por sua vez, refere-se a um modelo que se adequa muito bem os dados de treinamento. Porém, esse modelo aprende os detalhes e o ruído nos dados de maneira a afetar negativamente o seu desempenho em novos dados. Isso significa que os ruídos ou flutuações aleatórias nos dados de treinamento são captados e aprendidos como conceitos pelo modelo. O problema é que esses conceitos não se aplicam a novos dados e afetam negativamente a capacidade de generalização dos modelos (Brownlee, 2016).

A Figura 27, a seguir, apresenta três modelos: um modelo sofre de *underfitting*, outro de *overfitting*, e outro está bem ajustado. Pode-se notar que a configuração ótima, ainda que gere um pequeno erro na classificação, está mais adequada para generalizar o que foi aprendido para diferentes conjuntos de dados.

Figura 27 – *Underfitting e Overfitting*



Fonte: Udacity (2017)

Conforme pode-se notar na Figura 27, modelos muito simples de redes neurais não são capazes de modelar a complexidade de distribuição dos dados e tendem a resultar em *underfitting*. Porém, arquiteturas excessivamente complexas tendem a causar *overfitting*, pois resultam em modelos altamente não lineares com alta variância para adequar-se aos dados de treinamento.

3.3.2. Dados

Dados são o elemento mais importante para qualquer algoritmo de aprendizado de máquina (*machine learning*). Isso se deve ao fato de que esses algoritmos computacionais consistem em achar correlações entre os dados para fazer previsões.

No caso de Redes Neurais, cada ponto de dados deve ter um conjunto de características de input (*features*) e um conjunto de características de rótulo (*labels*). As características de input devem, em hipótese, ter alguma correlação com os rótulos. Além disso, os dados devem ser quantitativos ou categóricos, pois a rede neural consiste em um modelo matemático.

Por isso, grande parte do trabalho necessário para desenvolver uma rede neural consiste em encontrar, produzir, filtrar e estudar dados.

3.3.2.1. Conjuntos de treino, validação e teste

Conforme explicado no item 3.3.1, é necessário encontrar uma arquitetura da rede neural que não seja excessivamente simples ou complexa. Isso constitui um processo de tentativa e erro, testando o aprendizado com números diferentes de camadas e de perceptrons.

Para assegurar que a rede neural não gere bons resultados apenas nos dados de treinamento (*training data*), separa-se um segundo conjunto de dados, chamado de conjunto de validação (*validation data*). Os dados de validação não são usados no aprendizado, eles servem apenas para verificar a performance da rede em novos dados. Ou seja, servem para testar a capacidade de generalização do modelo depois de obter bons resultados com os dados de treinamento.

Na prática, treinam-se múltiplas arquiteturas de redes neurais com os dados de treinamento até atingir baixos erros de treinamento. Em seguida, avalia-se a performance dessas redes no conjunto de validação para escolher a melhor arquitetura.

Por fim, usa-se ainda mais um conjunto de dados para avaliar a performance da rede, chamado de conjunto de teste (*test data*). Esse último conjunto de dados tem como objetivo evitar que o modelo escolhido não seja adequado apenas para o conjunto de validação.

A divisão dos dados nesses três conjuntos é amplamente usada no desenvolvimento de qualquer modelo de *machine learning* para evitar *overfitting*.

3.3.2.2. Escalonamento de dados

Os dados para o aprendizado de uma rede neural podem ter diversas variáveis com diferentes escalas. Por exemplo, algumas podem estar em pés, quilômetros ou horas, outras podem ter valores negativos.

Diferenças nas escalas entre as variáveis de entrada podem aumentar a dificuldade do problema que está sendo modelado. Um exemplo disso é que grandes valores de entrada (por exemplo, um intervalo de centenas ou milhares de unidades) podem resultar em um modelo que aprende grandes valores de peso. Um modelo com grandes valores de peso geralmente é instável, o que significa que ele pode sofrer com um desempenho ruim durante o aprendizado e sensibilidade aos valores de entrada, resultando em maior erro de generalização (Brownlee, 2019b).

O segundo problema é que os dados com maior escala terão um sinal maior na rede neural, o que pode diminuir a influência de dados com escalas menores. Isso diminui a capacidade do modelo aprender com algumas características importantes do conjunto de dados.

Para evitar esse problema, os dados de input são escalonados antes de serem alimentados na rede neural. Esse escalonamento pode ocorrer de duas formas:

- **Normalização:** Consiste em mudar a escala dos dados de forma que fiquem em um intervalo pré-definido. Normalmente adota-se o intervalo de 0 a 1. Para usar a normalização, os dados de entrada devem variar entre um valor mínimo e máximo.
- **Padronização:** Se a distribuição dos dados é normal, recomenda-se que os dados sejam padronizados de forma a assumir uma distribuição com média igual a 0 e desvio padrão de 1.

Assim, o escalonamento dos dados de entrada garante que todas variáveis influenciem a rede neural com a mesma intensidade.

Quanto aos dados de rótulo, o escalonamento deve ocorrer para garantir que os rótulos variem no mesmo intervalo que a função de ativação no final da rede.

4. METODOLOGIA

Objetiva-se treinar uma rede neural capaz de detectar fissuras em imagens de superfície de concreto. Tal detecção consiste na delimitação da região ocupada pela fissura na imagem, conforme a Figura 28.

Figura 28 – Detecção de fissura usando rede neural convolucional

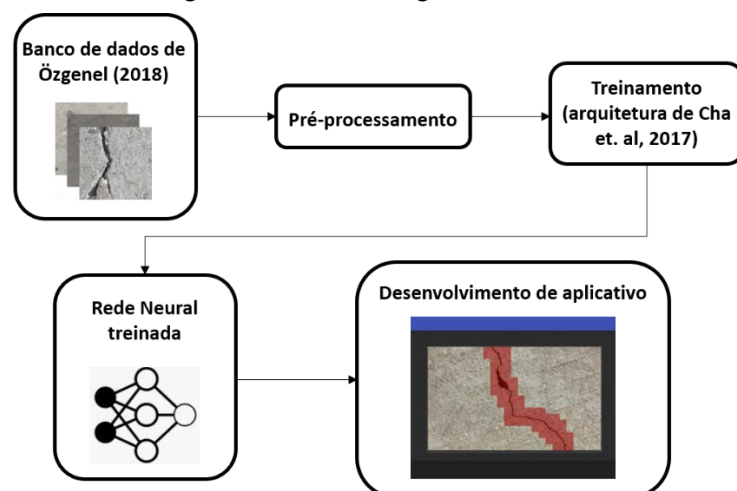


Fonte: Cha *et al.* (2017)

Almeja-se, portanto, uma visão computacional da fissura, o que possibilitará estudos futuros que podem automatizar diversos processos na área de estudo de patologias, como determinação de abertura de fissuras, comprimento total de fissuras, quantidade total de material de injeção necessário para recuperação de uma região fissurada, etc.

Para atingir tal objetivo, adota-se a seguinte metodologia: utiliza-se o banco de dados *open-source* criado por Özgenel (2018) para o treinamento da rede neural convolucional. A arquitetura dessa rede é a proposta por Cha *et al.* (2017). Em seguida, desenvolve-se um aplicativo *web* para a detecção de fissuras, que usa a rede treinada para fazer previsões em diversos recortes de uma imagem com fissura escolhida pelo usuário. A imagem a seguir apresenta um fluxograma da metodologia:

Figura 29 – Metodologia do trabalho



Os itens a seguir detalham a metodologia proposta, apresentando as ferramentas usadas, o banco de dados, os passos de pré-processamento, a arquitetura da rede neural, e a aplicação do modelo.

4.1. FERRAMENTAS

O desenvolvimento e treinamento da rede neural convolucional é realizado com o *Keras*, que é um pacote de *Deep Learning* para *python*. Esse pacote, por sua vez, utiliza o *tensorflow* para a realização de operações de *machine learning*, possibilitando que o pesquisador se preocupe apenas com os detalhes mais importantes da implementação da rede neural (arquitetura, dados, parâmetros de aprendizado). O ambiente de desenvolvimento será o *Jupyter Notebook*.

Assim, o *Keras* permite implementar cada camada de uma rede neural com uma linha de código, explicitando o tipo da camada (camada densa, convolucional, de *pooling* ou de ativação) e seus parâmetros (número de filtros e suas dimensões, tipo de ativação). Por fim, compila-se o modelo e definem-se diversos parâmetros para melhorar o aprendizado, como gradiente descendente estocástico, *dropout*, regularização de pesos e momento.

Depois que o modelo estiver pronto, basta salvar e usá-lo para a implementação desejada.

4.2. BANCO DE DADOS

Para treinar a rede neural de classificação, é necessário o uso de um banco de dados que contenha imagens com fissura e sem fissura. Quanto maior esse banco de dados, melhor será o treinamento (no estudo de Cha, utilizaram-se 40 mil imagens para validação e treino).

Assim, decidiu-se usar o banco de dados *open-source* criado por Özgenel (2018), que consiste em 40 mil imagens com resolução de 227x227 pixels, divididas em dois diretórios correspondentes aos rótulos de interesse: “Positive”, relativo à presença de fissura, e “Negative”, relativo à ausência. Cada rótulo tem a mesma quantidade de imagens (20 mil imagens contêm fissura e 20 mil imagens não).

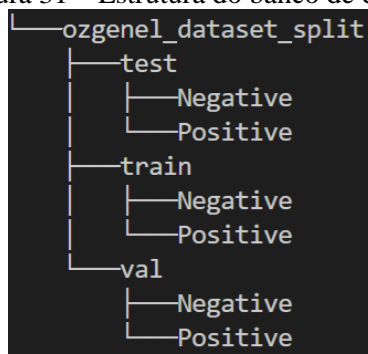
Este banco de dados foi usado em diversas pesquisas com o tópico de detecção de fissuras, sendo um exemplo a pesquisa de Dung (2018). Este é um fator que comprova a qualidade das imagens.

Figura 30 – Amostras do banco de dados de Özgenel (2018)



Para dividir as imagens em conjuntos de treino, validação e teste, escreveu-se um script em *python* que distribuiu as imagens aleatoriamente nos diretórios correspondentes em uma proporção 70%:15%:15% (anexo 10.1). Cada um desses conjuntos tem a mesma quantidade de imagens para ambos os rótulos (com e sem fissura). Esta proporção é comumente usada em *datasets* maiores (mais de 10 mil amostras), e está próxima da proporção usada por Cha *et al.* (2017).

Figura 31 – Estrutura do banco de dados



É importante ressaltar que o banco de dados usado tem fissuras mais abertas, diferente do banco de dados usado por Cha *et al.* (2017), formado por imagens com fissuras mais fechadas. Porém, o uso de imagens de treinamento com fissuras abertas também se mostrou promissor para a tarefa de classificação, como está apresentado nos resultados.

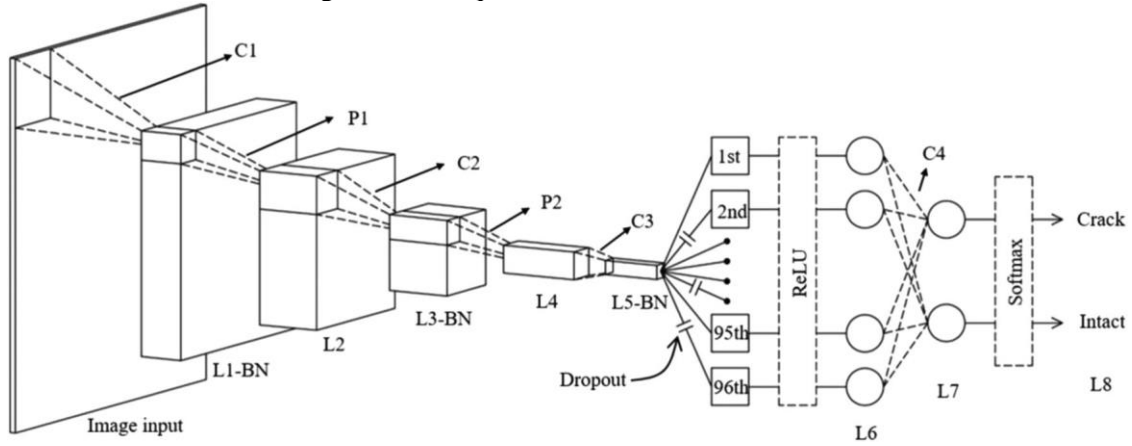
4.3. ARQUITETURA DO MODELO

Procura-se desenvolver uma rede neural convolucional de classificação, capaz de prever a presença ou ausência de fissura em uma imagem colorida de 227×227 px (dimensões das imagens do banco de dados).

Para realizar esta tarefa, será utilizada uma arquitetura inspirada na arquitetura proposta por Cha *et al.* (2017) (Figura 32), que consiste em uma rede neural simples de classificação, que trouxe bons resultados para a detecção de fissuras em concreto. Uma mudança chave nesta arquitetura é a dimensão da camada de *input*, de 227×227 em vez de 256×256 , a dimensão utilizada no estudo de Cha. Esta mudança foi implementada para adequar o modelo ao banco

de dados utilizado e descartar, portanto, a necessidade de redimensionamento de imagens durante o processo de treinamento. Além disso, o estudo de Dung (2018) apresentou resultados satisfatórios com esta resolução, o que foi mais um fator a favor desta escolha.

Figura 32 – Arquitetura da rede convolucional



Fonte: Cha *et al.* (2017)

Tabela 2: Dimensões das camadas da rede neural

Camada	Altura	Largura	Profundidade	Operação	Altura	Largura	Profundidade	Qtd.	Passo
Input	227	227	3	C1	20	20	3	3	2
BN	–	–	–	Batch Norm.	–	–	–	–	–
L1	114	114	3	P1	7	7	–	–	2
L2	54	54	3	C2	15	15	3	24	2
BN	–	–	–	Batch Norm.	–	–	–	–	–
L3	27	27	24	P2	4	4	–	–	2
L4	12	12	24	C3	10	10	48	48	2
BN	–	–	–	Batch Norm.	–	–	–	–	–
L5	6	6	48	ReLU	–	–	–	–	–
L6	6	6	48	C4	1	1	48	96	1
–	–	–	–	Flatten	–	–	–	–	–
L7	1	1	3456	C5	1	1	3456	96	1
L8	1	1	96	C6	1	1	96	2	–
L9	1	1	2	Softmax	–	–	–	–	–

A Tabela 2 descreve as camadas da rede neural. Após as primeiras camadas de convolução, implementam-se camadas de *batch normalization*, que aplicam uma padronização no seu input, fazendo com que seu output tenha uma média próxima de zero e desvio padrão próximo de 1. Essas camadas têm a função de acelerar o treinamento e reduzir a chance de *overfitting*, além de possibilitar processamento com valores em um intervalo controlado, o que geralmente é a melhor escolha quando se usa métodos numéricos/computacionais. Além disso, aplica-se apenas uma camada de ativação (ReLU) antes das camadas densas ao final da rede.

Na última camada, implementa-se uma camada densa com dois nós, seguida de uma camada *softmax*. Os outputs finais, portanto, são dois: uma probabilidade de ausência de fissura e outra probabilidade de presença de fissura (entropia cruzada categórica). Toma-se como

predição o rótulo correspondente ao maior valor entre os dois. Todas essas medidas foram usadas na arquitetura de Cha.

O código para a montagem da rede encontra-se no anexo 10.3, cujo número total de parâmetros treináveis resultou em 471955.

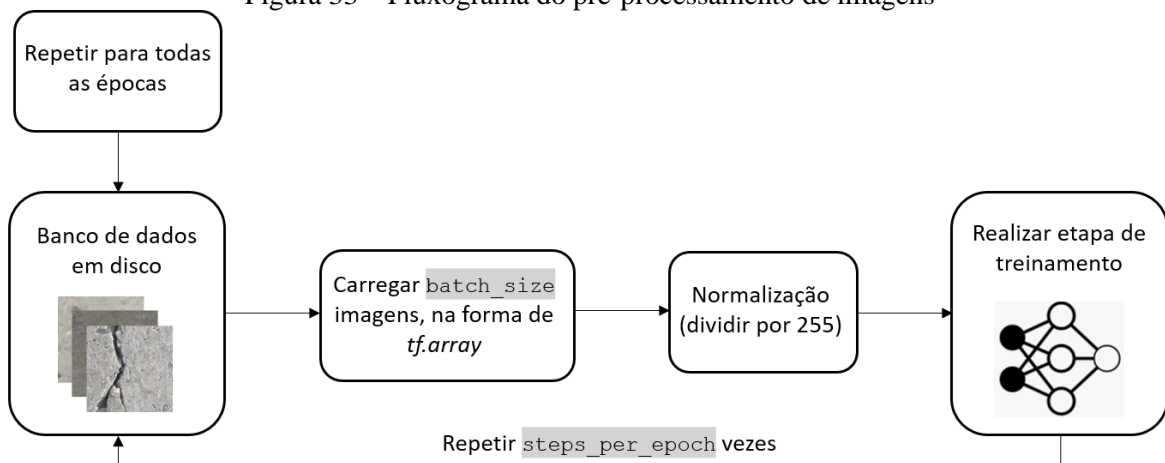
4.4. PRÉ-PROCESSAMENTO

Antes de as imagens serem alimentadas à rede neural para o seu treinamento, é necessário realizar uma etapa de pré-processamento. Essa etapa é realizada para os conjuntos de treino, validação e teste, e consiste em dois passos:

- O primeiro deles é a normalização dos valores dos pixels para que eles variem no intervalo de 0 a 1. Como cada pixel tem valor de intensidade de 0 a 255, basta dividir esses valores por 255. Essa normalização é bastante comum quando se trabalha com imagens, e serve para que os pesos mudem uniformemente ao longo do aprendizado. (Jordan, 2018b).
- O segundo passo consiste no carregamento das imagens. Quando se usa grandes bancos de dados, é impossível carregar todos os dados de treinamento de uma vez, pois há um limite de memória RAM. Assim, é necessário carregar as imagens do disco a medida em que o modelo é treinado.

Portanto, o pré-processamento das imagens da maneira descrita pela Figura 33, a seguir. Estes passos foram implementados usando a API de pré-processamento do *keras*, e o código apresenta-se no anexo 10.2.

Figura 33 – Fluxograma do pré-processamento de imagens



O valor máximo de `batch_size` depende da quantidade de memória RAM disponível. Sabendo que cada imagem resultará em um *array* de dimensões 3x227x227 (3 canais de cor, vezes 227 pixels de altura, vezes 227 pixels de largura), e que cada elemento deste *array* ocupará 4 bytes (float de 4x8 = 32 bits), calcula-se que cada imagem ocupará 600 KB. Assim, um *batch* de 1024 imagens ocuparia 600 MB de memória, por exemplo (equação (31)). Para garantir que em cada época o modelo tenha a chance de usar todas as imagens de treinamento, é usual definir o número de passos por época (`steps_per_epoch`) como o tamanho do conjunto de treinamento dividido pelo tamanho do batch (equação (30)).

$$\text{steps_per_epoch} = \text{dataset_size} / \text{batch_size} \quad (30)$$

$$memória_batch \text{ (bytes)} = altura * largura * profundidade * 4 * (8 \text{ bytes}) \quad (31)$$

No desenvolvimento do modelo, utilizou-se `batch_size` = 1024, o que resulta em valores de `steps_per_epoch` iguais a 28, 6 e 6 para os conjuntos de treino, validação e teste, respectivamente.

4.5. PARÂMETROS DE TREINAMENTO

Este item especifica as configurações utilizadas para o treinamento do modelo.

O otimizador utilizado foi o RMSProp (“model.compile”, no anexo 10.4), que é semelhante ao gradiente descendente com momento, mas calculado de uma maneira ligeiramente diferente. Este otimizador restringe atualizações de pesos que não causem uma alteração grande no erro, o que possibilita que o modelo atinja a convergência mais rápido, mesmo com uma taxa de aprendizado maior (Gandhi, 2018).

Quanto aos hiperparâmetros de aprendizado, foram utilizadas as configurações padrão do *Keras*, que apresentam-se na Tabela 3:

Tabela 3: Hiperparâmetros de aprendizado

Taxa de aprendizado	ρ	momento	ε
0.001	0.9	0.0	1e-07

O modelo é treinado ao longo de 50 épocas, pois foi por volta deste número que a rede desenvolvida por Cha *et al.* (2017) atingiu os melhores resultados.

5. RESULTADOS

Nesta seção, serão apresentados os resultados de treino, validação e teste do modelo treinado, assim como uma discussão sobre o que essas métricas significam.

O modelo foi treinado utilizando como métrica a acurácia de treino. O gráfico da Figura 34 mostra a evolução das acurácias de treino e validação, ao longo de 50 épocas. A Figura 35, por sua vez, mostra predições em uma pequena amostra do conjunto de teste.

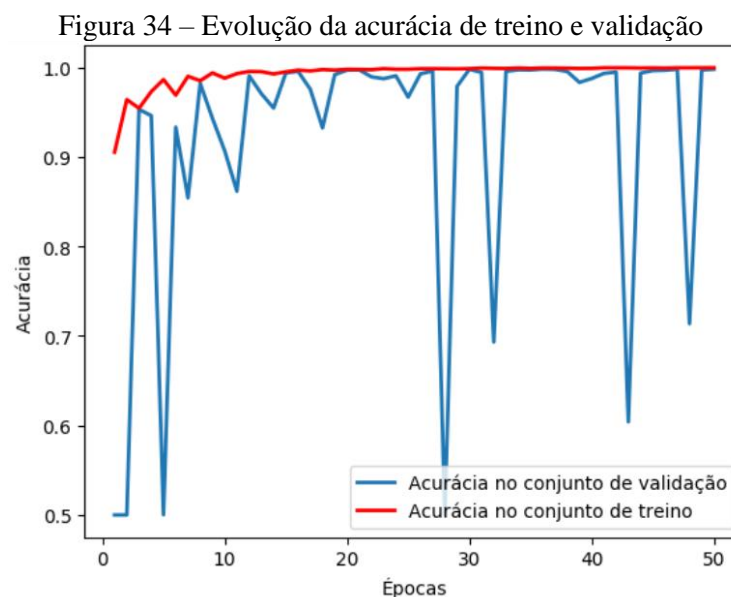
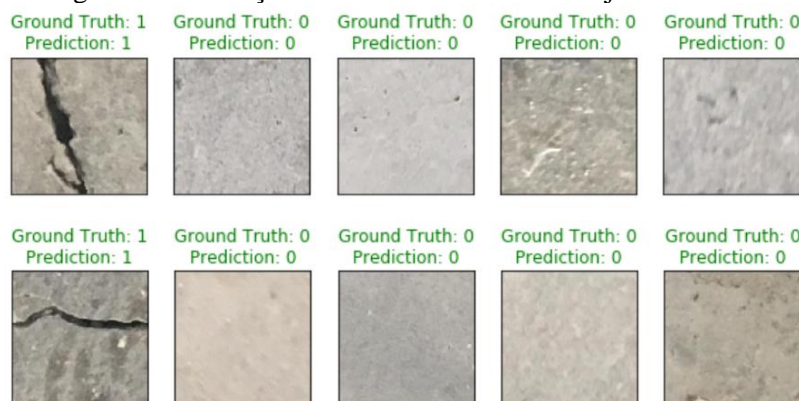


Figura 35 – Predições feitas em amostra do conjunto de teste



Como pode-se notar pelo gráfico acima, o modelo atingiu grandes acurácias em poucas épocas. Provavelmente, isso se deve ao fato de que a presença ou não de fissuras é relativamente fácil de ser aprendida pela rede neural usada, pois redes neurais de porte semelhante são capazes de lidar com problemas de classificação com muito mais categorias e complexidade de características.

Pode-se notar, também, que a acurácia no conjunto de validação oscilou bastante em certos momentos. Em alguns casos, isso pode ser sinal de *overfitting*, caracterizando uma situação em que o modelo aprende muito bem o conjunto de treinamento, mas não generaliza o aprendizado para acertar no conjunto de validação, o que causa uma oscilação na acurácia de validação que tende à aleatoriedade.

Porém, no presente estudo, essa oscilação na acurácia de validação pode ser atribuída ao fato de que, ao longo do aprendizado, o modelo estava variando entre diversas soluções ótimas, sendo que algumas delas não generalizavam bem para o conjunto de validação. Uma solução para este problema seria implementar regularização dos pesos, ou aumentar a razão das camadas de *dropout*.

De qualquer forma, o modelo foi capaz de chegar a uma solução ótima para ambos os conjuntos de teste e validação, com acurácia de 99,95% e 99,78%, respectivamente. A acurácia de teste ao final das épocas foi de 99,8%. Vale lembrar que essas porcentagens foram calculadas usando *batches* de 1024 imagens sobre os conjuntos de treino, validação e teste, que têm 28 mil, 6 mil e 6 mil imagens, respectivamente. Dessa forma, as porcentagens foram calculadas usando 28, 6 e 6 passos, respectivamente. Como os *batches* de imagens são carregados aleatoriamente, as porcentagens não devem ser tomadas como valores exatos, mas garante-se que todas elas estão acima de 99%.

Contudo, as altas acurácias de validação e teste não corresponderam a uma boa performance do modelo em todas as situações, como está explicado no item 7. Suspeita-se que o banco de dados não teve uma boa variedade em termos de abertura e contraste de fissuras. Assim, não se deve tomar essas métricas como bons indicadores da capacidade de generalização do modelo.

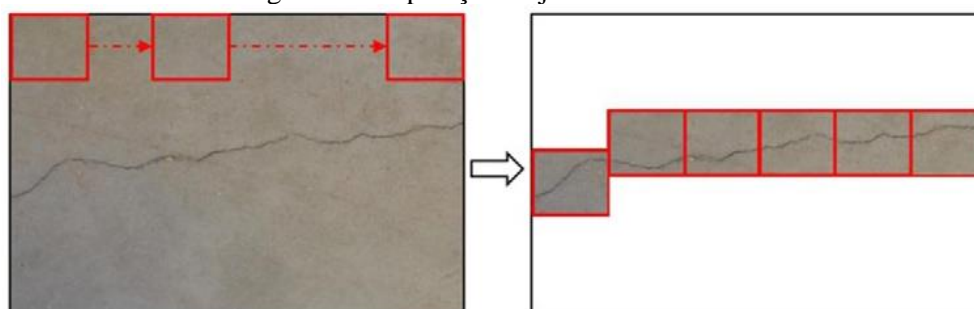
No item 7, apresentam-se os testes em imagens tiradas por celular e encontradas na internet. Desenvolveu-se um aplicativo que usa o modelo treinado para detectar fissuras em fotos de resoluções variadas. O seu desenvolvimento, assim como suas limitações, são discutidos nos itens a seguir.

6. DESENVOLVIMENTO DE APLICATIVO

O modelo treinado trouxe boas métricas de validação, treino e teste. Porém, o desenvolvimento de um aplicativo que use esta rede neural torna possível a sua aplicação em diversas situações diferentes, de uma maneira conveniente.

Para aplicar o modelo em imagens de diversas resoluções e gerar uma detecção mais sofisticada da fissura, adota-se uma solução simples: a ideia é dividir a imagem em diversos recortes e, para cada um deles, usar a rede treinada para fazer uma predição. Com isso, será possível delimitar a região ocupada pela fissura na imagem, conforme apresentado pelas figuras 28 e 36.

Figura 36 – Operação da janela deslizante*

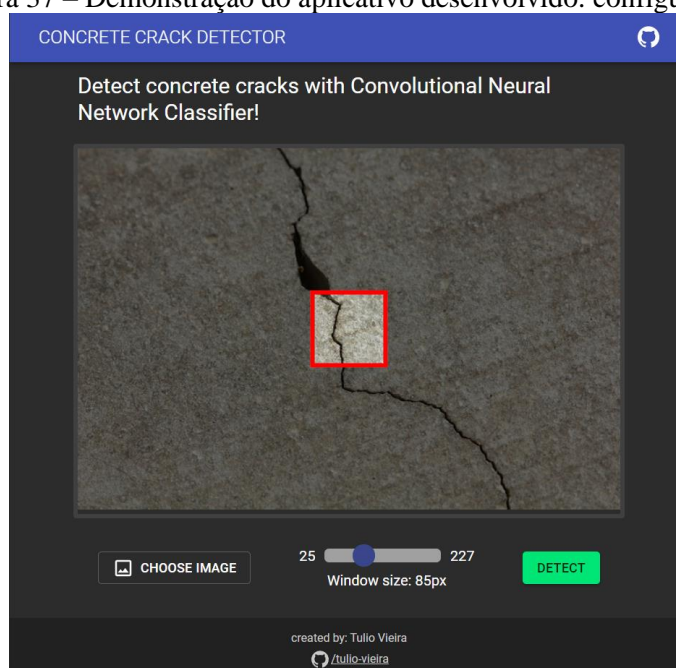


Fonte: Cha *et al.* (2017).

*A figura mostra uma janela com passo igual à sua largura, mas no trabalho presente utiliza-se um passo igual à metade da largura da janela, o que é recomendado por Cha.

O aplicativo implementa esta solução, e funciona da seguinte forma: o usuário escolhe uma foto de uma superfície de concreto com fissuras. Em seguida, escolhe o tamanho da janela deslizante (*sliding window*) que percorrerá a foto. Essa janela deslizante percorre a imagem usando um passo igual a metade da dimensão da janela. Em cada posição, o modelo faz uma predição quanto à ausência ou presença de fissura, e os resultados são apresentados graficamente de maneira a formar uma região de detecção da fissura.

Figura 37 – Demonstração do aplicativo desenvolvido: configurações

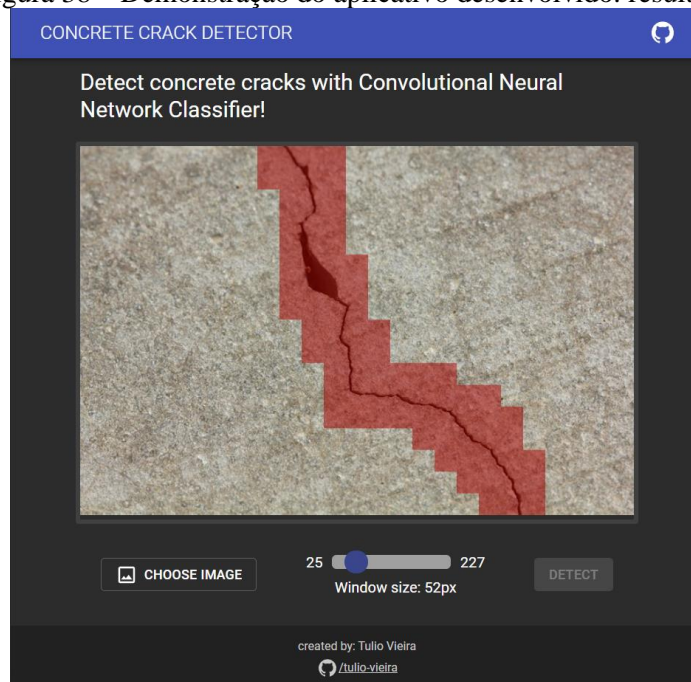


É importante ressaltar que o passo da janela foi usado para possibilitar que as detecções se sobreponham umas às outras, o que evita que uma fissura possa apenas ser encontrada no canto de uma imagem. Esse tipo de situação pode dificultar a detecção (Cha *et al.*, 2017), ainda que o modelo tenha sido treinado com um banco de imagens que contém figuras com fissuras nas bordas.

Para garantir que o modelo funcione com imagens de diversas resoluções e tamanhos de janela, redimensionam-se as seções de imagens geradas pela janela para a resolução de 227×227 pixels, que é a dimensão de entrada da rede neural.

O resultado da detecção é apresentado a seguir:

Figura 38 – Demonstração do aplicativo desenvolvido: resultado



Este aplicativo foi desenvolvido para funcionar no browser, usando *tensorflowjs*. Esta tecnologia possibilitou a conversão do modelo desenvolvido no *keras* para um modelo que possa ser usado em *javascript*, assim como a implementação dos passos de pré-processamento, incluindo a janela deslizante, nesta linguagem. Contudo, encontraram-se bugs do *tensorflowjs* quando tentou-se usar o aplicativo em celulares e máquinas menos potentes, pois esta tecnologia ainda é recente e precisa de mais desenvolvimento.

O aplicativo pode ser acessado neste link: <https://tulio-vieira.github.io/concrete-crack-detector-app>. Recomenda-se o uso do navegador *Google Chrome* para a utilização.

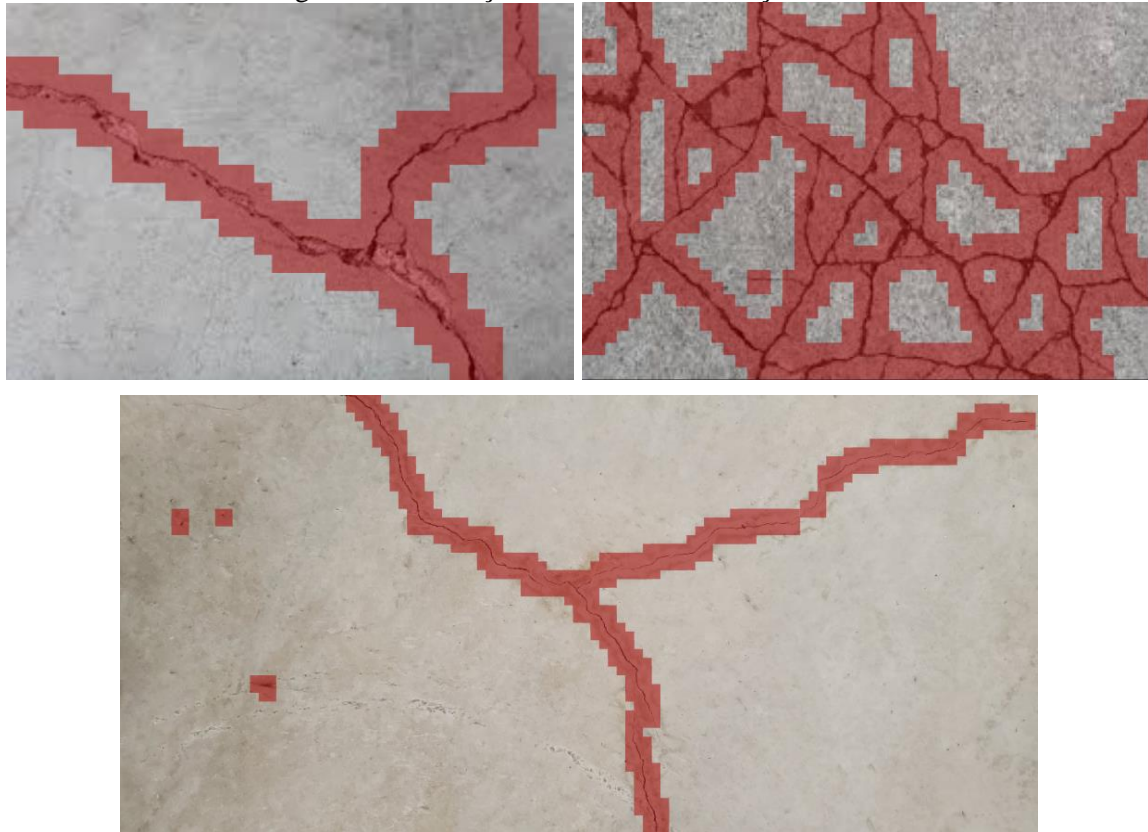
7. LIMITAÇÕES

Nesta seção, discutem-se as limitações de uso do aplicativo desenvolvido e, consequentemente, da rede neural implementada.

Pode-se afirmar que o aplicativo faz um bom trabalho para retratar as situações em que o modelo tem ou não boa performance. Basicamente, quanto mais os recortes gerados pela janela deslizante forem semelhantes às imagens de treino, melhor será a detecção. Assim, o

modelo traz os melhores resultados em fotos de fissura que tenham um contraste nítido entre a fissura e superfície intacta, com escolha de dimensão da janela que retrate uma abertura de fissura semelhante à apresentada na Figura 30. As figuras a seguir retratam essas condições ideais, em que foi possível obter uma boa detecção.

Figura 39 – Detecção de fissura em condições ideais



As cores de fundo não parecem ter efeito na predição, pois o banco de trouxe variedade suficiente de cores de concreto para que o modelo atingisse uma generalização nesse aspecto.

Situações em que a superfície de concreto apresente outros artefatos, como corrosão ou manchas, não resultam em bons resultados, pois o modelo não foi treinado para tais situações. Um exemplo pode ser visto a seguir.

Figura 40 – Detecção de fissura em superfície muito manchada



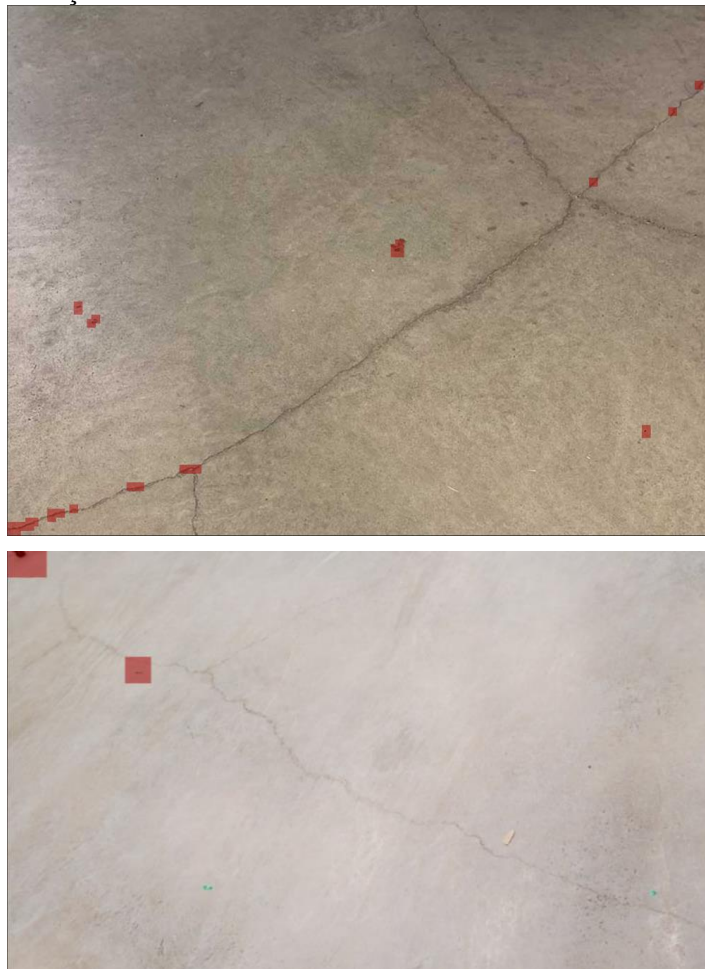
Mesmo pequenas manchas podem gerar falsos positivos, como pode ser visto a seguir.

Figura 41 – Falsos positivos devido a manchas



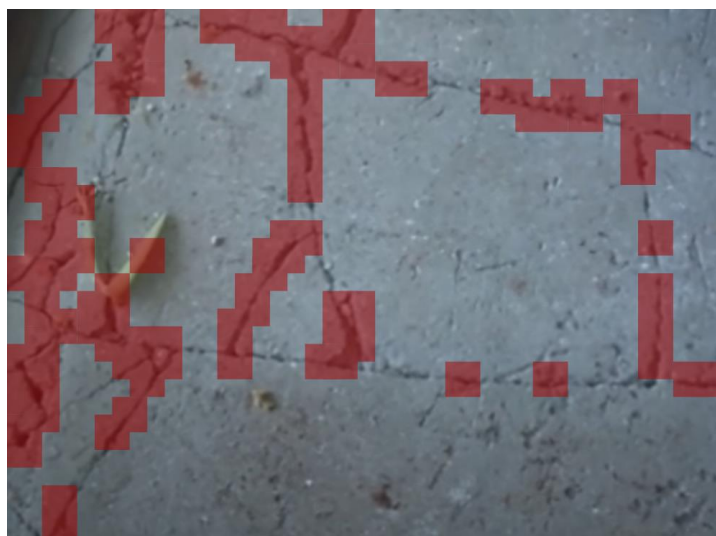
A seguir, apresentam-se exemplos em que o contraste que não foi alto suficiente para uma boa detecção.

Figura 42 – Detecção de fissura em foto com baixo contraste entre fissura e superfície



Também recomenda-se que a figura tenha uma qualidade aceitável, de forma a gerar recortes adequados com dimensões próximas de 227 pixels. O exemplo a seguir mostra um resultado com uma foto de qualidade ruim (embaçada, artefatos de compressão), com uma janela de 35×35 px.

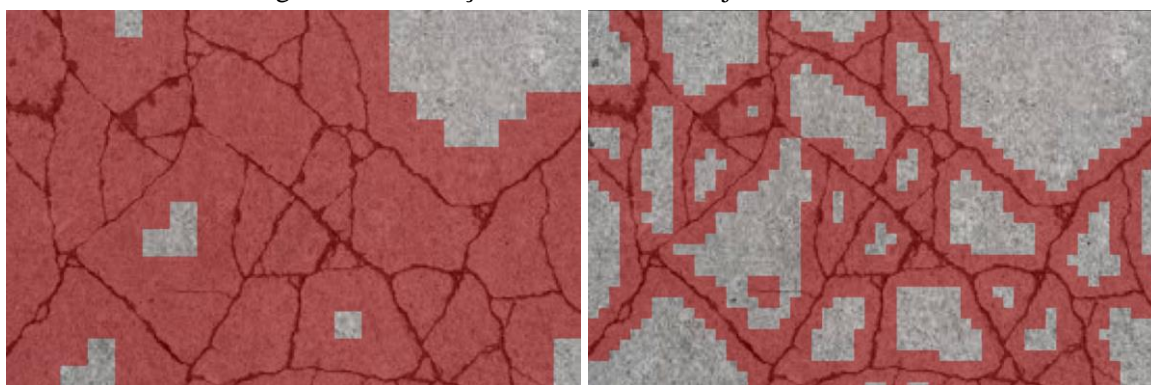
Figura 43 – Detecção de fissura em foto com baixa resolução



Vale notar que o tamanho da janela deslizante tem um efeito direto na qualidade de detecção, e foi por isso que se possibilitou a escolha de sua dimensão no aplicativo. Se a imagem atender todas as condições descritas (contraste, qualidade, ausência de manchas), será possível obter regiões de detecção com maior resolução. Contudo, deve-se lembrar que se a janela for menor que a abertura da fissura, não se terá bons resultados (recomenda-se mostrar abertura de fissura semelhante à apresentada na Figura 30).

Além disso, quanto menor a janela, maior será a chance de falsos positivos devido à detecção de pequenas manchas na imagem. Como essas limitações dependem das características de cada imagem, é difícil determinar um número mínimo de resolução da janela. A Figura 44, a seguir, mostra detecções em uma foto com resolução 600x399px, com tamanhos de janela iguais a 55px (à esquerda) e 22px (à direita).

Figura 44 – Detecção com tamanhos de janela diferentes



Por fim, quanto menor a janela, mais previsões serão necessárias, e maior será o tempo de processamento. No aplicativo, adotou-se uma dimensão mínima variável, apenas para evitar que o tempo de processamento se torne muito longo (limitam-se 1900 previsões, o que é um valor arbitrário, com um valor mínimo de janela de 10px).

Essa limitação de processamento se dá pela simplicidade do algoritmo da janela deslizante. Redes neurais próprias para a detecção de objetos implementam algoritmos complexos para a determinação da posição, de maneira a obter uma performance que possibilita detecção em tempo real (para cada frame de um vídeo ao vivo). Exemplos de redes capazes disso são *Faster R-CNN* (Ren *et. al*, 2017) e *YOLO* (Redmon, 2017).

8. CONCLUSÃO

O presente trabalho procurou demonstrar uma aplicação simples de rede neural (*deep learning*) para detecção de fissura, focando na classificação de recortes da imagem para gerar uma região de detecção.

Usando uma arquitetura relativamente simples, foi possível gerar bons resultados, desde que as condições ideais de detecção estejam presentes. Desenvolveu-se um aplicativo que usa o modelo treinado com uma janela deslizante, o que gera uma localização e delimitação da fissura com uma rede de classificação.

A simplicidade desse algoritmo prejudica a performance, pois é necessário analisar cada recorte da imagem para localizar a fissura. Porém, ele foi adotado apenas para demonstrar o uso da rede neural, e seria necessário utilizar alternativas mais complexas para melhorar o desempenho, como *Faster R-CNN* ou *YOLO*.

Estes resultados ainda podem ser melhorados com o uso de bancos de dados mais completos, que contenham fissuras em superfícies manchadas, com menos contraste, e com aberturas diferentes, que também podem ser usados em conjunto com bancos de dados *open-source*.

Foi possível, também, demonstrar o uso do *tensorflowjs*, que tornou possível o desenvolvimento do aplicativo de detecção de fissuras no navegador. Essa tecnologia facilitou imensamente a produção, pois possibilitou a conversão automática do modelo treinado para o ambiente *javascript*. No momento presente, a equipe de desenvolvimento do *tensorflow* vem lançando várias versões da tecnologia, incluindo versões para micro-controladores e outras linguagens, o que está consolidando ainda mais a sua posição como a plataforma de produção de modelos de *machine learning*.

É importante enfatizar, contudo, que o presente trabalho foca apenas em uma das tarefas relevantes para a detecção, que é a classificação da imagem. Para atingir detecções mais sofisticadas, é necessário implementar modelos capazes de realizarem a localização e segmentação semântica das fissuras. Essas tarefas incluem não só a implementação de redes neurais com arquiteturas complexas, mas também de algoritmos avançados que funcionam em conjunto a essas redes.

Em meio a essa complexidade, diversos autores vêm proposto diversas alternativas, com os artigos mais recentes datando do presente ano de escrita. O estado da arte na área de detecção de fissuras parece estar nas pesquisas de Young-Jin Cha, que desenvolveu um algoritmo capaz de detectar fissuras a nível de pixel em tempo real (Choi, W., e Cha, Y.J., 2020), usando uma arquitetura *auto-encoder*.

Apesar de o presente trabalho não acompanhar os desenvolvimentos mais recentes na área, este ainda apresenta os conceitos básicos de visão computacional e *machine learning*, que são as bases para o desenvolvimento de modelos e algoritmos mais complexos. Também foi possível retratar questões relativas à produção de um modelo de *machine learning*, cujo código e detalhes de implementação podem ser encontrados no repositório *github*: <https://github.com/tulio-vieira/concrete-crack-detector-app>. Os *notebooks* usados para treino, por sua vez, podem ser encontrados aqui: <https://github.com/tulio-vieira/concrete-crack-detector-train>.

9. SUGESTÕES PARA TRABALHOS FUTUROS

Sugestões para trabalhos futuros incluem a implementação de segmentação semântica e detecção de objetos para a detecção de fissuras. Modelos que se destacam nessas tarefas são *Faster R-CNN* e *YOLO network*, com muitos artigos descrevendo as suas aplicações para a detecção de patologias em concreto e pavimentos, dos quais pode-se citar Dung (2019) e Mandal (2018). Outros autores estão focados no desenvolvimento de seus próprios modelos, que é o caso de Choi e Cha (2020) com *SDDNet*, cujos resultados estão entre os mais avançados.

Para trabalhos que foquem na tarefa de classificação, recomenda-se o desenvolvimento de bancos de dados próprios, pois o banco de dados *open-source* não forneceu variedade suficiente de aberturas de fissuras e contraste para detecções em um conjunto grande de situações, frisando também a importância de regularização dos pesos para facilitar o treinamento.

Também pode-se estudar outros métodos de pré-processamento de imagem, como utilização de filtros para aumentar o contraste da imagem antes de ela ser analisada pela rede neural.

10. ANEXOS

10.1. Código para divisão dos conjuntos de treino, validação e teste

```
import os
import numpy as np
import shutil
import random
from tqdm import tqdm

# Creating Train / Val / Test folders (One time use)
root_dir = 'Concrete Crack Images for Classification'
new_dir = 'ozgenel_dataset_split'
classes_dir = ['/Positive', '/Negative']

val_ratio = 0.15
test_ratio = 0.15

for cls in classes_dir:
    os.makedirs(new_dir + '/train' + cls)
    os.makedirs(new_dir + '/val' + cls)
    os.makedirs(new_dir + '/test' + cls)

    # Creating partitions of the data after shuffling
    src = root_dir + cls # Folder to copy images from
    allFileNames = os.listdir(src)
    np.random.shuffle(allFileNames)
    train_FileNames, val_FileNames, test_FileNames =
        np.split(np.array(allFileNames),
                  [int(len(allFileNames)* (1 - val_ratio - test_ratio)),
                   int(len(allFileNames)* (1 - test_ratio))])

    train_FileNames = [src+'/'+ name for name in train_FileNames.tolist()]
    val_FileNames = [src+'/'+ name for name in val_FileNames.tolist()]
    test_FileNames = [src+'/'+ name for name in test_FileNames.tolist()]

    print('Handling class {}/{}'.format(classes_dir.index(cls) + 1,
                                         len(classes_dir)))
    print('Total images: ', len(allFileNames))
    print('Training: ', len(train_FileNames))
    print('Validation: ', len(val_FileNames))
```

```
print('Testing: ', len(test_FileNames))

# Copy-pasting images
for name in tqdm(train_FileNames):
    shutil.copy(name, new_dir + '/train' + cls)

for name in tqdm(val_FileNames):
    shutil.copy(name, new_dir + '/val' + cls)

for name in tqdm(test_FileNames):
    shutil.copy(name, new_dir + '/test' + cls)
```

10.2. Código para pré-processamento de dados

```
from tensorflow.compat.v1.keras.preprocessing.image import ImageDataGenerator
DATASET_DIRECTORY = 'ozgenel_dataset_split'
img_generator = ImageDataGenerator(rescale = 1/255)
train_dataset = img_generator.flow_from_directory(
    DATASET_DIRECTORY + '/train',
    target_size=(227, 227),
    batch_size = 1024,
    class_mode = "categorical")
val_dataset = img_generator.flow_from_directory(
    DATASET_DIRECTORY + '/val',
    target_size=(227, 227),
    batch_size = 1024,
    class_mode = "categorical")
test_dataset = img_generator.flow_from_directory(
    DATASET_DIRECTORY + '/test',
    target_size=(227, 227),
    batch_size = 1024,
    class_mode = "categorical")

# Checar conjuntos de treino, validação e teste
def checkDatasets(train, val):
    batchX, batchy = train.next()
    print('Train Dataset info:\nBatch shape=%s, min=%.3f, max=%.3f' % (batchX.shape, batchX.min(), batchX.max()))
    batchX, batchy = val.next()
    print('Validation Dataset info:\nBatch shape=%s, min=%.3f, max=%.3f' % (batchX.shape, batchX.min(), batchX.max()))

checkDatasets(train_dataset, val_dataset)
```


10.3. Código para definição de arquitetura do modelo

```
# this import statement guarantees that tensorflow 2.0 will behave like 1.x
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

from tensorflow import keras
from tensorflow.keras import layers, activations
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import Dense, Dropout, Flatten, BatchNormalization
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Activation
from tensorflow.keras.preprocessing.image import ImageDataGenerator

model = tf.keras.Sequential()
#C1
model.add(Conv2D(3, (20,20), strides=2, padding='same', input_shape=(227, 227,3))
)
#P1
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(7,7), strides=2))
#C2
model.add(Conv2D(24, (15,15), strides=2, padding='same'))
model.add(BatchNormalization())
#P2
model.add(MaxPooling2D(pool_size=(4,4), strides=2))
#C3
model.add(Conv2D(48, (10,10), strides=2, padding='same'))
model.add(BatchNormalization())

model.add(Dropout(0.5))
model.add(layers.Activation(activations.relu))
#C4
model.add(Conv2D(96, (1,1), strides=1, padding='same'))
model.add(Flatten())
model.add(Dense(96))
model.add(Dense(2, activation='softmax'))
model.summary()
```


10.4. Código para treinamento do modelo

```
# compile the model
model.compile(loss='categorical_crossentropy', optimizer='rmsprop',
              metrics=['accuracy'])
with tf.device('/gpu:0'):
# train the model
    hist = model.fit_generator(
        train_dataset,
        steps_per_epoch=28,
        validation_data=val_dataset,
        validation_steps=6,
        verbose=2,
        epochs=50
    )
```

11. REFERÊNCIAS BIBLIOGRÁFICAS

- ALBAWI, S., MOHAMMED, T. A., AL-ZAWI, S. (2017). *Understanding of a convolutional neural network*. 2017 *International Conference on Engineering and Technology (ICET)*. Disponível em: https://www.researchgate.net/publication/319253577_Understanding_of_a_Convolutional_Neural_Network. Acesso em 01/12/2019.
- BROWNLEE, J. (2016). *Overfitting and Underfitting With Machine Learning Algorithms*. Disponível em: <https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/>. Acesso em 01/12/2019.
- _____. (2019a). *How to Configure the Learning Rate When Training Deep Learning Neural Networks*. Disponível em: <https://machinelearningmastery.com/learning-rate-for-deep-learning-neural-networks/>. Acesso em 01/12/2019.
- _____. (2019b). *How to use Data Scaling Improve Deep Learning Model Stability and Performance*. Disponível em: <https://machinelearningmastery.com/how-to-improve-neural-network-stability-and-modeling-performance-with-data-scaling/>. Acesso em 01/12/2019.
- CHA, Y.-J., CHOI, W., & BÜYÜKÖZTÜRK, O. (2017). *Deep Learning-Based Crack Damage Detection Using Convolutional Neural Networks*. *Computer-Aided Civil and Infrastructure Engineering*, 32(5), 361–378.
- CHOI, W., CHA, Y.J. (2020). *SDDNet: Real-time crack segmentation*. *IEEE Transactions on Industrial Electronics*, (IF: 7.515), 67(9), 8016-8025. Disponível em: <https://ieeexplore.ieee.org/abstract/document/8863123>. Acesso em 14/12/2020.
- CS231n (2015). *CS231n Convolutional Neural Networks for Visual Recognition*. Disponível em: <http://cs231n.github.io/convolutional-networks/#fc>. Acesso em 01/12/2019.
- DUNG, C. V., ANH, L. D. (2018). *Autonomous Concrete Crack Detection Using Deep Fully Convolutional Neural Network*. *Automation in Construction* 99 (2019) 52–58.
- GANDHI, R. *A Look at Gradient Descent and RMSprop Optimizers*. Disponível em <https://towardsdatascience.com/a-look-at-gradient-descent-and-rmsprop-optimizers-f77d483ef08b>. Acesso em 14/12/2020.
- HOCHULI, A. G. (2016). *Redes Neurais Convolucionais*. Disponível em: http://www.inf.ufpr.br/aghochuli/caffe/CNN_PPT.pdf. Acesso em 01/12/2019.
- JORDAN, J. (2018a). *Common architectures in convolutional neural networks*. Disponível em: <https://www.jeremyjordan.me/convnet-architectures/>. Acesso em 01/12/2019.
- _____. (2018b). *Normalizing your data (specifically, input and batch normalization)*. Disponível em: <https://www.jeremyjordan.me/batch-normalization/>. Acesso em 14/12/2020.
- MAIA, R. (2019). *Detecção de Dano Estrutural em Prédios Históricos Utilizando Redes Neurais Artificiais*. Qualificação de Doutorado em Estruturas e Construção Civil, Departamento de Engenharia Civil e Ambiental, Universidade de Brasília, Brasília, DF, 59p.
- MANDAL, V., UONG, L., ADU, G. Y (2018). *Automated road crack detection using deep convolutional neural networks*, in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2018, pp. 5212–5215. Disponível em <https://ieeexplore.ieee.org/abstract/document/8622327>. Acesso em 14/12/2020.
- MELO JÚNIOR, C. M. (2016). *Metodologia para geração de mapas de danos de fachadas a partir de fotografias obtidas por veículo aéreo não tripulado e processamento digital de*

imagens. Tese de Doutorado em Estruturas e Construção Civil. Departamento de Engenharia Civil e Ambiental, Universidade de Brasília, Brasília, DF, 376 p.

NEHME, A. (2018). *Understanding Convolutional Neural Networks*. Disponível em: <https://towardsdatascience.com/understanding-convolutional-neural-networks-221930904a8e>. Acesso em 01/12/2019.

ÖZGENEL, Ç. F. (2018). *Concrete Crack Images for Classification*. Mendeley Data, v1. Disponível em: <https://data.mendeley.com/datasets/5y9wdsg2zt/1>. Acesso em 03/12/2019.

PASQUIER, J. (2018). *Countering Internal Covariate Shift with Batch Normalization*. Disponível em: <https://cai.tools.sap/blog/internal-covariate-shift/>. Acesso em 01/12/2019.

PEREIRA, F. C. (2015). Análise de desempenho de algoritmos para auxílio ao reconhecimento de fissuras em fachadas com revestimento de argamassa visando sua embarcação em VANTs. Dissertação (Mestrado em Engenharia Elétrica). Universidade Federal do Rio Grande do Sul. Porto Alegre, 2015.

REDMON, J., FARHADI, A. (2017). YOLO9000: better, faster, stronger. ArXiv preprint. Disponível em: <https://arxiv.org/abs/1612.08242>. Acesso em 03/12/2019.

REN S., HE. K., GIRSHICK, R., SUN, J. (2017). *Faster R-CNN: towards real-time object detection with region proposal networks*. IEEE Trans. Pattern Anal. Mach. Intell. 39 (6) 1137–1149. Disponível em <https://doi.org/10.1109/TPAMI.2016.2577031>. Acesso em 03/12/2019.

SKALSKI, P. (2019). *Gentle Dive into Math Behind Convolutional Neural Networks - Mysteries of Neural Networks Part V*. Disponível em: <https://towardsdatascience.com/gentle-dive-into-math-behind-convolutional-neural-networks-79a07dd44cf9>. Acesso em 01/12/2019.

SUKHOY, V., STOYTCHEV, A. (2010). *Learning to detect the functional components of doorbell buttons using active exploration and multimodal correlation*. Proc. of IEEE Humanoids, 572–579. Disponível em: https://www.researchgate.net/publication/224211521_Learning_to_detect_the_functional_components_of_doorbell_buttons_using_active_exploration_and_multimodal_correlation. Acesso em 01/12/2019.

TRASK, A. (2019). *Grokking Deep Learning*. Manning Publications, Shelter Island, NY, 301p.

UDACITY (2017). *Deep Learning Nanodegree*. Disponível em: <https://www.udacity.com>. Acesso em 01/12/2019.

VAHID, C. M. (2017). *Convolutional Neural Nets Using MXNet*. AWS Presentation transcript. Disponível em: <https://slideplayer.com/slide/13993644/>. Acesso em: 01/12/2019.

ZHANG, L., YANG, F., ZHANG Y. D., ZHU, Y. J. (2016). *Road Crack Detection Using Deep Convolutional Neural Network*. Image Processing (ICIP), 2016 IEEE International Conference on, IEEE, 2016, September, pp. 3708–3712. Disponível em: <https://doi.org/10.1109/ICIP.2016.7533052>. Acesso em 03/12/2019.

ZHANG, H., TAN, J., LIU, L., WU, Q.J., WANG, Y., JIE, L. (2017). *Automatic Crack Inspection for Concrete Bridge Bottom Surfaces Based on Machine Vision*. Chinese Automation Congress (CAC), IEEE 2017, October, pp. 4938–4943. Disponível em: <https://doi.org/10.1109/CAC.2017.8243654>. Acesso em 03/12/2019.

ZHANG, X., RAJAN, D., & STORY, B. (2019). *Concrete Crack Detection Using Context-Aware Deep Semantic Segmentation Network*. *Computer-Aided Civil and Infrastructure Engineering*, 2019, 1–21.