



GOVERNO DO ESTADO DO RIO DE JANEIRO
SECRETARIA DE ESTADO DE CIÊNCIA E TECNOLOGIA
FUNDAÇÃO DE APOIO À ESCOLA TÉCNICA
CENTRO DE EDUCAÇÃO PROFISSIONAL EM TECNOLOGIA DA INFORMAÇÃO
FACULDADE DE EDUCAÇÃO TECNOLÓGICA DO ESTADO DO RIO DE JANEIRO
FAETERJ/PETRÓPOLIS

***Local Binary Convolutional Neural Network para
Reconhecimento de Expressões Faciais de Emoções
Básicas***

Alexandra Miguel Raibolt da Silva

Petrópolis - RJ

Dezembro, 2018

Alexandra Miguel Raibolt da Silva

***Local Binary Convolutional Neural Network para
Reconhecimento de Expressões Faciais de Emoções
Básicas***

Trabalho de Conclusão de Curso apresentado à Coordenadoria do Curso de Tecnólogo em Tecnologia da Informação e da Comunicação da Faculdade de Educação Tecnológica do Estado do Rio de Janeiro Faeterj/Petrópolis, como requisito parcial para obtenção do título de Tecnólogo em Tecnologia da Informação e da Comunicação.

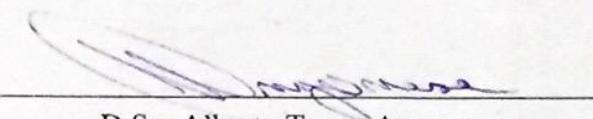
Orientador:
Alberto Torres Angonese

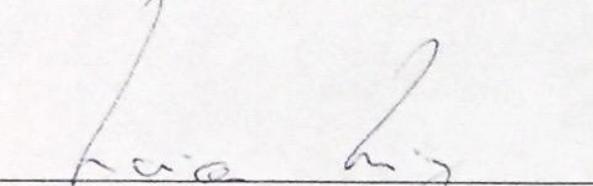
Petrópolis - RJ

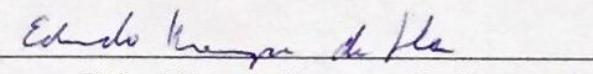
Dezembro, 2018

Folha de Aprovação

Trabalho de Conclusão de Curso sob o título “*Local Binary Convolutional Neural Network para Reconhecimento de Expressões Faciais de Emoções Básicas*”, defendida por Alexandra Miguel Raibolt da Silva e aprovada em 3 de Dezembro de 2018, em Petrópolis - RJ, pela banca examinadora constituída pelos professores:

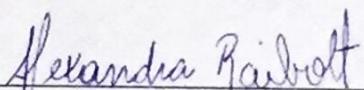

D.Sc. Alberto Torres Angonese
Orientador


M.Sc. Laion Luiz Fachini Manfroi
Faculdade de Educação Tecnológica do Estado
do Rio de Janeiro Faeterj/Petrópolis


D.Sc. Eduardo Krempser da Silva
Fundação Oswaldo Cruz (Fiocruz)

Declaração de Autor

Declaro, para fins de pesquisa acadêmica, didática e técnico-científica, que o presente Trabalho de Conclusão de Curso pode ser parcial ou totalmente utilizado desde que se faça referência à fonte e aos autores.



Alexandra Miguel Raibolt da Silva

Petrópolis, em 3 de Dezembro de 2018

Dedicatória

Dedico esta, aos meus pais Fátima e Alexandre, e ao meu companheiro Diego que, com muito incentivo e apoio constante, me ajudaram a chegar onde estou.

Agradecimentos

Primeiramente a Deus que me permitiu chegar onde estou.

Aos meus pais Fátima e Alexandre, e ao meu companheiro Diego, pelo amor, pelo incentivo e pelo apoio constante.

À Instituição – funcionários, direção, administração e seu corpo docente que apesar das inúmeras lutas e dificuldades, se mantém firme nos ideais em que acredita para proporcionar um ensino superior público de qualidade. Em especial aos professores D.Sc. Sicilia Judice e M.Sc. Laion Luiz Manfroi pelo suporte, pelo apoio e pelo carinho, vocês me inspiram!

E por último, mas não menos importante, ao professor D.Sc. Alberto Angonese, meu orientador, pela oportunidade, pela compreensão, pela amizade e pelo incentivo que garantiram a conclusão deste trabalho.

Epígrafe

"If knowledge can create problems, it is
not through ignorance that we can solve
them"

(Isaac Asimov)

Resumo

Há pouco tempo, com os esforços voltados para os avanços tecnológicos direcionados aos *hardwares*, os recursos da Inteligência Artificial (em inglês, *Artificial Intelligence* - AI) ganham força e, com isso, novas técnicas e métodos estão sendo aplicadas cada vez mais em nosso dia-a-dia. Assim sendo, algoritmos de Aprendizado Profundo (em inglês, *Deep Learning* - DL), como as Redes Neurais Convolucionais (em inglês, *Convolutional Neural Networks* - CNN), vêm sendo empregadas em sistemas de detecção de objetos, de reconhecimento de pessoas, de classificação de faces, etc. Neste trabalho, propusemos o uso da Rede Neural Convolucional Binária Local (em inglês, *Local Binary Convolutional Neural Network* - LBCNN) para a tarefa de reconhecimento de expressões faciais (em inglês, *Facial Expressions Recognition* - FER). O LBCNN é uma técnica que unifica o descritor de texturas Padrão Binário Local (em inglês, *Local Binary Pattern* - LBP) com uma arquitetura de CNN. Na abordagem LBCNN, em vez da otimização dos filtros convolucionais, o aprendizado otimiza os pesos lineares aprendíveis, reduzindo a complexidade do modelo. Para a tarefa FER, o modelo de treinamento foi gerado por meio de três bases de imagens de expressões faciais públicas, nos quais foi aplicado o processo de *Data Augmentation* para melhorar a representatividade das características extraídas e evitar *Overfitting*. O LBCNN implementado neste trabalho foi desenvolvido em *Python* usando o framework *TensorFlow* em uma arquitetura baseada em GPU (em inglês, *Graphics Processing Unit*). Com inúmeras aplicações do mundo real, a tarefa de FER é extensivamente um desafio. A ideia em utilizar o LBCNN para a tarefa de reconhecimento de expressão facial é produzir um modelo com desempenho computacional eficiente, bem como reduzir a complexidade computacional, o que motiva a integração com uma plataforma robótica autônoma em um ambiente real.

Abstract

Not long ago, with efforts focused on technological advancements aimed at hardware, the capabilities of Artificial Intelligence gain strength and, with this, new techniques and methods are being applied more and more in our day-to-day. Thus, Deep Learning algorithms, such as Convolutional Neural Networks, have been used in systems for detecting objects, recognizing people, classifying faces, and so on. In this paper, we proposed the use of Local Binary Convolutional Neural Network (LBCNN) for the facial expressions recognition (FER) task. The LBCNN is a technique that unifies the Local Binary Patterns (LBP) feature descriptor with a Convolutional Neural Network (CNN) architecture. In the LBCNN approach, instead of the optimization of the convolutional filters, the learning optimizes the learnable linear weights, reducing the model's complexity. For the FER task, the training model was generated using three databases of public facial expression images, on which the Data Augmentation process was applied in order to improve the representativity of the extracted features and to prevent Overfitting. The LBCNN implemented in this work was developed in Python using the TensorFlow framework in a GPU-based architecture. With numerous of real-world applications, FER is extensively a challenge. The idea in use LBCNN for the FER task is to produces a model with efficient computational performance, as well as to reduce the computational complexity, which motivates the integration with an autonomous robotic platform in a real environment.

Lista de Figuras

1.1	As seis expressões faciais de emoções básicas.	p. 17
(a)	Raiva.	p. 17
(b)	Nojo.	p. 17
(c)	Medo.	p. 17
(d)	Feliz.	p. 17
(e)	Triste.	p. 17
(f)	Surpreso.	p. 17
2.1	Principais tipos de Aprendizado de Máquina.	p. 23
2.2	Exemplo de clusterização.	p. 25
2.3	O Neurônio Biológico.	p. 29
2.4	O modelo de McCulloch-Pitts.	p. 29
2.5	Conjunto de dados linearmente separável.	p. 31
2.6	Arquitetura do modelo <i>Single-Layer Perceptron</i>	p. 32
2.7	Problema XOR: Conjunto de dados não-linearmente separável.	p. 32
2.8	Arquitetura básica do modelo <i>Multilayer Perceptron</i>	p. 34
2.9	Representação de uma matriz de valores de pixel.	p. 36
(a)	Imagen.	p. 36
(b)	Área 12x12 recortada da imagem (a).	p. 36
(c)	Matriz de valores de pixel da imagem (b).	p. 36
2.10	Matriz representativa de uma imagem com dimensão 10x10x3.	p. 37
2.11	Resultado da convolução.	p. 38
(a)	Imagen de entrada.	p. 38

(b)	Resultado da convolução.	p. 38
2.12	Operação de Max-Pooling com um filtro 2×2 e <i>stride</i> 2.	p. 39
2.13	Arquitetura básica do modelo <i>CNN</i> .	p. 40
2.14	Exemplo de formulação do LBP.	p. 42
2.15	Resultado do LBP.	p. 42
(a)	Imagen de entrada.	p. 42
(b)	LBP resultante.	p. 42
2.16	Histograma de intensidade dos pixels.	p. 43
2.17	Reformulação do LBP através de filtros convolucionais.	p. 44
3.1	Principais produtos oferecidos pelo GCP.	p. 47
3.2	Amostras presentes nas três bases de imagens.	p. 49
3.3	Resultado do processo de <i>Data Augmentation</i> .	p. 50
(a)	Imagen de entrada.	p. 50
(b)	Sub-amostras.	p. 50
3.4	Arquitetura básica do modelo <i>Local Binary Convolutional Neural Network</i> .	p. 55
4.1	Matrix de Confusão da CNN convencional.	p. 60
(a)	JAFFE.	p. 60
(b)	Extended-CK+.	p. 60
(c)	FER-2013.	p. 60
4.2	Matrix de Confusão do LBCNN.	p. 60
(a)	JAFFE.	p. 60
(b)	Extended-CK+.	p. 60
(c)	FER-2013.	p. 60

Lista de Tabelas

2.1	Exemplo de FrequênciA Absoluta.	p. 43
3.1	Conjunto de dados de treinamento e teste da base de imagens JAFFE.	p. 51
3.2	Conjunto de dados de treinamento e teste da base de imagens Extended CK+. . .	p. 51
3.3	Conjunto de dados de treinamento e teste da base de imagens FER-2013.	p. 51
3.4	Rótulos Categóricos x Valores Numéricos.	p. 53
3.5	Valores Numéricos x Codificação <i>One-Hot</i>	p. 54
4.1	Taxa de acurácia (porcentagem) na etapa de teste.	p. 58
4.2	Tempo gasto (horas) na etapa de treinamento.	p. 58
4.3	Custo (dólar) x tempo (horas) de execução por mês.	p. 59
4.4	Custo (dólar) x tempo (horas) de execução por hora.	p. 59
4.5	Custo efetivo ao executar o modelo LBCNN em comparação ao modelo de CNN convencional.	p. 59

Lista de Abreviaturas e Siglas

AI	<i>Artificial Intelligence</i>
APA	<i>American Psychological Association</i>
API	<i>Application Programming Interface</i>
BN	<i>Bayesian Network</i>
CLBP	<i>Compound Local Binary</i>
CNN	<i>Convolutional Neural Network</i>
CS-LBP	<i>CS-Center Symmetric</i>
CV	<i>Computer Vision</i>
DL	<i>Deep Learning</i>
DTR	<i>Decision Tree Regression</i>
Extended CK+	<i>Extended Cohn-Kanade</i>
FER	<i>Facial Expression Recognition</i>
FER-2013	<i>Facial Expression Recognition 2013</i>
GCE	<i>Google Compute Engine</i>
GPS	<i>Global Positioning System</i>
GPU	<i>Graphics Processing Unit</i>
HC	<i>Hierarchical Clustering</i>
IBM	<i>International Business Machines</i>
ICML	<i>Challenges in Representation Learning</i>
IFR	<i>International Federation of Robotics</i>
IaaS	<i>Infrastructure as a Service</i>
IoT	<i>Internet of Things</i>
JAFFE	<i>Japanese Female Facial Expression</i>
K-NN	<i>K-Nearest Neighbors</i>
LBC	<i>Local Binary Convolution</i>
LBCNN	<i>Local Binary Convolutional Neural Network</i>
LBP	<i>Local Binary Pattern</i>
ML	<i>Machine Learning</i>
MLP	<i>Multilayer Perceptron</i>
MLR	<i>Multiple Linear Regression</i>
MRELBP	<i>Median Robust Extended Local Binary Pattern</i>
NLP	<i>Natural Language Processing</i>
NN	<i>Neural Network</i>
PR	<i>Pattern Recognition</i>
RFR	<i>Random Forest Regression</i>
RGB	<i>Red, Green, Blue</i>
ROI	<i>Region of Interest</i>
ReLU	<i>Rectified Linear Units</i>
SLP	<i>Single-Layer Perceptron</i>

SVM *Support Vector Machine*

UCB *Upper Confidence Bound*

Sumário

1	Introdução	p. 16
1.1	Motivação	p. 18
1.2	Justificativa	p. 18
1.3	Objetivos	p. 20
1.3.1	Objetivo Geral	p. 20
1.3.2	Objetivo Específico	p. 20
1.4	Organização dos capítulos seguintes	p. 20
1.5	Notas sobre tradução e terminologia	p. 21
2	Fundamentação Teórica	p. 22
2.1	Aprendizado de Máquina	p. 22
2.1.1	Tipos de Aprendizado de Máquina	p. 23
2.1.2	Redes Neurais Artificiais	p. 27
2.2	Aprendizado Profundo	p. 34
2.2.1	Redes Neurais Convolucionais	p. 34
2.3	Descritores de Texturas	p. 40
2.3.1	Padrão Binário Local	p. 41
2.3.2	Reformulação do LBP através de filtros convolucionais	p. 44
3	Implementação	p. 46
3.1	Ferramentas	p. 46
3.1.1	<i>Google Cloud Platform</i>	p. 47

3.1.2	Dependências Utilizadas	p. 48
3.2	Base de Imagens	p. 48
3.3	Pré-processamento	p. 50
3.3.1	Formato .pkl	p. 52
3.3.2	Codificação <i>One-Hot</i>	p. 53
3.4	Rede Neural Convolucionial Binária Local	p. 54
4	Experimentos e Resultados	p. 57
5	Considerações Finais	p. 61
5.1	Trabalhos Futuros	p. 61
	Reconhecimentos	p. 63
	Referências	p. 64
	Apêndice A – Código Fonte do processo de reformulação do LBP através de filtros convolucionais	p. 71
	Apêndice B – Código Fonte da implementação do LBCNN proposto	p. 77
	Apêndice C – Código Fonte do processo de <i>Data Augmentation</i>	p. 95
	Apêndice D – Código Fonte do processo de ROI	p. 97
	Apêndice E – Código Fonte do módulo dateP para serialização com <i>Pickle</i>	p. 99
	Apêndice F – Código Fonte do processo de serialização com <i>Pickle</i>	p. 102

1 *Introdução*

Incorporando inúmeras perspectivas, como nos campos da psicologia, neurociência, linguística, psiquiatria, educação, entre outros, emoções e suas expressões vêm sendo alvo de grandes teorias e pesquisas por teóricos, pesquisadores e psicólogos [1, 2, 3, 4, 5, 6] ao tentar desvendar como ocorrem as emoções, compreender sua natureza, e como tais emoções podem afetar o comportamento humano.

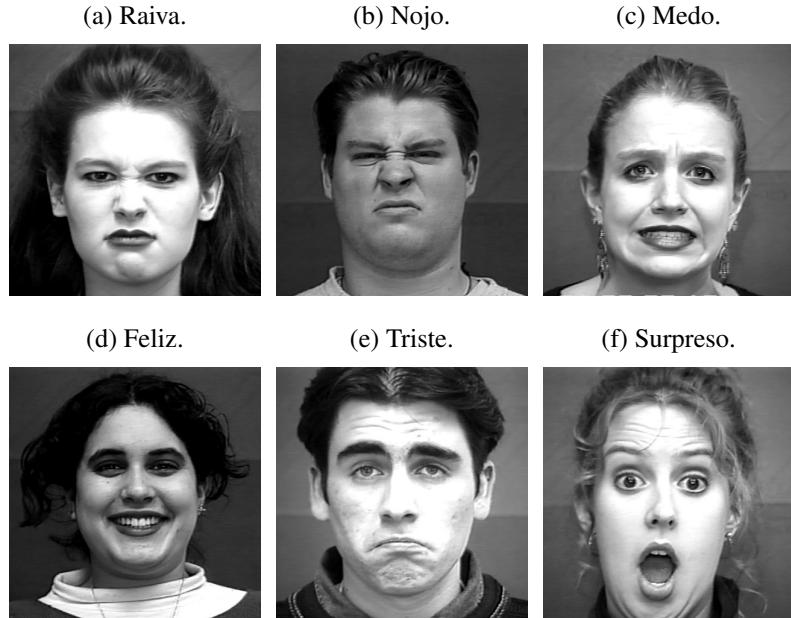
Quando falamos de emoções e suas expressões, podemos levar em consideração diversos traços encontrados no comportamento humano que refletem na expressão emocional de um indivíduo [6], tais expressões emocionais podem ser: expressões vocais, expressões corporais, aceleração dos batimentos cardíacos, suor excessivo, sem contar as expressões faciais que são um meio de comunicação não verbal durante o processo de comunicação, que fornece informações de um indivíduo sobre suas intenções, desejos, objetivos, estado de espírito, etc.

Pioneiro na análise das emoções e expressões faciais, o psicólogo Paul Ekman foi eleito em 2002 pela *American Psychological Association* (APA) como sendo um dos 100 psicólogos mais notáveis do século XX [7], além de, em 2009, ser eleito também como uma das 100 pessoas mais influentes pela revista Time [8]. Seis emoções básicas ou universais (Raiva, Nojo, Medo, Alegria, Tristeza, Surpresa), foram foco de estudo de Ekman [9], e consideradas por ele como sendo as emoções comuns aos seres humanos, independente de fatores culturais. As respectivas expressões faciais de emoções básicas consideradas por Ekman podem ser vistas na Figura 1.1.

É comum, entretanto, encontrar na literatura modelos teóricos com uma quantidade diferente de emoções básicas [3] ou possuindo variedades em relação a elas. Porém, as seis emoções básicas consideradas por Ekman são utilizadas por diversos pesquisadores [10, 11, 12, 13], mostrando assim, sua relevância e influência no âmbito de pesquisa.

Realizar a tarefa de reconhecimento de objetos, reconhecimento facial, reconhecimento de expressões faciais, torna-se uma tarefa fácil e corriqueira nas relações interpessoais dos seres humanos. Para o cérebro humano, tal tarefa é considerada extremamente fácil, porém, computacionalmente tal tarefa apresenta um alto grau de complexidade.

Figura 1.1: As seis expressões faciais de emoções básicas.



Fonte: Adaptado de [10, 11].

Reconhecimento automatizado de objetos, facial, e de expressões faciais é um importante desafio no campo da Visão Computacional (em inglês, *Computer Vision - CV*) devido à grande necessidade encontrada na interação robô-humano por sistemas de reconhecimento automatizados. Por conta de tais necessidades que surgem junto ao mercado, têm-se avançado surpreendentemente com o surgimento de novas técnicas capazes de oferecer resultados satisfatórios que dispõem de incontáveis aplicações do mundo real, como Robótica Assistiva [14], gerenciamento de estoque, sistema de vigilância [15], controle automático de acesso, detecção de transtornos mentais [16], autenticação, etc.

O crescente estudo de Aprendizado de Máquina (em inglês, *Machine Learning - ML*) e Reconhecimento de Padrões (em inglês, *Pattern Recognition - PR*) nas últimas duas décadas para solucionar tarefas de CV, conduziu pesquisas em busca de técnicas de extração de recursos de imagens. Tais técnicas como Máquina de Vetores de Suporte (em inglês, *Support Vector Machine - SVM*) [17, 18], Redes Bayesianas (em inglês, *Bayesian Networks - BN*) [19], Redes Neurais Convolucionais (em inglês, *Convolutional Neural Networks - CNN*) [20], apresentam bons resultados no processo de extração de características e padrões de classificação em imagens de faces para o reconhecimento e classificação de expressões faciais.

1.1 Motivação

As arquiteturas de CNN vêm conquistando espaço em desafios de reconhecimento e classificação de imagens a partir do ano de 2012 no desafio *Imagenet* [21], onde, originalmente a arquitetura foi proposta por LeCun [22]. A partir de estudos, pesquisas e aprimoramentos, hoje, existem diferentes modelos de arquiteturas de CNN, e seus resultados as tornam estado-da-arte para a resolução de problemas na área de CV. Tais arquiteturas em destaque são: *AlexNet*, *Inception*, *ResNet*. Além da variedade de arquiteturas disponíveis, as CNN avançaram também em outros aspectos, tais como: função de ativação, design de camadas, otimização, etc. Entretanto, um problema ainda enfrentado ao treinar tais arquiteturas, está relacionado ao poder computacional necessário, onde, torna-se um recurso caro, ou até mesmo indisponível.

Em contrapartida, a arquitetura do LBCNN [23] demonstra um desempenho computacional eficiente, com sua complexidade computacional reduzida em comparação a outras arquiteturas de CNN. Sob essa ótica, o objetivo central deste trabalho concentra-se na implementação de um sistema, utilizando como extrator de características uma adaptação do LBCNN que seja capaz de realizar a tarefa de reconhecimento e classificação de expressões faciais básicas. Espera-se alcançar resultados satisfatórios que serão posteriormente compartilhados com a comunidade acadêmica. A partir dos resultados obtidos, será analisada também a viabilidade de integrar o sistema proposto, que terá como objetivo agregar a funcionalidade da tarefa de FER para a área de computação Assistiva no robô apresentado em [24, 25].

1.2 Justificativa

Pesquisas realizadas em 2017 pela Federação Internacional de Robótica (em inglês, *International Federation of Robotics - IFR*) [26] apontam que no ano de 2016, o principal cliente consumidor de robôs industriais foi a indústria automotiva com uma taxa de participação de 35% da oferta total para o mesmo ano, enquanto a indústria eletroeletrônica atingiu uma taxa de 31% de participação da oferta total, tornando-se o principal cliente consumidor do mercado asiático (China, Japão, República da Coréia).

Por regiões, o mercado com o crescimento mais potente do mundo com um aumento contínuo de vendas de robôs industriais é a Ásia, que no ano de 2016 vendeu um total de 190.492 unidades, onde teve um aumento de 19% nas vendas. Vale a pena ressaltar que para os últimos quatro anos consecutivos, este foi o maior nível atingido de vendas já registrados.

Em 2016, os cinco países que representaram 74% do volume total de vendas globais de

robôs no mundo eram: China, República da Coréia, Japão, Estados Unidos e Alemanha. Enquanto que no Brasil a venda de robôs, para o mesmo ano, diminuiu de 1.407 unidades vendidas em 2015, para 1.207 unidades em 2016. A partir dos dados apresentados acima, é possível observar uma discrepância na comparação do Brasil com outros países. Isto se deve a aspectos socioeconômicos, ao fato de países desenvolvidos investirem no seu desenvolvimento tecnológico, apoiando financeiramente a educação e a pesquisa, enquanto no Brasil há déficits no apoio e investimento no desenvolvimento educacional, científico e tecnológico do país.

Ainda segundo a IFR [27], no que diz respeito a robôs de serviços, no ano de 2016 foram vendidas 59.706 unidades ao redor do mundo, enquanto que em 2015 este número era de 48.018 unidades vendidas. Portanto, as vendas de robôs de serviços tiveram um aumento de 24% entre os anos de 2015 a 2016.

Entre estes robôs de serviços, é possível destacar as seguintes categorias como sendo fortes setores em crescimento com vendas iguais ou superiores a 1.600 unidades registradas no ano de 2016: robôs com aplicações de defesa, veículos guiados automatizados para ambientes de fabricação e para ambientes não industriais, robôs para pecuária e agricultura, robôs médicos e exoesqueletos humanos.

Além disso, robôs de serviço para uso pessoal e doméstico (robôs de entretenimento e lazer - tais como robôs brinquedos, robôs cortadores de grama, sistemas de educação e pesquisa, etc.) atingiram no ano de 2016 a marca de aproximadamente 6,7 milhões de unidades vendidas. Entretanto, há categorias de robôs de serviços em que suas vendas unitárias não atingiram a marca de 1.000 unidades registradas no ano de 2016, tais categorias são: robôs de limpeza, robôs de construção e demolição, aplicações de resgate e segurança.

A partir dos dados apresentados acima, em um contexto mundial é possível observar que existe uma carência de tecnologias voltadas para o desenvolvimento de soluções para a Robótica Assistiva, além de aplicações de resgate e segurança.

Portanto, devido a carência tecnologia de uma produção brasileira de tecnologias voltadas para o desenvolvimento de soluções para a área da Robótica, se faz necessário o estudo de métodos e técnicas que contribuam de alguma forma para suprir tal carência. No caso deste trabalho, temos como objeto de estudo a análise de expressões faciais de emoções básicas através do uso de uma adaptação de arquitetura de CNN - o LBCNN - que aplicada a plataformas robóticas autônomas possa trazer valor acadêmico e científico para a comunidade acadêmica brasileira além de atender demandas da sociedade em vários aspectos, tais como: auxílio doméstico e engenharia de defesa.

1.3 Objetivos

1.3.1 Objetivo Geral

O objetivo geral deste trabalho constitui-se na implementação de um sistema capaz de realizar a tarefa de reconhecimento e classificação de expressões faciais básicas utilizando como extrator de características uma adaptação do LBCNN.

1.3.2 Objetivo Específico

São definidos como objetivos específicos:

- Estudo do estado-da-arte das CNN aplicadas em tarefas da CV e PR e do estado-da-arte das técnicas de extração de características;
- Implementação da adaptação do modelo de LBCNN capaz de realizar a tarefa de reconhecimento e classificação de expressões faciais básicas;
- Realização de testes em três bases de imagens de expressões faciais públicas;
- Validação e análise entre os resultados obtidos (precisão de classificação, consumo computacional, etc.) em comparação a uma CNN convencional e outros métodos do estado-da-arte.

1.4 Organização dos capítulos seguintes

Este Trabalho de Conclusão de Curso está organizado da seguinte forma: no **Capítulo 2** é apresentado uma visão geral da fundamentação teórica necessária para compreender os conceitos envolvidos das técnicas utilizadas neste trabalho. No **Capítulo 3** é apresentado a implementação para realizar a tarefa de reconhecimento e classificação de expressões faciais básicas utilizando um modelo de LBCNN adaptada proposta neste trabalho. Em seguida, no **Capítulo 4** são apresentados os experimentos realizados e os resultados obtidos pelo modelo proposto. Finalmente, no **Capítulo 5** é apresentado as considerações finais do trabalho e discussões sobre trabalhos futuros.

1.5 Notas sobre tradução e terminologia

Termos encontrados neste trabalho, especificamente *hardwares*, *insight*, *Soma*, *cluster*, *e-commerce*, *Perceptron*, *if*, *otherwise*, *soft-max*, *Overfitting*, *framework*, *on demand*, *expertise*, *streams*, *multiclass*, etc., são alguns termos em inglês que não foram traduzidos para a língua portuguesa pela falta de tradução que expresse adequadamente o mesmo significado do termo em inglês.

Alguns termos foram traduzidos para a língua portuguesa, porém, independente do termo traduzido ou não, optou-se por utilizar os acrônimos que correspondem aos termos em inglês, como: AI, APA, API, BN, CLBP, CNN, CS-LBP, CV, DL, Extended CK+, FER, FER-2013, GCE, GPS, GPU, HC, IBM, ICML, IFR, IaaS, IoT, JAFFE, K-NN, LBC, LBCNN, LBP, ML, MLP, MLR, MRELBP, NLP, ANN, PR, RGB, ROI, ReLU, SLP, SVM, UCB, etc. pelo fato de serem termos já difundidos entre pesquisadores e a comunidade científica.

Nomes como: *Imagenet*, *AlexNet*, *Inception*, *ResNet*, *Microsoft*, *Google*, *Facebook*, *Spotify*, *Amazon*, *Apriori*, *Eclat*, *Partition*, *FP-Growth*, *Bio-Summ*, *GistSUMM*, *Q-learning*, *Python*, *TensorFlow*, *YouTube*, *Coca-Cola*, *Snapchat*, *Ocado*, *Philips*, *Wix.com*, *Numpy*, *Scipy*, *SciKit-Learn*, *Matplotlib*, *GitHub*, *Adam Optimizer*, *Stochastic Gradient Descent*, *Pickle*, *Data Augmentation*, *One-Hot*, *Precision-Recall*, *Dropout*, *TensorBoard*, etc. não foram traduzidos por se tratarem de nomes próprios.

Vale ressaltar que as considerações mencionadas não se tratam de forma alguma, de desrespeito à língua portuguesa, e a utilização desses termos e acrônimos foram somente empregados pelas justificações apresentadas acima.

2 Fundamentação Teórica

Neste capítulo serão apresentados os conceitos teóricos que dão embasamento para o desenvolvimento deste trabalho.

2.1 Aprendizado de Máquina

Aprendizado de Máquina (em inglês, *Machine Learning*) é uma subárea da AI que tem como propósito automatizar máquinas através da criação de modelos analíticos. Tal ideia se baseia na premissa de que máquinas têm a habilidade de aprender, tomar decisões, extrair *insights* e reconhecer padrões (através de dados) sem a necessidade de serem explicitamente programadas, ou seja, sem interferência humana.

Os modelos aprendem a partir de experiências acumuladas que se dá através de soluções bem-sucedidas de cálculos anteriores, logo, possuem a capacidade de se adaptar de forma independente a medida em que são apresentados a novos dados. Por conta disto, é possível que os modelos de ML produzam resultados e decisões assertivas.

O ML vem se tornando popular nos últimos anos devido ao volume de dados que hoje temos disponíveis, além da variedade dos dados (tais como: textos, imagens, banco de dados tradicionais, áudios, GPS (em inglês, *Global Positioning System*), vídeos, movimentações financeiras, atividades em redes sociais).

Outros fatores que impulsionaram a popularização da implementação de modelos de ML na atualidade - dentre tantos - é a disponibilidade ao armazenamento de dados, e a disponibilidade de processamento robusto em geral, que pode auxiliar no processamento computacional de forma rápida e poderosa.

Assim sendo, ao desenvolver modelos de ML é possível realizar análises em um grande conjunto de dados complexo de forma automática e rápida.

Devido à grande aplicabilidade disponível, sendo possível ser aplicada a praticamente qual-

quer área que possua uma grande quantidade de dados, hoje, diversas organizações ao redor do mundo, como: *International Business Machines* (IBM), *Microsoft*, *Google*, *Facebook*, *Spotify* utilizam aplicações de ML em seus negócios.

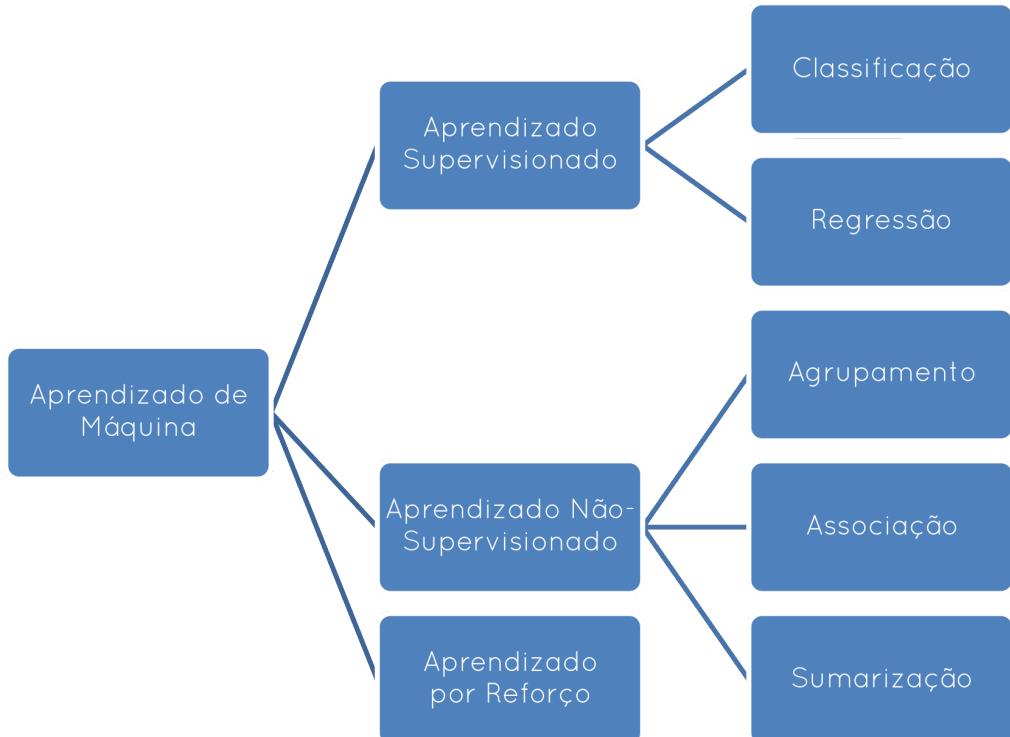
Mural de notícias do *Facebook*, tradução automática do *Google*, ofertas e recomendações da *Amazon*, são apenas alguns – entre tantos – exemplos comuns de como esta tecnologia está presente em nosso dia-a-dia.

2.1.1 Tipos de Aprendizado de Máquina

Segundo S. Marsland [28] podemos classificar os modelos de ML em quatro categorias distintas: Aprendizado Supervisionado, Aprendizado Não-Supervisionado, Aprendizado por Reforço e Aprendizado Evolutivo.

No entanto, no meio acadêmico alguns autores consideram três categorias principais de tipos de Aprendizado de Máquina: Aprendizado Supervisionado, Aprendizado Não-Supervisionado e Aprendizado por Reforço, como pode ser visto na Figura 2.1 a seguir:

Figura 2.1: Principais tipos de Aprendizado de Máquina.



Fonte: Elaborado pela autora.

Aprendizado Supervisionado

Um conjunto de dados de entradas e suas respectivas saídas desejadas (denominadas como classes) são fornecidas por um “professor” ao modelo/computador, mostrando que há relação entre os dados de entrada e de saída.

O objetivo deste tipo de aprendizado está em mapear as entradas para as suas respectivas saídas desejadas ao encontrar uma regra geral que seja apta a classificar novos dados entre as classes já presentes no conjunto de dados.

É possível que problemas de aprendizado supervisionado sejam categorizados como problemas de classificação e regressão.

Classificação: Em uma tarefa de classificação, o objetivo é mapear os tipos de entradas e categoriza-las em classes distintas, ou seja, realiza previsões onde os resultados de saída são de natureza discreta.

Algoritmos de classificação podem ser usados para problemas como: classificação de gênero, classificação de imagens médicas [29] e classificação de expressões faciais como poderá ser visto neste e em outros trabalhos [30, 31].

K-vizinhos mais próximos (em inglês, *K-Nearest Neighbors* - K-NN) [32, 33], SVM [17, 18], Baías Ingénuas (em inglês, *Naive Bayes*) [34, 35, 36] são exemplos de algoritmos de classificação.

Regressão: Em uma tarefa de regressão, o objetivo é mapear os tipos de entrada destinada a alguma determinada função contínua, ou seja, realiza previsões onde os resultados de saída são de natureza contínua.

Algoritmos de regressão podem ser usados para responder perguntas como: “Quanto custa?”, “Quantos existem?”, etc. além de poderem ser usados para problemas como: previsões de mercado, retenção de clientes e associação entre produtos.

Regressão Linear (em inglês, *Linear Regression*) [37], Regressão Linear Múltipla (em inglês, *Multiple Linear Regression* - MLR) [38], Regressão Logística (em inglês, *Logistic Regression*) [39] são exemplos de algoritmos de regressão.

Aprendizado Não-Supervisionado

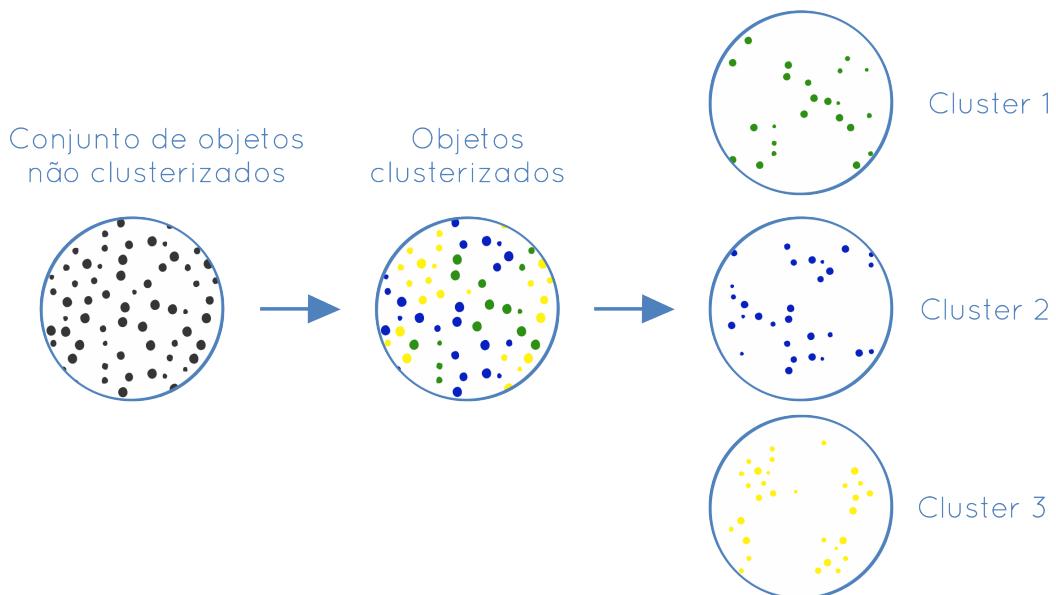
Nenhum conjunto de dados de entradas e de saídas desejadas é fornecido ao modelo/computador, portanto, não possui ajuda de um “professor”.

O objetivo deste tipo de aprendizado em si está em descobrir por conta própria novos padrões (classes) no conjunto de dados fornecido.

É possível que problemas de aprendizado não-supervisionado sejam categorizados como problemas de agrupamento, associação e sumarização.

Agrupamento (*Cluster*): Em uma tarefa de agrupamento, ou clusterização como também é chamado, o objetivo é encontrar padrões existentes no conjunto de objetos (que não possuem classes associadas) e agrupa-los em *Clusters* (grupos), ou seja, agrupa os objetos em classes baseando-se na descoberta de semelhanças e diferenças existentes entre eles. Na Figura 2.2 podemos ver este processo:

Figura 2.2: Exemplo de clusterização.



Fonte: Adaptado de [40, 41].

K-Means [42, 43], Clusterização Hierárquica (em inglês, *Hierarchical Clustering* - HC) [44, 45] são exemplos de algoritmos de clusterização.

Associação: Aprendizado Não-Supervisionado por regras de associação é um método utilizado para descobrir padrões ou relacionamentos que ocorrem com frequência entre variáveis, ou seja, identificar regras que descrevam elementos presentes em conjuntos de dados. A análise de cesto de compras presente na maioria dos *e-commerce*s é um exemplo de regra de Associação.

Apriori [46, 47, 48], *Eclat* [48], *Partition* [49], *FP-Growth* [48] são exemplos de regras de aprendizado de algoritmos de associação.

Sumarização: O método de sumarização consiste em produzir de forma a sintetizar ou resumir as principais informações, conteúdos e/ou indicadores presentes em um determinado

dado de entrada de maneira automática.

O aprendizado de sumarização pode ter a finalidade de: descrever o conteúdo presente no dado de entrada, sumarizar a informação presente no conteúdo do dado de entrada e analisar de forma crítica o conteúdo presente no dado de entrada.

A sumarização pode ser aplicada a diferentes tipos de dados de entrada, como por exemplo imagens e sons, entretanto, o principal tipo de dado utilizado em métodos de sumarização automática são dados textuais.

Um exemplo de sumarização automática é a produção de um texto resumido a partir da análise de sentenças de um grande documento, sem perder as principais informações contidas no documento de forma que a leitura, sentido e objetivo do texto não seja perdida.

Bio-Summ [50], *GistSUMM* [51, 52] são exemplos de algoritmos de sumarização.

Aprendizado por Reforço

Neste tipo de aprendizado não é especificado em nenhum momento como uma determinada tarefa deva ser executada, portanto, a dinâmica adotada neste método é a interação de um agente em um ambiente dinâmico. A principal função do agente é a de aprender através de tentativa e erro a se adaptar, se comportar e/ou atuar neste ambiente a fim de atingir determinado propósito (por exemplo, solucionar um cubo mágico de dimensão 3 x 3 x 3).

Uma vez em que o agente possui apenas a própria experiência com o ambiente como fonte de conhecimento e aprendizado, é apresentado ao agente feedback de recompensas e punições em relação ao ambiente na medida em que o agente interage com o ambiente do problema, com a finalidade de estimar vantagens que maximize determinadas tomadas de ações para os diferentes estados encontrados no ambiente.

Aprender a derrotar um adversário em um determinado jogo apenas através das experiências adquiridas ao jogar contra ele é um exemplo de aplicação de algoritmos de Aprendizado por Reforço.

Q-learning [53], *Upper Confidence Bound* (UCB) [54, 55], Amostragem de Thompson (em inglês, *Thompson Sampling - TS*) [56, 57] são exemplos de algoritmos de aprendizado por reforço.

Aprendizado Semi-Supervisionado

Há ainda outros autores como O. Chapelle, B. Schölkopf e A. Zien [58] e X. Zhu e A. B. Goldberg [59] que consideram a existência de outro tipo de Aprendizado de Máquina , denominado Aprendizado Semi-Supervisionado, este, encontra-se entre o Aprendizado Supervisionado e o Aprendizado Não-Supervisionado, onde um "professor" fornece ao modelo/computador um conjunto de dados de entradas e parcialmente suas respectivas saídas desejadas, ou seja, há ausência de dados de saídas desejadas. Neste sentido, é disponibilizado ao modelo um conjunto de treinamento incompleto.

Este tipo de aprendizado é muito utilizado em casos onde há uma pequena quantidade de dados de saídas desejadas, portanto, o Aprendizado Semi-Supervisionado aprende tanto a partir de dados rotulados, quanto de dados não rotulados.

Logo, é possível solucionar tarefas de classificação a partir dos dados de treinamento rotulados ao obter uma regra geral que seja eficiente ao classificar novos dados, além de ser possível também solucionar tarefas de clusterização ao encontrar padrões que auxiliam no processo de constituição de clusters.

2.1.2 Redes Neurais Artificiais

Uma das características mais poderosas e importantes do ser humano está na capacidade de aprender através da exposição de exemplos desde o seu nascimento, como por exemplo, a fala, a escrita, a identificação e classificação de imagens, etc. Já na vida adulta, tais habilidades tornam-se tarefas rápidas, naturais e sem o requerimento de esforços.

Tendo como exemplo, a identificação de objetos e ambientes distintos, é questionado como o cérebro humano é capaz de traduzir uma imagem na retina humana para algo onde um indivíduo consiga identificar e classificar. O ponto é que o ser humano possui tais habilidades onde é fácil para ele realizar, tão fácil que chega ao ponto de não precisar aplicar esforços conscientes para a realização de tarefas deste tipo.

Compreendendo toda a complexidade do cérebro humano, tornou-se necessário o estudo e compreensão de como os sistemas neurais biológicos funcionam, e a partir de tais estudos, pesquisadores foram capazes de proporcionar a estrutura básica para a construção de modelos de Redes Neurais Artificiais (em inglês, *Artificial Neural Networks - ANN*).

A estrutura do cérebro é formada por unidades básicas conhecidas como neurônios. Segundo R. Beale e T. Jackson [60] o cérebro possui aproximadamente dez bilhões (10^{10}) de

neurônios, onde cada neurônio está conectado a aproximadamente a dez mil (10^4) outros neurônios.

Os neurônios são fundamentais para a performance do corpo humano como também são determinantes para o desempenho do raciocínio humano.

O neurônio é formado por um corpo celular também conhecido como *Soma* em inglês.

Agregado ao corpo celular encontram-se o Núcleo, os Dendritos e os Axônios, os dois últimos são comumente chamados de fibras nervosas.

Os Dendritos são filamentos onde possuem formas irregulares, além de apresentarem um aspecto de ramificações presentes em uma árvore. Agem como sendo um canal de entrada do neurônio, pois através do ambiente ou de outros neurônios, transporta os impulsos recebidos ao corpo celular.

Já os Axônios, também filamentos como os Dendritos, porém mais longos. Agem como sendo um canal de saída do neurônio, pois conduzem os impulsos oriundos do corpo celular para os outros neurônios.

Toda a comunicação entre os neurônios é processada através de Sinapses.

A Sinapse é conhecida como sendo uma região especializada, pois a comunicação dos neurônios se dá através do contato entre dois neurônios, e na região deste contato é onde ocorre a Sinapse, na qual os impulsos são conduzidos entre eles.

É possível que um único neurônio possua em seus Dendritos muitas entradas sinápticas agindo sobre eles. Como também é possível possuir muitas saídas sinápticas que estejam conectando-as a outros neurônios.

A estrutura básica de um neurônio biológico pode ser vista na Figura 2.3.

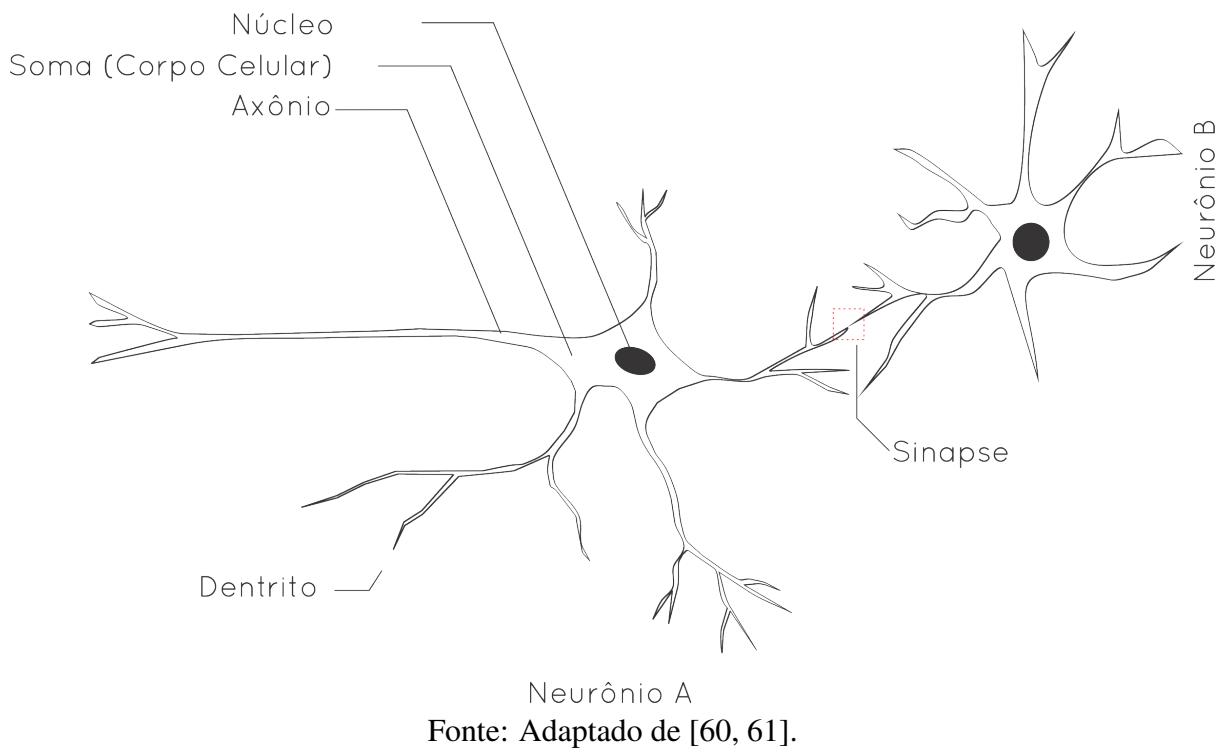
Podemos considerar os modelos de ANN como sendo o paralelismo que existe entre os sistemas neurais biológicos, possuindo comportamento semelhante ao cérebro, e tendendo a explorar situações relacionadas a tarefas de CV, agrupamento de dados, PR, generalização, dentre outras.

Os primeiros estudos sobre ANN iniciaram-se a partir do ano de 1943 com a teoria sobre como os neurônios funcionam através do desenvolvimento de um modelo de ANN para circuitos elétricos, desenvolvida pelo neurofisiologista W. McCulloch e pelo matemático W. Pitts [62].

A Figura 2.4 mostra a estrutura básica do modelo de McCulloch-Pitts.

Onde o modelo é estruturado por um vetor de entrada x que está associado a um vetor de

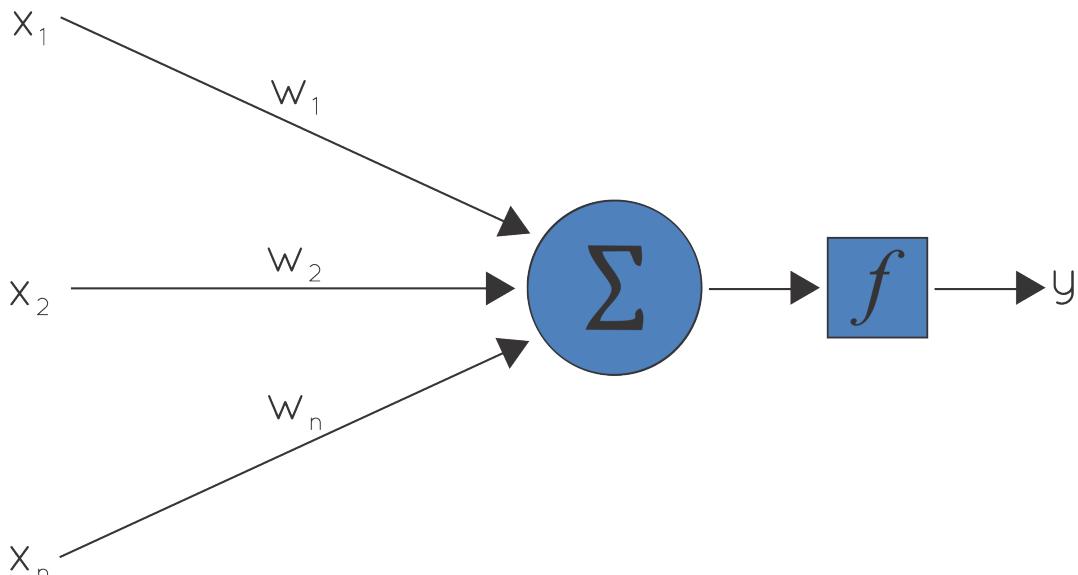
Figura 2.3: O Neurônio Biológico.



Neurônio A

Fonte: Adaptado de [60, 61].

Figura 2.4: O modelo de McCulloch-Pitts.



Fonte: Adaptado de [60].

pesos w , uma soma ponderada \sum , uma função de ativação f que determina o valor de saída y do modelo.

A soma das entradas ponderadas por seus pesos sinápticos associados é submetida à uma função de ativação (neste exemplo a função sinal – $sgn(x)$) onde, se a soma ponderada for

maior que um limiar de ativação (em inglês, *bias* - b) normalmente definido como 0, o neurônio é ativado, ou seja, a saída y será igual à 1, caso contrário o neurônio é desativado, logo, a saída y será igual à -1 , como pode ser visto na equação 2.1 abaixo:

$$y = \text{sgn}\left(\sum_{i=1}^n w_i x_i + b\right) \quad (2.1)$$

Onde n representa o tamanho do vetor de entrada.

No ano de 1957, F. Rosenblatt desenvolveu o que conhecemos hoje como a ANN mais antiga – o modelo *Perceptron*.

Perceptron de Camada Única

Perceptron de Camada Única (em inglês, *Single-Layer Perceptron* - SLP) é um modelo básico de ANN desenvolvida por F. Rosenblatt [63] onde possui um número definido de entradas, processamento, e uma única saída.

Uma das características deste modelo é ser um classificador linear, ou seja, realiza a classificação em casos onde os conjuntos de dados são linearmente separáveis em categorias binárias (1, 0), normalmente rotuladas como 1 e -1.

O modelo SLP soma todas as entradas ponderadas, como na equação 2.2.

$$\sum_{i=1}^n w_i x_i \quad (2.2)$$

Onde n representa o número de entradas para o modelo SLP, w_i representa o valor i no vetor de pesos que será multiplicado pelo valor x_i no vetor de entrada i .

Após este procedimento, as entradas ponderadas são passadas por uma função de ativação (a função de ativação mais utilizada em modelos SLPs é a função degrau) e, se a soma for maior que o limiar de ativação será considerado que o modelo SLP está ativado (a saída y será igual à 1). Como pode ser visto na função a seguir:

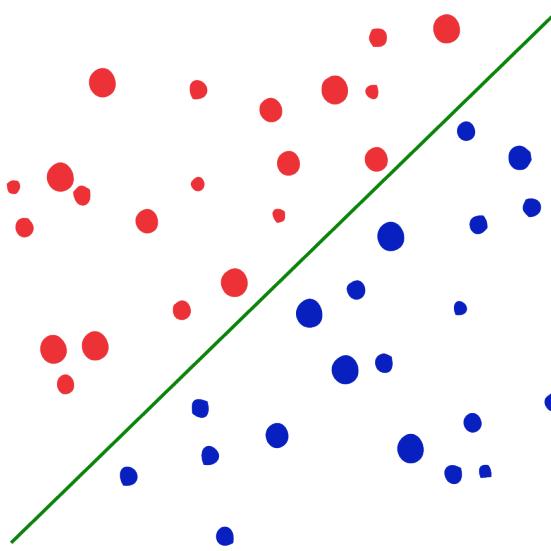
$$y = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases} \quad (2.3)$$

Onde w é um vetor de pesos, $w \cdot x$ é a soma ponderada $\sum_{i=1}^n w_i x_i$ e b é o limiar de ativação *bias*.

Os pesos são inicializados aleatoriamente, neste sentido, é feito um determinado ajuste de pesos caso a resposta esperada pela rede não seja a correta. Neste caso, o algoritmo busca por um novo conjunto de pesos que sejam os ideais para que a rede consiga classificar corretamente as entradas.

O modelo busca dividir o plano entre duas regiões, classificando as duas classes através de um separador linear conhecido em inglês como *decision boundary*. A Figura 2.5 mostra um conjunto de dados linearmente separável.

Figura 2.5: Conjunto de dados linearmente separável.



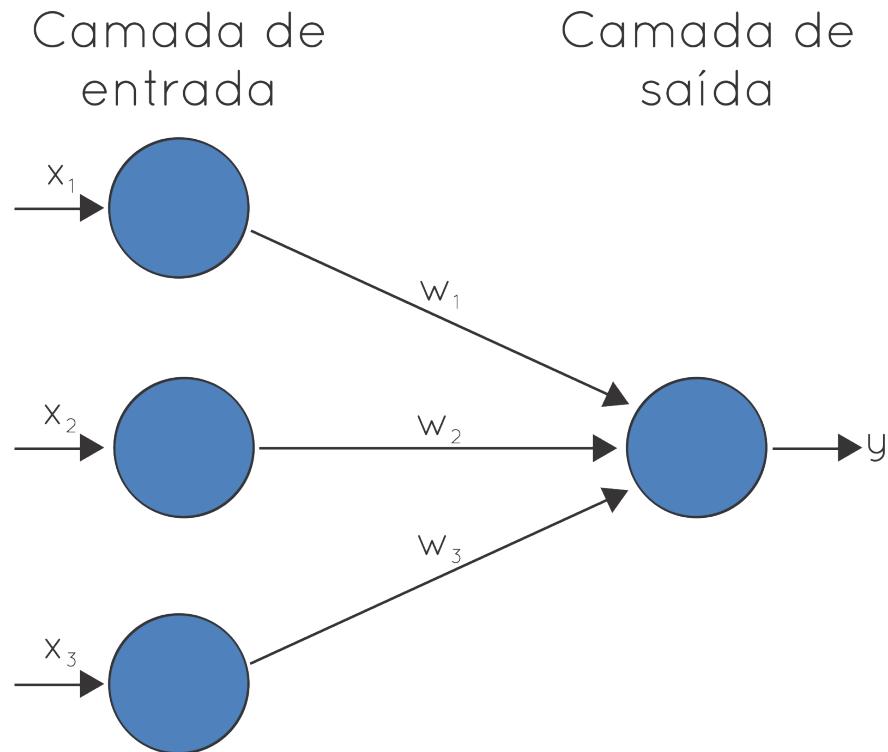
Fonte: Adaptado de [60].

A arquitetura básica do modelo SLP pode ser vista na Figura 2.6 abaixo.

Perceptron Multicamada

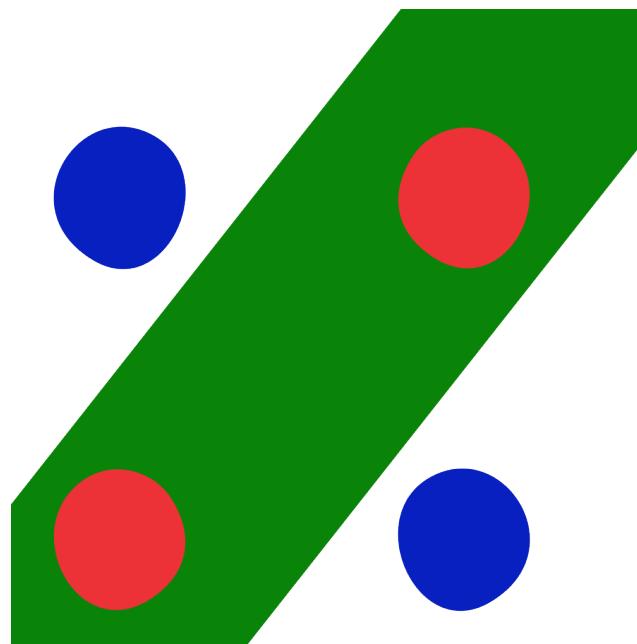
As características do modelo SLP se restringem à classificação de conjuntos linearmente separáveis, o que deixa fora de sua competência funções não-lineares, além de ser uma rede de camada única, ou seja, entre a camada de entrada e a camada de saída da rede, só existe um nó. Estas limitações foram demonstradas matematicamente em 1969 por M. Minsky e S. Papert [64].

Nesta ótica, estas limitações encontradas no modelo SLP foram solucionadas ao adicionar mais de uma camada de neurônios na rede, desta forma criando o modelo Perceptron Multicamada (em inglês, *Multilayer Perceptron* - MLP) que é capaz de realizar a classificação em casos onde os conjuntos de dados são não-linearmente separáveis, como por exemplo, o problema XOR como pode ser visto na Figura 2.7.

Figura 2.6: Arquitetura do modelo *Single-Layer Perceptron*.

Fonte: Adaptado de [60].

Figura 2.7: Problema XOR: Conjunto de dados não-linearmente separável.



Fonte: Adaptado de [60].

Arquitetura

Uma arquitetura básica de MLP pode ser dividida nas camadas listadas abaixo:

- **Camada de Entrada** (ou em inglês, *Input Layer*): Onde o conjunto de dados é inserido na rede.
- **Camada Escondida** (ou em inglês, *Hidden Layer*): É nesta camada onde ocorre a maioria dos processamentos da rede.
- **Camada de Saída** (ou em inglês, *Output Layer*): Onde o resultado obtido através do processamento realizado na camada escondida é exibido.

No qual cada nó de entrada x_i ($x_1, x_2 \dots x_n$) está associado a um peso sináptico w_i ($w_1, w_2 \dots w_n$), onde o valor dos pesos sinápticos são inicialmente inicializados de forma aleatória. A entrada possui também um limiar de ativação b (*bias*) que é a representação de um valor fixo diferente de 0.

Neste sentido, ocorre o processo de combinação linear, produzindo o potencial de ativação s , onde os nós de entrada são ponderados por seus respectivos pesos sinápticos associados ($x_1 \cdot w_1 + x_2 \cdot w_2 + \dots + x_n \cdot w_n$), e pelo limiar de ativação b como pode ser visto na equação 2.4.

$$s = \sum_{i=1}^n w_i x_i + b \quad (2.4)$$

Onde n representa o tamanho do vetor de entrada.

Após este processo, a soma das entradas ponderadas por seus pesos sinápticos associados mais o limiar de ativação, ou seja, o potencial de ativação s é submetido à uma função de ativação f (a função de ativação mais utilizada em modelos MLPs é a função sigmóide), como pode ser visto na equação 2.5 abaixo:

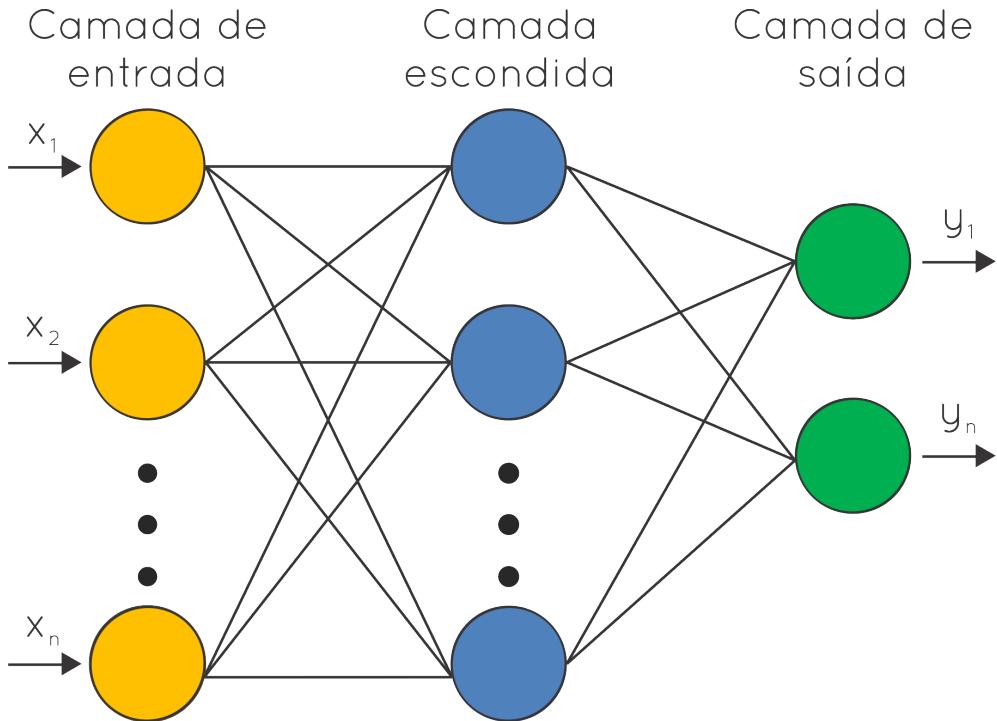
$$f(s) = \frac{1}{(1 + e^{-s})} \quad (2.5)$$

Onde s representa o potencial de ativação.

Logo, se o potencial de ativação s for maior que o limiar de ativação será considerado que o modelo MLP estará ativado (a saída y será igual à 1), e será considerado que o modelo MLP estará desativado, caso contrário.

A arquitetura básica do modelo MLP com uma única camada escondida pode ser vista na Figura 2.8.

Vale ressaltar que arquiteturas profundas de uma rede MLP, possuem n camadas escondidas.

Figura 2.8: Arquitetura básica do modelo *Multilayer Perceptron*.

Fonte: Adaptado de [60].

2.2 Aprendizado Profundo

Aprendizado Profundo (em inglês, *Deep Learning* - DL) pode ser caracterizado como sendo uma subárea do ML na qual grandes ANN com inúmeras camadas intermediárias são manipuladas com a finalidade de modelar abstrações e processar dados de alto nível. DL está por trás de tecnologias "inteligentes" em evolução que compreendem e reconhecem objetos visuais, que compreendem sons e textos, como a fala e a escrita humana, que reagem a situações adversas e ao mesmo tempo complexas.

Muitas empresas direcionadas ao futuro já incorporaram o DL em seus mais variados setores e processos de negócios, o que gera uma vantagem competitiva entre os concorrentes e muda totalmente a forma de pensar no que diz respeito a exploração de tecnologias e grande volume de dados presentes nas empresas, trazendo ao mercado uma nova forma de oferecer produtos e serviços "inteligentes", como por exemplo, desde diagnósticos confiáveis à drones autônomos.

2.2.1 Redes Neurais Convolucionais

Reconhecer os padrões existentes em imagens é um campo de pesquisa e desenvolvimento desafiador onde um algoritmo possui a capacidade de identificar objetos, lugares, sinais de

trânsito, animais, faces, pessoas, tumores entre outros aspectos visuais.

As diversas arquiteturas de CNN são aplicadas com sucesso em desafios de reconhecimento e classificação de imagens, embora sejam também utilizadas em processamento de vídeo [65, 66], e Processamento de Linguagem Natural [67, 68] (em inglês, *Natural Language Processing - NLP*).

Uma CNN pode ser caracterizada como sendo uma ANN de múltiplas camadas onde primariamente faz-se a suposição de que os dados de entrada sejam imagens. Como visto na seção 1.1, proposta inicialmente por LeCun, as CNN hoje, são consideradas estado-da-arte para a resolução de problemas na área de CV, possuindo uma variedade de modelos e de arquiteturas diferentes como já comentado.

Apesar da semelhança com a arquitetura MLP, a principal diferença está na operação que ocorre nas imagens inseridas na rede, chamada convolução, que será apresentada mais abaixo.

Arquitetura

Uma arquitetura básica de CNN pode ser dividida nas camadas listadas abaixo:

- **Camada de Entrada** (ou em inglês, *Input Layer*): Onde a imagem (ou base de imagens) é inserida na rede.

Computacionalmente falando, toda imagem pode ser representada como sendo uma matriz de valores de pixel, como pode ser visto na Figura 2.9.

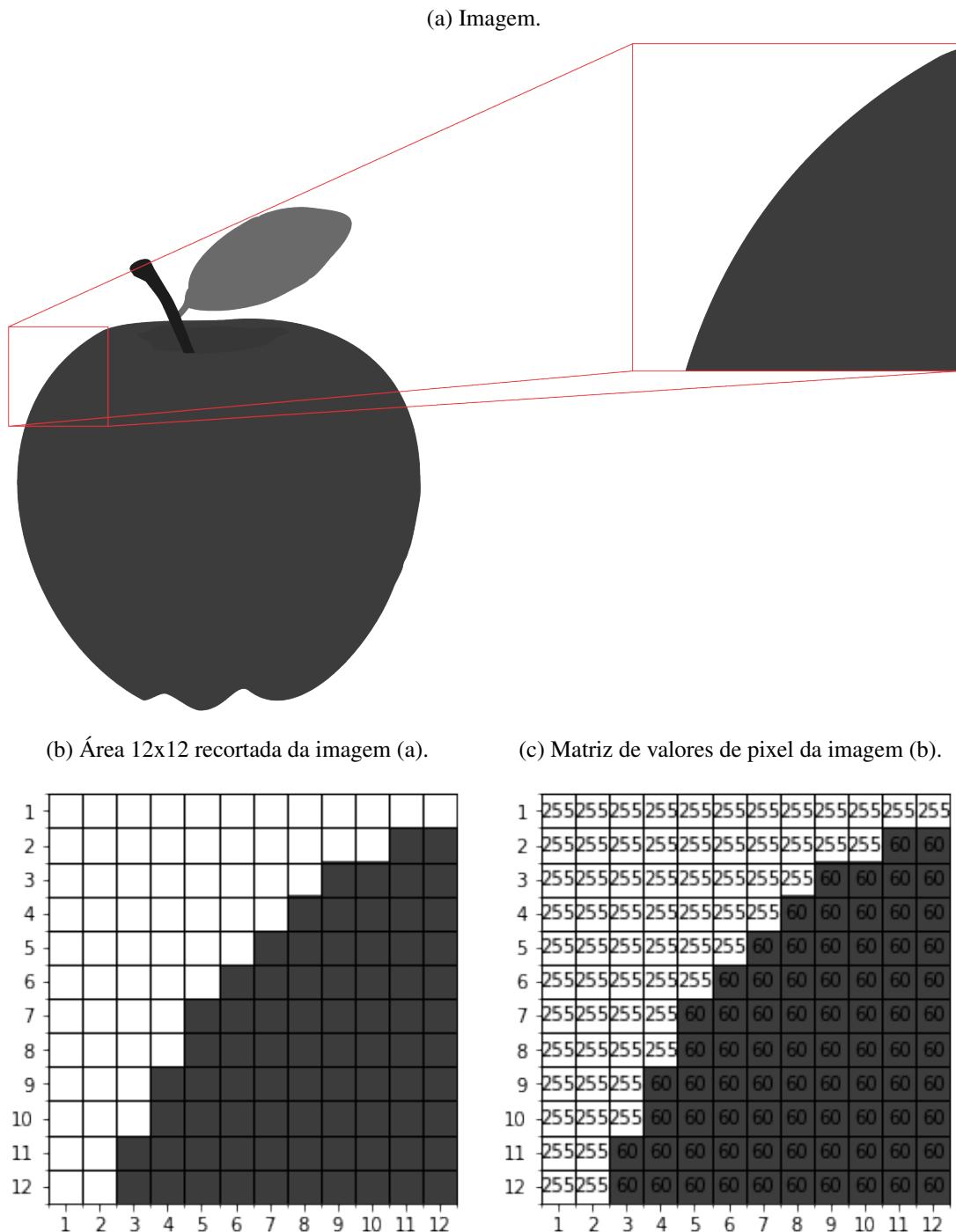
Portanto, ao receber uma imagem como entrada, o computador "enxergará" uma matriz de valores de pixel.

Ao supor que temos uma imagem de entrada colorida no formato .png e que seu tamanho é 10×10 , o computador verá uma matriz representativa de dimensão $10 \times 10 \times 3$, onde o valor 3 refere-se a um componente pré-estabelecido de uma imagem – Canal conhecido em inglês como *channel* – neste caso, o valor 3 refere-se aos valores RGB, onde, para cada um dos canais, será atribuído valores de intensidade de pixels dentro do intervalo de 0 à 255 (onde 0 representa pontos pretos e 255 representa pontos brancos) para cada ponto da matriz. A Figura 2.10 demonstra um exemplo de tal representação:

Entretanto, uma imagem de entrada em escala de cinza possui apenas 1 canal ($10 \times 10 \times 1$) como pode ser visto na Figura 2.9.

Para o propósito deste trabalho, de modo a reduzir ao máximo o poder computacional necessário na etapa de processamento de imagens, optamos em utilizar apenas imagens

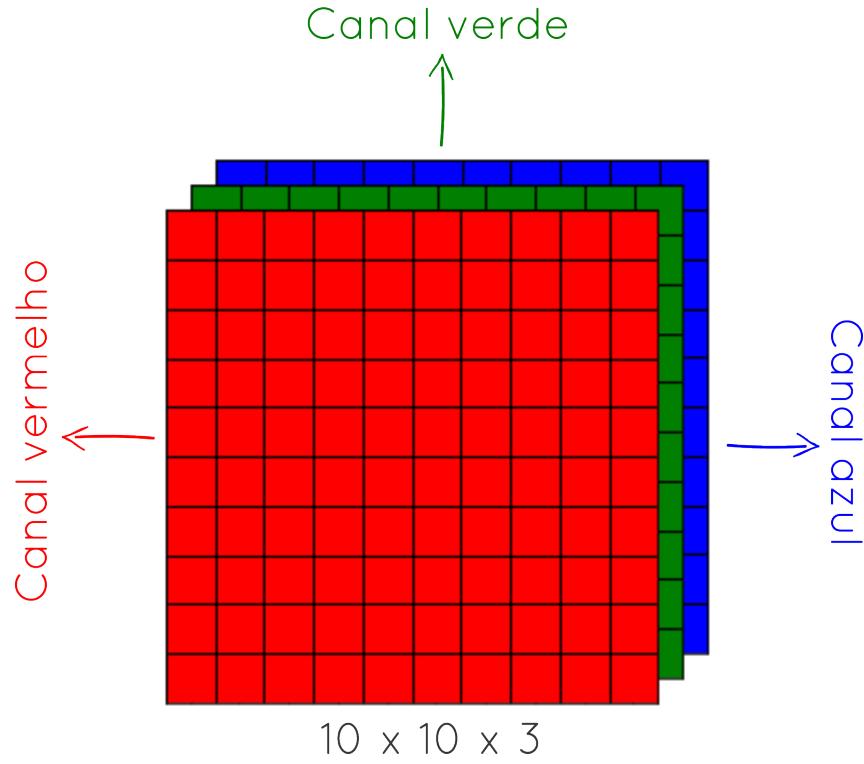
Figura 2.9: Representação de uma matriz de valores de pixel.



Fonte: Elaborado pela autora.

em escala de cinza, gerando desta maneira apenas uma única matriz 2D representativa para cada imagem.

- **Camada de Convolução** (ou em inglês, *Convolutional Layer*): É a principal operação que ocorre nas imagens inseridas na rede, e seu principal objetivo está em extrair recursos

Figura 2.10: Matriz representativa de uma imagem com dimensão $10 \times 10 \times 3$.

de imagens de entrada. Uma imagem g é gerada pela convolução de um filtro f (comumente conhecido na literatura em inglês como *kernel*) com a imagem de entrada I definida pela equação 2.6:

$$g(m, n) = \sum_{i=1}^q \sum_{j=1}^q f(i, j)I(m-i, n-j) \quad (2.6)$$

onde q é a dimensão da máscara de convolução.

Um filtro pode ressaltar determinada característica presente em uma imagem, como por exemplo, sombras, bordas, entre outros.

A Figura 2.11 apresenta o resultado de uma convolução aplicada a uma imagem de entrada.

Onde o resultado da convolução é comumente conhecido como um mapa de recursos (em inglês, *feature maps*), e onde o filtro convolucional f utilizado foi uma matriz de tamanho 3×3 como na representação 2.7 a seguir:

Figura 2.11: Resultado da convolução.

(a) Imagem de entrada.



(b) Resultado da convolução.



Fonte: Elaborado pela autora.

$$f = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (2.7)$$

- **Camada ReLU** (ou em inglês, *ReLU Layer*): A camada ReLU (em inglês, *Rectified Linear Units*) é utilizada para que a rede possa convergir mais rápido [21], aplica para as camadas escondidas da rede uma função de ativação não linear à saída x da camada anterior, como na equação 2.8:

$$f(x) = \max(0, x) \quad (2.8)$$

Portanto, a saída será 0 quando a entrada for menor que 0 ($x < 0$), ou a saída será 1 caso contrário. Desta forma a ativação é limitada exclusivamente a 0, logo, a função de ativação ReLU contribui em uma etapa de treinamento da rede mais rápida para arquiteturas profundas.

- **Camada de Subamostragem** (ou em inglês, *Pooling Layer*): Reduz a dimensionalidade dos *feature maps* que serão utilizados pelas camadas seguintes, entretanto, sem perder as informações mais relevantes. Desta forma, podemos definir a camada de subamostragem como uma convolução que possui o objetivo de reduzir o tamanho espacial da representação, de forma a contribuir com a diminuição de quantidade de parâmetros e processamento computacional da rede.

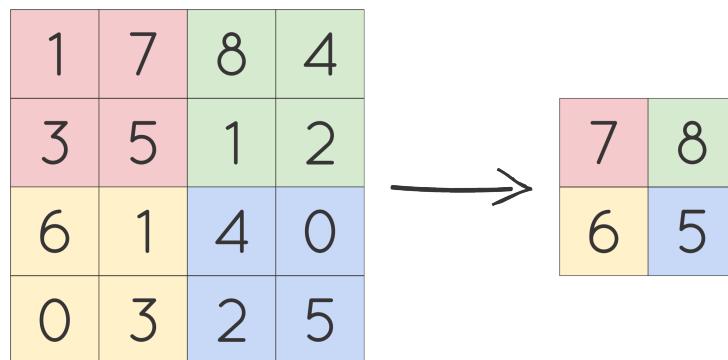
Existem diferentes tipos de Subamostragens, tais como: *Max*, *Min*, *Sum*, *Average*, etc., entretanto, a mais comum e utilizado neste trabalho é o *Max*, conhecido em inglês como

Max-Pooling.

Para aplicar o *Max-Pooling*, é necessário definir uma vizinhança espacial ("uma janela") como por exemplo, um filtro de tamanho 2×2 . É necessário definir também o "passo"(em inglês, *stride*) da janela, ou seja, de quantas em quantas células a janela deslizará sobre o *feature maps* da camada anterior, na literatura, o *stride* é comumente definido como sendo do mesmo comprimento que o tamanho do filtro, neste caso, o *stride* será definido como 2.

Após definir estes parâmetros, a função do *Max-Pooling* está em pegar o valor máximo dos elementos do *feature maps* presentes na janela 2×2 e agrupa-los. A Figura 2.12 mostra um exemplo da operação de *Max-Pooling* em um *feature maps* obtido após passar pelas camadas de convolução e ReLU.

Figura 2.12: Operação de Max-Pooling com um filtro 2×2 e *stride* 2.



Fonte: Elaborado pela autora.

- **Camada de *Flatten*** (ou em inglês, *Flatten Layer*): Após passar pelas camadas de convolução, ReLU e subamostragem a próxima etapa de uma CNN é a classificação, ou seja, é necessário passar pela camada totalmente conectada. A camada totalmente conectada aceita apenas como entrada, dados de dimensão $1D$. Entretanto, a dimensão dos dados produzidos até a camada de subamostragem são dados de dimensão $3D$, sendo necessário a conversão para um vetor de recursos $1D$.

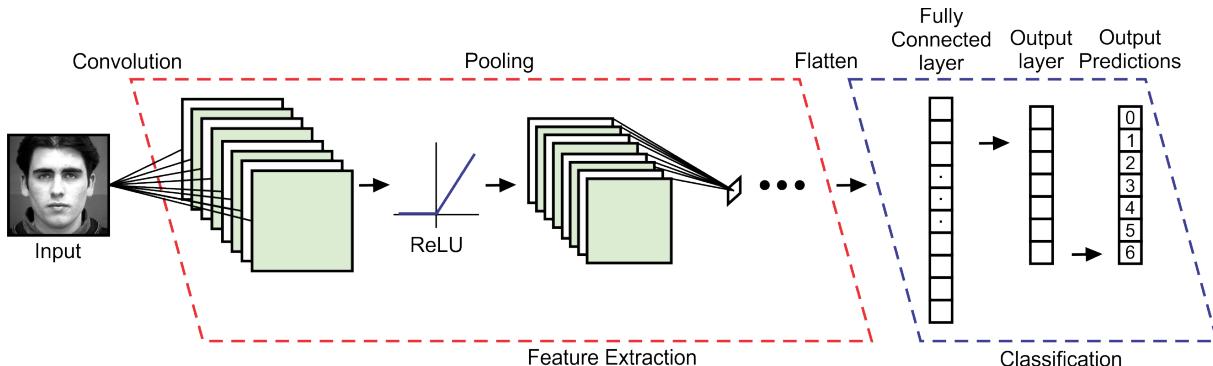
A camada *Flatten* possui o objetivo de "achatar" todo o volume de dados $3D$ em um único vetor $1D$ de características para posteriormente servir de entrada para a camada totalmente conectada.

- **Camada totalmente conectada** (ou em inglês, *Fully Connected Layer*): Essa camada age como um classificador, se localiza ao final da rede, geralmente é do tipo *soft-max* usado para classificar a imagem de entrada. Realiza tal tarefa ao reconhecer padrões que

foram gerados pelas camadas anteriores. Portanto, estima a probabilidade de quais das n classes a imagem de entrada pertence.

A Figura 2.13 mostra a arquitetura básica do modelo de CNN.

Figura 2.13: Arquitetura básica do modelo *CNN*.



Fonte: Adaptado de [69].

2.3 Descritores de Texturas

Na literatura existe diversas definições sobre o que é uma textura de acordo com as mais variadas pesquisas e áreas distintas. Portanto, percebe-se que a definição de textura está relacionada com a aplicação utilizada, não havendo uma definição precisa sobre o tema. Tal como:

"An image texture is described by the number and types of its primitives and the spatial organization or layout of its primitives. The spatial organization may be random, may have a pairwise dependence of one primitive on a neighboring primitive, or may have a dependence of n primitives at a time. The dependence may be structural, probabilistic, or functional (like a linear dependence)." [70, p. 786].

No campo de processamento de imagens digitais, realizar a classificação de padrões em imagens e objetos é uma das tarefas mais complexas, entretanto, torna-se uma valiosa técnica de análise, pois os métodos de classificação e reconhecimento de imagens utilizam a textura para a identificação de características e/ou padrões presentes em imagens de interesse, o que, consequentemente, a partir dos dados extraídos de uma imagem (tais dados como: luminosidade, distribuição espacial, arranjo estrutural das superfícies, rugosidade, suavidade, etc.), torna-se possível a realização de tomada de decisão em diferentes aspectos e áreas.

Portanto, podemos determinar uma textura de forma sucinta, como sendo uma característica ou padrão importante presente em imagens e objetos. Na literatura, é possível encontrar também abordagens distintas utilizadas para descrever a textura no processamento de imagens, tais como: Abordagem Estatística [70], Abordagem Estrutural [71] e Abordagem Espectral [72].

Descritores de textura são algoritmos que utilizam métodos e técnicas com o objetivo de extrair características e padrões presentes em imagens e objetos, o que, posteriormente, a partir da identificação de uma região específica encontrada na imagem, pode ser realizado a classificação de imagens como semelhantes.

Análise de imagens médicas [73], identificação biométrica [74], e sensoriamento remoto [75] são exemplos de aplicações onde as informações extraídas a partir da análise de textura são utilizadas.

2.3.1 Padrão Binário Local

LBP é um tipo de descritor de textura simples e eficiente [76], usado para reconhecimento e classificação de padrões de imagens com texturas. Vem ganhando notoriedade nesta tarefa a partir do ano de 2002 com o trabalho de Ojada [77], devido a sua baixa complexidade computacional, além da aptidão ao representar micro padrões [78, 79].

Devido ao seu sucesso, foram propostas novas variantes a partir dos fundamentos do LBP tradicional, como *Median Robust Extended Local Binary Pattern* (MRELBP) [80], *Compound Local Binary* (CLBP) [81], *CS- Center Symmetric LBP* (CS-LBP) [82, 83], entre diversas outras. Seu sucesso é tanto, que aplicado a problemas de CV, tornou-se uma abordagem popular na análise de imagens, como por exemplo, aplicado a análise de faces [84]. Apesar das inúmeras variantes disponíveis, neste trabalho utilizamos como método a ser aplicado o LBP tradicional.

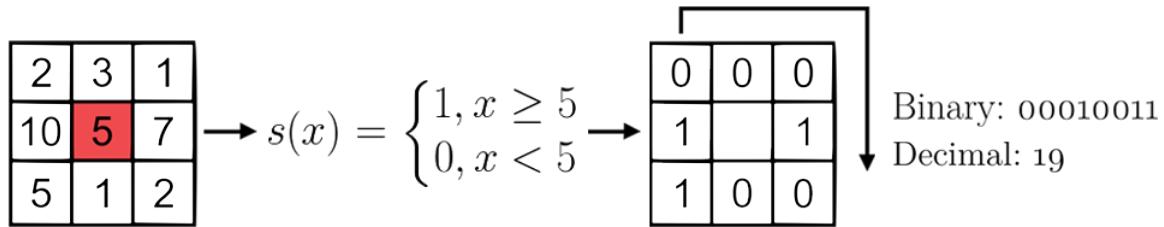
O objetivo do LBP tradicional está em calcular uma representação local de texturas, onde as informações são separadas entre padrão e intensidade, aspectos estes que se complementam.

O LBP tradicional considera para cada pixel em uma imagem em escala de cinza, uma vizinhança de tamanho 3×3 em torno do pixel central, em seguida, atribui-se 1, caso o pixel seja maior ou igual ao pixel central, ou atribui-se 0, caso o pixel seja menor ao pixel central, como pode ser vista na Figura 2.14:

Como visto na Figura 2.14 os valores obtidos em $s(x)$ são então concatenados, e em seguida, o valor binário resultante é convertido para a base decimal substituindo o valor do pixel central.

Este processo é então repetido em toda a imagem de entrada para cada pixel.

Figura 2.14: Exemplo de formulação do LBP.

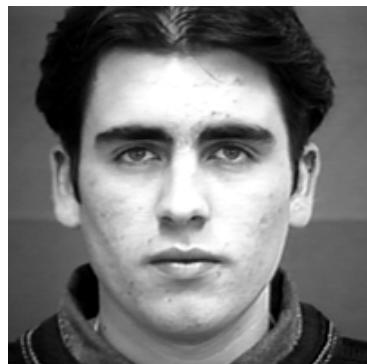


Fonte: Elaborado pela autora.

A Figura 2.15 a seguir mostra um exemplo completo do resultado desta técnica aplicado a uma imagem de entrada:

Figura 2.15: Resultado do LBP.

(a) Imagem de entrada.



(b) LBP resultante.



Fonte: Elaborado pela autora.

Após este procedimento, é calculado o histograma de intensidade dos pixels da imagem LBP resultante, gerada através do cálculo da frequência relativa.

Portanto, para um valor i , devemos considerar a frequência absoluta como sendo o número f_i de vezes em que certa variável assumiu o valor i . Ou seja, a frequência absoluta está interessada em descobrir a quantidade de vezes que o valor i aparece em uma amostra, ou, como no exemplo da amostra apresentada na Tabela 2.1 abaixo - o número de elementos que pertencem a uma determinada classe:

A partir da frequência absoluta, será determinada a frequência relativa (F_i) por meio da fórmula 2.9 a seguir:

$$F_i = \left(\frac{f_i}{N} \right) \quad (2.9)$$

Onde, f_i representa a frequência absoluta e N é igual ao somatório da frequência absoluta

Tabela 2.1: Exemplo de Frequência Absoluta.

Classes	Quantidade de imagens por classe
Raiva	10
Nojo	17
Medo	8
Alegria	13
Neutro	9
Tristeza	13
Surpresa	16
Total	86

$(\sum f_i)$ que representa o número de elementos da amostra.

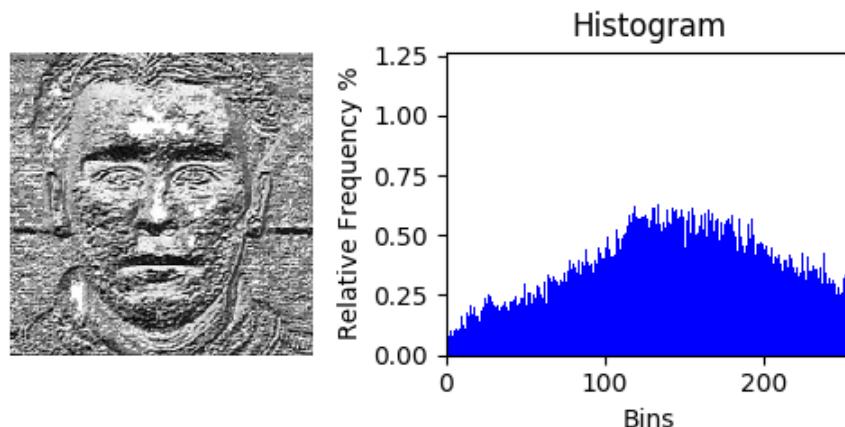
Podemos obter a porcentagem do valor dos elementos presentes na amostra em relação ao total da amostra ao realizar a multiplicação da frequência relativa por 100:

$$P_i = F_i \cdot 100 \quad (2.10)$$

Portanto, para calcular o histograma de intensidade dos pixels da imagem LBP resultante, os recursos resultantes são agrupados em um único vetor de característica onde o mesmo possui um valor mínimo de 0 e um valor máximo de 255, logo, construindo um vetor de característica $256 - dimensional$, uma vez que a vizinhança de tamanho 3×3 possui $2^8 = 256$ possíveis padrões.

A Figura 2.16 mostra o resultado do histograma de intensidade dos pixels do LBP resultante da Figura 2.15.

Figura 2.16: Histograma de intensidade dos pixels.



Fonte: Elaborado pela autora.

A partir deste momento, nosso vetor final de características – o histograma 256-*dimensional* – pode ser processado com algoritmos de ML (como por exemplo SVM) para realizar tarefas de análise de texturas e classificação de imagens.

2.3.2 Reformulação do LBP através de filtros convolucionais

Em [23], o autor demonstra que é possível reformular o LBP através de filtros convolucionais.

Para reformular o LBP através de filtros convolucionais e atingir o mesmo objetivo, é aplicado uma convolução de toda a imagem com 8 filtros convolucionais de tamanho 3x3, seguida de um operador de binarização não-linear, neste caso, a função Heaviside (veja a equação 2.11):

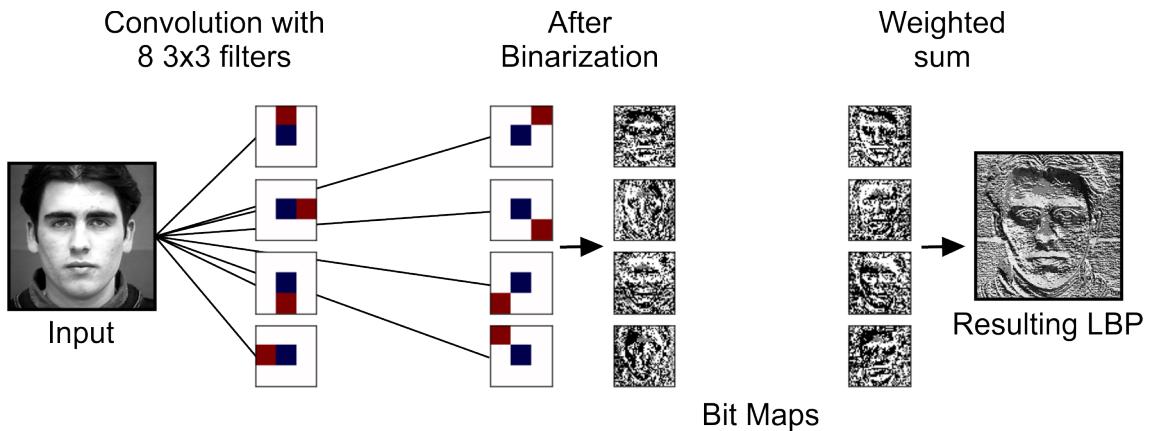
$$H(x) = \frac{1 + sgn(x)}{2} \quad (2.11)$$

Onde $sgn(x)$ se refere a função sinal, portanto, podemos obter a equação 2.12:

$$H(x) = \begin{cases} 0, & x < 0 \\ \frac{1}{2}, & x = 0 \\ 1, & x > 0 \end{cases} \quad (2.12)$$

Em seguida é realizada uma soma ponderada dos 8 bit maps usando um vetor de pesos pré-definidos, $v = [2^7, 2^6, 2^5, 2^4, 2^3, 2^2, 2^1, 2^0]$, onde são então extraídos os recursos LBP. A Figura 2.17 ilustra todo este processo descrito acima.

Figura 2.17: Reformulação do LBP através de filtros convolucionais.



Fonte: Adaptado de [23].

Como mostrado na Figura 2.17, os filtros de intensidade vermelha indicam que o filtro

terá uma resposta positiva aos pixels pretos nas imagens de entrada (1), enquanto os filtros de intensidade azul indicam que o filtro terá uma resposta negativa aos pixels pretos nas imagens de entrada (-1).

O código fonte para realizar a reformulação do LBP através de filtros convolucionais pode ser visto no repositório *GitHub*¹ e no Apêndice A.

¹<https://github.com/whoisraibolt/Reformulation-of-LBP-through-Convolutional-Filters>

3 *Implementação*

Neste capítulo é descrito os métodos e ferramentas utilizadas para realizar os experimentos propostos neste trabalho a fim de solucionar o problema apresentado.

Para realizar a tarefa de reconhecimento e classificação de expressões faciais básicas a partir de três bases de imagens públicas de expressão facial, adotamos como metodologia a execução das etapas básicas ao trabalhar com ANN, etapas estas que consiste no treinamento de uma CNN e posteriormente sua validação.

3.1 Ferramentas

A adaptação do LBCNN proposta neste trabalho, bem como o pré-processamento das bases de imagens utilizadas, foi implementada em *Python* utilizando o *framework TensorFlow* em uma arquitetura baseada em GPU.

Todos os experimentos realizados foram processados por meio de uma instância de máquina virtual de alto desempenho na infraestrutura do *Google - Google Compute Engine* (GCE) - um componente Infraestrutura como Serviço (em inglês, *Infrastructure as a Service* - IaaS).

Optamos em utilizar tal serviço devido à falta de equipamentos robustos a atenderem as demandas deste trabalho em acelerar o processamento computacional.

O GCE é um recurso disponível na plataforma *Google Cloud Platform* (GCP)¹. A instância de máquina virtual *on demand* é composta por uma GPU NVIDIA® Tesla® K80 com memória da GPU de 12GB GDDR5 e 8 vCPUs disponíveis, além de 16 GB de memória RAM e 50GB de HD e é acessada por meio da interface de linha de comandos. A adaptação do LBCNN implementada neste trabalho está disponível no repositório *GitHub*² e pode ser visto no Apêndice B.

¹<https://cloud.google.com>

²<https://github.com/whoisraibolt/LBCNN>

3.1.1 Google Cloud Platform

Oferecida pelo *Google*, o GCP é um conjunto de ferramentas, soluções e serviços de computação em nuvem que atua na mesma infraestrutura central do *Google*, ao lado de seus produtos, como o *YouTube* e o seu próprio mecanismo de busca.

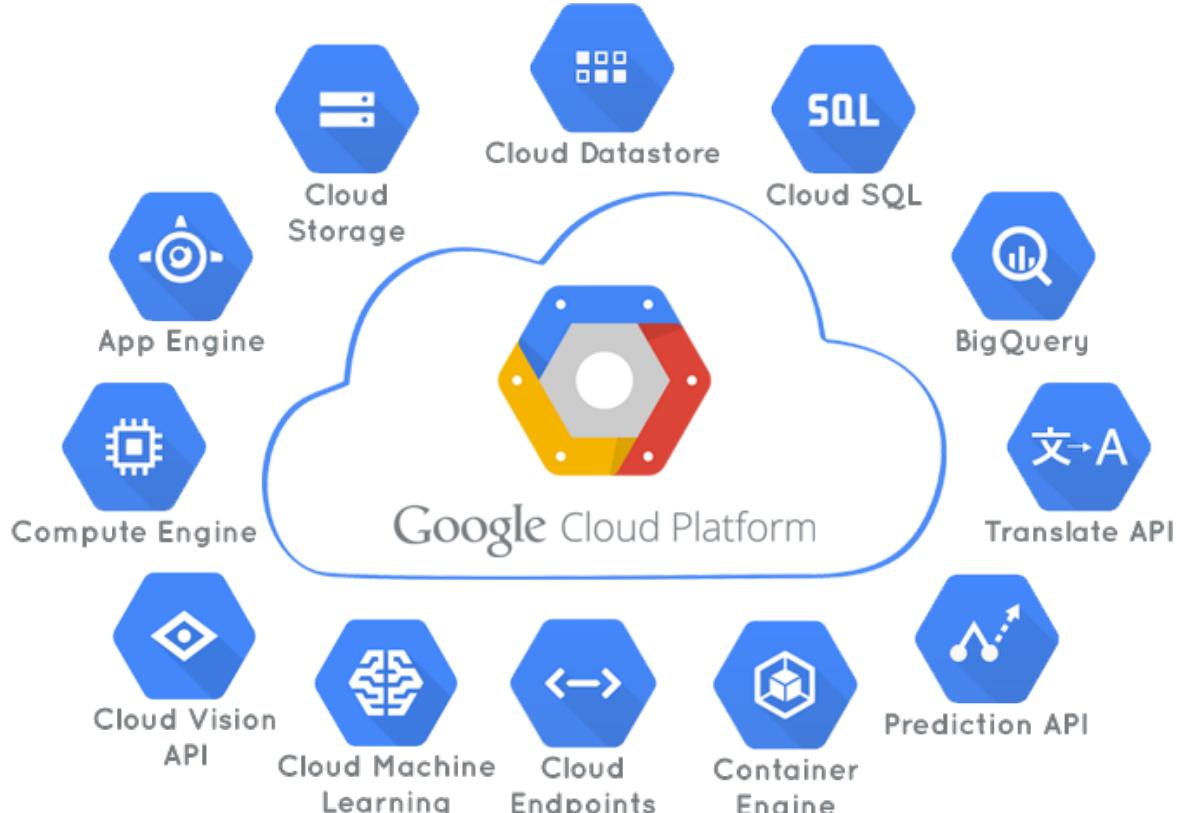
Os clientes do GCP disfrutam de uma infraestrutura segura, escalável, de alto desempenho, econômica (*on demand*), e com melhorias continuas.

O GCP disponibiliza uma série de soluções para processamento computacional, rede, armazenamento de dados, banco de dados escaláveis, exploração de dados, Internet das Coisas (em inglês, *Internet of Things* - IoT), ML, ferramentas de gerenciamento, monitoramento, registro, diagnósticos, entre outros.

Coca-Cola, Snapchat, Spotify, Ocado, Philips e Wix.com são apenas alguns dos clientes que utilizam os produtos oferecidos pelo GCP.

Na Figura 3.1 a seguir é apresentado os principais produtos oferecidos pelo GCP.

Figura 3.1: Principais produtos oferecidos pelo GCP.



Fonte: Adaptado de GCP³.

³<https://cloud.google.com/products>

3.1.2 Dependências Utilizadas

*NumPy*⁴: Alicerce para computação científica, apresenta objetos de grandes matrizes e matrizes multidimensionais, além de funções matemáticas e estatísticas de alto nível. Outras bibliotecas de nível superior utilizam a biblioteca *NumPy* como dado de suas entradas e saídas.

*SciPy*⁵: Biblioteca baseada em *NumPy*, estendendo seus recursos, acrescenta algoritmos e comandos para manipular e visualizar dados. Inclui diversos módulos tais como álgebra linear, teoria da probabilidade, otimização numérica, integração, etc., potencializando tarefas de ciências de dados.

*SciKit-Learn*⁶: Biblioteca baseada em *NumPy* e *SciPy*, adiciona uma quantidade enorme de algoritmos para tarefas de ML e mineração de dados. Inclui por exemplo técnicas de clusterização, classificação e regressão.

*Matplotlib*⁷: Biblioteca padrão para visualização em *Python*, portanto, permite a criação de gráficos e plotagens 2D em nível profissional. Outras bibliotecas de visualização são criadas para operar em conjunto como o *Matplotlib*, trazendo recursos interativos, exportação de gráficos, etc.

*TensorFlow*⁸: Framework desenvolvido pelo *Google*, usado especificamente para ML. Com ele, é possível implementar, modelar, e treinar diversos tipos de arquiteturas de ANN utilizando uma quantidade enorme de conjunto de dados.

3.2 Base de Imagens

Para avaliar a técnica apresentada e validá-la com a tarefa de FER, foram utilizadas três bases de imagens públicas de expressão facial: JAFFE [12], Extended CK+ [10, 11], e FER-2013 [13]. As três bases de imagens utilizadas neste trabalho possuem as seis expressões faciais básicas apresentadas no capítulo 1, e neutro, logo, formando sete emoções a serem classificadas pelo nosso modelo de LBCNN.

A Figura 3.2 mostra algumas amostras de imagens presentes nas bases de imagens utilizadas neste trabalho.

Da esquerda para a direita as colunas representam as imagens correspondem as respectivas

⁴<http://www.numpy.org>

⁵<https://www.scipy.org>

⁶<http://scikit-learn.org>

⁷<https://matplotlib.org>

⁸<https://www.tensorflow.org>

Figura 3.2: Amostras presentes nas três bases de imagens.



Fonte: Adaptado de [10, 11, 12, 13].

categorias: Raiva, Nojo, Medo, Feliz, Triste, Surpreso e Neutro. E de cima para baixo as linhas representam as imagens correspondentes as respectivas bases de imagens: JAFFE, Extended CK+, e FER-2013.

JAFFE

Japanese Female Facial Expression (JAFFE) consiste em 213 imagem de dez modelos japonesas. Selecionamos para o conjunto de treinamento 150 imagens, enquanto que para o conjunto de teste selecionamos 63 imagens.

Extended CK+

Extended Cohn-Kanade (Extended CK+) consiste em 593 sequencias de vídeo gravadas de 123 indivíduos. Selecionamos para o conjunto de treinamento 10.256 imagens, enquanto que para o conjunto de teste selecionamos 565 imagens.

FER-2013

Facial Expression Recognition 2013 (FER-2013) foi introduzido em 2013 no *Challenges in Representation Learning* (ICML 2013) [13], consiste em 28.709 imagens no conjunto de treinamento e 3.589 imagens no conjunto de teste. Criado com a API (em inglês, *Application Programming Interface*) de pesquisa de imagens do *Google*, suas imagens possuem variâncias, tais como, pose, iluminação, faixa etária, etc., o que torna a classificação desta base de imagens, em um grande desafio.

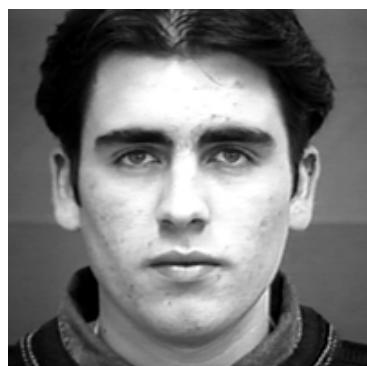
3.3 Pré-processamento

Devido à natureza escassa das bases de imagens utilizadas neste trabalho, aplicamos o processo de *Data Augmentation* onde o código fonte está disponível no repositório *GitHub*⁹ e pode ser visto no Apêndice C, processo este que é aplicado ao conjunto de treinamento com a finalidade de gerar sub-amostras diferentes para cada imagem, com a finalidade de alimentar, e consequentemente, melhorar os resultados do nosso classificador (*SoftMax*), além de ajudar a prevenir *Overfitting* [85] que consiste em um modelo que se ajusta bem ao conjunto de treinamento, entretanto, é ineficaz de generalizar-se bem a dados até então não vistos.

A Figura 3.3 mostra exemplos de sub-amostras extraídas de uma imagem de entrada.

Figura 3.3: Resultado do processo de *Data Augmentation*.

(a) Imagem de entrada.



(b) Sub-amostras.



Fonte: Elaborado pela autora.

As bases de imagens JAFFE, Extended CK+ e FER-2013 possuem respectivamente um total de 150, 10.256 e 28.709 imagens no conjunto de treinamento, todas, subdivididas em sete classes distintas.

Com o processo de *Data Augmentation*, estes números cresceram para um total de 2.360, 66.089 e 91.510 imagens no conjunto de treinamento respectivamente, indicando um aumento superior a 100% no conjunto de dados de todas as três bases de imagens, o que garante uma melhor representação dos dados de entrada.

Nas Tabelas 3.1, 3.2 e 3.3 é enumerado a distribuição realizada neste experimento para os conjuntos de dados de treinamento e para os conjuntos de dados de teste das bases de imagens JAFFE, Entended CK+ e FER-2013 respectivamente, onde pode ser visto a quantidade de imagens distribuídas em cada uma das sete classes.

⁹<https://github.com/whoisraibolt/Data-Augmentation>

Tabela 3.1: Conjunto de dados de treinamento e teste da base de imagens JAFFE.

Faces	Conjunto de treinamento	Conjunto de treinamento (após processo de DA)	Conjunto de teste
Raiva	20	311	10
Nojo	19	299	10
Medo	26	412	6
Feliz	21	328	10
Triste	23	361	8
Surpreso	21	334	9
Neutro	20	315	10
Total	150	2.360	63

Tabela 3.2: Conjunto de dados de treinamento e teste da base de imagens Extended CK+.

Faces	Conjunto de treinamento	Conjunto de treinamento (após processo de DA)	Conjunto de teste
Raiva	990	7.254	44
Nojo	501	5.789	59
Medo	983	8.677	55
Feliz	1.640	10.756	96
Triste	1.372	10.140	63
Surpreso	1.403	10.173	100
Neutro	3.367	13.300	148
Total	10.256	66.089	565

Tabela 3.3: Conjunto de dados de treinamento e teste da base de imagens FER-2013.

Faces	Conjunto de treinamento	Conjunto de treinamento (após processo de DA)	Conjunto de teste
Raiva	3.995	13.976	467
Nojo	436	3.979	56
Medo	4.097	13.928	496
Feliz	7.215	17.214	895
Triste	4.830	14.757	653
Surpreso	3.171	12.757	415
Neutro	4.965	14.899	607
Total	28.709	91.510	3.589

Além disso, foi utilizada a técnica de *Region of Interest* (ROI), onde o código fonte pode ser visto no repositório *GitHub*¹⁰ e no Apêndice D, esta técnica possui a finalidade de definir uma área de captura demarcando a face dos indivíduos da base de imagens Extended CK+, e posteriormente as mesmas foram convertidas para escala de cinza. Enquanto as bases de imagens JAFFE e FER-2013 não sofreram nenhuma alteração deste tipo, uma vez que já se encontravam de forma a atender as demandas deste trabalho (escala de cinza).

¹⁰<https://github.com/whoisraibolt/ROI-Multi-Images>

Realizamos nossos testes com as imagens em escala de cinza (1) pois as imagens das bases JAFFE e FER-2013 encontram-se em escala de cinza, e as imagens da base Extended CK+ encontram-se parcialmente em escala de cinza; (2) para reduzir a complexidade do modelo, uma vez que a complexidade de imagens em RGB (em inglês, *Red*, *Green*, *Blue*) é maior em relação as imagens em escala de cinza.

As imagens de todas as bases de imagens foram padronizadas para o mesmo tamanho (48x48).

3.3.1 Formato .pkl

Como já mencionado na Seção 3.2 utilizamos para avaliar a técnica apresentada e validá-la com a tarefa de FER três bases de imagens públicas de expressão facial: JAFFE, Extended CK+ e FER-2013. Neste sentido, utilizaremos as três bases de imagens como conjunto de dados de entrada para o LBCNN proposto neste trabalho.

A fim de tornar a leitura os conjuntos de dados mais eficiente em questões de desempenho, prevenção de retrabalho, etc., optamos em realizar a serialização das três bases de imagens para o formato .pkl ou *Pickle* como é conhecido. A serialização de dados transforma dados quaisquer em sequencias de bytes. Na linguagem *Python*, existem diferentes módulos para a serialização de dados, *Pickle* é um deles.

Pickle ou *pickling* (processo de serialização) é o termo empregado para o método de conversão aplicado a qualquer tipo de estrutura de objetos em *Python* para byte *streams* (0s e 1s). Existe também o termo *Unpickling* (processo de desserialização), termo este empregado para a realização do método oposto, ou seja, a conversão de byte *streams* (recriação) para objetos *Python*.

As vantagens em executar a serialização de dados está na possibilidade de salvar dados complexos, além da geração de algum nível de segurança de dados (mesmo que pequeno), uma vez que os dados serializados tornam-se quase ilegíveis.

O código fonte do processo de serialização de dados aplicado nas três bases de imagens utilizadas neste trabalho com o modulo *Pickle*, está disponível no repositório *GitHub*¹¹ e pode ser visto no Apêndice E e F.

¹¹<https://github.com/whoisraibolt/Tensorflow-With-Own-Data>

3.3.2 Codificação *One-Hot*

Identificar a melhor prática para realizar o processamento dos dados de acordo com o problema em questão torna-se uma etapa importante para a resolução de projetos de ML.

No caso deste trabalho, as classes que representam as seis expressões faciais básicas apresentadas no Capítulo 1, e neutro, logo, formando sete emoções, são representadas por valores numéricos, como pode ser visto na Tabela 3.4 a seguir:

Tabela 3.4: Rótulos Categóricos x Valores Numéricos.

Rótulos Categóricos	Valores Numéricos
Raiva	0
Nojo	1
Medo	2
Feliz	3
Triste	4
Surpreso	5
Neutro	6

Entretanto, existem algoritmos de ML que não aceitam rótulos categóricos como dados de entrada, ou seja, não operam com rótulos categóricos diretamente, como é o exemplo das CNN. Além disso, utilizar valores numéricos para representar rótulos categóricos em algoritmos de ML não é considerado como sendo uma boa prática, pois é possível que estes valores gerados possam prejudicar e influenciar a efetividade do algoritmo através do processo de aprendizado.

Na presença desta restrição, convertemos os valores numéricos para representar os rótulos categóricos das três bases de imagens utilizadas neste trabalho (tanto os valores numéricos do conjunto de treinamento quanto os valores numéricos do conjunto de teste) para vetores binários que representam cada um dos valores numéricos, através da codificação *One-Hot*.

A codificação *One-Hot* é uma codificação de estados que consiste em gerar n vetores binários para n valores numéricos que representam cada um dos rótulos categóricos.

Deste modo, *One-Hot* realiza a codificação para cada estado, de forma que cada valor numérico seja representado por um único vetor binário de tamanho n igual a quantidade de classes presentes no conjunto de dados. Neste sentido, Todas as variáveis de estado recebem o valor igual a 0, exceto uma, que receberá o valor igual a 1.

É possível encontrar no código fonte das implementações de CNN convencional e LBCNN proposto neste trabalho, a seguinte função que realiza a codificação *One-Hot* nos rótulos de treinamento e teste:

Listagem 3.1: Codificação One-Hot implementada.

```

# Class labels are One-Hot coded, meaning that each label is a vector with
# 7 elements,
# all of which are zero except one element
def dense_to_one_hot(labels_dense, num_classes):
    num_labels = labels_dense.shape[0]
    index_offset = np.arange(num_labels) * num_classes
    labels_one_hot = np.zeros((num_labels, num_classes))
    labels_one_hot.flat[index_offset + labels_dense.ravel()] = 1

    return labels_one_hot
10
# Call function dense_to_one_hot()
data_train_labels = dense_to_one_hot(labels_dense=data_train_labels,
    num_classes=num_classes)
data_test_labels = dense_to_one_hot(labels_dense=data_test_labels,
    num_classes=num_classes)

```

Desta forma, através da comparação entre os vetores binários, é possível avaliar a acurácia da etapa de treinamento.

O resultado da Codificação *One-Hot* pode ser visto na Tabela 3.5 a seguir:

Tabela 3.5: Valores Numéricos x Codificação *One-Hot*.

Valores Numéricos	Codificação One-Hot
0	1000000
1	0100000
2	0010000
3	0001000
4	0000100
5	0000010
6	0000001

3.4 Rede Neural Convolucional Binária Local

Neste trabalho propomos uma adaptação do modelo de LBCNN [23], para reconhecimento de expressões faciais com o objetivo de classificar expressões faciais básicas.

A fim de reduzir a complexidade computacional em CNN convencionais, o LBCNN se

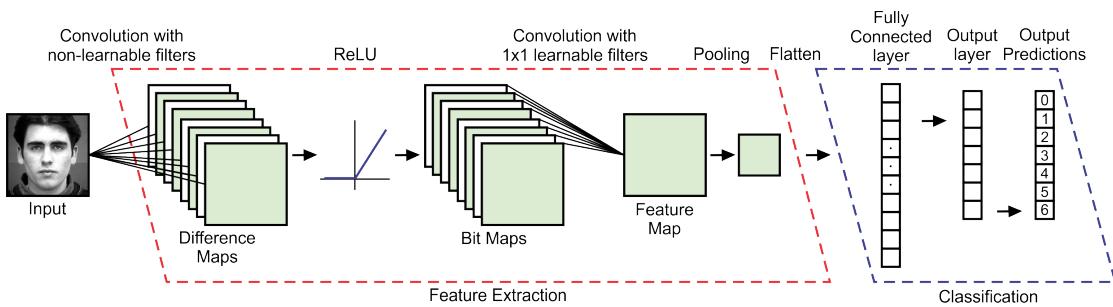
baseia nos princípios de LBP, e tráz como proposta, *Local Binary Convolution* (LBC), tornando-se uma poderosa alternativa para a camada convolucional em CNN convencionais.

Além de reduzir cálculos, e significativamente a quantidade de parâmetros aprendíveis durante a etapa de treinamento devido a sua natureza binária e esparsa, a camada LBC reduz a complexidade do modelo, consequentemente, acarretando em economias computacionais e requisitos de memória, tornando-se um modelo aplicável em ambientes reais que possuem recursos escassos e limitados.

A camada LBC é composta por, (1) um conjunto de filtros convolucionais de carácter binário, pré-definidos e fixos, ou seja, durante a etapa de treinamento, não são atualizados; (2) seguida por uma função de ativação não linear (ReLU); (3) que é seguida, por um conjunto de pesos lineares 1x1 aprendíveis.

Nosso modelo constitui-se em cinco camadas LBC, podendo ser ou não, seguida por *max-pooling*. A Figura 3.4 mostra a arquitetura básica do modelo de LBCNN proposto neste trabalho.

Figura 3.4: Arquitetura básica do modelo *Local Binary Convolutional Neural Network*.



Fonte: Adaptado de [23].

Utilizamos a distribuição de Bernoulli como uma generalização dos pesos em um LBP tradicional para gerar aleatoriamente nosso conjunto de filtros convolucionais de carácter binário, pré-definidos e fixos. Para isto, utilizamos uma variável aleatória para esta distribuição.

A distribuição discreta do espaço amostral da distribuição de Bernoulli consiste $\{0, 1\}$, portanto, não existe nenhum valor intermediário entre o seu espaço amostral.

Caso ocorra sucesso S a probabilidade de sucesso p será igual a 1, caso ocorra fracasso F a probabilidade de falha q será igual a $1 - p$, como pode ser visto na equação 3.1 quando a variável X é uma variável aleatória para esta distribuição:

$$\begin{aligned} P(X = 1) &= p \\ P(X = 0) &= 1 - p \end{aligned} \tag{3.1}$$

Além disto, a média E de uma variável aleatória de Bernoulli pode ser checada pela equação 3.2 abaixo:

$$E(X) = p \tag{3.2}$$

E a variância V de uma variável aleatória de Bernoulli pode ser checada pela equação 3.3 abaixo:

$$V(X) = p(1 - p) \tag{3.3}$$

Para o desenvolvimento deste trabalho, definimos para ser utilizado em todos os nossos experimentos, um nível de esparsidade igual a 0.5, em relação aos pesos que podem tolerar valores distintos de zero, em seguida atribuímos aleatoriamente 1 ou -1 a esses pesos.

4 Experimentos e Resultados

A eficiência do LBCNN implementado neste trabalho para a tarefa de FER foi avaliada comparando seu desempenho com uma CNN convencional, utilizando as bases de imagens já apresentadas.

A mesma arquitetura utilizada para implementar o modelo de CNN convencional serviu como base para a implementação do modelo de LBCNN proposto neste trabalho. Portanto, para fazer uma comparação junta entre as arquiteturas, todos os experimentos foram executados por 100 épocas, com um total de 20.000 iterações. Utilizamos uma taxa de aprendizado de $1e - 3$ e *Adam Optimizer*.

Desta forma, o número de filtros convolucionais, o número de camadas convolucionais, o número de unidades escondidas na camada totalmente conectada, foram os mesmos em ambos os modelos de redes.

Adam Optimizer [86] é um algoritmo de otimização como alternativa ao algoritmo Gradiente Estocástico de Descida (em inglês, *Stochastic Gradient Descent*) utilizado para realizar a atualização dos pesos de forma iterativa de acordo como o conjunto de dados de treinamento. Segundo D. Kinga e J. Ba. Adam [86] o algoritmo *Adam Optimizer* necessita de poucos requisitos de memória, além de ser computacionalmente eficiente.

Uma única taxa de aprendizado que não muda durante a etapa de treinamento é mantida pelo algoritmo *Stochastic Gradient Descent* para todas as atualizações de peso. O algoritmo *Adam Optimizer* não:

"The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients.". [86, p. 1].

Já em relação aos parâmetros do LBCNN, utilizamos filtros de tamanho 3x3 para gerar os filtros não aprendíveis como no LBP tradicional, por apresentar melhores resultados nos experimentos realizados, além de serem gerados aleatoriamente pela distribuição de Bernoulli, conforme descrito na Seção 3.4.

Os resultados obtidos através dos experimentos realizados podem ser vistos nas Tabelas 4.1 e 4.2, onde a Tabela 4.1 mostra a acurácia obtida na etapa de treinamento em cada base de imagem, enquanto a Tabela 4.2 mostra o tempo gasto na etapa de treinamento em cada base de imagem.

	CNN	LBCNN
JAFFE	73.0	77.8
Extended CK+	92.4	82.1
FER-2013	49.2	41.9

Tabela 4.1: Taxa de acurácia (porcentagem) na etapa de teste.

Na Tabela 4.1 é notável uma melhoria na taxa de acurácia entre o modelo LBCNN em comparação ao modelo CNN convencional no teste realizado com a base de imagens JAFFE. O modelo LBCNN teve um aumento de 4.8% na taxa de acurácia em comparação ao modelo CNN convencional.

Vale ressaltar que existem uma serie de dificuldades que podem ser encontradas ao analisar as três bases de imagens já apresentadas colocar referencia do capítulo, como por exemplo, no conjunto de treinamento e conjunto de teste da base de imagens Extended CK+, existem amostras de imagens em um ambiente controlado, entretanto, possui imagens de homens e mulheres, dificultando um pouco mais a generalização dos dados.

Já na base de imagens FER-2013, tanto em seu conjunto de treinamento quanto em seu conjunto de teste, existem amostras de imagens em ambientes não controlados, possui imagens de homens e mulheres de diversas etnias e faixas etárias, possuindo imagens com contrates escuros, imagens muito próximo aos rostos, imagens com expressões cobertas por mãos, etc.

Levando estes pontos em consideração, foi possível obter um melhor resultado no modelo LBCNN em comparação ao modelo CNN convencional no teste realizado com a base de imagens JAFFE, pois esta base de imagens possui em seu conjunto de treinamento e conjunto de teste, amostras de imagens geradas em um ambiente controlado, composto somente por imagens de mulheres, todas, possuindo a mesma etnia e faixa etária.

	CNN	LBCNN
JAFFE	0:46:15	0:16:52
Extended CK+	0:46:23	0:16:55
FER-2013	0:46:31	0:04:52

Tabela 4.2: Tempo gasto (horas) na etapa de treinamento.

Foi estimado no GCE o seu custo efetivo ao utilizar a instância de máquina virtual de alto

desempenho apresentada na seção 3.1, onde o custo x tempo de execução (gasto na etapa de treinamento) por mês é mostrado na Tabela 4.3 a seguir:

Custo	Tempo
\$ 608.49	730

Tabela 4.3: Custo (dólar) x tempo (horas) de execução por mês.

Além disto, o custo x tempo de execução (gasto na etapa de treinamento) por hora foi também estimado, como pode ser visto na Tabela 4.4 a seguir:

Custo	Tempo
\$ 0.836	1

Tabela 4.4: Custo (dólar) x tempo (horas) de execução por hora.

Como visto nas Tabelas 4.1 e 4.2, em média, o modelo de CNN convencional leva cerca de 46 minutos para realizar todo o seu processamento na etapa de treinamento em cada base de imagem, enquanto o modelo LBCNN implementado neste trabalho em média, leva cerca de 16 minutos para realizar todo o seu processamento na etapa de treinamento em cada base de imagem. Desta forma é possível estimar o custo efetivo gerado ao executar ambos os modelos apresentados, como pode ser visto na Tabela 4.5 a seguir:

	Custo	Tempo
CNN	0.639	46
LBCNN	0.222	16

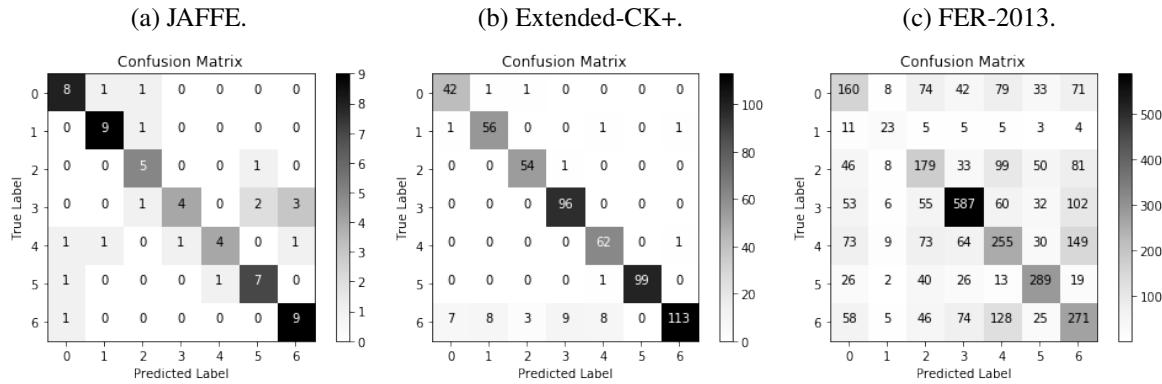
Tabela 4.5: Custo efetivo ao executar o modelo LBCNN em comparação ao modelo de CNN convencional.

Desta forma, é notável a eficiência e baixa complexidade computacional do LBCNN implementado neste trabalho, sendo possível obter uma taxa de classificação satisfatória nos testes realizados em cada base de imagem com um custo baixo, gerando uma economia de \$ 0.416 ao executar o modelo LBCNN em comparação ao modelo de CNN convencional.

As Figuras 4.1 e 4.2, respectivamente, mostram as matrizes de confusão das acuráciais obtidas na etapa de teste do modelo CNN convencional e LBCNN implementadas neste trabalho, onde foi avaliado o desempenho do algoritmo nas três bases de dados de expressões faciais públicas utilizadas neste trabalho.

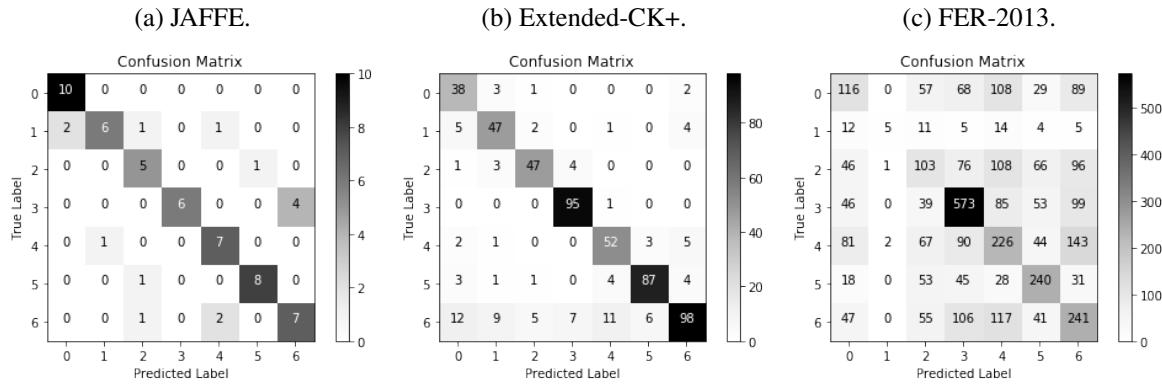
Ressaltamos que o objetivo deste trabalho não é obter uma maior precisão na etapa de teste em relação à CNN convencional, mas sim revelar sua eficiência, economia e baixa complexidade computacional em relação à CNN convencional para a tarefa de FER que resulta em um

Figura 4.1: Matrix de Confusão da CNN convencional.



Fonte: Elaborado pela autora.

Figura 4.2: Matrix de Confusão do LBCNN.



Fonte: Elaborado pela autora.

tempo gasto na etapa de treinamento relativamente baixa em relação a CNN convencional, enquanto a taxa de acurácia obtida é, em vezes relativamente melhor ou insignificante baixa (se comparado ao tempo gasto na etapa de treinamento).

5 Considerações Finais

Neste trabalho, propusemos o uso de LBCNN para a tarefa de FER, portanto, classificar as sete expressões faciais (Raiva, Nojo, Medo, Feliz, Triste e Surpresa) acrescentadas pela expressão neutra, compondo sete emoções básicas.

As bases teóricas que sustentaram a adaptação foram apresentadas. O LBCNN foi implementado em *Python* usando o *framework TensorFlow*. Experimentos foram realizados para comparar sua eficiência com um modelo CNN convencional.

Nossa abordagem mostrou-se eficiente em relação à sua precisão, custo efetivo e ao tempo gasto na etapa de treinamento, na qual é possível realizar a extração de características mais rapidamente.

5.1 Trabalhos Futuros

Embora existam várias pesquisas no campo de FER usando imagens estáticas usando CNN, ainda é um problema para aplicar esses modelos de redes em ambientes reais por causa das condições adversas nesses ambientes e um grande número de parâmetros; estes modelos tornam-se computacionalmente complexos.

No entanto, há uma grande necessidade na área de Robótica Assistiva, através de metodologias onde a FER realizada por dispositivos robóticos é necessária, como em monitoramento, comunicação e outros cenários.

Como continuação deste trabalho, propomos incorporar o modelo LBCNN apresentado neste trabalho em uma plataforma robótica autônoma. A plataforma robótica consiste de um robô Pioneer 3DX equipado com uma câmera RGBD, um laser de sensor Sick Lms200 e um computador usando o sistema operacional do robô (ROS), como descrito em [24, 25].

A ideia é integrar os funcionalistas já embutidos no robô com o método FER proposto neste trabalho. Na verdade, o robô é capaz de executar tarefas de detecção e identificação de pessoas e

tarefas simultâneas de localização e mapeamento (SLAM). O FER será uma função importante para tarefas Assistivas.

Ainda como continuação, pretendemos aplicar outras técnicas não abordadas neste trabalho, a fim de tornar o modelo de LBCNN proposto mais eficiente, tais técnicas como:

- Validação cruzada (em inglês, *Cross-validation*) [87] na etapa de treinamento a fim de ajudar a prevenir *Overfitting*;
- *Precision-Recall* para *Multiclass*;
- Regularização *Dropout* [88] para, durante a etapa de treinamento regularizar as camadas totalmente conectadas da rede a fim de ajudar a prevenir *Overfitting*;
- Utilização do conjunto de ferramentas de visualização TensorBoard [89] para a otimização e depuração do framework TensorFlow.

Reconhecimentos

A autora gostaria de agradecer à *Faculdade Tecnológica do Estado do Rio de Janeiro (FATEC RJ Petrópolis)* e ao *Laboratório Nacional de Computação Científica (LNCC)* por todo o suporte técnico.

Referências

- [1] António DAMÁSIO. “Em busca de Espinosa: prazer e dor na ciência dos sentimentos; adaptação para o português Brasil por Laura Teixeira Motta–São Paulo”. Em: *Companhia das Letras* (2004).
- [2] C Darwin. “A expressão das emoções no homem e nos animais. (LSL Garcia, Trad.)”. Em: *São Paulo: Cia das Letras. (Texto original publicado em 1872)* (2000).
- [3] Robert Plutchik. *Emotions and life: Perspectives from psychology, biology, and evolution.* American Psychological Association, 2003.
- [4] Paul Eckman. “Emotions revealed”. Em: *St. Martin’s Griffin, New York* (2003).
- [5] William James. “The Principles of”. Em: *Psychology* 2 (1890), p. 94.
- [6] Fabiano Koich Miguel. “Psicologia das emoções: uma proposta integrativa para compreender a expressão emocional”. Em: *Psico-usf* 20.1 (2015), pp. 153–162.
- [7] Steven J Haggstrom et al. “The 100 most eminent psychologists of the 20th century.” Em: *Review of General Psychology* 6.2 (2002), p. 139.
- [8] “The 2009 TIME 100: Paul Ekman.” Em: *Time, April 30th.* (2009).
- [9] Paul Ekman e Wallace V Friesen. “Constants across culture in the face and emotion”. Em: *Jenkins, Otaley, & Stein (eds.), Human Emotions: A Reader.* Malden, MA: Blackwell (1998).
- [10] Takeo Kanade, Jeffrey F Cohn e Yingli Tian. “Comprehensive database for facial expression analysis”. Em: *Automatic Face and Gesture Recognition, 2000. Proceedings. Fourth IEEE International Conference on.* IEEE. 2000, pp. 46–53.
- [11] Patrick Lucey et al. “The extended cohn-kanade dataset (ck+): A complete dataset for action unit and emotion-specified expression”. Em: *Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on.* IEEE. 2010, pp. 94–101.
- [12] Michael Lyons et al. “Coding facial expressions with gabor wavelets”. Em: *Automatic Face and Gesture Recognition, 1998. Proceedings. Third IEEE International Conference on.* IEEE. 1998, pp. 200–205.

- [13] Ian J Goodfellow et al. “Challenges in representation learning: A report on three machine learning contests”. Em: *International Conference on Neural Information Processing*. Springer. 2013, pp. 117–124.
- [14] Anthony M DiGioia III, Branislav Jaramaz e Bruce D Colgan. “Computer Assisted Orthopaedic Surgery: Image Guided and Robotic Assistive Technologies.” Em: *Clinical Orthopaedics and Related Research (1976-2007)* 354 (1998), pp. 8–16.
- [15] Ashraf Abbas M Al-modwahi et al. “Facial expression recognition intelligent security system for real time surveillance”. Em: *Proceedings of the International Conference on Computer Graphics and Virtual Reality (CGVR)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering e Applied Computing (WorldComp). 2012, p. 1.
- [16] Erin B McClure et al. “Facial expression recognition in adolescents with mood and anxiety disorders”. Em: *American Journal of Psychiatry* 160.6 (2003), pp. 1172–1174.
- [17] Corinna Cortes e Vladimir Vapnik. “Support-vector networks”. Em: *Machine learning* 20.3 (1995), pp. 273–297.
- [18] Roy de Groot. *Data Mining for Tweet Sentiment Classification: Twitter Sentiment Analysis*. LAP LAMBERT Academic Publishing, 2012.
- [19] Roberto Ligeiro Marques e INÊS Dutra. “Redes Bayesianas: o que são, para que servem, algoritmos e exemplos de aplicações”. Em: *Coppe Sistemas–Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brasil* (2002).
- [20] Zhengyou Zhang et al. “Comparison between geometry-based and gabor-wavelets-based facial expression recognition using multi-layer perceptron”. Em: *Automatic Face and Gesture Recognition, 1998. Proceedings. Third IEEE International Conference on*. IEEE. 1998, pp. 454–459.
- [21] Alex Krizhevsky, Ilya Sutskever e Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. Em: *Advances in neural information processing systems*. 2012, pp. 1097–1105.
- [22] Yann LeCun et al. “Gradient-based learning applied to document recognition”. Em: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.
- [23] Felix Juefei-Xu, Vishnu Naresh Boddeti e Marios Savvides. “Local Binary Convolutional Neural Networks”. Em: *arXiv preprint arXiv:1608.06049* (2016).

- [24] Alberto Torres Angonese e Paulo Fernando Ferreira Rosa. “Integration of People Detection and Simultaneous Localization and Mapping Systems for an Autonomous Robotic Platform”. Em: *Robotics Symposium and IV Brazilian Robotics Symposium (LARS/SBR), 2016 XIII Latin American*. IEEE. 2016, pp. 251–256.
- [25] Alberto Torres Angonese e Paulo Fernando Ferreira Rosa. “Multiple people detection and identification system integrated with a dynamic simultaneous localization and mapping system for an autonomous mobile robotic platform”. Em: *Military Technologies (ICMT), 2017 International Conference on*. IEEE. 2017, pp. 779–786.
- [26] International Federation of Robotics. *Executive Summary World Robotics 2017 Industrial Robots*. 2017. URL: https://ifr.org/downloads/press/Executive_Summary_WR_2017_Industrial_Robots.pdf (acesso em 11/06/2018).
- [27] International Federation of Robotics. *Executive Summary World Robotics 2017 Service Robots*. 2017. URL: https://ifr.org/downloads/press/Executive_Summary_WR_Service_Robots_2017_1.pdf (acesso em 11/06/2018).
- [28] Stephen Marsland. *Machine learning: an algorithmic perspective*. Chapman e Hall/CRC, 2011.
- [29] Roberta Barbosa Oliveira. “Método de detecção e classificação de lesões de pele em imagens digitais a partir do modelo chan-vese e máquina de vetor de suporte”. Em: (2012).
- [30] Fabio Abrantes Diniz et al. “RedFace: um sistema de reconhecimento facial baseado em técnicas de análise de componentes principais e autofaces”. Em: *Revista Brasileira de Computação Aplicada* 5.1 (2013), pp. 42–54.
- [31] JOSÉ MARIA PIRES DE MENEZES JUNIOR et al. “ANÁLISE DO RECONHECIMENTO DE FACES UTILIZANDO O ALGORITMO DE KNN”. Em: () .
- [32] Min-Ling Zhang e Zhi-Hua Zhou. “ML-KNN: A lazy learning approach to multi-label learning”. Em: *Pattern recognition* 40.7 (2007), pp. 2038–2048.
- [33] Min-Ling Zhang e Zhi-Hua Zhou. “A k-nearest neighbor based algorithm for multi-label classification”. Em: *Granular Computing, 2005 IEEE International Conference on*. Vol. 2. IEEE. 2005, pp. 718–721.
- [34] Nir Friedman, Dan Geiger e Moises Goldszmidt. “Bayesian network classifiers”. Em: *Machine learning* 29.2-3 (1997), pp. 131–163.
- [35] Mehran Sahami. “Learning Limited Dependence Bayesian Classifiers.” Em: *KDD*. Vol. 96. 1. 1996, pp. 335–338.

- [36] Pat Langley, Wayne Iba, Kevin Thompson et al. “An analysis of Bayesian classifiers”. Em: *Aaaai*. Vol. 90. 1992, pp. 223–228.
- [37] Xun Zhang, Jiale Deng e Rui Su. “The EM algorithm for a linear regression model with application to a diabetes data”. Em: *Progress in Informatics and Computing (PIC), 2016 International Conference on*. IEEE. 2016, pp. 114–118.
- [38] Shiqing Jia, Chunyan Hou e Jinsong Wang. “Software aging analysis and prediction in a web server based on multiple linear regression algorithm”. Em: *Communication Software and Networks (ICCSN), 2017 IEEE 9th International Conference on*. IEEE. 2017, pp. 1452–1456.
- [39] Peng Li et al. “Telecom customer churn prediction method based on cluster stratified sampling logistic regression”. Em: (2014).
- [40] Tatyana Bitencourt Soares de Oliveira. “Clusterização de dados utilizando técnicas de redes complexas e computação bioinspirada”. Tese de doutorado. Universidade de São Paulo, 2008.
- [41] Alexandre Xavier Ywata Carvalho et al. *Clusterização hierárquica espacial com atributos binários*. Rel. téc. Texto para Discussão, Instituto de Pesquisa Econômica Aplicada (IPEA), 2009.
- [42] John A Hartigan e Manchek A Wong. “Algorithm AS 136: A k-means clustering algorithm”. Em: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28.1 (1979), pp. 100–108.
- [43] Tapas Kanungo et al. “An efficient k-means clustering algorithm: Analysis and implementation”. Em: *IEEE Transactions on Pattern Analysis & Machine Intelligence* 7 (2002), pp. 881–892.
- [44] Stephen C Johnson. “Hierarchical clustering schemes”. Em: *Psychometrika* 32.3 (1967), pp. 241–254.
- [45] Fionn Murtagh. “A survey of recent advances in hierarchical clustering algorithms”. Em: *The Computer Journal* 26.4 (1983), pp. 354–359.
- [46] Rakesh Agarwal, Ramakrishnan Srikant et al. “Fast algorithms for mining association rules”. Em: *Proc. of the 20th VLDB Conference*. 1994, pp. 487–499.
- [47] Christian Borgelt e Rudolf Kruse. “Induction of association rules: Apriori implementation”. Em: *Compstat*. Springer. 2002, pp. 395–400.
- [48] Jeff Heaton. “Comparing dataset characteristics that favor the Apriori, Eclat or FP-Growth frequent itemset mining algorithms”. Em: *SoutheastCon, 2016*. IEEE. 2016, pp. 1–7.

- [49] Eui-Hong Han, George Karypis e Vipin Kumar. *Scalable parallel data mining for association rules*. Vol. 26. 2. ACM, 1997.
- [50] Bruno Vilela Oliveira. “BIO-SUMM–UMA ESTRATÉGIA DE REDUÇÃO DE COMPLEXIDADE DE INFORMAÇÃO NÃO ESTRUTURADA”. Tese de doutorado. Universidade Federal do Rio de Janeiro, 2012.
- [51] Thiago Alexandre Salgueiro Pardo, Lucia Helena Machado Rino e Maria das Graças Volpe Nunes. “GistSumm: A summarization tool based on a new extractive method”. Em: *International Workshop on Computational Processing of the Portuguese Language*. Springer. 2003, pp. 210–218.
- [52] Thiago Alexandre Salgueiro Pardo. “Gistsumm: Um sumarizador automático baseado na idéia principal de textos”. Em: *Série de Relatórios do Núcleo Interinstitucional de Linguística Computacional, São Paulo* (2002), pp. 1–10.
- [53] Christopher John Cornish Hellaby Watkins. “Learning from delayed rewards”. Tese de doutorado. King’s College, Cambridge, 1989.
- [54] Peter Auer. “Using confidence bounds for exploitation-exploration trade-offs”. Em: *Journal of Machine Learning Research* 3.Nov (2002), pp. 397–422.
- [55] Aurélien Garivier e Eric Moulines. “On upper-confidence bound policies for non-stationary bandit problems”. Em: *arXiv preprint arXiv:0805.3415* (2008).
- [56] Shipra Agrawal e Navin Goyal. “Thompson sampling for contextual bandits with linear payoffs”. Em: *International Conference on Machine Learning*. 2013, pp. 127–135.
- [57] Aditya Gopalan, Shie Mannor e Yishay Mansour. “Thompson sampling for complex online problems”. Em: *International Conference on Machine Learning*. 2014, pp. 100–108.
- [58] Olivier Chapelle, Bernhard Schölkopf e Alexander Zien. “Semi-Supervised Learning”. Em: () .
- [59] Xiaojin Zhu e Andrew B Goldberg. “Introduction to semi-supervised learning”. Em: *Synthesis lectures on artificial intelligence and machine learning* 3.1 (2009), pp. 1–130.
- [60] Russell Beale e Tom Jackson. *Neural Computing—an introduction*. CRC Press, 1990.
- [61] Ben Coppin. *Inteligência artificial*. Grupo Gen-LTC, 2015.
- [62] Warren S McCulloch e Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. Em: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.
- [63] F Rosenblatt. *The perceptron—a perceiving and recognizing automation*. Rel. téc. Report 85-460-1 Cornell Aeronautical Laboratory, Ithaca, 1957.

- [64] Marvin Minsky e Seymour Papert. “Perceptrons. 1969”. Em: *Cited on* (1990), p. 1.
- [65] Christophe Garcia e Manolis Delakis. “Convolutional face finder: A neural architecture for fast and robust face detection”. Em: *IEEE Transactions on pattern analysis and machine intelligence* 26.11 (2004), pp. 1408–1423.
- [66] Franck Mamalet, Sébastien Roux e Christophe Garcia. “Real-time video convolutional face finder on embedded platforms”. Em: *EURASIP Journal on Embedded Systems* 2007.1 (2007), p. 021724.
- [67] Yoon Kim. “Convolutional neural networks for sentence classification”. Em: *arXiv preprint arXiv:1408.5882* (2014).
- [68] Marc Moreno Lopez e Jugal Kalita. “Deep Learning applied to NLP”. Em: *arXiv preprint arXiv:1703.03091* (2017).
- [69] Maurice Peemen, Bart Mesman e Henk Corporaal. “Efficiency optimization of trainable feature extractors for a consumer platform”. Em: *International Conference on Advanced Concepts for Intelligent Vision Systems*. Springer. 2011, pp. 293–304.
- [70] Robert M Haralick. “Statistical and structural approaches to texture”. Em: *Proceedings of the IEEE* 67.5 (1979), pp. 786–804.
- [71] Bela Julesz. “Visual pattern discrimination”. Em: *IRE transactions on Information Theory* 8.2 (1962), pp. 84–92.
- [72] Rafael C Gonzalez e RC Woods. *Processamento digital de imagens. tradução: Cristina yamagami e leonardo piamonte*. 2010.
- [73] G Castellano et al. “Texture analysis of medical images”. Em: *Clinical radiology* 59.12 (2004), pp. 1061–1069.
- [74] Wenfeng Jin, Yunhong Wang e Tieniu Tan. “Text-independent writer identification based on fusion of dynamic and static features”. Em: *Advances in Biometric Person Authentication*. Springer, 2005, pp. 197–204.
- [75] Luo Bo e Cheng Jian. “Segmentation algorithm of high resolution remote sensing images based on LBP and statistical region merging”. Em: *Audio, Language and Image Processing (ICALIP), 2012 International Conference on*. IEEE. 2012, pp. 337–341.
- [76] Francesco Bianconi et al. “Automatic classification of granite tiles through colour and texture features”. Em: *Expert Systems with Applications* 39.12 (2012), pp. 11212–11218.
- [77] Timo Ojala, Matti Pietikainen e Topi Maenpaa. “Multiresolution gray-scale and rotation invariant texture classification with local binary patterns”. Em: *IEEE Transactions on pattern analysis and machine intelligence* 24.7 (2002), pp. 971–987.

- [78] Timo Ojala, Matti Pietikäinen e David Harwood. “A comparative study of texture measures with classification based on featured distributions”. Em: *Pattern recognition* 29.1 (1996), pp. 51–59.
- [79] Timo Ahonen, Abdenour Hadid e Matti Pietikainen. “Face description with local binary patterns: Application to face recognition”. Em: *IEEE transactions on pattern analysis and machine intelligence* 28.12 (2006), pp. 2037–2041.
- [80] Li Liu et al. “Median robust extended local binary pattern for texture classification”. Em: *IEEE Transactions on Image Processing* 25.3 (2016), pp. 1368–1381.
- [81] Faisal Ahmed et al. “Compound local binary pattern (CLBP) for robust facial expression recognition”. Em: *Computational Intelligence and Informatics (CINTI), 2011 IEEE 12th International Symposium on*. IEEE. 2011, pp. 391–395.
- [82] Marko Heikkilä, Matti Pietikäinen e Cordelia Schmid. “Description of interest regions with center-symmetric local binary patterns”. Em: *Computer vision, graphics and image processing*. Springer, 2006, pp. 58–69.
- [83] Marko Heikkilä, Matti Pietikäinen e Cordelia Schmid. “Description of interest regions with local binary patterns”. Em: *Pattern recognition* 42.3 (2009), pp. 425–436.
- [84] Matti Pietikäinen et al. *Computer vision using local binary patterns*. Vol. 40. Springer Science & Business Media, 2011.
- [85] Douglas M Hawkins. “The problem of overfitting”. Em: *Journal of chemical information and computer sciences* 44.1 (2004), pp. 1–12.
- [86] D Kinga e J Ba Adam. “A method for stochastic optimization”. Em: *International Conference on Learning Representations (ICLR)*. 2015.
- [87] M Stone. “Cross-validation: A review”. Em: *Statistics: A Journal of Theoretical and Applied Statistics* 9.1 (1978), pp. 127–139.
- [88] Nitish Srivastava et al. “Dropout: a simple way to prevent neural networks from overfitting”. Em: *The Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.
- [89] Sanjay Surendranath Girija. “Tensorflow: Large-scale machine learning on heterogeneous distributed systems”. Em: (2016).

APÊNDICE A – Código Fonte do processo de reformulação do LBP através de filtros convolucionais

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# Imports
5 from PIL import Image
from os import walk
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf

10 def load_dataset(path):
    f = []

    # Find all images in folder
    for (dirpath, dirnames, filenames) in walk(path):
        f.extend(filenames)

        #print dirpath
        #print dirnames
20        #print filenames

    images = []

    # For each image in folder
    for item in f:
        # Open and grayscale image
        image = Image.open(path+'/'+item).convert('L')

        # Get pixels from image (each pixel is into 0 to 255)
```

```

30     px1 = np.array(list(image.getdata()))
      px1 = px1.reshape(-1)

      images.append(px1)

35     images = np.array(images)

      return images, image.size,

# Get all images and their hidth and height
40 data, (img_sizeW, img_sizeH) = load_dataset('input')

img_size_flat = img_sizeW * img_sizeH

def heaviside(x):
45   # if -1: (-1+1)/2 = 0
   # if 0: (0+1)/2 = 0.5
   # if 1: (1+1)/2 = 1
   return 0.5 * (tf.sign(x) + 1) # 0.5 equal /2

50 H = [
      [[ 0, 0, 0, 0, 0, 0, 0, 1]],
      [[ 1, 0, 0, 0, 0, 0, 0, 0]],
      [[ 0, 1, 0, 0, 0, 0, 0, 0]]
    ],
    [
55   [[ 0, 0, 0, 0, 0, 0, 1, 0]],
   [[-1,-1,-1,-1,-1,-1,-1,-1]],
   [[ 0, 0, 1, 0, 0, 0, 0, 0]]
    ],
    [
60   [[ 0, 0, 0, 0, 0, 1, 0, 0]],
   [[ 0, 0, 0, 0, 1, 0, 0, 0]],
   [[ 0, 0, 0, 1, 0, 0, 0, 0]]
  ]
]

65 anchor_weights = tf.cast(H, tf.float32)

x = tf.placeholder(tf.float32, shape=[None, img_size_flat])

70 x_image = tf.reshape(x, [-1, img_sizeW, img_sizeH, 1])

bit_maps = tf.nn.conv2d(input=x_image,

```

```

    filter=anchor_weights ,
    strides=[1, 1, 1, 1],
    padding='SAME')

# Apply heaviside activation
bit_maps = heaviside(bit_maps)

80 v = [2**7,2**6,2**5,2**4,2**3,2**2,2**1,2**0]

# TensorFlow session
session = tf.Session()

85 # Initializing variables
session.run(tf.global_variables_initializer())

# Get convoluted images with heaviside activation applied
out = session.run([bit_maps], feed_dict={x: data})[0]

90
ances = []

# For each image (we have 8 bit maps per image now)
for i in range(out.shape[0]):
    # Get array for one image (8 bit maps)
    tmp = out[i]
    tmp = np.transpose(tmp,(2, 0, 1))

    for j in range(8):
        tmp[j] *= v[j]

    tmp = tmp.reshape(8, -1)

    # Get weighted sum
105 ans = np.sum(tmp, axis=0)
    ans = ans.reshape(img_sizeH, img_sizeW)
    ances.append(ans)

    ances = np.array(ances)

110 output_path = 'output/'

for i in range(len(ances)):
    # For each pixel it into (0 – 255) now
    arr = ances[i].astype('uint8')

```

```

# Get image from that array
image = Image.fromarray(arr)

# Save as .jpg
image.save(output_path+str(i)+'.png')

# Helper-function for plotting an image
def plot_image(image):
    plt.imshow(image.reshape(200, 200),
               interpolation='nearest',
               cmap='gray')

    plt.show()

image = data[0] # Change 0 to see other images

plot_image(image)

# Helper-function for plotting convolutional weights
def plot_conv_weights(weights, input_channel=0):
    # Assume weights are TensorFlow ops for 4-dim variables
    # e.g. weights_conv1 or weights_conv2

    # Retrieve the values of the weight-variables from TensorFlow
    # A feed-dict is not necessary because nothing is calculated
    w = session.run(weights)

    # Get the lowest and highest values for the weights
    # This is used to correct the colour intensity across
    # the images so they can be compared with each other
    w_min = np.min(w)
    w_max = np.max(w)

    # Number of filters used in the conv. layer
    num_filters = w.shape[3]

    # Create figure with a grid of sub-plots
    fig, axes = plt.subplots(4, 2)

    # Plot all the filter-weights
    for i, ax in enumerate(axes.flat):
        # Only plot the valid filter-weights

```

```

160     if i < num_filters:
161         # Get the weights for the i'th filter of the input channel
162         # See new_conv_layer() for details on the format
163         # of this 4-dim tensor
164         image = w[:, :, input_channel, i]
165
166         # Plot image
167         ax.imshow(image, vmin=w_min, vmax=w_max,
168                   interpolation='nearest', cmap='seismic')
169
170         # Remove ticks from the plot
171         ax.set_xticks([])
172         ax.set_yticks([])
173
174         plt.show()
175
176 # Plot weights
177 plot_conv_weights(weights=anchor_weights)
178
179 def plot_conv_feature_maps(layer, image):
180     # Assume layer is a TensorFlow op that outputs a 4-dim tensor
181     # which is the output of a convolutional layer,
182     # e.g. layer_conv1 or layer_conv2
183
184     # Create a feed-dict containing just one image
185     # Note that we don't need to feed y_true because it is
186     # not used in this calculation
187     feed_dict = {x: [image]}
188
189     # Calculate and retrieve the output values of the layer
190     # when inputting that image
191     values = session.run(layer, feed_dict=feed_dict)
192
193     # Number of filters used in the conv. layer
194     num_filters = values.shape[3]
195
196     # Create figure with a grid of sub-plots
197     fig, axes = plt.subplots(4, 2)
198
199     # Plot the output images of all the filters
200     for i, ax in enumerate(axes.flat):
201         # Only plot the images for valid filters
202         if i < num_filters:

```

```
# Get the output image of using the i'th filter
# See new_conv_layer() for details on the format
# of this 4-dim tensor
205   image = values[0, :, :, i]

# Plot image.
ax.imshow(image, interpolation='nearest', cmap='gray')

# Remove ticks from the plot
210   ax.set_xticks([])
       ax.set_yticks([])

plt.show()

215 # Plot Feature Maps
plot_conv_feature_maps(layer=bit_maps, image=image)

# Close session
220 session.close()
```

APÊNDICE B – Código Fonte da implementação do LBCNN proposto

```
#!/usr/bin/python
# coding: utf-8

# Imports
5 from datetime import timedelta
from scipy.stats import bernoulli
from sklearn.externals import joblib
from sklearn.metrics import confusion_matrix
import gzip
10 import itertools
import matplotlib.pyplot as plt
import numpy as np
import os
import tensorflow as tf
15 import time

# Dimensions of the data

# We know that images are 48 pixels in each dimension
20 in_height = 48
in_width = 48

# Images are stored in one-dimensional arrays of this length
image_size_flat = in_height * in_width
25

# Tuple with height and width of images used to reshape arrays
image_shape = (in_height, in_width)

# Classes info
30 classes = ['Angry', 'Disgust', 'Fear', 'Happy', 'Sad', 'Surprise', 'Neutral'
    ]
```

```

# Number of classes
num_classes = len(classes)

35 # Number of colour channels for the images: 1 channel for gray-scale
# Channels mean number of primary colors
num_channels = 1

# Placeholder variables
40
# Placeholder variable for the input images
x = tf.placeholder(tf.float32, shape=[None, image_size_flat])

# Reshape 'x'
45 x_image = tf.reshape(x, shape=[-1, in_height, in_width, num_channels])

# Placeholder variable for the true labels associated with the images
y_true = tf.placeholder(tf.float32, shape=[None, num_classes])

# We could also have a placeholder variable for the class-number,
# but we will instead calculate it using argmax
50 y_true_cls = tf.argmax(y_true, dimension=1)

# Load Dataset
55 def load_dataset(filename):
    upload_file = gzip.open(filename, 'rb')
    images_set, labels_set = joblib.load(upload_file)
    upload_file.close()

    return images_set, labels_set

# Returns the values of the function for images_set, labels_set on train-
# set
60 data_train_images, data_train_labels = load_dataset('Datasets/JAFFE/JAFFE-
    with-DA-training-set.pkl.gz')

# Returns the values of the function for images_set, labels_set on test-set
65 data_test_images, data_test_labels = load_dataset('Datasets/JAFFE/JAFFE-
    test-set.pkl.gz')

# Class labels are One-Hot coded, meaning that each label is a vector with
# 7 elements,
# all of which are zero except one element

```

```
70 def dense_to_one_hot(labels_dense , num_classes):
    num_labels = labels_dense.shape[0]
    index_offset = np.arange(num_labels) * num_classes
    labels_one_hot = np.zeros((num_labels , num_classes))
    labels_one_hot.flat[index_offset + labels_dense.ravel()] = 1
75
    return labels_one_hot

# Call function dense_to_one_hot()
data_train_labels = dense_to_one_hot(labels_dense=data_train_labels ,
    num_classes=num_classes)
80 data_test_labels = dense_to_one_hot(labels_dense=data_test_labels ,
    num_classes=num_classes)

print("Size of:")
print("- Training-set:\t\t{}\n".format(len(data_train_images)))
print("- Test-set:\t\t{}\n".format(len(data_test_images)))

85 data_test_cls = np.argmax(data_test_labels , axis=1)
# print data_test_cls

# Helper-function for catch the class
90 # number and print its corresponding name
def get_emotion(num_class):
    if num_class == 0:
        return 'Angry'
    elif num_class == 1:
        return 'Disgust'
95    elif num_class == 2:
        return 'Fear'
    elif num_class == 3:
        return 'Happy'
    elif num_class == 4:
100       return 'Sad'
    elif num_class == 5:
        return 'Surprise'
    elif num_class == 6:
        return 'Neutral'

105

# Helper-function for plotting images
def plot_images(images , cls_true , cls_pred=None):
    assert len(images) == len(cls_true) == 9
```

```
# Create figure with 3x3 sub-plots
fig, axes = plt.subplots(3, 3)
fig.subplots_adjust(hspace=0.3, wspace=0.3)

115 for i, ax in enumerate(axes.flat):
    # Plot image
    ax.imshow(images[i].reshape(image_shape), cmap='gray')

    # Show true and predicted classes
120    # T = True, and P = Predicted
    if cls_pred is None:
        xlabel = "T: {0}".format(get_emotion(cls_true[i]))
    else:
        xlabel = "T: {0}, P: {1}".format(get_emotion(cls_true[i]),
                                         get_emotion(cls_pred[i]))

125    # Show the classes as the label on the x-axis
    ax.set_xlabel(xlabel)

    # Remove ticks from the plot
130    ax.set_xticks([])
    ax.set_yticks([])

plt.show()

135 # Plot a few images to see if data is correct

# Get the first images from the test-set
images = data_test_images[0:9]

140 # Get the true classes for those images
cls_true = data_test_cls[0:9]

# Plot the images and labels using our helper-function above
plot_images(images=images, cls_true=cls_true)

145 # Convolutional Layer 1

# Convolution filters are 3 x 3 pixels
filter_height_1 = 3
filter_width_1 = 3

150 # There are 8 of these filters
```

```
num_filters_1 = 8

155 # Convolutional Layer 2

# Convolution filters are 3 x 3 pixels
filter_height_2 = 3
filter_width_2 = 3

160 # There are 16 of these filters
num_filters_2 = 16

# Convolutional Layer 3

165 # Convolution filters are 3 x 3 pixels
filter_height_3 = 3
filter_width_3 = 3

170 # There are 32 of these filters
num_filters_3 = 32

# Convolutional Layer 4

175 # Convolution filters are 3 x 3 pixels
filter_height_4 = 3
filter_width_4 = 3

# There are 64 of these filters
180 num_filters_4 = 64

# Convolutional Layer 5

# Convolution filters are 3 x 3 pixels
185 filter_height_5 = 3
filter_width_5 = 3

# There are 128 of these filters
num_filters_5 = 128

190 # Fully-connected layer
# Number of neurons in fully-connected layer
fc_size = 128

195 # Non-trainable filters initialized with distribution
```

```

# of Bernoulli as in article and then it's non-trainable
def new_weights_non_trainable(h,
                               w,
                               num_input,
                               num_output,
                               sparsity=0.5):

    # Number of elements
    num_elements = h * w * num_input * num_output

    # Create an array with n number of elements
    array = np.arange(num_elements)

    # Random shuffle it
    np.random.shuffle(array)

    # Fill with 0
    weight = np.zeros([num_elements])

    # Get number of elements in array that need be non-zero
    ind = int(sparsity * num_elements + 0.5)

    # Get it piece as indexes for weight matrix
    index = array[:ind]

    for i in index:
        # Fill those indexes with bernoulli distribution
        # Method rvs = random variates
        weight[i] = bernoulli.rvs(0.5)*2-1

    # Reshape weights array for matrix that we need
    weights = weight.reshape(h, w, num_input, num_output)

    # print weights

    return weights

def new_weights(shape):
    return tf.Variable(tf.truncated_normal(shape, stddev=0.05))

def new_biases(length):
    # Equivalent to y intercept
    # Constant value carried over across matrix math

```

```

    return tf.Variable(tf.constant(0.05, shape=[length]))

240 # Think of it as one block

def new_LBC_layer(input,                      # The previous layer
                  filter_height,        # Height of each filter
245                  filter_width,         # Width of each filter
                  in_channels,          # Num. channels in prev. layer
                  out_channels,         # # Number of filters
                  use_pooling=True):   # Use 2x2 max-pooling

250 # The out_channels of the previous layer are the in_channels of the
# next layer

# Shape of the filter-weights for the convolution
# This format is determined by the TensorFlow API
# shape = [filter_height, filter_width, in_channels, out_channels]

255 # Non-trainable filters
anchor_weights = tf.Variable(new_weights_non_trainable(
    h=filter_height,
    w=filter_width,
260    num_input=in_channels,
    num_output=out_channels).astype(np.float32),
    trainable=False)

# Difference Maps
265 difference_maps = tf.nn.conv2d(input=input,
    filter=anchor_weights,
    strides=[1, 1, 1, 1],
    padding='SAME')

270 # Non-linear unit is ReLU, as in article

# Bit Maps
bit_maps = tf.nn.relu(difference_maps)

275 # Set of learnable linear weights is a convolution with 1x1 kernels,
# without bias and without non-linear unit

shape = [1, 1, out_channels, 1]

280 weights = new_weights(shape)

```

```

# Feature Maps
feature_maps = tf.nn.conv2d(input=bit_maps,
                            filter=weights,
285                           strides=[1, 1, 1, 1],
                            padding='SAME')

# Use pooling to down-sample the image resolution?
if use_pooling:
    # This is 2x2 max-pooling, which means that we
    # consider 2x2 windows and select the largest value
    # in each window. Then we move 2 pixels to the next window
    feature_maps = tf.nn.max_pool(value=feature_maps,
                                    ksize=[1, 2, 2, 1],
295                           strides=[1, 2, 2, 1],
                                    padding='SAME')

# We will plot them later
return feature_maps, anchor_weights, weights
300

# Convolutional layer 1
feature_maps_conv_1, anchor_weights_conv_1, weights_conv_1 = new_LBC_layer(
    input=x_image,
    filter_height=filter_height_1,
305    filter_width=filter_width_1,
    in_channels=num_channels,
    out_channels=num_filters_1,
    use_pooling=False)

# Convolutional layer 2
feature_maps_conv_2, anchor_weights_conv_2, weights_conv_2 = new_LBC_layer(
    input=feature_maps_conv_1,
    filter_height=filter_height_2,
    filter_width=filter_width_2,
315    in_channels=1,
    out_channels=num_filters_2,
    use_pooling=False)

# Convolutional layer 3
feature_maps_conv_3, anchor_weights_conv_3, weights_conv_3 = new_LBC_layer(
    input=feature_maps_conv_2,
    filter_height=filter_height_3,
    filter_width=filter_width_3,
320

```

```
in_channels=1,  
325  out_channels=num_filters_3 ,  
     use_pooling=False)  
  
# Convolutional layer 4  
feature_maps_conv_4 , anchor_weights_conv_4 , weights_conv_4 = new_LBC_layer(  
330  input=feature_maps_conv_3 ,  
     filter_height=filter_height_4 ,  
     filter_width=filter_width_4 ,  
     in_channels=1,  
     out_channels=num_filters_4 ,  
335  use_pooling=False)  
  
# Convolutional layer 5  
feature_maps_conv_5 , anchor_weights_conv_5 , weights_conv_5 = new_LBC_layer(  
340  input=feature_maps_conv_4 ,  
     filter_height=filter_height_5 ,  
     filter_width=filter_width_5 ,  
     in_channels=1,  
     out_channels=num_filters_5 ,  
     use_pooling=True)  
345  
def flatten_layer(layer):  
    # Get the shape of the input layer  
    layer_shape = layer.get_shape()  
  
    # The shape of the input layer is assumed to be:  
    # layer_shape == [num_images, image_height, image_width, num_channels]  
  
    # The number of features is: image_height * image_width * num_channels  
    # We can use a function from TensorFlow to calculate this  
355  num_features = layer_shape[1:4].num_elements()  
  
    # Reshape the layer to [num_images, num_features]  
    # Note that we just set the size of the second dimension  
    # to num_features and the size of the first dimension to -1  
    # which means the size in that dimension is calculated  
    # so the total size of the tensor is unchanged from the reshaping  
360  layer_flat = tf.reshape(layer, [-1, num_features])  
  
    # The shape of the flattened layer is now:  
    # [num_images, image_height * image_width * num_channels]  
365
```

```

# Return both the flattened layer and the number of features
return layer_flat, num_features

370 layer_flat, num_features = flatten_layer(feature_maps_conv_5)

def new_fc_layer(input,           # The previous layer
                 num_inputs,      # Num. inputs from prev. layer
                 num_outputs,     # Num. outputs
375                 use_relu=True): # Use Rectified Linear Unit (ReLU)?

    # Create new weights and biases
    weights = new_weights(shape=[num_inputs, num_outputs])
    biases = new_biases(length=num_outputs)

380    # Calculate the layer as the matrix multiplication of
    # the input and weights, and then add the bias-values
    layer = tf.matmul(input, weights) + biases

    # Use ReLU?
    if use_relu:
        layer = tf.nn.relu(layer)

    return layer

390 # Fully-Connected Layer 1
layer_fc_1 = new_fc_layer(input=layer_flat,
                           num_inputs=num_features,
                           num_outputs=fc_size,
                           use_relu=True)

395 # Fully-Connected Layer 2 (Classes)
layer_fc_2 = new_fc_layer(input=layer_fc_1,
                           num_inputs=fc_size,
                           num_outputs=num_classes,
                           use_relu=False)

# Normalization of class-number output
y_pred = tf.nn.softmax(layer_fc_2)

400 # The class-number is the index of the largest element
y_pred_cls = tf.argmax(y_pred, dimension=1)

# Cross-entropy

```

```
410 cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=layer_fc_2,
                                                          labels=y_true)

# Average of the cross-entropy
loss = tf.reduce_mean(cross_entropy)

415 # Optimization method
optimizer = tf.train.AdamOptimizer(learning_rate=1e-3).minimize(loss)

# Measures of performance
420 correct_prediction = tf.equal(y_pred_cls, y_true_cls)

accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

# Calculates the total number of learning parameters
425 total_parameters = 0

for variable in tf.trainable_variables():
    variable_parameters = 1

430    for i in variable.get_shape():
        variable_parameters *= i.value

    total_parameters += variable_parameters

435 print("\nTotal number of learning parameters in the network model: %d" %
       total_parameters)

# Saver-object which is used for storing and retrieving
# all the variables of the TensorFlow graph
saver = tf.train.Saver()

440 epochs_completed = 0
index_in_epoch = 0
num_examples = data_train_images.shape[0]

# Serve data by batches
445 def next_batch(batch_size):

    global data_train_images
    global data_train_labels
    global index_in_epoch
    global epochs_completed
```

```
    start = index_in_epoch
    index_in_epoch += train_batch_size

455    # When all training data have been already used, it is reorder randomly
    if index_in_epoch > num_examples:
        # Finished epoch
        epochs_completed += 1

460    # Shuffle the data
    perm = np.arange(num_examples)
    np.random.shuffle(perm)

465    data_train_images = data_train_images[perm]
    data_train_labels = data_train_labels [perm]

        # Start next epoch
        start = 0
470    index_in_epoch = train_batch_size

        assert train_batch_size <= num_examples
    end = index_in_epoch

475    return data_train_images [start:end], data_train_labels [start:end]

# TensorFlow session
# Verify your GPU – Find out which device is use
session = tf.Session(config=tf.ConfigProto(log_device_placement=True))

480
# Restore or initialize variables

# We therefore save checkpoints during training so we can continue training
# at
# another time (e.g. during the night), and also for performing analysis
# later
485 # without having to train the neural network every time we want to use it

# If you want to restart the training of the neural network, you have to
# delete
# the checkpoints first

490 # Directory used for the checkpoints
save_dir = 'Checkpoints/JAFFE/'
```

```
# Create the directory if it does not exist
if not os.path.exists(save_dir):
    os.makedirs(save_dir)

# Base-filename for the checkpoints
save_path = os.path.join(save_dir, 'JAFFE')

# First try to restore the latest checkpoint. This may fail and raise an
# exception e.g.
# if such a checkpoint does not exist, or if you have changed the
# TensorFlow graph

try:
    print("Trying to restore last checkpoint...\n")
    # Find the latest checkpoint - if any
    last_checkpoint_path = tf.train.latest_checkpoint(checkpoint_dir=
        save_dir)

    # Try and load the data in the checkpoint
    saver.restore(session, save_path=last_checkpoint_path)

    # If we get to this point, the checkpoint was successfully loaded
    print("\nRestored checkpoint from" + last_checkpoint_path)

except:
    # If the above failed for some reason, simply
    # initialize all the variables for the TensorFlow graph
    print("\nFailed to restore checkpoint. Initializing variables instead.")
    session.run(tf.global_variables_initializer())

# Split the training-set into smaller batches of this size
train_batch_size = 128

# Counter for total number of iterations performed so far
total_iterations = 0

# Helper-function for performing a number of optimization iterations so as
# to gradually
# improve the variables of the network layers
```

```

530 # In each iteration , a new batch of data is selected from the training-set
      and then
# TensorFlow executes the optimizer using those training samples

# The progress is printed every 100 iterations
def training(num_iterations):

535     # Ensure we update the global variable rather than a local copy
     global total_iterations

# Start-time used for printing time-usage below
start_time = time.time()

540     for i in range(total_iterations ,
                      total_iterations + num_iterations):

545         # Get a batch of training examples:
         # x_batch now holds a batch of images and
         # y_true_batch are the true labels for those images
         x_batch, y_true_batch = next_batch(train_batch_size)

550         # Put the batch into a dict with the proper names
         # for placeholder variables in the TensorFlow graph
         feed_dict_train = {x: x_batch ,
                            y_true: y_true_batch}

555         # Run the optimizer using this batch of training data
         # TensorFlow assigns the variables in feed_dict_train
         # to the placeholder variables and then runs the optimizer
         session.run(optimizer, feed_dict=feed_dict_train)

560         # Print status every 100 iterations
         if i % 100 == 0:
             # Calculate the accuracy on the training-set
             acc = session.run(accuracy, feed_dict=feed_dict_train)

565             # Message for printing
             msg = "Optimization Iteration: {0:>6}, Training Accuracy:
{1:>6.1%}"

             # Print it
             print(msg.format(i + 1, acc))

```

```
# Save a checkpoint to disk every 1000 iterations
if i % 1000 == 0:
    # Save all variables of the TensorFlow graph to a checkpoint
    # Append the global_step counter to the filename so we save
575    # the last several checkpoints
    saver.save(session, save_path=save_path, global_step=i)

    print("Saved checkpoint.")

# Update the total number of iterations performed.
580 total_iterations += num_iterations

# Ending time.
end_time = time.time()

585 # Difference between start and end-times.
time_dif = end_time - start_time

# Print the time-usage.
590 print("Time usage: " + str(timedelta(seconds=int(round(time_dif)))))

# Helper-function to plot example errors
def plot_example_errors(cls_pred, correct):
    # This function is called from print_test_accuracy() below

595    # cls_pred is an array of the predicted class-number for
    # all images in the test-set

    # correct is a boolean array whether the predicted class
    # is equal to the true class for each image in the test-set

    # Negate the boolean array
600    incorrect = (correct == False)

    # Get the images from the test-set that have been
    # incorrectly classified
    images = data_test_images[incorrect]

    # Get the predicted classes for those images
605    cls_pred = cls_pred[incorrect]

    # Get the true classes for those images
    cls_true = data_test_cls[incorrect]
```

```
615 # Plot the first 9 images
    plot_images(images=images[0:9],
                cls_true=cls_true[0:9],
                cls_pred=cls_pred[0:9])

620 # Helper-function to plot confusion matrix
def plot_confusion_matrix(cls_pred):
    # This is called from test() below

    # cls_pred is an array of the predicted class-number for
    # all images in the test-set

    # Get the true classifications for the test-set
    cls_true = data_test_cls

    # Get the confusion matrix using sklearn
    cm = confusion_matrix(y_true=cls_true,
                           y_pred=cls_pred)

    # Print the confusion matrix as text
    # print(cm)

    # Plot the confusion matrix as an image
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j,
                  i,
                  cm[i, j],
                  horizontalalignment='center',
                  color='white' if cm[i, j] > thresh else 'black')

640 # Make various adjustments to the plot
    plt.imshow(cm, interpolation='nearest', cmap='Greys')
    plt.colorbar()
    tick_marks = np.arange(num_classes)
    plt.xticks(tick_marks, range(num_classes))
    plt.yticks(tick_marks, range(num_classes))
    plt.title('Confusion Matrix')
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')

650
    plt.show()
```

```
# Split the test-set into smaller batches of this size
test_batch_size = 32
660
# Helper-function for showing the performance
def test(show_example_errors=False,
         show_confusion_matrix=False):
    # Number of images in the test-set
    num_test = len(data_test_images)

    # Allocate an array for the predicted classes which
    # will be calculated in batches and filled into this array
    670   cls_pred = np.zeros(shape=num_test, dtype=np.int)

    # Now calculate the predicted classes for the batches
    # We will just iterate through all the batches

    # The starting index for the next batch is denoted i
    675   i = 0

    while i < num_test:
        # The ending index for the next batch is denoted j
        j = min(i + test_batch_size, num_test)

        # Get the images from the test-set between index i and j
        images = data_test_images[i:j, :]

        # Get the associated labels.
        685   labels = data_test_labels[i:j, :]

        # Create a feed-dict with these images and labels
        feed_dict = {x: images,
                    y_true: labels}

        # Calculate the predicted class using TensorFlow
        690   cls_pred[i:j] = session.run(y_pred_cls, feed_dict=feed_dict)

        # Set the start-index for the next batch to the
        # end-index of the current batch
        i = j

        # Convenience variable for the true class-numbers of the test-set
```

```
700     cls_true = data_test_cls

    # Create a boolean array whether each image is correctly classified
    correct = (cls_true == cls_pred)

705    # Calculate the number of correctly classified images
    # When summing a boolean array, False means 0 and True means 1
    correct_sum = correct.sum()

    # Classification accuracy is the number of correctly classified
710    # images divided by the total number of images in the test-set
    acc = float(correct_sum) / num_test

    # Print the accuracy
    msg = "Accuracy on Test-Set: {0:.1%} ({1} / {2})"
715    print(msg.format(acc, correct_sum, num_test))

    # Plot some examples of mis-classifications, if desired
    if show_example_errors:
        print("\nExample errors:")
        plot_example_errors(cls_pred=cls_pred, correct=correct)

720    # Plot the confusion matrix, if desired
    if show_confusion_matrix:
        print("Confusion Matrix:")
        plot_confusion_matrix(cls_pred=cls_pred)

    # Performance after 20000 optimization iterations
    training(num_iterations=20000)

725    test(show_example_errors=True,
          show_confusion_matrix=True)

    # Close session
    session.close()
```

APÊNDICE C – Código Fonte do processo de Data Augmentation

```
fill_mode='nearest')

30
# Import folder
def load_dataset(path):
    f = []

35    # Find all images in folder
    for (dir_path, dir_names, file_names) in walk(path):
        f.extend(file_names)

    # For each image in folder
40    for item in f:
        image = Image.open(path+'/'+item).convert('L')

        # Create a numpy array with shape (1, 500, 500)
        x = img_to_array(image)
        #x = np.asarray(x)

        # Convert to a numpy array with shape (1, 1, 500, 500)
        x = x.reshape((1,) + x.shape)

50    i = 0
        for batch in data_gen.flow(x, save_to_dir='Data Augmentation
Generated', save_prefix='DA', save_format='jpg'):
            i += 1
            if i > 9:
                break

55    print('Done!\n')

load_dataset('Test')
```

APÊNDICE D – Código Fonte do processo de ROI

```

#!/usr/bin/python
# -*- coding: utf-8 -*-

# Imports
5 import cv2
import glob

def process(filename , key):
    # Read image
10   image = cv2.imread(filename)

    # Convert to grayscale
    # image = cv2.cvtColor(image , cv2.COLOR_BGR2GRAY)

15    # Crop the image using array slices – ROI
    # First – height , Second – width
    cropped = image[150:450 , 500:800]

    # Display ROI images
20    # cv2.imshow("Cropped" , cropped)

    # Save images
    cv2.imwrite('Images/cropped_{}.jpg'.format(key) , cropped)

25    # Waits to press any key
    # cv2.waitKey(0)

    # Closes displayed windows
    # cv2.destroyAllWindows()

30    # Grab all images in a given directory
    for (i , image_file) in enumerate(sorted(glob.glob('Images/*.jpg'))):
        process(image_file , i)

```

```
35 # Message  
print 'Done! '
```

APÊNDICE E – Código Fonte do módulo dateP para serialização com Pickle

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# Imports
5 import cv2
import gzip
import numpy as np
import pickle
import random

10 class DateP(object):
    def __init__(self, file, total_size):
        self.file = file
        self.total_size = total_size

15    def date_process(self):
        label = []
        all_data = []
        print('\nReading...\n')

20    # Read train.txr or test.txt
        with open(self.file) as f:
            for line in f:

25        # Separate images and labels
            image_file = line.strip().split(' ')[0]
            label_file = line.strip().split(' ')[1]

            ...
            print 'Image file:',image_file
            print 'Label file:',label_file
```

```
    '''

# Append all labels
35    label.append(str(int(label_file)))

# To get path of images
path_image = 'dataset/' + image_file

40    # Get data location information and original size
        # and send to data_pro()
        data = data_pro(path_image, 0, 0, 384, 256)

# Append all data
45    all_data.append(data)

image = np.asarray(all_data, dtype=float)
labels = np.asarray(label, dtype=int)

50    # It divides left operand with the right
        # operand and assign the result to left operand
        image /= 255

# returns the number of elements in all_data
55    lens = len(all_data)

# Gives a new shape to image without changing its data.
image = image.reshape(lens, self.total_size)

60    # To zip
        toZip = list(zip(image, labels))
        random.shuffle(toZip)
        datas, labelss = map(list, zip(*toZip))

Data = np.asarray(datas, dtype=float)
Lable = np.asarray(labelss, dtype=int)
All = Data, Lable

70    return All

def data_pro(src, x1, y1, x2, y2):
    # Read image
    image_ = cv2.imread(src)
```

```
75 # ROI (region of interest)
    ROI = image_[x1:x2, y1:y2]

    # New size
    size = (48, 48)

80 # Resize image
    image = cv2.resize(image_, size)

    # Converts the image to grayscale
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    bb = np.zeros((image.shape[0], image.shape[1]), dtype=image.dtype)
    bb[:, :] = gray_image[:, :]

    cc = bb.reshape(1, image.shape[0] * image.shape[1])

90 return cc
```

APÊNDICE F – Código Fonte do processo de serialização com Pickle

```

#!/usr/bin/python
# -*- coding: utf-8 -*-

#joblib is usually significantly faster on large numpy arrays because it
5 # has a special handling for the array buffers of the numpy data structure

# Imports
from dateP.dateP import DateP
from sklearn.externals import joblib
10 import cv2
import gzip
import numpy as np
import os
import pickle
15 import random
import shutil

# text.txt or train.txt file path, should have 'imagepath label' as each
# row in .txt
file = 'dataset/train.txt'
20

# train or test and the total size of one image (48*48)
Object = DateP(file, 2304)
O1 = Object.date_process()
d = O1

25
# Print message
print('writing...\n')

# os.path.join([local], [filename])
filename = os.path.join('/home/alexandra', 'dataset-training-set.pkl')
30

```

```
# Generate the pkl file , save as .gz
# Dumping in a gzip compressed file using a compress level of 3
file = joblib.dump(d, filename + '.gz', compress=('gzip', 3))

35
print('Done !\n')
```