CENTRO PAULA SOUZA
FACULDADE DE TECNOLOGIA DE FRANCA
“DR. THOMAZ NOVELINO”

TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

GUILHERME MALAQUIAS FERREIRA

ESTUDO DE UM MODELO DE RECONHECIMENTO DE FALA CONSTRUÍDO COM O *FRAMEWORK TENSORFLOW*

FRANCA/SP
2017

GUILHERME MALAQUIAS FERREIRA

ESTUDO DE UM MODELO DE RECONHECIMENTO DE FALA CONSTRUÍDO COM O *FRAMEWORK TENSORFLOW*

Trabalho de Graduação apresentado à Faculdade de Tecnologia de Franca - "Dr. Thomaz Novelino", como parte dos requisitos obrigatórios para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Orientador(a): Profa. Dra. Jaqueline Brigladori Pugliesi.

FRANCA/SP

2017

GUILHERME MALAQUIAS FERREIRA

ESTUDO DE UM MODELO DE RECONHECIMENTO DE FALA CONSTRUÍDO COM O *FRAMEWORK TENSORFLOW*

Trabalho de Graduação apresentado à Faculdade de Tecnologia de Franca – "Dr. Thomaz Novelino", como parte dos requisitos obrigatórios para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Trabalho avaliado e aprovado pela seguinte Banca Examinadora:

Orientador(a).....: _____
Nome .....: Profa. Dra. Jaqueline Brigladori Pugliesi.
Instituição.....: Faculdade de Tecnologia de Franca – "Dr. Thomaz Novelino"

Examinador(a) 1 : _____
Nome .....: Prof. Me. Carlos Alberto Lucas.
Instituição.....: Faculdade de Tecnologia de Franca – "Dr. Thomaz Novelino".

Examinador(a) 2 : _____
Nome .....: Prof. Me. Jorge Luis Takahashi Hattori.
Instituição.....: Faculdade de Tecnologia de Franca – "Dr. Thomaz Novelino".

Franca, 16 de novembro de 2017.

AGRADECIMENTO

Agradecer aos corresponsáveis pela conclusão de um objetivo é sempre um clichê, mas um clichê necessário. Assim, gostaria de agradecer aos meus familiares, principalmente meus pais, Valmir Aparecido Ferreira e Miria Malaquias Ferreira, por terem acreditado em mim, sempre com amor e carinho, e por terem me proporcionado os princípios que definiram a minha liberdade de pensamento.

Também gostaria de agradecer aos inúmeros educadores, professores ou não, que contribuíram (e contribuem) para minha formação intelectual e humana, sempre me estimulando a buscar o raciocínio questionador, progressista e crítico, longe dos sensos comuns e das conclusões fáceis.

Por fim, e não menos importante, gostaria de fazer um agradecimento especial à minha companheira Letícia Saud (sem a qual, absolutamente, este trabalho não teria sido possível), pelo infinito amor, carinho e paciência em mim depositados.

Dedico este trabalho aos meus pais Valmir Aparecido Ferreira e Miria Malaquias Ferreira e à minha companheira Letícia Saud.

Há uma teoria que diz que, se um dia alguém descobrir exatamente para que serve o universo e por que ele está aqui, ele desaparecerá instantaneamente e será substituído por algo ainda mais bizarro e inexplicável.

Há outra que diz que isso já aconteceu.

Douglas Adams

RESUMO

O campo de pesquisa das Redes Neurais Artificiais (RNAs) possui grande relevância no mundo contemporâneo, sendo utilizado com sucesso na resolução dos mais diversos tipos de problema, como no tratamento do câncer e no reconhecimento de imagens e de fala. Especificamente no reconhecimento de fala, as RNAs têm sido extensivamente utilizadas, pois estas atingem um alto nível de precisão e adaptabilidade aos ruídos do ambiente. Assim, existem diversos *frameworks* e bibliotecas de programação que simplificam a construção de modelo de RNAs. Dessa forma, o objetivo deste trabalho é explicar o desenvolvimento de uma RNA utilizando o *framework TensorFlow*, demonstrando assim a capacidade e os recursos da ferramenta na construção de modelos preditivos. Para isso, foi utilizado como exemplo uma RNA construída para a classificação das palavras em inglês dos números de zero a nove. Além dos números, a rede distingue momentos de silêncio e palavras desconhecidas, ou seja, classifica um som como algum dos números – "*one*", "*two*", etc., como "*silence*" em casos de silêncio ou como "*unknown*" para palavras desconhecidas. Para a explicação do modelo preditivo, foram abordados os principais conceitos do fenômeno físico da fala humana, das RNAs e do *framework TensorFlow*, que possui diversos recursos para a construção de RNAs. Por fim, após a contextualização da teoria e dos algoritmos da rede utilizada, foram abordados o treinamento e as simulações realizadas para a verificação da precisão do modelo, feitas com um áudio de dez minutos (com sons contínuos) e com vinte sons isolados. Após a execução do algoritmo, a rede apresentou classificações com altos níveis de acurácia e baixos níveis de erros e falsos positivos, tanto no treinamento, quanto nas validações, nos testes e nas simulações posteriores. Dessa forma, utilizando o *TensorFlow*, em (razoavelmente) poucas linhas, usando vários métodos prontos e um alto nível de abstração, desenvolveu-se uma robusta RNA, o que permitiu concluir que, com o *framework*, é possível construir poderosos modelos preditivos de forma simplificada.

Palavras-chave: Reconhecimento de Fala. Redes Neurais Artificiais. *Framework TensorFlow*. Redes Neurais Convolucionais. Aprendizado de Máquina.

ABSTRACT

The research field of Artificial Neural Networks (RNAs) has great relevance in the contemporary world, being successfully used in the resolution of the most diverse types of problem, as in the treatment of cancer and in the recognition of images and speech. Specifically in speech recognition, ANNs have been extensively used because they achieve a high level of accuracy and adaptability to environmental noise. Thus, there are several frameworks and programming libraries that simplify the RNA model construction. Thus, the objective of this work is to explain the development of an RNA using the TensorFlow framework, thus demonstrating the tool's capabilities and capabilities in the construction of predictive models. For this, an RNA constructed for the classification of English words from numbers zero to nine was used as an example. Besides the numbers, the network distinguishes moments of silence and unknown words, that is, it classifies a sound as one of the numbers - "one", "two", etc., as "silence" in cases of silence or as "unknown" for unknown words. For the explanation of the predictive model, the main concepts of the physical phenomenon of human speech, of the RNAs and of the TensorFlow framework, which has several resources for the construction of RNAs, were discussed. Finally, after the contextualization of the theory and algorithms of the network used, the training and the simulations performed to verify the accuracy of the model, made with a ten minutes audio (with continuous sounds) and with twenty isolated sounds, were approached. After the execution of the algorithm, the network presented classifications with high levels of accuracy and low levels of errors and false positives, both in training, validations, tests and subsequent simulations. Thus, using the TensorFlow, in (reasonably) few lines, using several ready methods and a high level of abstraction, a robust RNA was developed, which allowed to conclude that, with the framework, it is possible to construct powerful predictive models of form simplified.

Keywords: Speech Recognition. Artificial Neural Networks. Framework TensorFlow. Convolutional Neural Networks. Machine Learning.

LISTA DE FIGURAS

Figura 1 – Aparelho fonador humano. 20
Figura 2 – *Televox*..... 21
Figura 3 – Diagrama do sistema nervoso humano. 25
Figura 4 – Neurônio biológico..... 26
Figura 5 – Modelo do neurônio de McCulloch e Pitts..... 28
Figura 6 – Gráficos das funções de ativação..... 30
Figura 7 – Funções linearmente separável (a) e não linearmente separável (b)..... 32
Figura 8 – Rede *feedforward* de camada única. 34
Figura 9 – Rede *feedforward* com múltiplas camadas. 35
Figura 10 – Rede *Perceptron* com realimentação. 36
Figura 11 – Outro exemplo de rede recorrente..... 36
Figura 12 – Diagrama do aprendizado supervisionado..... 39
Figura 13 – Diagrama do aprendizado não supervisionado..... 40
Figura 14 – *Perceptron* de uma camada. 44
Figura 15 – Algoritmo de treinamento do *Perceptron*. 45
Figura 16 – Algoritmo de operação do *Perceptron.* 45
Figura 17 – Treinamento do *Perceptron* de camada única. 46
Figura 18 – *Perceptron* de múltiplas camadas..... 47
Figura 19 – Algoritmo *backpropagation.* 50
Figura 20 – Algoritmo de operação do MLP. 50
Figura 21 – Representação do campo receptivo local de uma RNC..... 51
Figura 22 – Comparação entre uma MLP e uma RNC. 52
Figura 23 – Representação de uma imagem em três canais para uma RNC. 53
Figura 24 – Operação de convolução em uma imagem de um canal. 54
Figura 25 – Operação de *ReLU* em uma imagem cinza. 55
Figura 26 – Operação de *max-pooling* 56
Figura 27 – Arquitetura de uma RNC. 57
Figura 28 – Algoritmo de treinamento de uma RNC. 58
Figura 29 – Modelo original do neurônio de McCulloch e Pitts. 59
Figura 30 – Neurocomputador *Mark I – Perceptron*..... 60
Figura 31 – Histórico das RNAs. 63

Figura 32 – Instalação do programa *Docker*. 66
Figura 33 – Configuração do *Docker*. 67
Figura 34 – Execução de teste do *Docker*. 67
Figura 35 – *Download* e execução do contêiner do *TensorFlow*. 67
Figura 36 – Instalação de dependências para o projeto. 67
Figura 37 – Instalação de outra dependência e configuração do horário. 68
Figura 38 – Instalação do *software Bazel*. 68
Figura 39 – Ajuste da variável de ambiente "*PATH"*. 68
Figura 40 – *Script* de teste de funcionamento do *TensorFlow.* 68
Figura 41 – Clonando o repositório do *TensorFlow* no *GitHub*. 69
Figura 42 – Perguntas da configuração do *TensorFlow*. 69
Figura 43 – Persistindo mudanças do contêiner. 70
Figura 44 – Iniciando um novo contêiner baseado na imagem atualizada. 70
Figura 45 – Exemplo de criação de um tensor de dimensão 2. 71
Figura 46 – Exemplo de criação de uma variável com o método *tf.get_variable*..... 72
Figura 47 – Exemplo de utilização de um grafo *tf.Variable* 73
Figura 48 – Exemplo de atribuição de valor a uma variável..... 73
Figura 49 – Representação de um grafo com a operação *tf.add*. 74
Figura 50 – Exemplo de criação de um grafo para uma operação de soma. 74
Figura 51 – Utilização da *tf.Session* para a execução de uma operação de soma. . 75
Figura 52 – Utilização do método *tf.Session.run* para uma operação de soma. 75
Figura 53 – Etapas da rede *cnn-trad-fpool3*. 79
Figura 54 – Dicionário retornado com o método *models_prepare_model_settings*.. 84
Figura 55 – Execução do *script train.py*..... 91
Figura 56 – Informações do treinamento e validação da rede. 92
Figura 57 – Matriz de confusão do processo de testes da rede. 93
Figura 58 – Execução do *script freeze.py*. 93
Figura 59 – Execução do *script generate_streaming_test_wav.py*. 94
Figura 60 – Execução do programa *test_streaming_accuracy.cc*. 94
Figura 61 – Execução do *script label_wav.py*. 94

LISTA DE QUADROS

Quadro 1 – Exemplos de tipos de dados do *TensorFlow*. 72

LISTA DE TABELAS

Tabela 1 – Matriz de confusão. 42
Tabela 2 – Outro exemplo de uma matriz de confusão. 43
Tabela 3 – Testes de reconhecimento dos sons de “*zero*” a “*four*”. 95
Tabela 4 – Testes de reconhecimento dos sons de “*five*” a “*nine*”. 95
Tabela 5 – Testes de reconhecimento com sons desconhecidos (a). 96
Tabela 6 – Testes de reconhecimento com sons desconhecidos (b). 96

LISTA DE SIGLAS

AM – Aprendizado de Máquina
AP – Aprendizado Profundo
API – Application Programming Interface
CPU – Central Processing Unit
GPU – Graphics Processing Unit
IA – Inteligência Artificial
MCP – Modelo de McCulloch e Pitts
MFCC – Mel Frequency Cepstral Coefficients
MLP – Multilayer Perceptron
RNA – Rede Neural Artificial
RNC – Rede Neural Convolucional
RNP – Rede Neural Profunda

SUMÁRIO

1 INTRODUÇÃO 16

2 SOM E SISTEMAS DE RECONHECIMENTO DE FALA 19
2.1 OS FENÔMENOS FÍSICOS DO SOM E DA VOZ HUMANA 19
2.2 HISTÓRICO DOS SISTEMAS DE RECONHECIMENTO DE FALA 21

3 REDES NEURAIS ARTIFICIAIS 24
3.1 CONCEITOS GERAIS 24
3.2 O CÉREBRO HUMANO 25
3.3 NEURÔNIOS ARTIFICIAIS 28
3.4 FUNÇÕES DE ATIVAÇÃO 30
3.5 ARQUITETURAS DE REDE 32
3.5.1 Redes alimentadas adiante com camada única 33
3.5.2 Redes alimentadas adiante com múltiplas camadas 34
3.5.3 Redes recorrentes 35
3.6 APRENDIZADO E TREINAMENTO 36
3.6.1 Aprendizado supervisionado 38
3.6.2 Aprendizado não supervisionado 39
3.6.3 Conjunto de dados 40
3.6.4 Matriz de confusão 41
3.7 *PERCEPTRON* DE CAMADA ÚNICA 44
3.8 *PERCEPTRON* DE MÚLTIPLAS CAMADAS 46
3.8.1 Conceitos gerais 46
3.8.2 Algoritmo *backpropagation* 48
3.9 REDES NEURAIS CONVOLUCIONAIS 50
3.10 HISTÓRIA DAS REDES NEURAIS 59

4 *FRAMEWORK TENSORFLOW* 64
4.1 CONCEITOS INICIAIS 64
4.2 INSTALAÇÃO 65
4.3 FUNCIONAMENTO GERAL 70
4.3.1 Tensores 71
4.3.2 Tipos de dados e variáveis 72
4.3.3 Grafos e sessões 73
4.3.4 Estimadores 75

5 MODELO DE RECONHECIMENTO DE FALA 77
5.1 REDE *CNN-TRAD-FPOOL3* 77
5.1.1 Módulo de extração de características 78
5.1.2 Rede neural profunda *cnn-trad-fpool3* 78
5.1.3 Módulo de manuseio posterior 80
5.2 ESPECTROGRAMA E MÉTODO MFCC 80
5.3 FUNCIONAMENTO DAS FERRAMENTAS DE *SOFTWARE* 81
5.3.1 Ferramenta *train.py* 81
5.3.2 Ferramenta *freeze.py* 87
5.3.3 Ferramenta *generate_streaming_test_wav.py* 87
5.3.4 Ferramenta *test_streaming_accuracy.cc* 88
5.3.5 Ferramenta *label_wav.py* 89
5.4 GERAÇÃO DO MODELO PREDITIVO 89

CONSIDERAÇÕES FINAIS 97

REFERÊNCIAS 101

APÊNDICE A 108

1 INTRODUÇÃO

Desde 1940 um ramo da ciência que vem ganhando grande evidência é o da Inteligência Artificial (IA). Como discutido no trabalho de Minsky (1961, p. 8), não existe nenhuma definição universalmente aceita de inteligência, logo, delimitar o que é Inteligência Artificial é um assunto complexo e polêmico.

No entanto, um conceito que pode ser visto na pesquisa de Bellman (1978) é que IA pode ser vista como a automatização de atividades associadas ao pensamento humano, como a tomada de decisões, a resolução de problemas e o aprendizado. Ou ainda, de acordo com Kurzweil (1990), IA é a arte de criar máquinas que executam funções que exigem inteligência quando executadas por pessoas.

Por conseguinte, em um mundo cada vez mais conectado, é consenso que na história da humanidade nunca se produziu tantos dados quanto atualmente. Estima-se que até 2020 estejam sendo gerados 44 zetabytes (44 trilhões de gigabytes) de dados por ano (IDC, 2014). Nesse contexto, a IA surge como uma importante ferramenta para tratar e interpretar esse colossal volume de dados.

Conforme abordado por Russel e Norvig (2013), atualmente a IA é relevante para qualquer tarefa intelectual, abrangendo uma enorme variedade de subcampos, tais como: Sistemas *Fuzzy*, Árvores de Decisão, Algoritmos Genéticos, Mineração de Dados e Redes Neurais Artificiais (RNAs). Desses subcampos, um dos que mais tem suscitado estudos e aplicações nos últimos anos é o das RNAs, bastante relacionado ao Aprendizado de Máquina[1] (AM).

As RNAs foram baseadas no funcionamento do cérebro humano, e são utilizadas extensivamente na resolução de diversos tipos de problema, desde o tratamento do câncer até a solução de questões relacionadas a pecuária e astronomia, por exemplo.

Sendo assim, percebe-se que a programação completa, “do zero” do código de modelos de RNAs para a resolução de problemas complexos pode ser onerosa e dispendiosa. Dessa forma, nos últimos anos surgiram vários tipos de APIs (*Application Programming Interface*)[2], bibliotecas de *software* e *frameworks*[3] que têm

[1] Aprendizado de Máquina: Área que busca dar capacidade de aprendizado aos sistemas computacionais.
[2] API: Interface de Programação de Aplicativos, conjunto de padrões de programação para acesso a um sistema de *software*.

por objetivo facilitar o desenvolvimento de modelos de RNAs. Esses sistemas possibilitam simplicidade no desenvolvimento do código, maior produtividade e disponibilizam poderosos recursos para análise dos modelos de AM.

Dessa forma, tendo em vista a importância das RNAs na resolução de problemas e o recente surgimento destes *frameworks* de AM, o objetivo deste trabalho é explicar a construção de uma RNA utilizando o *framework* de Aprendizado de Máquina *TensorFlow*. Sendo assim, o trabalho visa responder, principalmente, a seguinte questão problema: “Como utilizar o *framework TensorFlow* no desenvolvimento de uma RNA”?

Para responder a essa questão, foi necessário utilizar uma RNA real, para que fosse possível explicar as etapas de seu desenvolvimento e o funcionamento de seu código. Assim, foi utilizada a RNA para reconhecimento de comandos de voz *cnn-trad-fpool3*, apresentada no trabalho de Sainath e Parada (2015) e construída pelo grupo TensorFlow Team (2017b).

Para o trabalho, a rede foi treinada com um banco de voz para a classificação das palavras em inglês dos números de zero a nove. Assim, foram abordados o funcionamento teórico, a lógica do código da implementação e as etapas da execução das ferramentas de *software* com o código da rede.

Por fim, a respeito da estrutura do trabalho, no segundo capítulo tentou-se responder o seguinte questionamento: “Como é produzido o som da voz humana”? Para isso, foi realizada uma caracterização da natureza física do som e da voz humana, além da apresentação de um breve histórico dos sistemas de reconhecimento de fala.

O objetivo do terceiro capítulo foi responder as questões: “O que são as RNAs e como podem ser utilizadas na resolução de problemas?” Assim, foi desenvolvida uma apresentação dos principais conceitos sobre as Redes Neurais Artificiais e especificamente das Redes Neurais Convolucionais, além de um histórico geral das RNAs.

Já no quarto capítulo buscou-se responder “O que é, como instalar e como funciona o *TensorFlow*”? Para isso, abordou-se os principais conceitos, as instruções de instalação e o funcionamento geral do *framework*.

[3] *Framework*: Abstração de programação para determinada linguagem ou tecnologia.

No quinto capítulo tentou-se responder o questionamento: “Como o *framework TensorFlow* foi utilizado no desenvolvimento da RNA para reconhecimento de fala”? Assim, foram discutidos os conceitos teóricos da RNA utilizada, além das ferramentas de *software* que foram usadas para a criação da rede e para a geração de estatísticas de precisão desta. Por fim, foram apresentados os passos executados para a geração do modelo preditivo, com a precisão das classificações da rede.

Nas Considerações finais foram discutidos os resultados e conclusões do trabalho e, no Apêndice A, foi apresentada a ferramenta de *software train.py*, utilizada para a geração do modelo de classificação.

2 SOM E SISTEMAS DE RECONHECIMENTO DE FALA

Inicialmente, antes de se falar propriamente do desenvolvimento do modelo de reconhecimento de comandos de voz utilizando o *framework TensorFlow*, é importante realizar uma contextualização sobre a natureza física do som e da produção da fala humana.

Dessa forma, neste capítulo são discutidos os fenômenos físicos do som e da voz humana, além da descrição de um breve histórico dos sistemas de reconhecimento de fala.

2.1 OS FENÔMENOS FÍSICOS DO SOM E DA VOZ HUMANA

De acordo com Young et al. (2003, p. 289), a definição mais geral de som consiste em dizer que este é uma onda longitudinal — os deslocamentos são paralelos à direção da propagação da onda — se propagando em um meio, geralmente, o ar. O ouvido humano possui uma impressionante sensibilidade, capaz de detectar ondas sonoras com uma intensidade muito pequena.

As ondas sonoras mais simples são as senoidais, que possuem valores definidos para a amplitude, a frequência e o comprimento de onda. O ouvido humano é sensível aos sons com frequências entre 20 e 20000 Hz.

Por conseguinte, de acordo com Callou e Leite (1990, p. 11), o campo que estuda os sons da fala humana como entidades físico-articulatórias isoladas é chamado de fonética. Assim, cabe à fonética descrever os sons da linguagem e analisar suas particularidades articulatórias, acústicas e perceptivas.

Segundo Cavaliere (2011, p. 11) e Cunha e Cintra (1985, p. 18), praticamente todos os sons da fala resultam das modificações vibratórias derivadas da ação de certos órgãos sobre a corrente de ar vinda dos pulmões. Para a produção do som, três condições são necessárias:

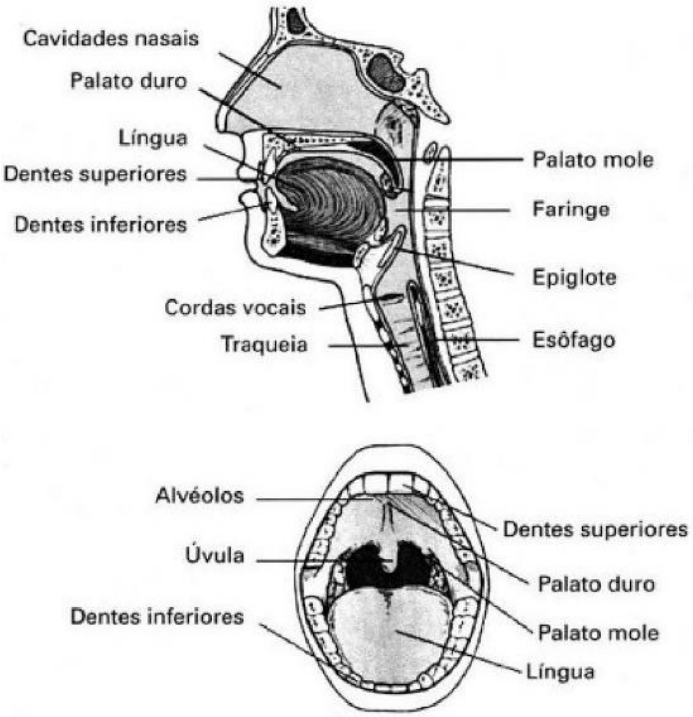
- Corrente de ar.
- Obstáculo encontrado por essa corrente de ar.
- Caixa de ressonância.

Ainda segundo Cunha e Cintra (1985, p. 18), estas condições são criadas pelos órgãos da fala denominados, em conjunto, aparelho fonador, que é composto por:

- Pulmões, brônquios e traqueia: órgãos respiratórios que fornecem a corrente de ar, matéria-prima da fonação.
- Laringe, onde se localizam as cordas vocais, que produzem energia sonora utilizada na fala.
- Cavidades supralaríngeas (faringe, boca e fossas nasais), que funcionam como caixas de ressonância, sendo que a cavidade bucal pode variar profundamente de forma e de volume, graças aos movimentos dos órgãos ativos, sobretudo da língua.

Os órgãos do aparelho fonador podem ser vistos na Figura 1.

Figura 1 – Aparelho fonador humano.



Fonte: Cavalieri, 2011, p. 35.

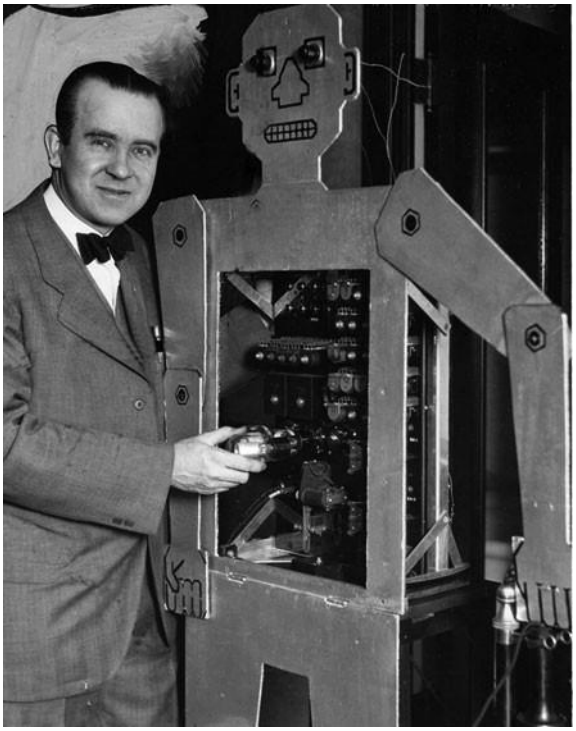
Por fim, em outros termos, articular um som implica atribuir-lhe um conjunto de características que o distingue dos demais sons, mediante estreitamentos ou oclusões que modulam a corrente de ar nas cavidades faríngea, bucal e nasal.

2.2 HISTÓRICO DOS SISTEMAS DE RECONHECIMENTO DE FALA

O objetivo de um sistema de reconhecimento automático de fala é compreender corretamente o enunciado de um indivíduo para a realização de alguma ação posterior.

De acordo com Pierrel (1987 apud HUGO, 1995, p. 19), por volta de 1930, o americano R. J. Wensley construiu o *Televox*, primeiro autômato capaz de receber ordens por telefone e executar alguns movimentos correspondentes. Uma foto do *Televox* pode ser vista na Figura 2.

Figura 2 – *Televox*.



Fonte: History of Computers, 2012, online.

No entanto, os primeiros sistemas de reconhecimento de fala apareceram somente na década de 1950. Segundo Bresolin (2008, p. 3), na década de 1950 vários pesquisadores tentaram explorar as ideias fundamentais de acústica e fonética. Em 1952, na Bell Labs, Davis, Biddulph e Balashek (1952) construíram um sistema para o reconhecimento de dígitos isolados (números de 0 a 9) para um único locutor.

Nos laboratórios da RCA (*Radio Corporation of America*), Olson e Belar (1956) tentaram desenvolver um sistema para reconhecer dez sílabas diferentes

faladas por uma pessoa. No final da década de 50, Fry (1959) e Denes (1959), pesquisadores da *University College of England* (Colégio Universitário da Inglaterra), construíram um reconhecedor de fonemas para detectar quatro vogais e nove consoantes, utilizando um analisador de espectro e uma combinação de padrões para fazer a decisão do reconhecimento.

Outro esforço notável neste período foi o reconhecedor de vogais de Forgie e Forgie (1959), construído no MIT (*Michigan Intitute of Tecnology* – Instituto de Tecnologia de Michigan), no *Lincoln Laboratories* (Laboratórios Lincoln) em 1959, utilizando um banco de filtros.

De acordo com Hugo (1995, p. 19), na década de 1960 a aparição dos métodos numéricos e a utilização do computador dão uma nova dimensão a estas pesquisas. Em 1966, sistemas em laboratório conseguem identificar corretamente de trinta a cinquenta palavras ditas por diferentes pessoas. Estas experiências eram baseadas na comparação das formas das palavras.

Ainda na década de 60, conforme Bresolin (2008, p. 3), é quando os laboratórios japoneses entram na área de reconhecimento de fala. O primeiro sistema japonês foi descrito por Suzuki e Nakata (1961) do *Tokyo Research Laboratory* (Laboratório de Pesquisas de Tóquio), e era um reconhecedor de vogais. Outro esforço japonês na construção de um sistema de reconhecimento de fala foi o trabalho de Sakai e Doshita (1962) da *Kyoto University* (Universidade de Kyoto), onde construíram um reconhecedor de fonemas que usava a análise de passagem por zero para fazer o reconhecimento de fala.

Segundo Bresolin (2008, p. 3), ainda na década de 1960, importantes projetos foram desenvolvidos, os quais tiveram uma grande implicação nas pesquisas na área de reconhecimento de fala nos vinte anos seguintes. Um deles foi a pesquisa de Martin, Nelson e Zadell (1964) e outros pesquisadores dos laboratórios RCA no final dos anos 60.

Martin desenvolveu um conjunto de métodos elementares de normalização no tempo baseado na habilidade de detectar o início e o fim da fala. Quase ao mesmo tempo, na então União Soviética, Vintsyuk (1968) propôs o uso de métodos de programação dinâmica. Outro projeto foi a pesquisa pioneira de Reddy (1966) no campo de reconhecimento de fala contínua por fonemas dinâmicos.

Ainda de acordo com Bresolin (2008, p. 4), na década de 70 a área de reconhecimento de palavras isoladas ou expressões discretas tornou-se viável com

a tecnologia baseada nos fundamentos estudados por Velichko e Zagoruyko (1970) na Rússia e Sakoe e Chiba (1978) nos Estados Unidos. Os russos estudaram o uso de padrões de reconhecimento e os japoneses os métodos de programação dinâmica. Além disso, os laboratórios *AT&T* e *Bell*, nos Estados Unidos, iniciaram uma série de pesquisas visando construir um sistema de reconhecimento de fala que pudesse entender uma pessoa falando.

Segundo Rabiner e Juang (1993 apud MARTINS, 1997, p. 3), a década de 80 caracterizou-se pela difusão de métodos baseados na modelagem estatística, como por exemplo os Modelos Ocultos de Markov. Nesse período também foi introduzido o uso de Redes Neurais Artificiais para reconhecimento de fala e foi dado um grande impulso para a implementação de sistemas robustos de reconhecimento de fala contínua para grandes vocabulários.

Nessa época foram obtidos sistemas com altas taxas de reconhecimento. De acordo com Martins (1997, p. 3), como exemplos desses sistemas, podem ser citados: o BYBLOS, de Kubala et al. (1988), com taxa de acerto de 93%, e o SPHINX, de Lee (1999) com taxa de acerto de 96,2%.

Ainda de acordo com Martins (1997, p. 3), nos anos 90 as pesquisas continuaram com a busca de um sistema de reconhecimento de fala contínuo, como o DARPA (*Defense Advanced Research Projects Agency* – Agência de Projetos Avançados de Pesquisa de Defesa), que visava reconhecer continuadamente e sem erros palavras dentro de um vocabulário de 1000 elementos.

Também nos anos 90 surgiu uma ferramenta desenvolvida por Cortes e Vapnik (1995), a chamada Máquina de Vetor de Suporte (*Suport Vector Machine* – SVM), que é uma poderosa classe de redes de aprendizagem supervisionada, utilizada principalmente para reconhecimento de padrões e regressão.

Na primeira década dos anos 2000 a maioria dos sistemas de reconhecimento de fala tinha como base o Modelo Oculto de Markov em conjunto com os descritores MFCC (*Mel-Frequency Cepstral Coefficient*) (BRESOLIN, 2008).

Por fim, desde meados dos anos 2006, uma forte tendência nas pesquisas nas áreas de reconhecimento de fala são as abordagens de *Deep Learning* (Aprendizado Profundo) e *Machine Learning*. Com isso, surgiram muitos produtos de reconhecimento de fala, tais como o *Google Cloud Speech API*, em Google (2017a), o *Google Now*, em Google (2017b), a *Siri*, da Apple Inc. (2017), a *Cortana*, da Microsoft Corporation (2017) e a *Alexa*, da Amazon Inc. (2017).

3 REDES NEURAIS ARTIFICIAIS

Neste capítulo são abordados os conceitos gerais das Redes Neurais Artificiais e, especificamente, das Redes Neurais Convolucionais.

3.1 CONCEITOS GERAIS

Segundo Rezende (2005, p. 3), o objetivo das pesquisas em IA é capacitar o computador para executar funções que são desempenhadas pelo ser humano usando conhecimento e raciocínio. Para que se possa aspirar à ação inteligente, deve-se analisar todos os aspectos relativos ao desenvolvimento e uso da inteligência.

Assim, a área das Redes Neurais Artificiais, também conhecida como neurocomputação, redes conexionistas ou processadores paralelamente distribuídos, é o campo que pesquisa modelos matemáticos baseados no funcionamento do cérebro humano, como abordado no trabalho de Braga, Carvalho e Ludermir (2000 apud REZENDE 2005, p. 142).

No entanto, como abordado por Kovács (2006, p. 21), é importante esclarecer que, apesar do conhecimento científico sobre o funcionamento do sistema nervoso humano ter crescido bastante nos últimos tempos, este ainda é muito incompleto. O que se conhece até o agora são apenas generalizações de observações isoladas realizadas em sistemas com condições controladas. Apesar disso, a estrutura fisiológica básica do cérebro é conhecida, e é exatamente nesta estrutura que se baseiam as RNAs.

Assim, de acordo com Haykin (2007, p. 27), a pesquisa em redes neurais tem sido motivada desde o início pela concepção de que o cérebro humano processa informações de uma maneira bastante diferente do computador digital tradicional. O cérebro é um sistema de processamento de informação altamente complexo, não linear e paralelo. Ele pode organizar seus elementos, os neurônios, de forma a realizar certos processamentos muito mais rapidamente que o computador mais veloz hoje existente.

Assim, tem-se que “um cérebro em “desenvolvimento” é sinônimo de um cérebro plástico: a plasticidade permite que o sistema nervoso em desenvolvimento se adapte ao meio ambiente” (HAYKIN, 2007, p. 27).

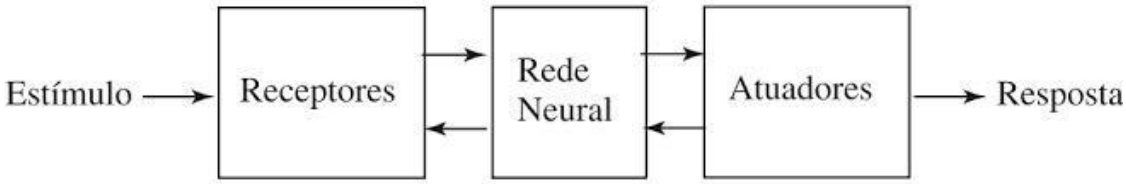
Dessa forma, de acordo com Aleksander e Morton (1990 apud Haykin, 2007, p. 28), uma rede neural pode ser vista como uma máquina adaptativa, ou seja, um processador maciçamente paralelo e distribuído, constituído de unidades de processamento simples, que têm propensão natural para armazenar conhecimento experimental e torná-lo disponível para o uso.

Por fim, as redes artificiais são normalmente implementadas com componentes eletrônicos ou então simuladas por programação em um computador digital, como abordado por Haykin (2007, p. 28).

3.2 O CÉREBRO HUMANO

De acordo com Haykin (2007, p. 32), o sistema nervoso humano pode ser visto como um sistema de três estágios, como mostrado no diagrama da Figura 3.

Figura 3 – Diagrama do sistema nervoso humano.



Fonte: Arbib, 1987 apud HAYKIN, 2007, p. 32.

Na figura 3, o centro do sistema é o cérebro, representado pela rede neural, que recebe continuamente informação, percebe esta e toma decisões apropriadas. As setas que apontam da esquerda para a direita indicam a transição para frente do sinal portador de informação, através do sistema.

Já as setas que apontam da direita para a esquerda representam a presença de realimentação do sistema. Os receptores convertem estímulos do corpo humano ou do ambiente externo em impulsos elétricos que transmitem informação para a rede neural (cérebro). Por sua vez, os atuadores convertem os impulsos elétricos gerados pela rede neural em respostas discerníveis como saídas do sistema.

Por conseguinte, de acordo com Haykin (2007, p. 32), verifica-se que os neurônios são de cinco a seis ordens de grandeza mais lentos que as portas lógicas de silício. Os eventos em um circuito de silício acontecem em nanossegundos (10^{-9} s), enquanto eventos neurais ocorrem em milissegundos (10^{-3} s). Entretanto, o

cérebro compensa a taxa de operação relativamente lenta dos neurônios pelo número realmente espantoso de neurônios (células nervosas), cada um com um volume maciço de conexões entre si.

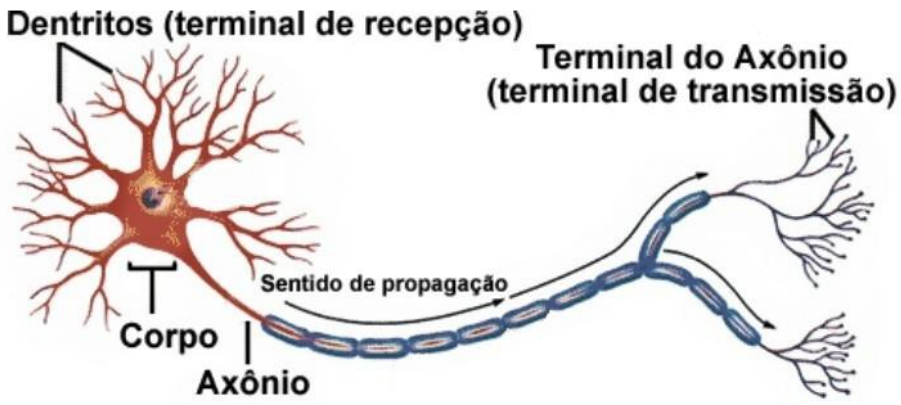
Assim, como verificado no trabalho de Silva, Spatti e Flauzino (2010, p. 30-31), estima-se que a rede neural de um humano adulto seja composta por cerca de 100 bilhões (10^{11}) de neurônios. Cada neurônio está interligado por conexões sinápticas em média a outros 6000 neurônios, perfazendo um total de 600 trilhões de sinapses.

De acordo com Faggin (1991 apud HAYKIN, 2007, p.32), esta organização neural resulta em uma estrutura extremamente eficiente. Mais especificamente, a eficiência energética do cérebro é de aproximadamente 10^{-16} joules por operação por segundo, enquanto o valor correspondente para os melhores computadores atuais é de cerca de 10^{-6} joules por operação por segundo.

Por conseguinte, conforme abordado no trabalho de Kovács (2006, p. 15), nas décadas de 1950 e 1960 passou-se a entender o neurônio biológico como sendo basicamente o dispositivo computacional elementar do sistema nervoso, que possuía muitas entradas e uma única saída. Este é um dos principais conceitos das redes neurais biológicas utilizado nas redes neurais artificiais.

Assim, ainda segundo Kovács (2006, p. 13), a célula nervosa, ou neurônio, foi identificada anatomicamente e descrita com notável detalhe pelo neurologista espanhol Cajal (1911). Um neurônio possui cerca de 100 *μm*, e é composto basicamente pelo corpo da célula, ou soma, de onde projetam-se extensões filamentares, os dendritos e o axônio, como pode ser visto na Figura 4.

Figura 4 – Neurônio biológico.



Fonte: Adaptado de Roberta et al., 2016, online.

De acordo com Kovács (2006, p. 13), os dendritos cobrem um volume muitas vezes maior do que o próprio corpo celular e formam uma árvore dendrital. O axônio, também conhecido como fibra nervosa, serve para realizar a conexão entre os neurônios.

Segundo Braga, Carvalho e Ludermir (2000, p. 6-7), os dendritos têm por função receber os impulsos nervosos, vindos de outros neurônios e conduzi-los até o corpo celular, onde a informação é processada e novos impulsos são gerados. Estes impulsos são transmitidos a outros neurônios, passando através do axônio até os dendritos dos neurônios seguintes.

O neurônio possui geralmente um único axônio, embora esse possa apresentar algumas ramificações, que podem se estender por vários metros. Alguns tipos de neurônios (a grande maioria dos que constituem o sistema nervoso central dos vertebrados) possui uma capa segmentada de mielina, que serve para acelerar a transmissão da informação pelo axônio.

Os sinais que chegam pelos axônios são pulsos elétricos conhecidos como impulsos nervosos ou potenciais de ação, e constituem a informação que o neurônio processará, de alguma forma, para produzir como saída um impulso nervoso no seu axônio.

Por conseguinte, as sinapses são regiões eletroquimicamente ativas compreendidas entre duas membranas celulares dos neurônios: a pré-sináptica, que é por onde chega um estímulo proveniente de uma outra célula, e a pós-sináptica, que é a do dendrito.

De acordo com Kovács (2006, p. 13), nesta região intersináptica o estímulo nervoso que chega até a sinapse é transferido à membrana dendrital através de substâncias conhecidas como neurotransmissores. O resultado dessa transferência é uma alteração no potencial elétrico da membrana pós-sináptica.

Segundo Braga, Carvalho e Ludermir (2000, p. 7), o tipo de neurotransmissor (há em torno de 100 diferentes tipos de neurotransmissores) liberado determinará a polarização ou a despolarização do corpo do neurônio seguinte. Assim, de acordo com o neurotransmissor liberado, a sinapse poderá ser inibitória ou excitatória.

A contribuição de todos os nós pré-sinápticos na polarização do neurônio pós-sináptico determinará se este gerará ou não um impulso nervoso. Portanto, o percentual de disparo de um neurônio é determinado pelo acúmulo de um número

grande de entradas inibitórias e excitatórias, medido pelo corpo da célula em um pequeno intervalo de tempo.

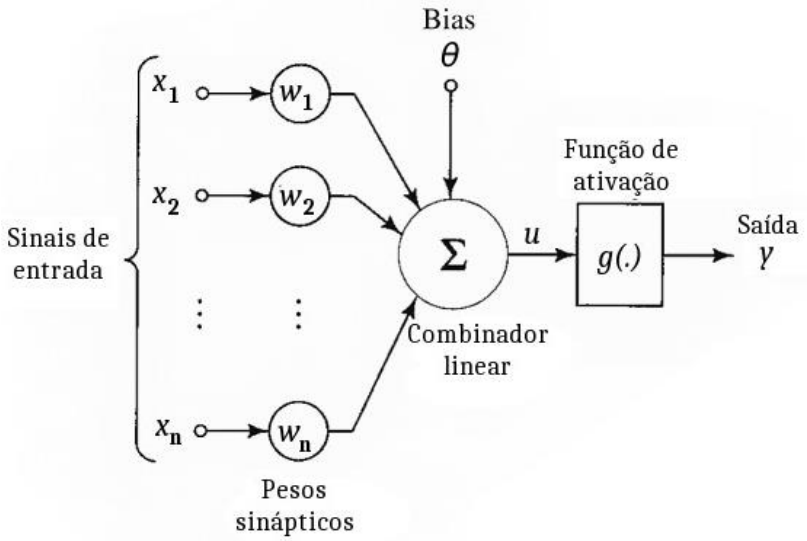
Depois de gerar um impulso, o neurônio entra em um período de refração (período em que o axônio não pode ser novamente estimulado), durante o qual retorna ao seu potencial de repouso enquanto se prepara para a geração de um novo impulso.

Por fim, como abordado por Braga, Carvalho e Ludermir (2000, p. 7), há uma diferença de potencial (em Volts) entre o interior e o exterior do neurônio, ocasionada pela diferença entre a concentração de potássio dentro da célula que cria um potencial elétrico de -70 mV (potencial de repouso) em relação ao exterior. Assim, para que a célula dispare, produzindo um potencial de ação (impulso nervoso), é necessário que os impulsos das sinapses sejam reduzidos para cerca de -50 *mV*.

3.3 NEURÔNIOS ARTIFICIAIS

Segundo Silva, Spatti e Flauzino (2010, p. 33), o modelo do neurônio de McCulloch e Pitts (1943), o MCP, ainda é o mais utilizado nas diferentes arquiteturas de RNAs. Uma representação do neurônio MCP pode ser vista na Figura 5.

Figura 5 – Modelo do neurônio de McCulloch e Pitts.



Fonte: Adaptado de Haykin, 2007, p. 36.

De acordo com Silva, Spatti e Flauzino (2010, p. 34) e Braga, Carvalho e Ludermir (2000, p. 8-9), o neurônio artificial do modelo MCP possui os seguintes elementos:

1. Terminais de entrada $x_1, x_2, \dots, x_n$ que representam os dendritos.
2. Pesos $w_1, w_2, \dots, w_n$ associados aos terminais de entrada que emulam o comportamento das sinapses. Os valores podem ser positivos ou negativos, dependendo se as sinapses forem inibitórias ou excitatórias.
3. Combinador linear $\sum$ que emula o corpo do neurônio biológico, cuja função é realizar a soma dos valores $x_i w_i$ recebidos pelo neurônio (soma ponderada) e decidir se o neurônio deve ou não disparar (saída igual a 1 ou a 0), comparando a soma obtida ao limiar de ativação do neurônio.
4. Limiar de ativação θ (também chamado de *bias* e *threshold*) que é a variável que especifica qual será o patamar apropriado para que o resultado produzido pelo combinador linear possa gerar um valor de disparo em direção à saída do neurônio.
5. Potencial de ativação u, que é o resultado da diferença entre o combinador linear e o limiar de ativação. Se $u \geq \theta$, então o neurônio produz um potencial excitatório; caso contrário, o potencial será inibitório.
6. Função de ativação $g(.)$, que ativa ou não a saída, dependendo do valor da soma ponderada das suas entradas.
7. Sinal de saída y que representa o axônio. Consiste do valor final produzido pelo neurônio em relação a um determinado conjunto de sinais de entrada, podendo ser também utilizado por outros neurônios que estão sequencialmente interligados.

Dessa forma, duas expressões que podem sintetizar o resultado do neurônio MCP são dadas pelas equações 1 e 2. A equação 1 representa o potencial de ativação do neurônio MCP, onde n representa o número de entradas do neurônio, w_i é o peso associado à entrada x_i e θ é o limiar de ativação. Já a equação 2 representa o sinal de saída do neurônio MCP, onde y é o sinal de saída e $g(u)$ é a função de ativação aplicada ao potencial de ativação do neurônio.

$$u = \sum_{i=1}^{n} (x_i w_i) - \theta \quad \textbf{(1)}$$

$$y = g(u) \quad \textbf{(2)}$$

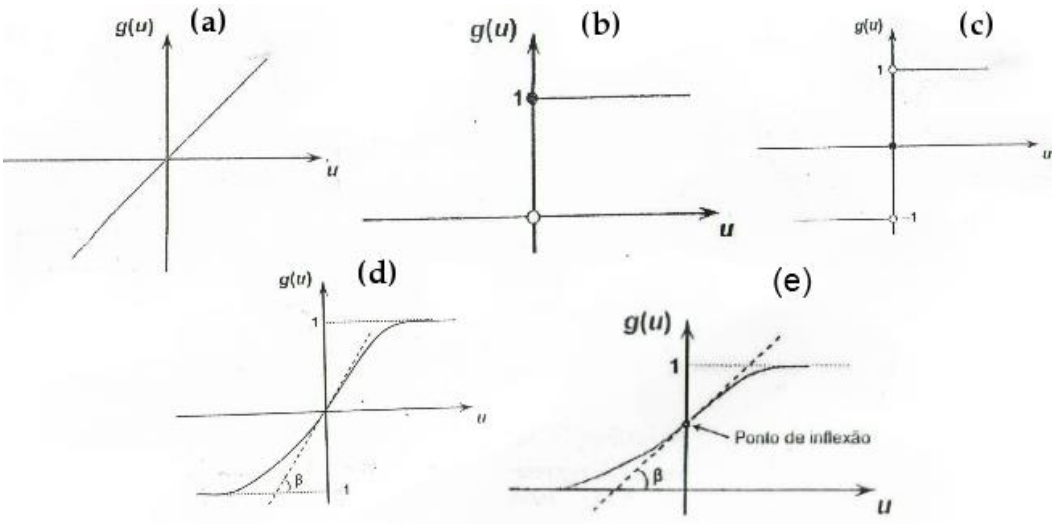
Dessa forma, um neurônio MCP terá sua saída y ativada quando $u \geq \theta$, ou seja, quando $\sum_{i=1}^{n}(x_i w_i) \geq \theta$.

3.4 FUNÇÕES DE ATIVAÇÃO

A partir do modelo proposto por McCulloch e Pitts, foram derivados vários outros modelos que permitem a produção de uma saída qualquer, não necessariamente zero ou um, e com diferentes funções de ativação (BRAGA; CARVALHO; LUDERMIR, 2000, p. 10).

Algumas funções de ativação bastante utilizadas, como verificado nos trabalhos de Braga, Carvalho e Ludermir (2000, p. 10-11), Silva, Spatti e Flauzino (2010), Kovács (2006, p. 21-23) e Haykin (2007, p. 38-40) são listadas nos gráficos da Figura 6.

Figura 6 – Gráficos das funções de ativação.



Fonte: Adaptado de Silva, Spatti e Flauzino, 2010, p. 36-42.

Estas funções são apresentadas nas equações de 3 a 7.

a) Função linear, dada pela equação 3.

$$g(u) = \alpha \mathrm{u} \quad \textbf{(3)}$$

Nesta função, α é um número real que define a saída linear para os valores de entrada, $g(u)$ é a saída e u é a entrada. O gráfico da função pode ser visto na Figura 6 (a).

b) Função degrau, limiar, *heavyside* ou *hard limiter*, dada pela equação 4.

$$g(u) = \begin{cases} 1, se\ u \geq 0 \\ 0, se\ < 0 \end{cases} \quad \textbf{(4)}$$

O resultado produzido pela aplicação da função degrau assumirá valores unitários positivos quando o potencial de ativação do neurônio for maior ou igual a zero; caso contrário, assumirá valores nulos. O gráfico da função pode ser visto na Figura 6 (b).

c) Função degrau bipolar, sinal, *symmetric hard limiter*, linear por partes ou rampa, dada pela equação 5.

$$g(u) = \begin{cases} +1, se\ u > 0 \\ 0, se\ u = 0 \\ -1, se\ u < 0 \end{cases} \quad \textbf{(5)}$$

O resultado produzido pela aplicação desta função de ativação assumirá valores unitários positivos quando o potencial de ativação do neurônio for maior que zero, valor nulo quando o potencial for também nulo e valores negativos quando o potencial for menor que zero. O gráfico da função pode ser visto na Figura 6 (c).

d) Função tangente hiperbólica, dada pela equação 6.

$$g(u) = \frac{1 - e^{-\beta u}}{1 + e^{-\beta u}} \quad \textbf{(6)}$$

O resultado de saída sempre assumirá valores reais entre −1 e 1, onde β está também associado ao nível de inclinação da função tangente hiperbólica em relação ao seu ponto de inflexão. O gráfico da função pode ser visto na Figura 6 (d).

e) Função sigmoidal, logística ou *s-shape*, dada pela equação 7.

$$g(u) = \frac{1}{1 + e^{-\beta u}} \quad \textbf{(7)}$$

Na função, β é uma constante real associada ao nível de inclinação da função logística frente ao seu ponto de inflexão. O resultado de saída da

função logística assumirá sempre valores reais entre 0 e 1. O gráfico da função pode ser visto na Figura 6 (e).

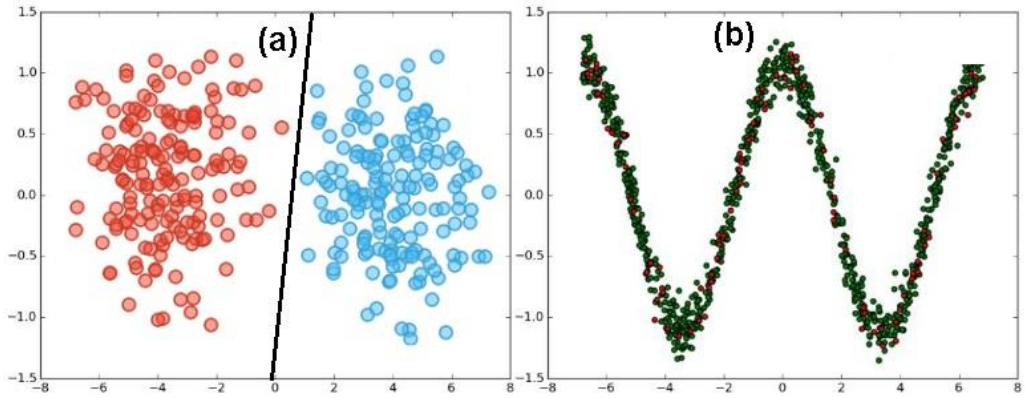
3.5 ARQUITETURAS DE REDE

De acordo com Braga, Carvalho e Ludermir (2000, p. 11-12) e Haykin (2007, p. 46), a arquitetura de uma RNA define os tipos de problema que a rede pode resolver e também está intimamente ligada com o algoritmo de aprendizagem utilizado.

Assim, um importante conceito relacionado ao tipo de problemas que uma RNA pode resolver é o de problemas linearmente separáveis. Um problema linearmente separável é dado quando as classes deste, representadas em um hiperplano, podem ser separadas por uma reta.

Dessa forma, redes MCP, que possuem uma única camada de neurônios, por exemplo, só conseguem resolver problemas linearmente separáveis. Na Figura 7, em (a) é possível verificar uma função linearmente separável e em (b) uma não linearmente separável.

Figura 7 – Funções linearmente separável (a) e não linearmente separável (b).



Fonte: Autoria própria.

Por conseguinte, conforme Silva, Spatti e Flauzino (2010, p. 45), o treinamento de uma arquitetura de rede consiste da aplicação de um conjunto de passos ordenados com o intuito de ajustar os pesos e os limiares de seus neurônios. Assim, o ajuste, também conhecido como algoritmo de aprendizagem, visa

sintonizar a rede para que as suas respostas estejam próximas dos valores desejados.

A arquitetura de uma RNA possui os seguintes parâmetros: número de camadas de rede, número de neurônios por camada, tipo de conexão entre os neurônios e a topologia da rede. Ainda de acordo com Silva, Spatti e Flauzino (2010, p. 46), uma RNA pode ser dividida em três partes, denominadas camadas, que são:

- Camada de entrada: É a camada responsável pelo recebimento de dados do meio externo.
- Camadas escondidas, intermediárias, ocultas ou invisíveis: São aquelas compostas de neurônios que extraem as características associadas ao processo ou sistema a ser inferido. Quase todo o processamento interno da rede é realizado nessas camadas.
- Camada de saída: Nesta camada, também composta por neurônios, é onde são produzidos os resultados finais da rede.

Também é possível classificar uma RNA quanto à sua conectividade. Quando todos os neurônios da rede estão conectados, esta é chamada de completamente conectada, e quando não estão, é chamada de parcialmente conectada. Há representações de redes completamente conectadas nas Figuras 8, 9 e 10 e de uma rede parcialmente conectada na Figura 11.

Por fim, de acordo com Haykin (2007, p. 46-49), existem basicamente três classes de arquiteturas de redes: as alimentadas adiante com camada única, as alimentadas diretamente com múltiplas camadas e as recorrentes. Tais classes são melhor descritas nas próximas duas subseções.

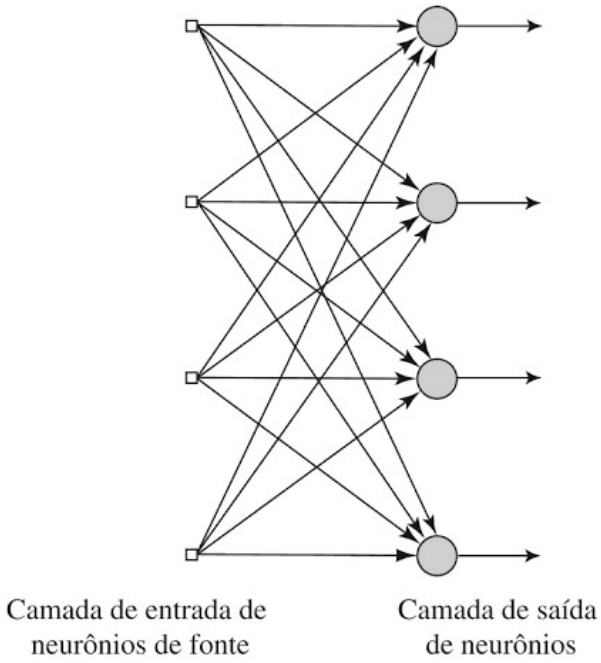
3.5.1 Redes alimentadas adiante com camada única

Segundo Haykin (2007, p. 46-47) e Silva, Spatti e Flauzino (2010, p. 46-47), as redes alimentadas adiante com camada única, também chamadas de *feedforward* ou acíclicas, o fluxo dos dados se dá para frente, ou seja, não existe realimentação e são tipicamente utilizadas em problemas envolvendo classificação de padrões e filtragem linear.

Ainda de acordo com Silva, Spatti e Flauzino (2010, p. 46-47), os principais tipos de redes *feedforward* com camada única são o *Perceptron* e o *Adaline*. Nestes

tipos de rede, a (única) camada funciona tanto como de entrada quanto de saída e os algoritmos de aprendizado utilizados são baseados, respectivamente, na regra de Hebb e na regra delta. Uma rede *feedforward* de camada única é representada na Figura 8.

Figura 8 – Rede *feedforward* de camada única.



Fonte: Haykin, 2007, p. 47.

3.5.2 Redes alimentadas adiante com múltiplas camadas

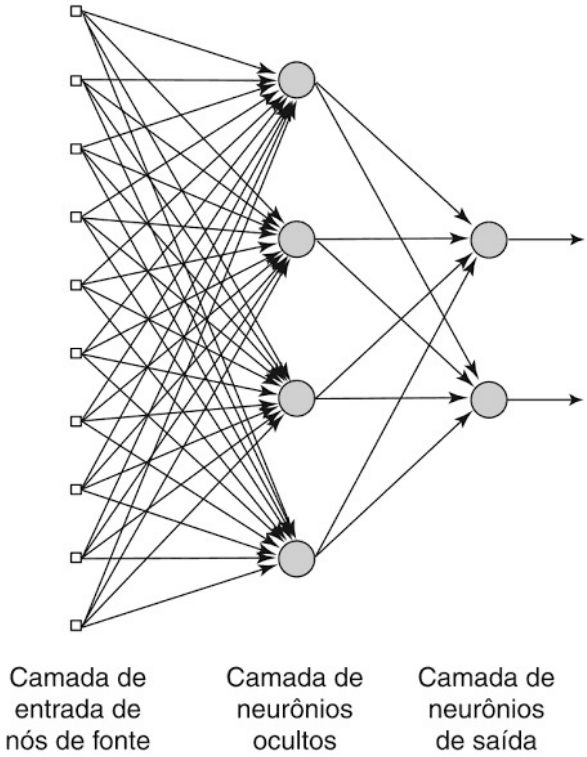
Segundo Haykin (2007, p. 47) e Silva, Spatti e Flauzino (2010, p. 47-48), a classe de redes *feedforward* de múltiplas camadas distingue-se pela presença de uma ou mais camadas ocultas. Os nós de entrada da rede fornecem os dados de entrada aos neurônios da segunda camada e assim por diante, até a camada de saída, que constitui a resposta global da rede.

Ainda de acordo Silva, Spatti e Flauzino (2010, p. 48), os principais modelos de redes com este tipo de arquitetura são o *Perceptron* Multicamadas (*Multilayer Perceptron* - MLP) e as Redes de Base Radial (*Radial Basis Function*), cujos algoritmos de aprendizado são, respectivamente, baseados na regra delta generalizada e na regra delta/competitiva.

Por conseguinte, de acordo com Silva, Spatti, Flauzino (2010, p. 47), estas redes são empregadas na solução de diversos tipos de problemas, como

aproximação de funções, classificação de padrões, otimização, etc. Uma representação de uma rede *feedforward* de múltiplas camadas é demonstrada na Figura 9.

Figura 9 – Rede *feedforward* com múltiplas camadas.



Fonte: Haykin, 2007, p. 48.

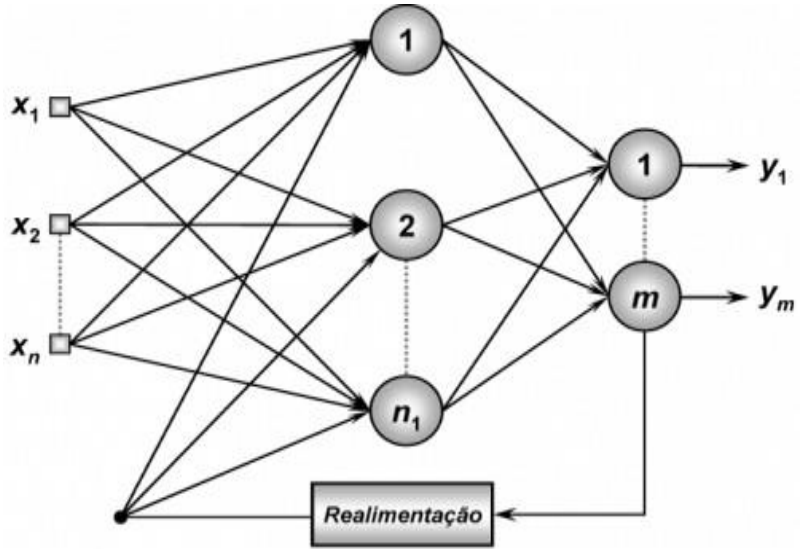
3.5.3 Redes recorrentes

As redes *feedforward* de múltiplas camadas que possuem realimentação (ou recorrência) são chamadas de redes recorrentes, e são utilizadas no processamento dinâmico de informações, como previsão de séries temporais, otimização e identificação de sistemas (SILVA; SPATTI; FLAUZINO, 2010, p. 49).

Os principais tipos redes dessa classe são a *Hopfield* e a *Perceptron* multicamadas com realimentação, cujos algoritmos de aprendizado são baseados, respectivamente, na minimização de funções de energia e na regra delta generalizada.

Na Figura 10 há a representação de uma rede *Perceptron* recorrente com dois sinais da camada de saída retroalimentando a camada intermediária.

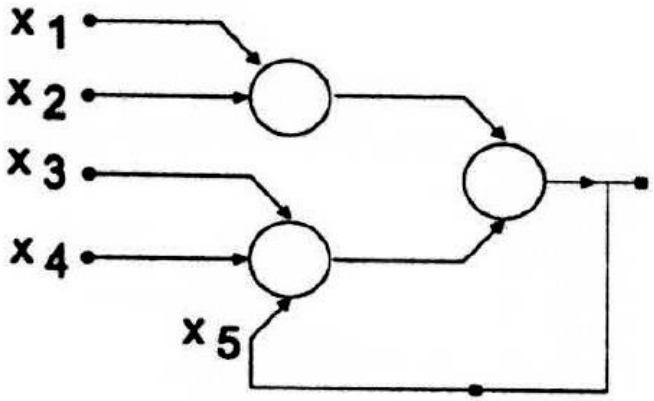
Figura 10 – Rede *Perceptron* com realimentação.



Fonte: Silva, Spatti e Flauzino, 2010, p. 49.

Outro exemplo de rede recorrente pode ser visto na Figura 11.

Figura 11 – Outro exemplo de rede recorrente.



Fonte: Braga, Carvalho e Ludermir, 2000, p. 12.

3.6 APRENDIZADO E TREINAMENTO

De acordo com Braga, Carvalho e Ludermir (2000, p. 35), o conceito de aprendizado em RNAs foi introduzido com o trabalho de Rosenblatt (1958). O modelo proposto pelo autor, conhecido como *Perceptron*, era composto por uma estrutura de rede que utilizava neurônios MCP e uma regra de aprendizado.

Posteriormente, Rosenblatt demonstrou o teorema de convergência do *Perceptron*, que mostra que um neurônio MCP treinado com o algoritmo de

aprendizado do *Perceptron* sempre converge para uma resposta caso o problema em questão seja linearmente separável.

Apesar das várias possíveis interpretações do conceito de aprendizagem, uma possível definição realizada por McLaren (1970 apud BRAGA; CARVALHO; LUDERMIR, 2000, p. 15-16) no contexto das redes neurais é a seguinte:

> Aprendizagem é o processo pelo qual os parâmetros de uma rede neural são ajustados através de uma forma continuada de estímulo pelo ambiente no qual a rede está operando, sendo o tipo específico de aprendizagem realizada definido pela maneira particular como ocorrem os ajustes realizados nos parâmetros.

Dessa forma, de acordo com Haykin (2007, p. 75-76), uma RNA aprende sobre seu ambiente através de um processo interativo de ajustes de seus pesos sinápticos e níveis de *bias*. Assim, um conjunto bem definido de regras para a solução de aprendizagem é chamado de algoritmo de aprendizagem, e a aplicação do algoritmo nos dados de entrada do domínio do problema é chamada de treinamento.

Segundo Braga, Carvalho e Ludermir (2000, p. 15), existem diversos tipos de algoritmos de aprendizagem, cada um com suas vantagens e desvantagens, onde o que os diferencia é basicamente a forma pela qual o ajuste dos pesos é feito. Assim, Zaccone (2016, p. 90) define que o procedimento de aprendizagem da rede é iterativo: modifica ligeiramente os pesos sinápticos usando um conjunto selecionado chamado conjunto de treinamento para cada ciclo de aprendizagem, denominado época.

Em cada época, os pesos devem ser modificados para minimizar uma função de custo (ou erro), que é específica para o problema em questão. As funções de custo mais comumente utilizadas são a do erro quadrático e a *cross-entropy*. Já os algoritmos de minimização do erro mais comuns são baseados no gradiente descendente, que basicamente é um algoritmo de otimização iterativa que encontra os mínimos de uma função.

Dessa forma, ao longo da aplicação do algoritmo de aprendizagem, a rede neural será capaz de extrair características do sistema por meio dos dados de entrada do domínio do problema.

Por fim, ainda de acordo com Braga, Carvalho e Ludermir (2000, p. 15), é possível agrupar os algoritmos de aprendizado em dois grupos: aprendizado

supervisionado e não supervisionado, que são abordados nas duas subseções seguintes.

3.6.1 Aprendizado supervisionado

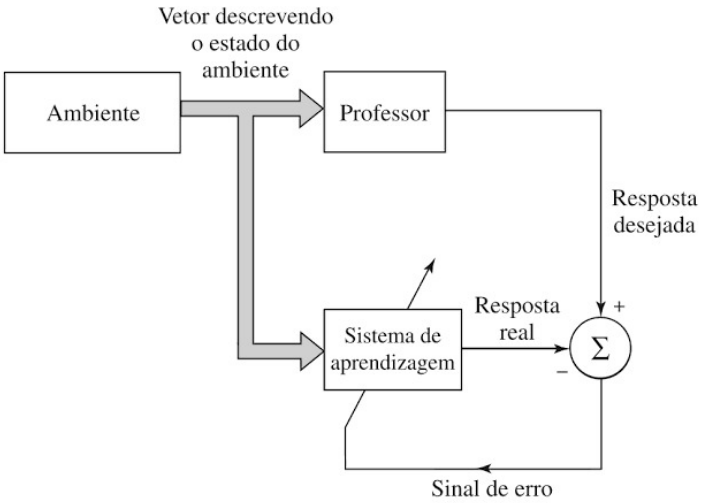
De acordo com Braga, Carvalho e Ludermir (2000, p. 16), o aprendizado supervisionado é o mais comumente utilizado no treinamento de RNAs, onde um supervisor (professor) fornece as entradas e a saída desejadas para a rede. Segundo Haykin (2007, p. 88-89) e Braga, Carvalho e Ludermir (2000, p. 16), o professor indica quais são os comportamentos “bons” e “ruins” para direcionar o treinamento da rede.

Dessa forma, neste método, a cada entrada compara-se a resposta desejada com a resposta real calculada, ajustando-se os pesos das conexões para minimizar o erro, que é definido como a diferença entre a resposta desejada e a resposta dada pela rede. Segundo Braga, Carvalho e Ludermir (2000, p. 16), a soma quadrada dos erros de todas as saídas é normalmente utilizada como medida de desempenho da rede e também como função de custo a ser minimizada pelo algoritmo de treinamento.

Por conseguinte, como também abordado por Braga, Carvalho e Ludermir (2000, p. 16), a desvantagem do aprendizado supervisionado é que, na ausência de um professor, o aprendizado para situações não previstas no treinamento não é possível.

Por fim, verifica-se que os modelos de aprendizado supervisionado mais conhecidos são os baseados na regra delta e no algoritmo *backpropagation*. Na Figura 12 há um diagrama em blocos representando o processo da aprendizagem supervisionada.

Figura 12 – Diagrama do aprendizado supervisionado.



Fonte: Haykin, 2007, p. 88.

3.6.2 Aprendizado não supervisionado

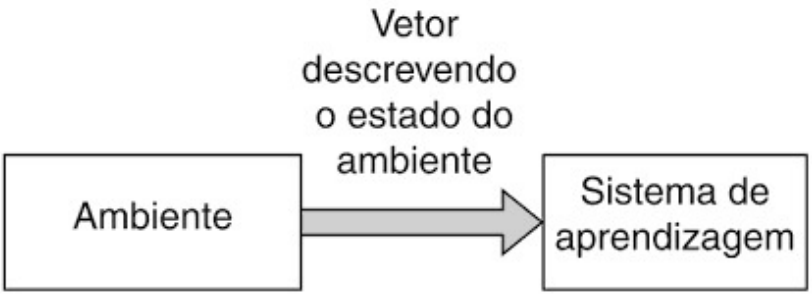
No aprendizado não supervisionado, como o próprio nome diz, não há um professor para supervisionar o processo de aprendizagem, logo, não existe o fornecimento de entradas e saídas desejadas para a rede. Ao invés disso, a própria rede deve se auto organizar em relação às particularidades existentes entre os elementos componentes do conjunto total de amostras, identificando subconjuntos (*clusters*) que contenham similaridades.

Os pesos sinápticos e os limiares dos neurônios da rede são então ajustados pelo algoritmo de aprendizagem de forma a refletir esta representação dentro da própria rede (SILVA; SPATTI; FLAUZINO, 2010, p. 56).

De acordo com Braga, Carvalho e Ludermir (2000, p. 19), a estrutura do sistema de aprendizado não supervisionado pode adquirir uma variedade de formas diferentes. Ela pode, por exemplo, consistir em uma camada de entrada, uma camada de saída, conexões *feedforward* da entrada para a saída e conexões laterais entre os neurônios da camada de saída.

Como tratado no trabalho de Braga, Carvalho e Ludermir (2000, p. 19), alguns possíveis modelos para aprendizado não supervisionado são o aprendizado hebbiano, a Regra de Oja, a Regra de Yuille e o aprendizado por competição. Por fim, na Figura 13 há um diagrama em blocos representando o funcionamento geral da aprendizagem não supervisionada.

Figura 13 – Diagrama do aprendizado não supervisionado.



Fonte: Haykin, 2007, p. 91.

3.6.3 Conjunto de dados

Durante o desenvolvimento de um modelo de RNA é importante analisar o conjunto de dados para que este auxilie no aprendizado e na convergência da rede. Assim, geralmente o conjunto de dados é dividido aleatoriamente em três partes: um subconjunto de treinamento, um de validação e um de teste.

O subconjunto de treinamento será utilizado exclusivamente para o treinamento da rede e os subconjuntos de validação e teste serão utilizados para verificar a precisão da rede para entradas não utilizadas no treinamento, ou seja, para analisar o grau de generalização do modelo.

Assim, de acordo com Braga, Carvalho e Ludermir (2010, p. 50-51), cerca de 60% a 90% das amostras de entrada devem ser utilizadas para o treinamento e de 10% a 40% devem ser usadas para o teste. Já de acordo com Silva (1998, apud HAYKIN, 2007, p. 119), algumas abordagens sugerem que 50% dos dados devem ser escolhidos para treinamento, 25% para validação e 25% para teste.

Segundo Silva, Spatti e Flauzino (2010, p. 50-51), o subconjunto de treinamento deve ser composto por um valor entre 60% e 90% do conjunto total e o de teste deve ser composto por um valor entre 10% e 40% do conjunto total.

Por conseguinte, de acordo com Braga, Carvalho e Ludermir (2000, p. 56-57) e Silva (1998, p. 153-154), existe um fenômeno relacionado ao aprendizado das RNAs que é chamado de *overfitting*, em português, super-ajustamento. Este fenômeno acontece quando a rede “memoriza” os dados de seu treinamento, ou seja, quando ela consegue classificar corretamente somente os dados que fizeram parte de seu treinamento.

Nessas ocorrências, o erro durante a fase de aprendizado tende a ser bem baixo; contudo, durante a fase de generalização, frente aos subconjuntos de teste, o erro tende a assumir valores bem elevados.

Por outro lado, de acordo com Silva (1998, p. 117), a situação em que a rede ainda não treinou o suficiente para ter uma boa classificação é chamada de *underfitting*. Nesses casos, ainda segundo Braga, Carvalho e Ludermir (2000, p. 56-57), os erros tanto na fase de aprendizado como na fase de teste serão bastante significativos.

Dessa forma, existe um método chamado validação cruzada (*cross-validation*), que é utilizado para detectar *overfitting* e interromper o treinamento antes que isso ocorra (SILVA, 1998, p. 114).

De acordo com Silva (1998, p. 117-118), o processo de *cross-validation* basicamente é constituído pelos seguintes processos:

- Dividir aleatoriamente o conjunto de dados em três subconjuntos: de treinamento, validação e teste.
- Treinar a rede somente com o subconjunto de treinamento e avaliar o erro da rede com o subconjunto de validação a cada *k* iterações (épocas).
- Interromper o treinamento quando o erro do subconjunto de validação for maior do que era *k* iterações atrás.
- Utilizar o subconjunto de pesos anteriores como o resultado de treinamento.

Por fim, o método *cross-validation* utiliza o conjunto de validação para antecipar o comportamento em situações reais, assumindo que o erro em ambos os casos será semelhante. Assim, o erro de validação geralmente é uma estimativa do erro de generalização.

3.6.4 Matriz de confusão

De acordo com Ting (2011, p. 209) e Richert e Coelho (2013, p. 188-189), uma matriz de confusão (ou matriz de erro) é uma forma de medir o desempenho de um algoritmo de aprendizado (geralmente supervisionado).

Assim, basicamente uma matriz de confusão é uma representação bidimensional do cruzamento entre as informações das amostras reais e as classificações realizadas por um sistema de classificação. O nome da ferramenta

decorre do fato de que é possível verificar se o modelo treinado está confundindo as classes.

Ainda segundo Ting (2011, p. 209), os dados da matriz são extraídos das classificações realizadas com os conjuntos de validação ou de teste. Assim, cada linha da matriz representa as amostras reais do conjunto, enquanto cada coluna representa as previsões realizadas pelo sistema de classificação. Na Tabela 1 pode ser conferido o exemplo de uma matriz de confusão para duas classes, a palavra "*one*" e a palavra "*two*".

Tabela 1 – Matriz de confusão.

		Classe prevista	
		zero	**one**
Classe real	**zero**	a	b
	one	c	d

Fonte: Adaptado de Ting, 2011, p. 209.

As entradas na matriz de confusão têm o seguinte significado:

- O número de classificações corretas da instância "*one*" é o elemento *a*.
- O número de classificações incorretas da instância "*two*" é o elemento *b*.
- O número classificações incorretas da instância "*one*" é o elemento *c*.
- O número de classificações corretas da instância "*two*" é o elemento *d*.

Dessa forma, com a matriz de confusão é possível encontrar várias métricas importantes para o campo da Aprendizagem de Máquina, para verificar como o modelo preditivo está dividindo as classes. Algumas dessas estatísticas são:

- A acurácia (AC) é a proporção do número total de previsões corretas, determinada utilizando a equação: $AC = \frac{a+d}{a+b+c+d}$.
- A taxa positiva verdadeira (TP) é a proporção de casos da classe "*two*" que foram identificados corretamente, calculada usando a equação: $TP = \frac{d}{c+d}$.
- A taxa de falso positivo (FP) é a proporção de casos "*one*" classificados incorretamente como "*two*", calculada usando a equação: $FP = \frac{a}{a+b}$.

- A taxa de falso negativo (FN) é a proporção de casos "*two*" que foram classificados incorretamente como "*one*", calculada usando a equação: $FN = \frac{c}{c+d}$.
- A precisão (P) é a proporção dos casos "*two*" classificados corretamente, calculada usando a equação: $P = \frac{d}{b+d}$.

Dessa forma, todas as classificações da diagonal principal são acertos, enquanto todas as outras são erros; portanto, é fácil analisar a qualidade das previsões do modelo. Assim, um modelo perfeito teria uma diagonal principal totalmente preenchida, com os outros valores zerados.

Assumindo uma rede treinada e uma amostra de 27 palavras, sendo 8 "*zero*", 6 "*one*" e 13 "*two*", um exemplo de uma matriz de confusão é apresentada na Tabela 2.

Tabela 2 – Outro exemplo de uma matriz de confusão.

		Classe prevista		
		zero	**one**	**two**
Classe real	**zero**	5	3	0
	one	2	3	1
	two	0	2	11

Fonte: Adaptado de Ting, 2011, p. 209.

Na matriz, quando a palavra era "*zero*", o sistema classificou como "*zero*" 5 vezes, como "*one*" 3 vezes e como "*two*" 0 vezes. Quando a palavra era "*one*", o sistema classificou como "*zero*" 2 vezes, como "*two*" 3 vezes e como "*two*" 1 vez. Por fim, quando a palavra era "*two*", o sistema classificou como "*zero*" 0 vezes, como "*one*" 2 vezes e como "*two*" 11 vezes.

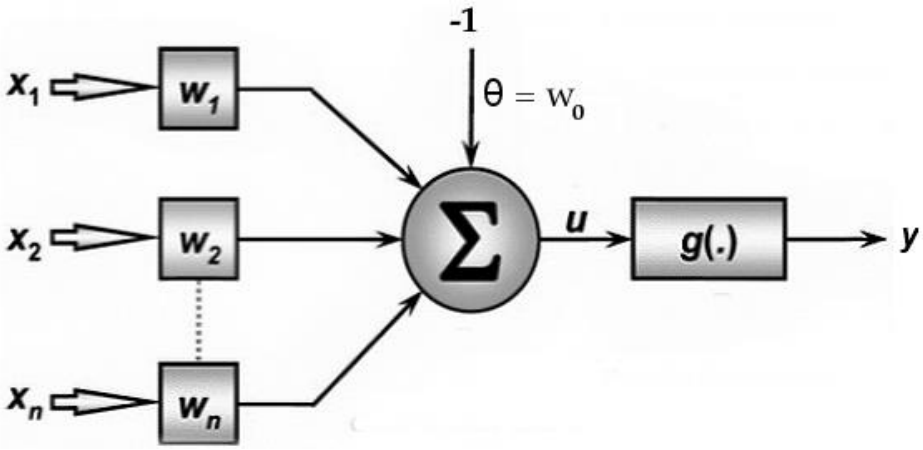
Dessa forma, percebe-se que o modelo tem resultados ruins na diferenciação entre “*one*” e “*zero*”, mas bons resultados na diferenciação entre “*two*” e as outras palavras.

3.7 *PERCEPTRON* DE CAMADA ÚNICA

As redes *Perceptron* de camada única possuem apenas uma camada neural com apenas um neurônio, cujo treinamento acontece de forma supervisionada. Rosenblatt (1958) demonstrou o teorema de convergência do *Perceptron* de uma camada, que mostra que um nó MCP treinado com seu algoritmo de aprendizado sempre converge para uma resposta correta caso o problema em questão seja linearmente separável.

Conforme abordado no trabalho de Silva, Spatti e Flauzino (2010, p. 60), geralmente as funções de ativação utilizadas no *Perceptron* de uma camada são a degrau (que terá como saída 0 ou 1) e a degrau bipolar (que terá como saída -1 ou 1). Uma representação do *Perceptron* de uma camada pode ser vista na Figura 14.

Figura 14 – *Perceptron* de uma camada.



Fonte: Silva, Spatti e Flauzino, 2010, p. 70.

Ainda de acordo com Silva, Spatti e Flauzino (2010, p. 64), o ajuste dos pesos e limiar do *Perceptron* é realizado por meio da regra de aprendizado de Hebb (1949), que pode ser expresso pela equação 8.

$$w_{atual} = w_{anterior} + \eta(d^k - y)x^{(k)} \quad \textbf{(8)}$$

Na equação 8, $x^{(k)} = [-1, x_1^{(k)}, x_2^{(k)}, \dots, x_n^{(k)}]^T$ é a *k-ésima* amostra de treinamento, w_{atual} é o vetor contendo o limiar e os pesos da iteração atual, $w_{anterior}$ é o vetor contendo o limiar e os pesos da iteração anterior, $d^{(k)}$ é o valor desejado para a *k-ésima* amostra de treinamento, y é o valor de saída produzido pelo *Perceptron* e η é uma constante que define a taxa de aprendizado da rede, que

expressa o quão rápido o processo de treinamento da rede produzirá a convergência. Geralmente esta constante é um valor entre 0 e 1.

Assim, segundo Silva, Spatti e Flauzino (2010, p. 65-66), um algoritmo de treinamento do *Perceptron* de camada única em pseudocódigo pode verificado na Figura 15.

Figura 15 – Algoritmo de treinamento do *Perceptron*.

```
Obter o conjunto de amostras de treinamento {x(k)}
Associar a saída desejada {d(k)} para cada amostra obtida
Iniciar o vetor w com valores aleatórios pequenos
Especificar a taxa de aprendizagem {η}
Iniciar o contador de número de épocas: época = 0
Repetir:
    erro = "inexiste"
    Para todas as amostras de treinamento {x(k), d(k)}, fazer:
      u = wTx(k)
      y = g(u), onde g(u) é geralmente ou a função degrau ou a degrau bipolar
      Se y ≠ d(k) então:
        w = w + η(d(k) − y)x(k)
        erro = "existe"
    época = época + 1
Até que: erro = "inexiste"
```

Fonte: Adaptado de Silva, Spatti e Flauzino, 2010, p. 65-66.

Dessa forma, uma vez treinada, o *Perceptron* simples estará apto para classificar padrões (lineares) diante de novas amostras. Após o treinamento, para utilizar o *Perceptron* treinado é utilizado o algoritmo de operação (em pseudocódigo), apresentado no algoritmo da Figura 16.

Figura 16 – Algoritmo de operação do *Perceptron.*

```
Obter uma amostra a ser classificada {x}
Utilizar um vetor w ajustado durante o treinamento
Fazer:
    u = wTx
    y = g(u), onde g(u) é geralmente ou a função degrau ou a degrau bipolar
    Se y = −1, então:
        Amostra x ∈ {Classe A}
    Se y = 1, então:
        Amostra x ∈ {Classe B}
```

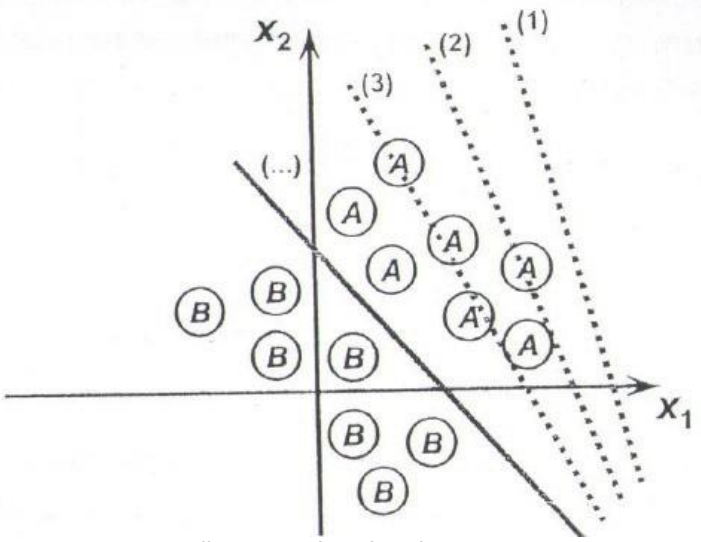
Fonte: Adaptado de Silva, Spatti e Flauzino, 2010, p. 65-66.

Por fim, como retratado por Silva, Spatti e Flauzino (2010, p. 66) é possível abstrair o processo de treinamento do *Perceptron* de camada única como o

processo de mover continuamente uma linha de classificação até que seja alcançada uma fronteira de separação que permite dividir as classes.

Como exemplo, é possível verificar na Figura 17 uma ilustração do processo de treinamento do *Perceptron* simples visando o alcance de uma fronteira de separabilidade para uma rede com duas entradas $\{x_1, x_2\}$, onde *(1), (2), (3), (...)* são as épocas de treinamento.

Figura 17 – Treinamento do *Perceptron* de camada única.



Fonte: Silva, Spatti e Flauzino, 2010, p. 67

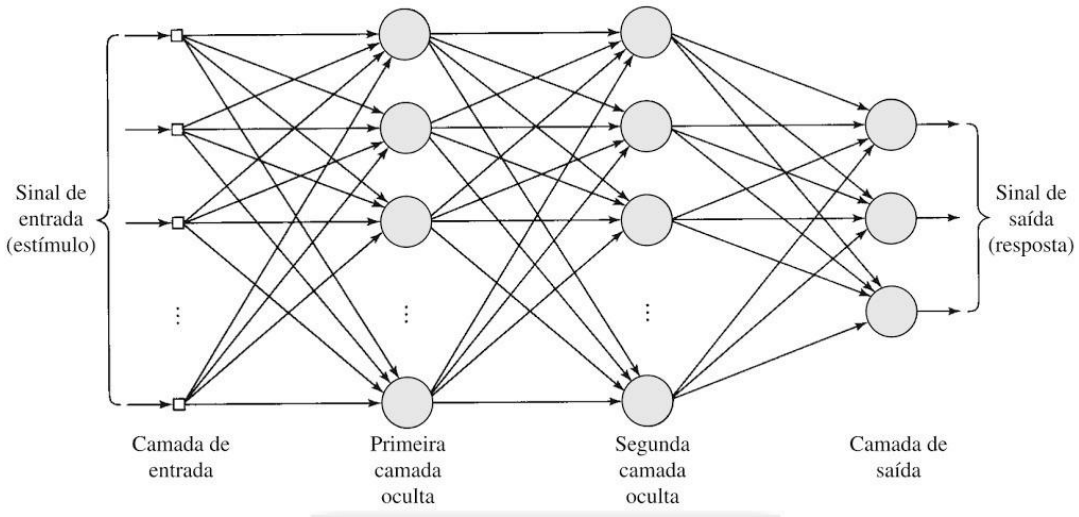
3.8 *PERCEPTRON* DE MÚLTIPLAS CAMADAS

Nesta seção são apresentados os conceitos gerais das redes *Perceptron* de múltiplas camadas, além do algoritmo de aprendizagem *backpropagation*.

3.8.1 Conceitos gerais

As redes *Perceptron* de múltiplas camadas, ou MLP (*Multilayer Perceptron*), são redes completamente conectadas, caracterizadas por uma camada de entrada e uma ou mais camadas intermediárias (ocultas) de neurônios. Cada neurônio possui uma função de ativação não linear (mas diferenciável em qualquer ponto), onde um tipo comum é a sigmoide. Uma representação de uma MLP pode ser vista na Figura 18.

Figura 18 – *Perceptron* de múltiplas camadas.



Fonte: Haykin, 2007, p. 186.

‘

A utilização de uma camada intermediária é suficiente para aproximar qualquer função contínua, e duas camadas intermediárias são suficientes para aproximar qualquer função matemática (BRAGA; CARVALHO; LUDERMIR, 2000, p. 55).

O treinamento acontece de forma supervisionada pelo algoritmo de retropropagação de erro (*error backpropagation*), também conhecido como regra delta generalizada. De acordo com Haykin (2007, p. 183), a aprendizagem por retropropagação de erro consiste de dois passos através das diferentes camadas da rede: um passo para frente, a propagação, e um passo para trás, a retropropagação.

No passo para frente, um vetor de entrada é aplicado aos nós sensoriais da rede e seu efeito se propaga através desta, camada por camada, onde os pesos sinápticos são fixos, até a produção da saída. Já durante o passo para trás, segundo Haykin (2007, p. 183), os pesos sinápticos são ajustados de acordo com uma regra de correção de erro.

Basicamente, a resposta real é subtraída de uma resposta desejada para produzir um sinal de erro, que é propagado para trás através da rede, no sentido contrário das conexões sinápticas. Assim, ainda de acordo com Haykin (2007, p. 183), os pesos sinápticos são ajustados para fazer com que resposta real da rede se aproxime da resposta desejada.

3.8.2 Algoritmo *backpropagation*

Inicialmente, é importante definir algumas variáveis e parâmetros para auxiliar no desenvolvimento do algoritmo *backpropagation*. É possível tomar então as seguintes variáveis, como abordado por Silva, Spatti e Flauzino (2010, p. 95-97):

- $W_{ji}^{(L)}$, são matrizes de pesos cujos elementos denotam o valor do peso sináptico conectando o *j-ésimo* neurônio da camada (L) ao *i-ésimo* neurônio da camada $(L-1)$.
- $I_{ji}^{(L)}$ são vetores cujos elementos são a entrada ponderada em relação ao *j-ésimo* neurônio da camada (L).
- n é a quantidade de neurônios por camada (L).

Na equação 9, há a definição genérica do elemento $I_{ji}^{(L)}$.

$$I_j^{(L)} = \sum_{i=0}^{n} W_{ji}^{(L)} x_i \leftrightarrow I_j^{(L)} = W_{j,0}^{(L)} x_0 + W_{j,1}^{(L)} x_1 + \cdots + W_{j,q}^{(L)} x_{j,q} \quad \textbf{(9)}$$

- $Y_j^{(L)}$ são vetores cujos elementos representam a saída do *j-ésimo* neurônio em relação à camada *(L)*, como definido pela equação 10.

$$Y_j^{(L)} = g(I_j^{(L)}) \quad \textbf{(10)}$$

Existem vários tipos de funções de erro, dentre elas a do erro quadrático, que mostra o desvio entre as respostas produzidas pelos neurônios de saída da rede em relação aos respectivos valores desejados, como abordado no trabalho de Silva, Spatti e Flauzino (2010, p. 99). A função genérica do erro quadrático pode ser vista na equação 11.

$$E(k) = \frac{1}{2}\sum_{j=1}^{n}(d_j(k) - Y_j^{(L)}(k))^2 \quad \textbf{(11)}$$

Na equação 11, n é novamente a quantidade de neurônios na camada oculta, $Y_j^{(L)}(k)$ é o valor produzido pelo *j-ésimo* neurônio de saída da rede considerando-se a *k-ésima* amostra de treinamento e $d_j(k)$ é o respectivo valor desejado. Assim, assumindo um conjunto de treinamento composto por amostras, a medição da evolução do desempenho global do algoritmo *backpropagation* pode ser efetuada por meio da avaliação do erro quadrático médio, definido pela Equação 12. Nessa

equação, $E(k)$ é o erro quadrático obtido na equação 11 e p é a quantidade de amostras do conjunto de treinamento.

$$E_M = \frac{1}{p}\sum_{k=1}^{p} E(k) \quad \textbf{(12)}$$

Ainda de acordo com Silva, Spatti e Flauzino (2010, p. 100-108), a equação do gradiente local $\delta_j^{(L)}$ em relação ao *j-ésimo* neurônio da camada de saída, que mede o erro do neurônio j da camada (L), é dada pela equação 13.

$$\delta_j^{(L)} = \sum_{k=1}^{n+1} \delta_k^{(L+1)} W_{kj}^{(L+1)}\, g'(I_j^{(L)}) \quad \textbf{(13)}$$

O ajuste da matriz de pesos $W_{ji}^{(L)}$ pode ser descrito pela equação 14.

$$W_{ji}^{(L)}(t+1) = W_{ji}^{(L)}(t) + \eta \delta_j^{(L)} x_i \quad \textbf{(14)}$$

Na equação 14, η é a taxa de aprendizagem do algoritmo *backpropagation*. O resultado da equação 14, conforme verificado em Silva, Spatti e Flauzino (2010, p. 100-108) pode ser descrito pela seguinte notação algorítmica, na equação 15.

$$W_{ji}^{(L)} = W_{ji}^{(L)} + \eta \delta_j^{(L)} x_i \quad \textbf{(15)}$$

Por conseguinte, como descrito por Silva, Spatti e Flauzino (2010, p. 109-110), o critério de parada do processo do algoritmo *backpropagation* fica estipulado em função do erro quadrático médio (calculado na equação 12), levando em consideração as amostras de treinamento utilizadas.

Assim, o algoritmo converge quando o erro quadrático médio entre duas épocas sucessivas for sucessivamente pequeno, o que pode ser descrito na equação 15, onde ε é a precisão requerida para o processo de convergência.

$$\left| E_M^{atual} - E_M^{anterior} \right| \leq \varepsilon \quad \textbf{(16)}$$

Por fim, uma representação do treinamento do MLP com algoritmo *backpropagation*, em pseudocódigo, como verificado em Silva, Spatti e Flauzino (2010, p. 110) pode ser visto na Figura 19.

Figura 19 – Algoritmo *backpropagation.*

```
1  v = quantidade de épocas
2  época = 0
3  Obter o conjunto de amostras de treinamento {x^(k)}
4  Associar o vetor de saída desejada {d^(k)} para cada amostra
5  Iniciar W_ji^(1), W_ji^(2), …, W_ji^(L) com valores aleatórios pequenos
6  Especificar a taxa de aprendizagem {η} e a precisão requerida {ε}
7  Repetir:
8       E_M^anterior = E_M, conforme a equação 12
9       Para todas as amostras de treinamento {x^(k), d^(k)} e as camadas (L), fazer:
10          # Passo forward
11          Obter I_j^(L) conforme equação 9
12          Obter Y_j^(L) conforme equação 10
13          # Passo backward
14          Determinar δ_k^(L), conforme equação 13
15          Ajustar W_ji^L, conforme a equação 15
16      E_M^atual = E_M, conforme equação 12
17      época = época + 1
18 Até que: (|E_M^atual − E_M^anterior| ≤ ε) E (época ≤ v)
```

Fonte: Adaptado de Silva, Spatti e Flauzino, 2010, p. 110.

Por fim, como ainda mostrado em Silva, Spatti e Flauzino (2010, p. 111), após o treinamento do MLP, para utilizar o modelo treinado, é possível definir o seguinte um algoritmo de operação, como apresentado na Figura 20.

Figura 20 – Algoritmo de operação do MLP.

```
1 Obter uma amostra {x}
2 Assumir W_ji^(1), W_ji^(2), …, W_ji^(L) já ajustadas no treinamento
3 Executar:
4       # Passo forward
5      Obter I_j^(L) conforme equação 9
6      Obter Y_j^(L) conforme equação 10
7 Disponibilizar as saídas da rede de Y_j^(L), (L) = última camada (saída)
```

Fonte: Adaptado de Silva, Spatti e Flauzino, 2010, p. 111.

3.9 REDES NEURAIS CONVOLUCIONAIS

Inicialmente, de acordo com Gollapudi (2016, p. 333) e Zaccone (2016, p. 127), uma Rede Neural Convolucional (RNC) possui quatro elementos fundamentais em sua arquitetura:

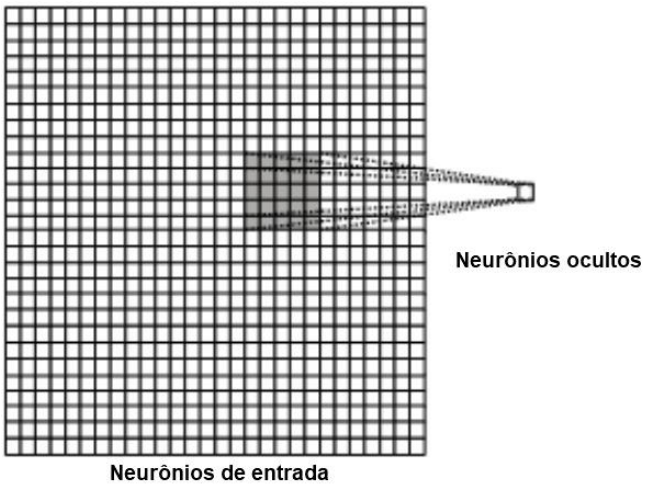
- Convolução (*convolution*).
- Inserção de não linearidade (*non-linearity*).

- Agrupamento (*polling*) ou camada de subamostragem (*subsampling layer*).
- Camadas totalmente conectadas (*fully connected layers*).

Segundo com Zaccone (2016, p. 128), as RNCs utilizam correlações espaciais que existem dentro dos dados de entrada. Cada neurônio da primeira camada subsequente está conectado a alguns dos neurônios de entrada. Esta região é chamada de campo receptivo local.

Na Figura 21 há a representação do campo receptivo local de uma RNC, que é o quadrado mais escuro de dimensão 5×5 que converge para um neurônio oculto. Assim, como cada conexão aprende um peso, neste caso existiriam 25 pesos aprendidos.

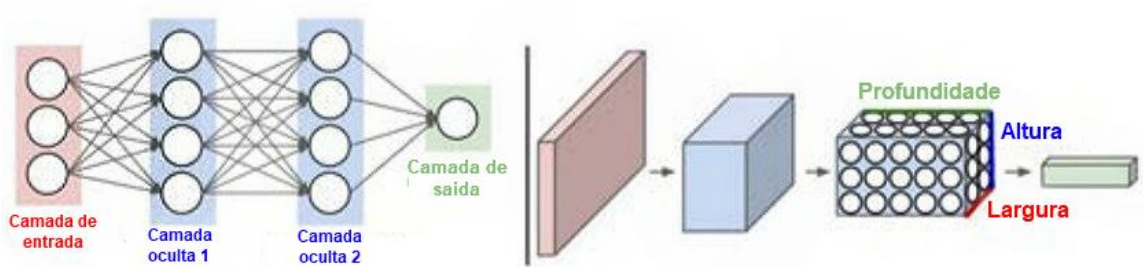
Figura 21 – Representação do campo receptivo local de uma RNC.



Fonte: Adaptado de Zaccone, 2016, p. 128.

Como verificado em Gollapudi (2016, p. 333), em uma RNC, os dados de entrada podem ser representados como vetores de $\boldsymbol{n}$ dimensões, mas geralmente são tridimensionais, com profundidade, altura e largura. Isso define a organização e o funcionamento da rede. No diagrama da Figura 22 há uma comparação entre uma rede MLP comum de três camadas com uma RNC de três camadas.

Figura 22 – Comparação entre uma MLP e uma RNC.



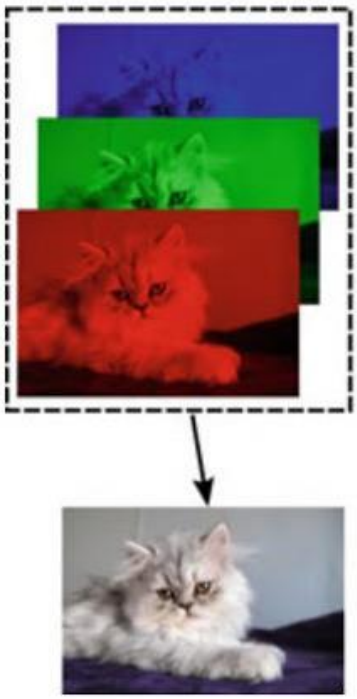
Fonte: Adaptado de Gollapudi, 2016, p. 333.

Por conseguinte, uma imagem de entrada[4] em uma RNC pode ser representada como a sobreposição de matrizes de características desta imagem. Cada matriz de características possui valores que representam a intensidade dos pixels da imagem para cada canal de cor.

Assim, para uma imagem de três canais, RGB[5], por exemplo, a representação desta seria composta de três matrizes sobrepostas, cada uma representando a intensidade dos pixels de uma das três cores. Na Figura 23 há a representação tridimensional dos canais de uma imagem RGB.

[4] Apesar dos exemplos serem de classificação de imagens, a classificação de sons utiliza uma abordagem semelhante à relatada nesta seção.
[5] RGB: Acrônimo para *Red*, *Green* and *Blue,* em português, Vermelho, Verde e Azul.

Figura 23 – Representação de uma imagem em três canais para uma RNC.

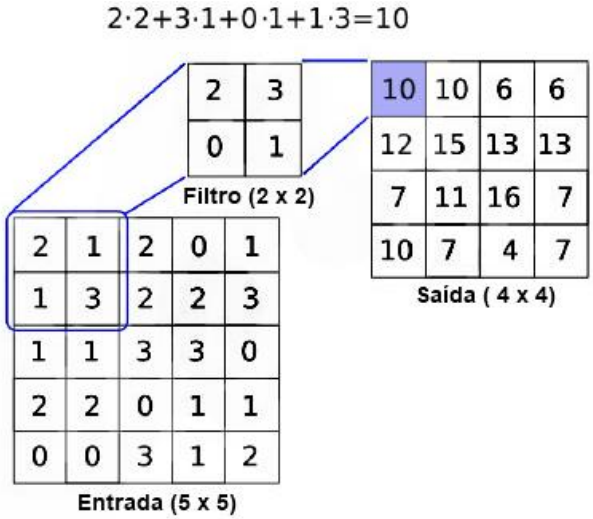


Fonte: Aghdam e Heravi, 2017, p. 91.

Por conseguinte, como descrito no trabalho de McClure (2017, p. 232), na matemática, uma convolução é definida como uma função que é aplicada sobre a saída de outra função. Assim, para imagens, uma convolução é a aplicação de um filtro (ou *kernel*) nas matrizes de características da imagem.

Esse filtro basicamente representa a multiplicação entre as matrizes de características da imagem e a matriz que representa o filtro. Esta operação resulta em outras matrizes, de dimensões menores. No diagrama da Figura 24 há a demonstração do funcionamento de uma convolução em uma imagem em escala de cinza (logo, com único canal), onde o vetor representativo é bidimensional.

Figura 24 – Operação de convolução em uma imagem de um canal.



Fonte: Adaptado de McClure, 2017, p. 232.

Na Figura 24, com a aplicação do filtro convolucional na imagem, é gerada uma nova matriz de características, também chamada de *feature layer* (camada de recursos) ou *feature map* (mapa de recursos). O filtro convolucional, de dimensão 2×2, opera realizando o produto escalar entre os elementos da matriz da imagem no local onde o filtro está posicionado e a matriz do filtro, gerando uma matriz menor, de dimensão 4×4.

Sobre o movimento do filtro, existem os chamados espaços válidos, que são todas as posições possíveis do filtro “dentro” da matriz. Um espaço não válido, por exemplo, seria uma posição em que o filtro ficasse “para fora” das bordas da matriz da imagem. Além disso, a movimentação do filtro convolucional é chamada de passo (*stride*) e a distância entre uma posição e a subsequente é chamada de tamanho do passo. No exemplo da Figura 24, o filtro se movimenta em ambas as direções, e tem um passo de tamanho 1.

Por conseguinte, de acordo com Karn (2016), O’Shea e Nash (2015, p. 4-5) e Wu (2017, p. 10), também existe uma importante operação realizada após cada Convolução, que é a inserção de não-linearidade, onde a função mais comum para isso é a *ReLU* (*Rectified Linear Unit* – Unidade Linear Retificada).

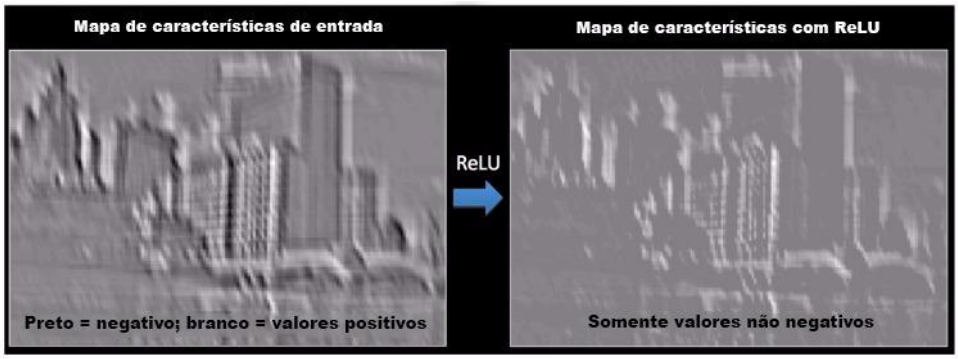
Segundo Karn (2016), o objetivo desta etapa é introduzir não linearidade na RNC, uma vez que a operação de Convolução é uma operação linear, e a maioria dos dados do mundo real são não-lineares. Sendo assim, existem outras funções não lineares, como a tangente hiperbólico e a sigmoide, mas a função *ReLU* é a

mais utilizada para propósitos gerais, pois funciona bem para a maioria das situações e é computacionalmente eficiente. A função *ReLU* é apresentada na equação 17.

$$f(u) = \max(0, u) \quad \textbf{(17)}$$

Dessa forma, como exemplo, a aplicação da *ReLU* em uma imagem poderia ser compreendida como uma operação que substitui todos os valores referentes a pixels negativos no mapa de recursos por zero, como pode ser visto na Figura 25.

Figura 25 – Operação de *ReLU* em uma imagem cinza.



Fonte: Adaptado de Fergus, 2015, p. 37.

Na Figura 25, a operação de *ReLU* é inserida em uma imagem em escala de cinza, onde valores negativos representam *pixels* pretos e valores positivos representam pixels brancos. Assim, após a realização da *ReLU*, todos os valores negativos são substituídos por 0, ou seja, as áreas pretas da imagem são suavizadas, ficando mais claras.

Com efeito, como descrito por Zaccone (2016, p. 130), em uma RNC podem existir várias camadas de convolução (com ou sem aplicação posterior de ReLU) e, entre essas camadas de convolução, existe a chamada *pooling layer* (ou *subsampling*). A camada de *pooling* é responsável por simplificar a informação de saída da camada anterior (a de convolução), gerando um *feature map* condensado.

De acordo com Gollapudi (2016, p. 334), na operação de *polling* a simplificação dos dados implica em reduzir, para cada *feature layer*, o número de parâmetros ou a quantidade de cálculos na rede, o que também diminui as chances de ocorrência de *overfitting*.

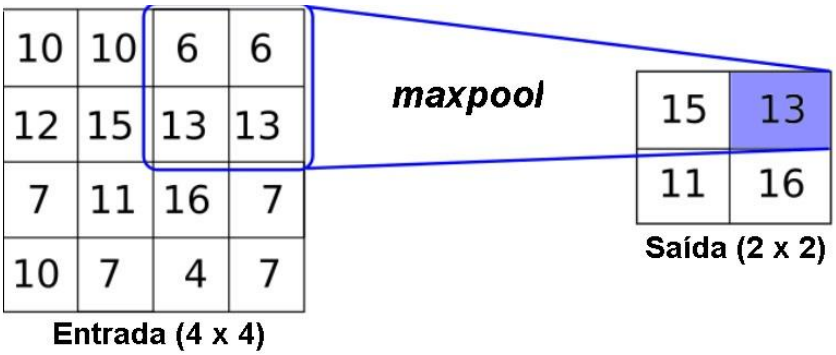
Segundo Gollapudi (2016, p. 334), normalmente as operações de *pooling* aplicam filtros de dimensão 2×2 ao longo da largura e da altura do vetor, o que

pode descartar cerca de 75% das ativações. A função de *pooling* mais comumente utilizada é a *max-pooling*.

A função *max-pooling* atua de forma simples sobre a matriz da *feature map*. A função possui uma dimensão (ou janela) $i \times j$, $(i > 0, j > 0)$, opera nos espaços válidos da matriz e possui um tamanho de passo fixo ou variável por direção de movimento. Basicamente, ela se move pela matriz e, em cada espaço válido, seleciona o maior número, gerando uma matriz de dimensão menor.

Um exemplo do funcionamento da função *max-pooling* é apresentado na Figura 26, onde a camada de entrada tem dimensão 4×4 e é a mesma da saída representada na Figura 25 após a inserção de *ReLU* [6]. A função *max-pooling* possui uma janela de dimensão 2×2 e opera em ambas as direções nos espaços válidos com passo de tamanho 2, gerando ao fim um uma matriz de dimensão 2×2.

Figura 26 – Operação de *max-pooling*.



Fonte: Adaptado de McClure, 2017, p. 233.

Em seguida, após a camada de *polling*, na arquitetura genérica de uma RNC existe uma camada totalmente conectada, que é muito semelhante à organização dos neurônios das MLPs tradicionais. Esta camada conecta todos os neurônios da camada de *polling* a cada um dos neurônios de saída, utilizando uma função de ativação (comumente a *softmax*) na camada de saída.

O objetivo da camada totalmente conectada é usar os dados das camadas anteriores para classificar a imagem de entrada em várias classes com base no conjunto de dados de treinamento. Por conseguinte, para a obtenção do erro total

[6] Como a saída apresentada na Figura 25 não possui valores negativos, após a operação de *ReLU* os resultados continuam inalterados.

utiliza-se alguma função de custo, comumente a do erro quadrático médio, descrita anteriormente na equação 12 do algoritmo *backpropagation*.

Outra função de custo bastante utilizada em RNCs é a *cross-entropy* (entropia cruzada). Assim, conforme abordado por Nielsen (2017), dado um neurônio MCP com os seguintes elementos:

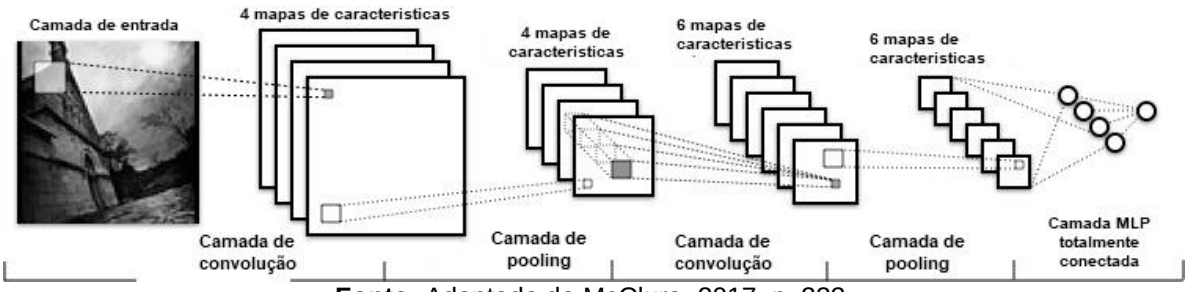
- n entradas $\{x_1, x_2, \dots, x_n\}$.
- Pesos $\{w_1, w_2, \dots, w_n\}$.
- *Bias* b.
- Soma ponderada das entradas $z = \sum_j x_j w_j + b$.
- Saída do neurônio $a = \sigma(z)$.
- Saída desejada y.

É possível definir a função de custo *cross-entropy* C da seguinte forma, como expresso na equação 18.

$$C = \frac{1}{n}\sum_x (y \ln a + (1-y)\ln(1-a)) \quad \textbf{(18)}$$

Por fim, com os quatro elementos básicos de uma RNC, convolução, *ReLU*, *polling* e uma camada totalmente conectada, é possível construir a arquitetura de uma RNC completa, como pode ser visto na Figura 27.

Figura 27 – Arquitetura de uma RNC.



Fonte: Adaptado de McClure, 2017, p. 233.

Dessa forma, de acordo com Karn (2016), a RNC é então treinada utilizando o algoritmo *backpropagation* e o gradiente descendente. Assim, o processo simplificado de treinamento de uma RNC é apresentado no algoritmo em pseudocódigo da Figura 28.

Figura 28 – Algoritmo de treinamento de uma RNC.

```
1  Fixar parâmetros: número de filtros, tamanhos de filtro e arquitetura da rede
2  Fazer:
3      k = quantidade de imagens do conjunto de treinamento
4      Inicializar todos os filtros, parâmetros e pesos com valores aleatórios
5  Para s de 1 até k, fazer:
6      Usar a imagem s de treinamento como entrada
7      Convolução
8      ReLU, conforme equação 17
9      Propagação na camada totalmente conectada
10     Encontrar as probabilidades de saída para cada classe
11     Calcular o erro total na camada de saída, conforme equação 12
12     Utilizando o algoritmo backpropagation, calcular:
13         Os gradientes de erro em relação a todos os pesos da rede
14     Utilizando o gradiente descendente atualizar:
15         Os valores e pesos do filtro
16         Os valores dos parâmetros
```

Fonte: Adaptado de Karn, 2016, online

Ainda de acordo com Karn (2016), após o treinamento da RNC, os pesos e parâmetros estarão otimizados para classificar as imagens do conjunto de treinamento. Quando uma nova imagem (“não vista”) é inserida no RNC, a rede passaria pelo passo de propagação direta e emitiria uma probabilidade para cada classe.

Dessa forma, verifica-se que a precisão da classificação da rede depende da intensidade do treinamento e do tamanho da base de dados sobre a qual foi realizado o treinamento. Se ambas as condições forem razoáveis, como abordado por Karn (2016), a rede irá generalizar bem as novas imagens e classificá-las em categorias corretas.

Outro importante aspecto das RNCs, é que sua arquitetura é variável, podendo contar com quantas camadas de convolução, *pooling* e *ReLU* forem necessárias, além de possíveis outras camadas com métodos para melhorarem seu tempo de treinamento, que é um fator fundamental, principalmente para problemas grandes, com milhões de parâmetros e multiplicações.

Dessa forma, um dos métodos para aumentar sua velocidade, é o método *Linear low-rank*. Este método é utilizado para reduzir o tamanho do modelo da rede neural para acelerar o processo de treinamento reduzindo o tamanho da camada final (diminuindo a quantidade de parâmetros da matriz de pesos), como tratado no trabalho de Sainath et al. (2013, p. 6655-6656).

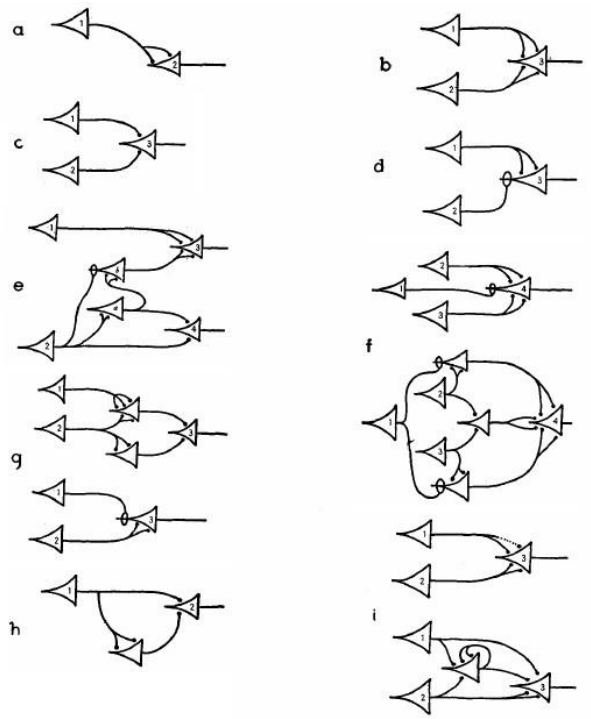
3.10 HISTÓRIA DAS REDES NEURAIS

Segundo Silva, Spatti e Flauzino (2010, p. 25), a primeira publicação relacionada a neurocomputação data de 1943, por meio do artigo elaborado por McCulloch e Pitts (1943). Neste trabalho, os autores realizaram a primeira modelagem matemática inspirada no neurônio biológico, o que resultou na primeira concepção de um neurônio artificial.

De acordo com Haykin (2007, p. 63), esse trabalho unificou os estudos de neurofisiologia e lógica matemática. Com neurônios suficientes e com conexões sinápticas ajustadas, os autores mostraram que uma rede desse tipo calcularia, a princípio, qualquer função computável.

Este era um resultado muito significativo e com ele é geralmente aceito o nascimento das áreas de RNA e a IA. O modelo original de McCulloch e Pitts pode ser visto na Figura 29.

Figura 29 – Modelo original do neurônio de McCulloch e Pitts.



Fonte: McCulloch e Pitts, 1943, p. 130.

Como descrito por Braga, Carvalho e Ludermir (2000, p. 2-3), o aprendizado relacionado às redes biológicas e artificiais veio a ser objeto de estudo somente

alguns anos depois do trabalho de McCulloch e Pitts. O primeiro trabalho de que se tem notícia relacionado ao aprendizado foi apresentado no trabalho de Hebb (1949).

Na pesquisa de Hebb demonstra-se como a plasticidade da aprendizagem das redes neurais é conseguida através da variação dos pesos de entrada dos nós. Ele propôs uma teoria para explicar o aprendizado em nós biológicos baseada no reforço das ligações sinápticas entre nós excitados. A regra de Hebb, como é conhecida a sua teoria na comunidade de RNAs, foi interpretada do ponto de vista matemático, e é hoje utilizada em vários algoritmos de aprendizado.

De acordo com Silva, Spatti e Flauzino (2010, p. 26), diversos outros pesquisadores continuaram o trabalho de desenvolvimento de modelos matemáticos fundamentados no neurônio biológico, gerando uma série de topologias (estruturas) e de algoritmos de aprendizado. Entre as linhas de pesquisa que surgiram destaca-se o trabalho de Rosenblatt (1958), que, no período compreendido entre 1957 e 1958, desenvolveu o primeiro neurocomputador, denominado *Mark I – Perceptron*, idealizando o modelo básico do *Perceptron*, um método de aprendizagem supervisionada. Uma foto do neurocomputador pode ser vista na Figura 30.

Figura 30 – Neurocomputador *Mark I – Perceptron*.

Fonte: TechGenix, 2017, online.

O *Perceptron* comporta-se como um classificador de padrões, dividindo o espaço de entrada em regiões distintas para cada uma das classes existentes, para

padrões que sejam linearmente separáveis (BRAGA; CARVALHO; LUDERMIR, 2000, p. 3).

Dessa forma, segundo Silva, Spatti e Flauzino (2010, p. 25), este modelo despertou interesse devido à sua capacidade em reconhecer padrões simples. Widrow e Hoff (1960) desenvolveram um tipo de rede denominada Adaline, que é a abreviatura de *ADAptive LINear Element*. Posteriormente, propôs-se o *Madaline*, a Adaline múltipla. Em seguida, foi escrito um importante artigo na área de IA, o *Steps toward artificial intelligence*, de Minsky (1961), onde há uma grande seção abordando as RNAs.

Em 1967 foi publicado o livro *Computation: finite and infinite machines* de Minsky (1967), que estendeu os resultados de McCulloch e Pitts e os colocou no contexto da teoria dos autômatos e da teoria da computação (HAYKIN, 2007, p. 64).

Durante o período dos anos 1960 existia a impressão de que as redes neurais poderiam realizar qualquer coisa. No entanto, como descrito por Silva, Spatti e Flauzino (2010, p. 25), em 1969 a área sofreu um revés com a publicação do clássico livro *Perceptrons: An introduction to computational geometry*, de Minsky e Papert (1969).

Os autores demonstraram de forma enfática a limitação de RNAs de apenas uma camada, como o *Perceptron* e o *Adaline*, em aprender o relacionamento entre as entradas e saídas de funções lógicas bem simples como o *XOR* (ou-exclusivo). Também afirmavam que não havia razão para supor que qualquer uma das limitações do Perceptron de camada única poderia ser superada na versão de múltiplas camadas.

Segundo Haykin (2007, 65-66), nos anos 70 houve um desinteresse pela pesquisa na área de RNAs, em boa parte motivado pelo trabalho de Minsky e Papert. Muitos pesquisadores, com exceção daqueles que trabalhavam em psicologia e em neurociências, abandonaram a área durante esta década.

De acordo com Silva, Spatti e Flauzino (2010, p. 26) e Haykin (2007, p. 66), em 1982 o físico Hopfield (1982) descreveu as redes recorrentes baseadas em funções de energia, desencadeando um grande interesse dos físicos teóricos pela modelagem neural. Este modelo de rede tornou-se conhecida como rede de Hopfield.

No final dos anos 1980 os pesquisadores voltaram a ter interesse na área de RNAs, devido a fatores como o desenvolvimento de computadores com maior

processamento, a criação de algoritmos de otimização mais eficientes e robustos e as novas descobertas sobre o sistema nervoso biológico.

Além disso, em 1986, foi desenvolvido no trabalho de Rumelhart, Hinton e Williams (1986) o algoritmo de treinamento de redes *Perceptron* de múltiplas camadas chamado de algoritmo de retropropagação (*backpropagation*). Naquele mesmo ano, foi publicado o livro *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, editado por Rumelhart, McClelland e PDP Research Group (1986).

O algoritmo *backpropagation* também foi bastante importante para reacender o interesse na área de RNAs, pois este permitia o ajuste dos pesos em uma rede com múltiplas camadas, sendo capaz de resolver problemas que o *Perceptron* simples não era capaz, como por exemplo o *XOR*.

De acordo com Haykin (2007, p. 69), no início dos anos 1990 nos trabalhos de Boser, Guyon e Vapnik (1992), Cortes e Vapnik (1995), Vapnik (1995) e Vapnik (1998), foram desenvolvidas as Máquinas de Vetor de Suporte, uma classe de redes de aprendizagem poderosa do ponto de vista computacional, que seriam utilizadas em reconhecimento de padrões, regressão e problemas de estimação densidade.

Segundo Géron (2017, p. 3), no trabalho de Hinton e Salakhutdinov (2006) foi demonstrado como treinar Redes Neurais de várias camadas, chamadas de Redes Neurais Profunda (RNP), que eram capazes de reconhecer dígitos escritos à mão em um nível de precisão de estado da arte, com cerca de 98%. Os autores chamaram esta técnica de *Deep Learning,* área que tem sido amplamente estudada desde os anos 2000, e que também é colocada como um campo do AM.

O artigo de Hinton e Salakhutdinov reavivou o interesse da comunidade científica na área de RNAs e demonstrou que a Aprendizagem Profunda não era apenas possível, mas capaz de realizar explicações que nenhuma outra técnica de AM poderia combinar (com a ajuda de um tremendo poder de computação e grandes quantidades de dados).

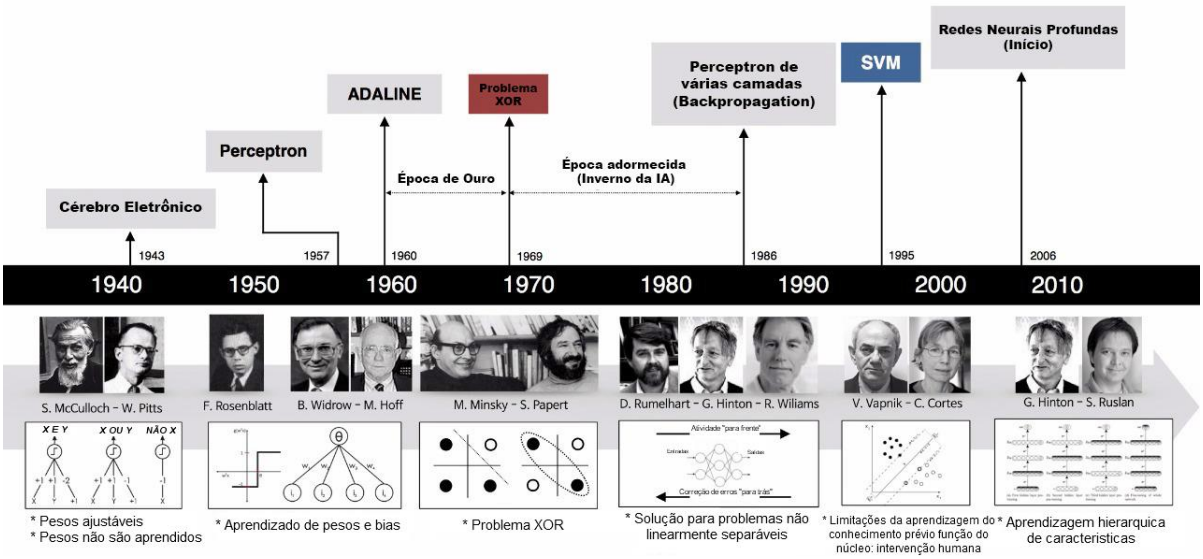
No século XXI, com o grande avanço da capacidade de processamento dos equipamentos de *hardware* foi possível treinar complexos modelos de redes, o que que antes não eram possível ou demorava muito tempo. Nesse contexto, as GPUs

(*Graphics Processing Unit*)[7] têm desempenhado um importante papel, pois são equipamentos eficientes nas manipulações numéricas e vetoriais envolvidas no AM. Dessa forma, atualmente o treinamento das redes pode ser realizado em modernas estruturas de *datacenters* de alto desempenho, com várias GPUs trabalhando em paralelo.

Como abordado por Géron (2017, p. 3), o rápido avanço do AM também impactou fortemente a indústria tecnológica, tornando-se o coração deste setor. Algumas áreas que tiverem bastante desenvolvimento neste sentido foram as de visão computacional e reconhecimento automática de imagens e de fala.

Por fim, verifica-se que atualmente existem várias tecnologias, tais como bibliotecas de *software* e *frameworks open source*[8], úteis para o desenvolvimento de modelos de AM e AP, assunto que é melhor abordado no quarto capítulo. Um resumo com os principais pontos da história das RNAs pode ser visto na Figura 31.

Figura 31 – Histórico das RNAs.



Fonte: Adaptado de Beam, 2017, online.

[7] GPU: Unidade de Processamento Gráfico, comumente conhecida por Placa de Vídeo.
[8] *Open source*: Código aberto, modelo de desenvolvimento de *software* que promove o licenciamento livre do código fonte, como alternativa para a indústria de *software*.

4 *FRAMEWORK TENSORFLOW*

Existem atualmente diversos sistemas disponíveis que visam a simplificação do desenvolvimento de modelos de RNAs; dentre eles, alguns, organizados por linguagem, são:

- Java: *Deeplearning4j*, do *Deeplearning4j Development Team* (2017).
- Python: *Theano*, de Al-Rfou et al. (2016), *Keras*, de Chollet (2015), *Microsoft Cognitive Toolkit*, do grupo Microsoft Research (2017), *Caffe*, de Jia et al. (2014), *Chainer*, de Tokui et al. (2015), *TensorFlow*, de Abadi et al. (2015).
- C++: *Kaldi*, de Povey et al. (2011).

Dessa forma, neste capítulo são descritos os principais conceitos do *framework TensorFlow*, além de sua instalação e seu funcionamento geral.

4.1 CONCEITOS INICIAIS

Como pode ser visto no trabalho de Abadi et al. (2015, p. 1), da equipe que desenvolveu a ferramenta, o *TensorFlow* é uma interface para expressar algoritmos de Aprendizado de Máquina e também uma implementação para executar tais algoritmos.

O sistema foi originalmente desenvolvido por pesquisadores da *Google Brain Team*, surgido do *Distbelief*, que era um sistema proprietário também de Aprendizado de Máquina lançado em 2011 (e descontinuado em 2015). O *TensorFlow* foi lançado como um projeto *open source* sob a licença Apache 2.0 em novembro de 2015, disponibilizado na plataforma *Github*[9], e está atualmente na versão 1.3, lançada em agosto de 2017.

Ainda segundo Abadi et al. (2015, p. 1), o sistema foi desenvolvido por pesquisadores de IA, mas ele é genérico o suficiente para ser aplicado em uma ampla variedade de outros domínios, incluindo os de treinamento e inferência para modelos de redes neurais profundas nas áreas de reconhecimento de fala, visão computacional, robótica, descoberta de medicamentos, etc.

Uma operação de computação com o *TensorFlow* pode ser executada em uma ampla variedade de arquiteturas / sistemas, desde dispositivos móveis, como *smartphones* e *tablets*, até sistemas distribuídos com milhares de GPUs e CPUs (*Central Processing Unit*)[10]. Atualmente, além da própria *Google*, a ferramenta é utilizada por empresas como *Airbnb*, *Ebay*, *Dropbox*, *Intel*, *IBM*, *Snapchat*, *Twitter* e *Uber*.

Por conseguinte, apesar do termo *TensorFlow* geralmente se referir à API utilizada para desenvolvimento e treinamento de modelos de Aprendizagem de Máquina, pode-se dizer que o *TensorFlow* é um pacote de *software* composto por:

- *TensorFlow*: É a API de aprendizado de máquina, onde os modelos são desenvolvidos nas linguagens Python ou C++ (ABADI et al. 2015, p. 1).
- *TensorBoard*: Ainda de acordo com Abadi et al. (2015, p. 12), o *TensorBoard* é uma ferramenta complementar ao *TensorFlow* que permite a visualização de estruturas gráficas e estatísticas. Ela facilita no entendimento da estrutura dos grafos de computação e do comportamento geral dos modelos de aprendizado de máquina. Toda a visualização é interativa, onde os usuários podem ampliar os nós agrupados do grafo para visualizar os detalhes.
- *TensorFlow Serving*: De acordo com Matos (2016), este é o *software* que facilita a instalação de modelos *TensorFlow* pré-treinados. Usando funções *TensorFlow* embutidas, o usuário pode exportar seu modelo para um arquivo que pode então ser lido nativamente pelo *TensorFlow Serving*. Assim, é possível iniciar um servidor simples de alto desempenho que pode receber dados de entrada, passá-los para o modelo treinado e retornar a saída do modelo.

4.2 INSTALAÇÃO

A documentação do *site* oficial da ferramenta, TensorFlow Team (2017a), descreve detalhadamente a instalação do sistema. É possível instalar o *software* nos

[9] *GitHub*: Plataforma *web* de hospedagem e compartilhamento de projetos de *software* que utiliza o programa de controle de versionamento *git*.
[10] CPU: Unidade Central de Processamento, mais conhecida como processador.

sistemas operacionais *Linux*, *Mac OS* e *Windows*, com as arquiteturas de processamento CPU e GPU. É possível instalar o sistema das seguintes formas:

1. Utilizando o gerenciador de pacotes *pip* dentro de um ambiente virtual isolado *virtualenv*.
2. Usando o gerenciador de pacotes *pip* sem um ambiente virtual isolado *virtualenv*.
3. Por meio de um contêiner *Docker*.
4. Utilizando o gerenciador de pacotes *Anaconda*.
5. Compilando e instalando os binários fonte.

Assim, para a realização do trabalho, o sistema foi instalado em uma máquina com arquitetura CPU e sistema operacional *GNU/Linux Debian*[11], utilizando um contêiner *Docker* – por ser a forma mais simples de instalação e ter sempre as bibliotecas mais atualizadas.

Por conseguinte, para a instalação do *Docker*, conforme a própria documentação oficial do sistema, em Docker (2017), foram utilizados os seguintes comandos, listados em ordem na Figura 32.

Figura 32 – Instalação do programa *Docker*.

```
$ sudo apt update
$ sudo apt install -y apt-transport-https ca-certificates curl gnupg2 software-
properties-common
$ curl -fsSL https://download.docker.com/linux/$(. /etc/os-release; echo
"$ID")/gpg | sudo apt-key add -
$ sudo apt-key fingerprint 0EBFCD88
$ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/$(.
/etc/os-release; echo "$ID") $(lsb_release -cs) stable"
$ sudo apt update && sudo apt install -y docker-ce
```

Fonte: Adaptado de Docker, 2017, online.

Por conseguinte, para que não fosse necessário utilizar o *Docker* com o usuário *root* do sistema hospedeiro, foi executado o seguinte comando, listado na Figura 33 (posteriormente, a máquina hospedeira foi reiniciada).

[11] Vale citar que, apesar da documentação oficial utilizar a distribuição *GNU/Linux Ubuntu*, a instalação também funciona (de forma idêntica) para a distribuição *Debian*.

Figura 33 – Configuração do *Docker*.

```
$ sudo usermod -aG docker $USER
```

Fonte: Adaptado de Docker, 2017, online.

Em seguida, para verificar se o sistema havia sido corretamente instalado, foi utilizado o seguinte comando, representado na Figura 34, que roda uma imagem de testes "*hello-world*", e imprime a mensagem "*Hello from Docker!*".

Figura 34 – Execução de teste do *Docker*.

```
$ docker run hello-world
```

Fonte: Adaptado de Docker, 2017, online.

Em seguida, para baixar a imagem *Docker tensorflow:nightly-py3*[12] e iniciar um novo contêiner, foi utilizado o seguinte comando, listado na Figura 35.

Figura 35 – *Download* e execução do contêiner do *TensorFlow*.

```
$ docker run -it tensorflow/tensorflow:nightly-py3 bash
```

Fonte: Adaptado de TensorFlow, 2017a, online.

Na primeira execução do comando, a imagem começará a ser baixada e, após o *download*, será iniciada uma nova sessão com o usuário *root* em um *bash* do contêiner. Para iniciar o contêiner em outros momentos, foi utilizado o mesmo comando. Após a inicialização do contêiner, foi necessário instalar alguns programas úteis para a execução do projeto com o seguinte comando apresentado na Figura 36.

Figura 36 – Instalação de dependências para o projeto.

```
# apt update && apt install -y vim git openjdk-8-jdk curl build-essential
libcurl3-dev libfreetype6-dev libpng-dev libzmq3-dev pkg-config python-dev
python-numpy python-pip software-properties-common swig zip zlib1g-dev
```

Fonte: Adaptado de TensorFlow, 2017a, online.

[12] A imagem *Docker* "*tensorflow", que é a versão* estável mais recente recomendada na documentação oficial (que corresponde a versão 1.3 do *TensorFlow*), não possui, até o atual momento, algumas bibliotecas utilizadas no projeto. Dessa forma, foi utilizada a imagem *tensorflow:nightly-py3* (que é uma imagem do tipo *nightly*, ou seja, é uma versão "não fechada" do *TensorFlow*, onde os desenvolvedores ainda estão trabalhando no código), que possui as bibliotecas necessárias.

Por conseguinte, outro programa instalado foi o *tzdata*, que configura o horário do contêiner. O programa pode ser instalado e configurado com os seguintes comandos, representados na Figura 37.

Figura 37 – Instalação de outra dependência e configuração do horário.

```
# apt install tzdata
# dpkg-reconfigure tzdata
```

Fonte: Autoria própria.

Com isso, o programa pergunta qual região geográfica deve ser utilizada, sendo necessário colocar a opção "*2. America*" e, em seguida, "*132. Sao_Paulo*". Também foi necessário instalar outra dependência: o programa *Bazel*. Para instalá-lo, foram utilizados os seguintes comandos, listados em ordem na Figura 38.

Figura 38 – Instalação do *software Bazel*.

```
# echo "deb [arch=amd64] http://storage.googleapis.com/bazel-apt stable jdk1.8" |
tee /etc/apt/sources.list.d/bazel.list
# curl https://bazel.build/bazel-release.pub.gpg | apt-key add -
# apt install bazel
```

Fonte: Adaptado de TensorFlow, 2017a, online.

Posteriormente, foi necessário configurar a variável de ambiente "*PATH*", editando arquivo "*~/.bashrc*", inserindo a seguinte linha de código, apresentada na Figura 39.

Figura 39 – Ajuste da variável de ambiente "*PATH"*.

```
1 export PATH="$PATH:$HOME/bin"
```

Fonte: Adaptado de TensorFlow, 2017a, online.

Em seguida, para testar se o *TensorFlow* funcionava corretamente, foi criado um *script* com o nome "*helloWorld.py*", com o seguinte código, apresentado na Figura 40.

Figura 40 – *Script* de teste de funcionamento do *TensorFlow.*

```
1 import tensorflow as tf
2 hello = tf.constant('Hello, world!')
3 sess = tf.Session()
4 print(sess.run(hello))
```

Fonte: Adaptado de TensorFlow, 2017a, online.

Assim, como a execução do *script* gerou a saída "*Hello, world!*", sem nenhum aviso de erro, foi possível concluir que o programa estava funcionando corretamente[13]. Posteriormente, clonou-se o repositório do *TensorFlow* no *GitHub* (no *branch master*), com o comando da Figura 41, que baixa a pasta "*tensorflow*", com os *scripts* e bibliotecas necessárias para o projeto.

Figura 41 – Clonando o repositório do *TensorFlow* no *GitHub*.

```
# cd ~
# git clone https://github.com/tensorflow/tensorflow.git
```

Fonte: Autoria própria.

Em seguida, dentro do diretório "*~/tensorflow*", foi necessário executar o comando "**.**/*configure"* para configurar o sistema*,* e então a ferramenta realizou as seguintes perguntas, listadas na Figura 42.

Figura 42 – Perguntas da configuração do *TensorFlow*.

```
01 Please specify the location of python. [Default is /usr/bin/python]: /usr/bin/python3
02 Please input the desired Python library path to use. Default is
[/usr/local/lib/python3.5/dist-packages]:
03 Do you wish to build TensorFlow with jemalloc as malloc support? [Y/n]: n
04 Do you wish to build TensorFlow with Google Cloud Platform support? [Y/n]: n
05 Do you wish to build TensorFlow with Hadoop File System support? [Y/n]: n
06 Do you wish to build TensorFlow with Amazon S3 File System support? [Y/n]: n
07 Do you wish to build TensorFlow with XLA JIT support? [y/N]: N
08 Do you wish to build TensorFlow with GDR support? [y/N]: N
09 Do you wish to build TensorFlow with VERBS support? [y/N]: N
10 Do you wish to build TensorFlow with OpenCL support? [y/N]: N
11 Do you wish to build TensorFlow with CUDA support? [y/N]: N
12 Do you wish to build TensorFlow with MPI support? [y/N]: N
13 Please specify optimization flags to use during compilation when bazel option "--
config=opt" is specified [Default is -march=native]:
```

Fonte: Autoria própria.

[13] É normal que apareçam alguns avisos dizendo que as operações de computação poderiam ser mais rápidas com certas bibliotecas. Esses avisos aparecem quando a instalação não foi feita utilizando os binários fonte.

Por conseguinte, foi utilizado o *script train.py* (explicado na seção 5.3.1), para que fosse inicializado o *download* do corpus oral. Após a mensagem de conclusão do *download*, o *script* foi interrompido.

Por fim, foi necessário persistir as mudanças realizadas no contêiner, ou então elas seriam perdidas quando a máquina hospedeira fosse desligada. Para isso, foi criada uma nova imagem com o nome *tensorflow-atualizado*, baseada no contêiner atualizado, com o seguinte comando[14], apresentado na Figura 43 (onde "*7bbea35054e8*" é o *id* do contêiner).

Figura 43 – Persistindo mudanças do contêiner.

```
$ docker commit 7bbea35054e8 tensorflow-atualizado
```

Fonte: Adaptado de Docker, 2017, online.

Para iniciar um novo contêiner baseado na imagem *Docker* atualizada (após o computador ter sido reiniciado), foi utilizado o seguinte comando, apresentado na Figura 44.

Figura 44 – Iniciando um novo contêiner baseado na imagem atualizada.

```
$ docker -it tensorflow-atualizado bash
```

Fonte: Autoria própria.

4.3 FUNCIONAMENTO GERAL

O *TensorFlow* fornece um conjunto extenso de funções e classes que permitem aos usuários definir modelos matemáticos a partir do zero. Ou seja, a princípio, é possível desenvolver qualquer modelo de Redes Neurais Artificiais (MATOS, 2016).

A ferramenta disponibiliza métodos pré-construídos para diversos tipos de algoritmos de AM, inclusive RNAs. Assim, existem vários métodos prontos para as etapas de uma RNA, desde o tratamento dos dados de entrada, até o treinamento, validação e testes. Dessa forma, nesta seção são apresentados os principais elementos do *TensorFlow*.

[14] É necessário executar o comando na máquina hospedeira, mas com o contêiner ainda ativado.

4.3.1 Tensores

De acordo com a documentação oficial, disponível em TensorFlow Team (2017a), o *TensorFlow*, como o próprio nome indica, é uma estrutura para definir e executar cálculos envolvendo os chamados tensores.

Um tensor é uma generalização de vetores e matrizes para dimensões potencialmente maiores. Internamente, o *TensorFlow* representa os tensores como matrizes de diversos tipos de dados de dimensão n. Um objeto *tf.Tensor* possui as seguintes propriedades: um tipo de dado, como um *float32*, e uma forma, que representa o número de dimensões do tensor.

Assim, ao escrever um programa utilizando a biblioteca, o principal objeto manipulado é o *tf.Tensor*. Um objeto *tf.Tensor* representa uma computação parcialmente definida que eventualmente produzirá um valor. Os programas escritos com a biblioteca funcionam inicialmente criando um grafo dos objetos *tf.Tensor*, detalhando como cada tensor é calculado com base nos outros tensores disponíveis e, em seguida, executando partes desse grafo para alcançar os resultados desejados.

Os principais tipos de tensores são: *tf.Variable*, *tf.Constant*, *tf.Placeholder* e o *tf.SparseTensor*. Com exceção do objeto *tf.Variable*, o valor de um tensor é imutável, o que significa que, no contexto de execução, um tensor tem apenas um valor. O que define a classificação de um objeto *tf.Tensor* é o seu número de dimensões. Assim, um objeto *tf.Tensor* de classificação *2* consiste em pelo menos uma linha e uma coluna. Na Figura 45, estão representadas duas formas de criar um tensor de dimensão 2.

Figura 45 – Exemplo de criação de um tensor de dimensão 2.

```
1  mymat = tf.Variable([[7],[11]], tf.int16)
2  myxor = tf.Variable([[False, True],[True, False]], tf.bool)
```

Fonte: Adaptado de TensorFlow, 2017a, online.

Por fim, a forma de um tensor (*shape*) representa o número de elementos em cada dimensão. Assim, as formas podem ser representadas através de listas Python, de tuplas de inteiros ou com o objeto *tf.TensorShape*. Dessa forma, um tensor *a* com os dados [1, 2, 3, 4, 5, 6, 7, 8, 9] tem a forma [9], e um tensor *b* com os dados [[1, 2, 3], [4, 5, 6], [7, 8, 9]] tem a forma [3, 3].

4.3.2 Tipos de dados e variáveis

A classe *tf.Dtype*, conforme descrito na documentação oficial, representa os tipos de dados do *TensorFlow*. Alguns desses tipos são apresentados no Quadro 1.

Quadro 1 – Exemplos de tipos de dados do *TensorFlow*.

Tipo de dado	Descrição
`tf.float64`	Número de ponto flutuante de dupla precisão de 64 bits
`tf.float16`	Número de ponto de flutuação de meia precisão de 16 bit
`tf.complex128`	Número complexo de dupla precisão de 128 bits.
`tf.int8`	Número inteiro com sinal de 8 bits.
`tf.int32`	Número inteiro com sinal de 32 bits.
`tf.uint8`	Número Inteiro sem sinal de 8 bits.
`tf.bool`	Valor booleano.
`tf.string`	Cadeia de caracteres.

Fonte: Adaptado de TensorFlow, 2017a.

As variáveis no *TensorFlow* são manipuladas através da classe *tf.Variable*, que representa um tensor cujo valor pode ser alterado executando operações sobre ele. Ao contrário dos objetos do tipo *tf.Tensor*, um objeto *tf.Variable* existe fora do contexto de uma chamada *session.run*. Assim, internamente, um objeto *tf.Variable* armazena um tensor persistente, onde operações específicas permitem ler e modificar os valores desse tensor.

A melhor forma de criar uma variável é chamar a função *tf.get_variable*, que requer o nome da variável. Para criar uma variável com o método *tf.get_variable*, basta fornecer um nome e uma forma, como pode ser verificado no código da Figura 46.

Figura 46 – Exemplo de criação de uma variável com o método *tf.get_variable*.

```
1 my_variable = tf.get_variable("my_variable", [1, 2, 3])
```

Fonte: Adaptado de TensorFlow, 2017a, online.

Isso cria uma variável chamada *my_variable*, que é um tensor tridimensional com forma [1, 2, 3], do tipo *tf.float32* e valor inicial aleatório. Para usar o valor de um

grafo *tf.Variable*, basta utilizá-lo como um objeto *tf.Tensor* comum, como apresentado no código da Figura 47.

Figura 47 – Exemplo de utilização de um grafo *tf.Variable*.

```
1  v = tf.get_variable("v", shape=(), initializer=tf.zeros_initializer())
2  w = v + 1
3  # w é um tf.Tensor que é calculado baseado no valor de v.
4  # Sempre que uma variável é usada em uma expressão, ela é
5  # convertida em um tf.Tensor que representa seu valor.
```

Fonte: Adaptado de TensorFlow Team (2017a)

Por fim, para atribuir um valor a uma variável, podem ser utilizados os métodos *assign* e *assign_add* da classe *tf.Variable*. Uma chamada ao método *assignment* pode ser conferido na Figura 48.

Figura 48 – Exemplo de atribuição de valor a uma variável.

```
1  v = tf.get_variable("v", shape=(), initializer=tf.zeros_initializer())
2  assignment = v.assign_add(1)
```

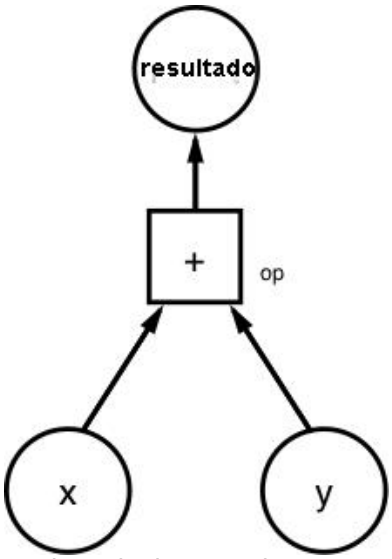
Fonte: Adaptado de TensorFlow, 2017a, online.

4.3.3 Grafos e sessões

Conforme descrito na documentação oficial, o *TensorFlow* usa um grafo de fluxo de dados para representar sua computação em termos das dependências entre operações individuais. Isso leva a um modelo de programação de baixo nível no qual primeiro se define o grafo de fluxo de dados e, em seguida, se cria uma sessão para executar partes do grafo.

O fluxo de dados é um modelo de programação comum para computação paralela. Em um grafo de fluxo de dados, os nós representam unidades de computação, e as bordas representam os dados consumidos ou produzidos por uma computação. Dessa forma, em um grafo *TensorFlow*, a operação *tf.add* corresponderia a um único nó com duas bordas de entrada (os valores a serem somados) e uma borda de saída (o resultado da adição). Esse exemplo pode ser visto na Figura 49.

Figura 49 – Representação de um grafo com a operação *tf.add*.



Fonte: Adaptado de Mesquita, 2017, online.

Sendo assim, como verificado em Mesquita (2017), um grafo é composto por um conjunto de objetos *tf.Operation*, que representam as unidades com as operações e um conjunto de objetos *tf.Tensor*, que representam as unidades com os dados. Assim, um código que cria o grafo da operação da soma de dois números pode ser verificado na Figura 50.

Figura 50 – Exemplo de criação de um grafo para uma operação de soma.

```
1  import tensorflow as tf
2  my_graph = tf.Graph()
3  with my_graph.as_default():
4      x = tf.constant([1,3,6])
5      y = tf.constant([1,1,1])
6      op = tf.add(x,y)
```

Fonte: Adaptado de Mesquita, 2017, online.

Dessa forma, no ciclo de trabalho do *TensowFlow*, inicialmente define-se o grafo e só depois são realizadas as computações (executar a operação de cada nó do grafo). Para isso, é necessário criar a classe *tf.Session*. O *TensorFlow* usa a classe *tf.Session* para, dentre outros usos, armazenar em cache informações sobre o *tf.Graph* para que seja possível executar com eficiência a mesma computação várias vezes.

Assim, segundo Mesquita (2017), um objeto do tipo *tf.Session* encapsula o ambiente onde as operações do grafo são executadas e os tensores são avaliados.

Para isso, é necessário definir qual grafo será utilizado na sessão, como pode ser visto no código apresentado na Figura 51, ainda com o exemplo da soma de dois números.

Figura 51 – Utilização da *tf.Session* para a execução de uma operação de soma.

```
1  import tensorflow as tf
2  my_graph = tf.Graph()
3  with tf.Session(graph=my_graph) as sess:
4      x = tf.constant([1,3,6])
5      y = tf.constant([1,1,1])
6      op = tf.add(x,y)
```

Fonte: Adaptado de Mesquita, 2017, online.

Por conseguinte, para executar as operações é necessário utilizar o método *tf.Session.run*, que é o mecanismo principal para executar o grafo ou avaliar um objeto *tf.Tensor*. Assim, é possível passar uma ou mais operações *tf.Operation* ou objetos *tf.Tensor* para o método *tf.Session.run* e então o *TensorFlow* executará as operações necessárias para o cálculo do resultado. Esse método executa um "passo" da computação do grafo.

É possível definir o que será executado através do argumento *fetches*, que pode ser um elemento do grafo, uma lista arbitrária, um dicionário, etc. No caso apresentado no código da Figura 52, será executado um passo do exemplo da operação de adição, que retorna o vetor [2 4 7].

Figura 52 – Utilização do método *tf.Session.run* para uma operação de soma.

```
1  import tensorflow as tf
2  my_graph = tf.Graph()
3  with tf.Session(graph=my_graph) as sess:
4      x = tf.constant([1,3,6])
5      y = tf.constant([1,1,1])
6      op = tf.add(x,y)
7      result = sess.run(fetches=op)
8  print(result)
```

Fonte: Adaptado de Mesquita, 2017, online.

4.3.4 Estimadores

Ainda de acordo com a documentação oficial da ferramenta, os estimadores são uma API de alto nível que simplifica a configuração, o treinamento e a avaliação de uma variedade de modelos de aprendizagem de máquina. Eles são baseados na

classe *tf.estimator* e encapsulam as ações de treinamento, avaliação, predição e exportação de modelos. Algumas características importantes dos estimadores são:

- Criam o grafo e a sessão automaticamente.
- Fornecem um ciclo de treinamento distribuído seguro que controla como e quando construir o grafo, inicializar as variáveis, lidar com as exceções, recuperar-se de falhas e salvar os sumários para o *TensorBoard*.

É possível construir estimadores manualmente ou utilizar os estimadores prontos do sistema. Os estimadores prontos permitem o uso de diferentes arquiteturas com poucas alterações de código. Alguns exemplos de estimadores prontos comuns do *tf.estimator* são:

- *DNNClassifier*: Classificador para RNAs profundas *feedforward*.
- *DNNRegressor*: Método de regressão para modelos RNP.
- *LinearClassifier*: Modelo de classificador linear.
- *LinearRegressor*: Estimador para problemas de regressão linear.

5 MODELO DE RECONHECIMENTO DE FALA

Neste capítulo, na primeira seção, foi descrito o funcionamento geral do modelo teórico da RNC utilizada e, na segunda seção, foram abordados o espectrograma e o método MFCC, que são transformações aplicadas nos áudios de entrada.

Já na terceira seção foi descrito o funcionamento das cinco ferramentas de *software* utilizadas para a geração do modelo de reconhecimento de voz. Por fim, na quarta seção discutiu-se os passos executados para a geração do modelo preditivo.

5.1 REDE *CNN-TRAD-FPOOL3*

O tipo de RNA utilizada para o modelo de reconhecimento de fala foi uma RNC com treinamento supervisionado, que foi apresentada no trabalho de Sainath e Parada (2015), conhecida por *cnn-trad-fpool3*. Este modelo de RNC foi implementada pelo grupo TensorFlow Team (2017b) e disponibilizado no repositório *GitHub* da ferramenta, que é explicado no *site* oficial do sistema, em TensorFlow Team (2017a).

Assim, no código da implementação utilizada, existem quatro modelos de RNAs disponíveis, são eles:

- *Single hidden fully-connected layer*: Um modelo com uma única camada oculta totalmente conectada, com resultados não muito precisos, mas com treinamento rápido e simples.
- *cnn-trad-fpool3*: É um modelo convolucional padrão, com boa precisão, mas que consome um volume razoável de processamento.
- *cnn-one-fstride4*: Modelo convolucional com baixos requisitos de computação.
- *Low latency SVDF Model*: Modelo com baixos requisitos de computação, mas também com menor precisão.

O modelo utilizado, o *cnn-trad-fpool3*, possui basicamente três componentes principais: (i) um módulo de extração de características, (ii) a rede neural profunda propriamente dita e (iii) um módulo de tratamento posterior. Os três componentes são abordados nas próximas subseções.

5.1.1 Módulo de extração de características

Como descrito no trabalho de Sainath e Parada (2015, p. 1478), o módulo de extração de características realiza a detecção de atividade de voz e gera um vetor de características de cada quadro de 10 ms dos arquivos de áudio.

Esses vetores de características são "empilhados" para criar um vetor maior (uma matriz), que é utilizado como entrada para a rede neural profunda. No caso, na saída tem-se uma matriz de dimensão $t \times f$, $t = 32$, $f = 40$, t representando o tempo e f a frequência do som.

5.1.2 Rede neural profunda *cnn-trad-fpool3*

Segundo Sainath e Parada (2015, p. 1478-1479), este módulo é composto especificamente pela rede neural profunda *cnn-trad-fpool3*, que é formada por duas camadas ocultas, uma camada de *linear low-rank*, uma camada profunda *feedforward* completamente conectada e uma camada com a função *softmax*, que produz uma estimativa da parte posterior de cada rótulo de saída – um rótulo para cada uma das palavras esperadas e mais um para as desconhecidas.

Os pesos e os *bias* da rede são estimados com o método *cross-entropy* e a taxa de aprendizado é realizada com um método chamado *asynchronous stochastic gradient descent with an exponential decay* (gradiente descendente estocástico assíncrono com uma queda exponencial).

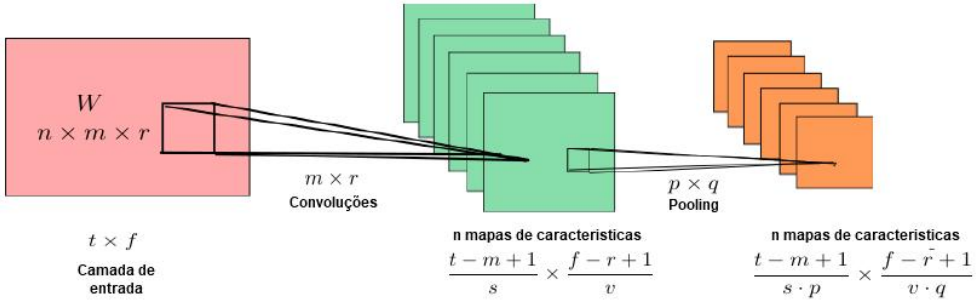
Dessa forma, ainda segundo Sainath e Parada (2015, p. 1478), assumindo que a o módulo de extração de características dê como entrada um vetor de dimensão $t \times f$, $t = 32$, $f = 40$, t representando o tempo e f a frequência do som, a rede possui as seguintes etapas / elementos:

- Um sinal de entrada de dimensão $t \times f$, $t = 32$, $f = 40$.
- Primeira camada oculta: Convolução com um filtro de dimensão $m \times r$, $m = 20$, $r = 8$. O filtro de convolução dá um passo de tamanho $s = 1$ (em relação ao tempo) e $v = 3$ (em relação à frequência).
- Com a primeira convolução, são geradas $n = 64$ *feature maps* (matrizes), cada um com dimensões $\frac{t - m + 1}{s} \times \frac{f - r + 1}{v} = \frac{32 - 20 + 1}{1} \times \frac{40 - 8 + 1}{1} = 13 \times 39$.

- Operação de *max-pooling* de dimensão $p \times q$, p em relação ao tempo e q em relação a frequência, $p = 1$, $q = 3$.
- Após a *max-pooling*, os *feature maps* terão dimensão $\frac{t-m+1}{sp} \times \frac{f-r+1}{vq} = \frac{32-20+1}{1*1} \times \frac{40-8+1}{1*3} = 13 \times 11$.
- Segunda camada oculta: Convolução com um filtro de dimensão $m \times r$, $m = 10$, $r = 4$. O filtro de convolução dá um passo de tamanho $s = 1$ (em relação ao tempo) e $r = 1$ (em relação à frequência).
- Com a segunda convolução, são geradas $n = 64$ *feature maps*, cada um com dimensões $\frac{t-m+1}{s} \times \frac{f-r+1}{v} = \frac{32-10+1}{1} \times \frac{40-4+1}{1} = 23 \times 37$.
- Terceira camada: Aplicação do método *Linear low-rank*, com $n = 32$ nós.
- Quarta camada: Camada *feedforward* totalmente conectada com $n = 128$ nós e inserção de *ReLU.*
- Quinta camada oculta: Função de ativação *softmax*, com $n = 4$ nós.

Estas etapas são resumidas na Figura 53.

Figura 53 – Etapas da rede *cnn-trad-fpool3*.



Fonte: Adaptado de Sainath e Parada, 2015, p. 1478.

Por fim, esta rede, apesar de possuir um alto grau de precisão para o reconhecimento de fala, demanda um alto grau de processamento. No fim de todas as etapas, a rede terá atualizado 244,2 mil parâmetros e realizado 9,7 milhões de operações de multiplicação (principalmente nas convoluções).

5.1.3 Módulo de manuseio posterior

Finalmente, o módulo de manuseio posterior, conforme tratado no trabalho de Sainath e Parada (2015, p. 1478-1479), combina a saída da RNP (os rótulos baseados nos quadros dos áudios) com uma pontuação de confiança das palavras-chave em questão. Dessa forma, uma decisão é tomada (a escolha de algum rótulo) caso a confiança exceder certo limite pré-definido.

5.2 ESPECTROGRAMA E MÉTODO MFCC

Antes da explicação dos algoritmos da próxima seção, é importante abordar duas transformações pelas quais as amostras de som passam antes de servirem como entrada para a rede. Estas transformações são a geração do espectrograma e a aplicação do método MFCC (*Mel Frequency Cepstral Coefficients*).

O espectrograma é uma forma de se representar as informações de áudio como uma série de fatias de informações de frequência, uma para cada janela (intervalo) de tempo do áudio. Assim, unindo essas fatias em uma sequência, forma-se uma representação única do som ao longo do tempo, chamada de impressão digital (*fingerprint*) MFCC.

A geração do espectrograma utiliza um algoritmo chamado *Fast Fourier Transform*, em português, Transformada Rápida de Fourier. Assim, o algoritmo que gera o espectrograma utilizado no trabalho recebe os seguintes parâmetros de entrada: os dados do áudio (armazenados como números reais no intervalo de -1 a 1), a largura da janela das amostras (480 ms) e o valor do passo especificando o quão longe mover a janela entre as fatias (160 ms). A partir disso, o algoritmo gera uma saída tridimensional com:

- Amplitude de cada frequência durante a fatia de tempo (é a menor dimensão).
- Tempo, com as sucessivas fatias de frequência.
- Canais do áudio de entrada, onde uma entrada de áudio estéreo, por exemplo, teria dois canais.

Por conta da ordem da memória do *TensorFlow*, na representação em uma imagem, o espectrograma gerado teria o tempo representado pelo eixo Y e a

frequência representada pelo eixo X, ao contrário da convenção usual para espectrogramas.

Posteriormente, o algoritmo utilizado aplica no espectrograma o método MFCC, que é uma maneira de representar os dados dos áudios de entrada para serem utilizados para o aprendizado de máquina. Utilizando o espectro de um espectrograma (um *cepstrum*), o algoritmo do MFCC descarta algumas das frequências mais altas que são menos significativas para o ouvido humano, transformando os dados em uma forma útil para o reconhecimento de fala.

5.3 FUNCIONAMENTO DAS FERRAMENTAS DE *SOFTWARE*

Nesta seção é descrito o funcionamento dos códigos dos programas utilizados[15] para a criação do modelo de reconhecimento de fala, sendo que todos foram desenvolvidos e disponibilizados pelo grupo TensorFlow Team (2017b). O principal *script* utilizado, o *train.py*, pode ser conferido no Apêndice A, já os outros *scripts*, bibliotecas e módulos podem ser encontrados no repositório oficial da ferramenta no *GitHub*, disponível em TensorFlow Team (2017b).

5.3.1 Ferramenta *train.py*

Inicialmente, no código desta ferramenta são realizadas as parametrizações iniciais do programa. Assim, entre as linhas 69 e 83 são importadas as bibliotecas e módulos necessários.

Em seguida, entre as linhas 286 e 426, com a verificação de execução *if __name__ == '__main__':* são parametrizados os argumentos de chamada do *script*, sendo eles:

- *data_url*: É um argumento que recebe uma *url* para *download* da base de dados de voz. Por padrão o parâmetro está configurado para baixar a base já citada, de Warden (2017).
- *data_dir*: É um argumento que define o diretório onde será baixada a base de dados de voz. Por padrão, tem o caminho "*/tmp/speech_dataset/*"'.

[15] A ferramenta *test_streaming_accuracy.cc* foi desenvolvida em C++, já as outras foram desenvolvidas em Python

- *background_volume*: Define o quão alto o ruído de fundo deve ser, entre 0 e 1. Por padrão, o valor é 0.1.
- *background_frequency*: Define quantas amostras de treinamento deverão ter ruído de fundo misturado. O valor padrão é 0.8, ou seja, 80%.
- *silence_percentage*: Argumento que define quantas amostras devem ser de silêncio, tendo por padrão a quantidade de 10%.
- *unknown_percentage*: Define quantas amostras devem ser de palavras desconhecidas, tendo por padrão o valor de 10%.
- *time_shift_ms*: Parâmetro que envolve um deslocamento aleatório no tempo dos dados da amostra de treinamento, de modo que uma pequena parte do começo ou do final dos áudios é cortada (em milissegundos) e a seção oposta é preenchida com zeros, o que imita as variações naturais no tempo de início nos dados de treinamento. O argumento tem como padrão o valor de 100 ms.
- *testing_percentage*: Porcentagem de amostras que serão utilizadas como conjunto de testes. Tem como padrão 10%.
- *validation_percentage*: Porcentagem de amostras que serão utilizadas como conjunto de validação. Tem como padrão o valor de 10%.
- *sample_rate*: Taxa de amostragem (em Hz) esperada dos arquivos *.wav*; tem como valor padrão 16000.
- *clip_duration_ms*: Duração esperada (em milissegundos) dos arquivos de áudio. Tem 1000 como valor padrão.
- *window_size_ms*: Controla a distância (em milissegundos) entre cada amostra de análise de frequência. Tem como padrão valor 10.0.
- *window_stride_ms*: controla o quão distante (em milissegundos) cada amostra de análise de frequência é da anterior. Tem como padrão valor 10.0.
- *dct_coefficient_count*: Define quantos *bins* usar para o método MFCC *fingerprint*, ou seja, quantos compartimentos serão usados para a contagem de frequência. Este parâmetro tem como padrão o valor 40.
- *how_many_training_steps*: Define quantos *loops* de treinamento deverão ser realizados, em duas etapas. O parâmetro tem como padrão o valor

"15000,3000", ou seja, a primeira etapa terá 15000 repetições e a segunda 3000 repetições.

- *eval_step_interval*: Parâmetro que controla com que frequência serão realizadas as avaliações dos resultados do treinamento (validações), onde o padrão é 400. Ou seja, a cada 400 repetições de treinamento será realizada uma validação dos resultados.
- *learning_rate*: Tamanho da taxa de aprendizado, que controla a velocidade das atualizações dos pesos da rede utilizados no treinamento. Tem como valor padrão o valor "0.001,0.0001", ou seja, na primeira etapa dos *loops* do treinamento (no caso os primeiros 15000) a taxa de aprendizado será de 0.001 e na segunda (os 3000 restantes) a taxa será de 0.0001.
- *batch_size*: Quantos itens serão treinados por vez, onde o valor padrão é 100.
- *summaries_dir*: Onde serão salvos os registros de resumo para o *TensorBoard*, onde o valor padrão é o caminho "*/tmp/retrain_logs*".
- *wanted_words*: Rótulos das palavras que serão usadas para o treinamento. O valor padrão foi alterado para "*zero,one,two,three,four,five,six,seven,eight,nine*". O *script* adicionará por padrão os rótulos "*_silence_*", para amostras de silêncio e o "*_unknown_*" para amostras que não coincidem com as palavras esperadas.
- *train_dir*: Caminho do diretório para escrever os *logs* de eventos e os pontos de verificação. O valor padrão é "*/tmp/speech_commands_train*".
- *save_step_interval*: Frequência de salvamento do ponto de verificação do modelo de treinamento, onde o valor padrão é 100.
- *model_architecture*: Argumento que define o modelo de rede que será utilizado. No caso, o valor padrão é "*conv*", que corresponde a rede *cnn-trad-fpoll3*, de Sainath e Parada (2015), abordada anteriormente.
- *check_nans*: Parâmetro que define se será realizada a verificação de números inválidos durante o processamento. Tem *false* como valor padrão.

Por conseguinte, após a realização das parametrizações, no método principal *main(_)*, entre as linhas 88 e 192, algumas configurações são ajustadas para serem

utilizadas no treinamento. Assim, com o código *tf.logging.set_verbosity(tf.logging.INFO)* são ativadas as mensagens de *log* e com *sess = tf.InteractiveSession()* é iniciada uma nova sessão do *TensorFlow*.

Em seguida, na linha 98, com o código *model_settings = models.prepare_model_settings(...)*, a função *prepare_model_settings()* do módulo *models.py* é chamada, onde é retornado um dicionário com os seguintes parâmetros necessários para o treinamento a rede, conforme representado na Figura 54.

Figura 54 – Dicionário retornado com o método *models_prepare_model_settings*.

```
{
  'desired_samples': 16000,
  'window_size_samples': 480,
  'window_stride_samples': 160,
  'spectrogram_length': 97,
  'dct_coefficient_count': 40,
  'fingerprint_size': 3880,
  'label_count': 10,
  'sample_rate': 16000
}
```

Fonte: Autoria própria.

Em seguida, na linha 102, com o código *audio_processor = input_data.AudioProcessor(...)* é importada a classe *AudioProcessor* do módulo *input_data.py*, que manipula o carregamento, o particionamento e a preparação dos dados do treinamento, que possui os seguintes métodos:

- *maybe_download_and_extract_database*: Verifica se a base de dados já foi baixada ou não. Se a base já tiver sido baixada, ela será utilizada para o treinamento. Caso negativo, o programa realiza o *download* do arquivo e o descompacta.
- *prepare_data_index*: Analisa as pastas dentro do diretório "*data_dir*", calcula os rótulos corretos para cada arquivo de áudio (com base no nome do subdiretório a que pertence) e atribui a uma partição as amostras de entrada.
- *prepare_background_data*: Procura uma pasta (com o nome *_background_noise_*) para os áudios de ruído de fundo e os carrega na memória para uso posterior.
- *prepare_processing_graph*: Constrói um grafo *TensorFlow* para aplicar as distorções de entrada: cria um grafo que carrega um arquivo .*wav*, o

decodifica, escala seu volume, faz seu deslocamento no tempo, acrescenta ruído de fundo, calcula seu espectrograma e, em seguida cria sua impressão digital MFCC.

- *set_size*: Calcula o número de amostras na partição do conjunto de dados.
- *get_data*: Reúne amostras do conjunto de dados, aplicando transformações conforme necessário (como a criação de amostras de silêncio).
- *get_unprocessed_data*: Recupera os dados de amostra para a partição fornecida sem realizar transformações.

Em seguida, na linha 125, com o código *fingerprint_input = tf.placeholder(...)* é gerado um tensor com o nome *fingerprint_input*, do tipo *float32*, na forma de um *array [None, fingerprint_size]*, com *fingerprint_size* igual a 3880, ou seja, é um tensor que representa uma matriz com uma quantidade de linhas ainda não definida e 3800 colunas.

Por conseguinte, na linha 128, nas variáveis *logits* e *dropout_prob* são salvos os resultados do método *create_model* do módulo *models.py*, que constrói a estrutura da arquitetura *conv* (*cnn-trad-fpoll3*) com os parâmetros adequados de uma rede convolucional: camadas de convolução, parametrização dos pesos e *bias*, inserção de *ReLU*, *max-pooling*, etc.

Em seguida, na linha 135, com o código *ground_truth_input = tf.placeholder(...)* é gerado um tensor com o nome *groundtruth_input*, de tipo indefinido, na forma de um *array [None, label_count]*, com *label_count = 10*, ou seja, é um tensor que representa uma matriz com uma quantidade de linhas ainda não definida e 10 colunas. Este tensor será utilizado para definir o erro e o otimizador.

Entre as linhas 141 e 161 é criada a estrutura para o *backpropagation* e a avaliação de treinamento no grafo. Na linha 166 as variáveis globais são salvas, assim, caso a execução do *script* seja interrompida, em uma próxima execução as variáveis possam ser recarregadas com os mesmos valores.

Entre as linhas 169 e 172 todos os resumos do treinamento e da validação são integrados e escritos em "*/tmp/retrain_logs/*". Na linha 176 a variável *start_step* é inicializada com o número 1, indicando que o treinamento irá inicializar na época 1.

Na linha 178, caso o programa tenha sido chamado com o parâmetro *start_checkpoint*, será recarregado o modelo pré-treinado anteriormente, com todas as variáveis e parâmetros definidos, inclusive a variável *start_step*, indicando em qual época o treinamento irá inicializar.

Na linha 182, com *tf.logging.info(...)*, o passo inicial do treinamento é impresso na tela. Já na linha 185, o grafo do treinamento e todos os parâmetros são salvos no arquivo "*conv.pbtxt*" no diretório "*/tmp/speech_commands_train*" e na linha 189 é salvo um arquivo com o nome "*conv_labels.txt*" no mesmo diretório com os rótulos definidos (mais o *_silence_* e o *_unknown_*).

Por conseguinte, entre as linhas 194 e 259 são realizados o treinamento e as validações. Assim, entre as linhas 196 e 200 são definidos os valores das taxas de aprendizado: para as primeiras 15000 repetições, a taxa é de 0.001 e para as 3000 restantes a taxa é de 0.0001.

Na linha 202 são inseridas as amostras de áudio que serão utilizadas para o treinamento, e entre as linhas 206 e 216 é executado o grafo com estas amostras. Já na linha 220, com o código *train_writer.add_summary(...)*, as informações do modelo e do treinamento são salvas. Na linha 221 as seguintes informações são impressas:

- *step*: É o número do passo (*loop*).
- *rate*: É valor do *learning rate*, que é de 0.001 para os 15000 primeiros *loops* e 0.0001 para os 3000 últimos.
- *accuracy*: Representa a porcentagem de classes previstas corretamente em um *loop* do treinamento. O modelo emite uma matriz de números, um para cada rótulo, em que cada número representa a probabilidade da entrada ser a classe prevista. O rótulo previsto é selecionado escolhendo-se a entrada com a pontuação mais alta, e a pontuação está sempre entre zero e um, onde valores mais altos representam maior confiança no resultado.
    - *cross entropy*: Representa o valor da entropia cruzada, que é o resultado da função de custo utilizada para orientar o processo de treinamento. A pontuação é obtida comparando o vetor de pontuações da execução do *loop* de treinamento com os rótulos corretos.

Em seguida, da linha 225 a 251, a cada 400 passos (e no último passo) do treinamento, o código realiza a validação e imprime a matriz de confusão para as predições do lote de amostras e, em seguida, baseado na matriz de confusão, imprime a acurácia da validação.

Na linha 254, a cada 100 passos o código salva o modelo completo em um arquivo com o nome “*conv.ckpt-passo*” (onde "*passo*" é o passo no momento), no diretório “*/tmp/speech_commands_train*”.

Por fim, o treinamento é repetido, até serem completadas todos os 18000 passos. Após a realização de todas as iterações do treinamento, o teste é então executado e a matriz de confusão com a acurácia da etapa é impressa.

5.3.2 Ferramenta *freeze.py*

Após o treinamento da rede, é possível gerar um arquivo binário do tipo *GraphDef* (também chamado de modelo congelado) com todas as informações da rede, que pode ser utilizado para reconhecer os comandos para os quais foi treinado.

Assim, o *script freeze.py* pode ser chamado com diversos parâmetros, a maioria semelhante (e com os mesmos valores padrão) aos utilizados no *script train.py*, e, dentre eles, o parâmetro *wanted_words*, na linha 175, foi alterado para "*default='zero,one,two,three,four,five,six,seven,eight,nine*'".

5.3.3 Ferramenta *generate_streaming_test_wav.py*

Após gerar o modelo congelado, é importante realizar testes para verificar a precisão da rede, onde a melhor forma de estimar o real desempenho desta é utilizá-la com um fluxo contínuo de som (ao invés de amostras individuais).

Assim, o *script generate_streaming_test_wav.py* gera um arquivo de áudio (longo) com palavras do conjunto de teste (ou seja, amostras que não foram utilizadas no treinamento da rede), inseridas a cada dois segundos, com ruídos de fundo e trechos de silêncio em posições aleatórias. Além disso, o *script* também gera um arquivo de texto com as informações do momento em que cada palavra foi pronunciada.

A ferramenta pode ser chamada com diversos parâmetros, a maioria semelhante (e com os mesmos valores padrão) aos parâmetros utilizados no *script train.py*, e, dentre eles, o parâmetro *wanted_words*, da linha 250, foi alterado para "*default='zero,one,two,three,four,five,six,seven,eight,nine*'".

Por padrão, o arquivo de som terá o nome "*streaming_test.wav*", o formato *.wav* e a duração de dez minutos, já o arquivo de texto terá o nome "*streaming_test_labels.txt*", e ambos serão salvos no diretório "*/tmp/speech_commands_train*".

5.3.4 Ferramenta *test_streaming_accuracy.cc*

Em seguida, é necessário testar o modelo congelado com o áudio gerado utilizando o programa *test_streaming_accuracy.cc*, que gera estatísticas de precisão. A ferramenta processa o fluxo de áudio, aplica o modelo congelado e acompanha quantos erros e sucessos o modelo obteve. Assim, é possível utilizar os seguintes parâmetros na chamada ao programa:

- *verbose*: Define se os logs de cada teste serão impressos em tela.
- *graph*: Parâmetro relativo ao caminho onde foi salvo o modelo congelado.
- *labels*: Parâmetro relativo ao caminho do arquivo de texto com os rótulos do treinamento.
- *wav*: Relativo ao caminho do arquivo de som contínuo.
- *ground_truth*: Relativo ao caminho do arquivo de texto com as informações dos momentos de pronúncia das palavras.

Dessa forma, o *script* imprime em tela três informações da precisão do modelo, são elas:

- *matched*: Porcentagem compatível, que representa o número total de sons que foram corretamente classificados. Uma classificação correta acontece quando o rótulo correto é escolhido para a amostra dentro de um determinado período de tempo definido pelo parâmetro "*time_tolerance_ms*" (que tem por padrão o valor 750 ms).
- *wrongly*: Porcentagem de erros, que representa a quantidade total de sons que desencadearam uma detecção (ou seja, modelo detectou que

não era silêncio ou ruído de fundo), mas a classe detectada estava errada.

- *false positives*: Porcentagem de falsos positivos, que representa quantas vezes o modelo disparou quando não havia nenhuma palavra real falada (silêncio ou ruído de fundo).

5.3.5 Ferramenta *label_wav.py*

Por fim, o *script* em *label_wav.py* testa o modelo congelado com um determinado arquivo de som, imprimindo as estatísticas de acerto. O programa recebe os seguintes parâmetros:

- *graph*: O caminho do arquivo com o modelo congelado.
- *labels*: O caminho do arquivo de texto com os do treinamento da rede.
- *wav*: O caminho do arquivo de áudio que será testado.
- *how_many_labels*: Mostra a quantidade de classificações que serão mostradas para o som analisado. Este parâmetro, na linha 129 do *script*, foi alterado para "*default=10*" para mostrar dez resultados.

5.4 GERAÇÃO DO MODELO PREDITIVO

Para gerar o modelo de reconhecimento dos dez comandos de voz (construído com o *TensorFlow*), foram executadas as seguintes etapas:

1. Executar o *script train.py* que:
    a. Monta os conjuntos de treinamento, validação e testes.
    b. Carrega os áudios e aplica as transformações de nivelamento, fatiamento e adição de ruído.
    c. Recebe os conjuntos de áudios transformados e gera os espectrogramas.
    d. Recebe os espectrogramas e aplica o método MFCC, que retorna uma matriz de características.
    e. Recebe as matrizes de características como entrada para a RNC *cnn-trad-fpoll3* e realiza o treinamento, as validações e os testes.

2. Executar o *script freeze.py*, que gerar um modelo congelado da rede treinada.
3. Executar o *script generate_streaming_test_wav.py*, que gera um arquivo de som contínuo para ser testado.
4. Executar o programa *test_streaming_accuracy.cc*, que testa a precisão do modelo congelado em relação ao arquivo de som contínuo.
5. Executar o *script label_wav.py* que testa a precisão do modelo congelado em relação a arquivos de som específicos.

Para a execução de todas as ferramentas de *software* foi utilizado um computador com as seguintes especificações:

- *Hardware*:
  - Memória: 6 GB, DDR3, 1333 MHz.
  - Armazenamento: SSD, 120 GB, R/W 540 MB/s.
  - SWAP: 8 GB (no SSD).
  - Processador: i3-3110M, 64 bits, 4 núcleos, 2,4 GHz.
- *Software*:
  - *GNU/Linux Debian* 9.0.1.
  - *Docker 17.06.2*.
  - Imagem *Docker: tensorflow/tensorflow:nightly-py3*.

Dessa forma, para treinar a rede, foi utilizada uma base gravações de voz (ou *corpus* oral) em inglês, pela dificuldade de se obter uma base gratuita em português. O corpus oral utilizado, disponível em Warden (2017), é distribuído sob a licença "*Creative Commons Attribution 4.0 International Public License*", e é composto por 64.727 arquivos de áudio (compactados em um arquivo de 2.3 GB) de 1 s, cada um contendo uma palavra falada em por uma pessoa diferente. Os arquivos possuem o formato *16-bit little-endian PCM-encoded WAVE* e uma taxa de amostragem de 16000 Hz.

Os mantenedores da base de dados coletaram os áudios utilizando *crowdsourcing*[16], por meio da plataforma *web* de gravação de voz do grupo AIY Projects (2017). As gravações são de pessoas de diversas localidades, gêneros e

[16] *Crowdsourcing*: Construção colaborativa.

idades dizendo as palavras: *"Yes", "No", "Up", "Down", "Left", "Right", "On", "Off", "Stop", "Go", "Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine", "Bed", "Bird", "Cat", "Dog", "Happy", "House", "Marvin", "Sheila", "Tree" e "Wow".*

Por conseguinte, para a geração do modelo de reconhecimento de fala propriamente dito, após a instalação e a configuração do *TensorFlow*, foi utilizado o *script train.py*, que é a principal ferramenta para a criação do modelo da RNA.

Para a execução desta ferramenta, foram realizadas alterações no código para que posteriormente fosse possível verificar o tempo das execuções total, do treinamento e validações e dos testes.

Dessa forma, para executar o programa *train.py*, foi executado o comando[17] apresentado na Figura 55, onde as saídas relativas ao tempo de execução foram escritas no arquivo "*~/tempo_execucao.txt*" e as relativas aos *logs* do modelo foram escritos no arquivo "*~/logs_tensorflow.txt*".

Figura 55 – Execução do *script train.py*.

```
# python3 train.py 1> ~/tempo_execucao.txt 2> ~/logs_tensorflow.txt
```

Fonte: Adaptado de TensorFlow, 2017b, online.

Assim, a execução do *script* consumiu um processamento em torno de 99% dos quatro núcleos da CPU com os seguintes tempos de execução:

- Duração total do programa: 31 horas, 21 minutos e 12 segundos.
- Duração do treinamento e validações: 31 horas, 17 minutos e 10 segundos.
- Duração dos testes: 1 min e 13 segundos.

Na Figura 56 estão representados o *log* do último passo da execução do treinamento, a matriz de confusão da validação realizada no último passo e o *log* das estatísticas da matriz de confusão da validação.

[17] Para todos os comandos executados nesta seção, é necessário estar no diretório "*~/tensorflow/tensorflow/examples/speech_commands*".

Figura 56 – Informações do treinamento e validação da rede.

```
INFO:tensorflow:Step #18000: rate 0.000100, accuracy 89.0%, cross entropy 0.384587
INFO:tensorflow:Confusion Matrix:
 [[250   0   0   0   0   0   0   0   0   0   0   0]
 [  0 179   5   9   1   8   2  11  10   7   5  13]
 [  1  11 245   0   1   0   0   0   2   0   0   0]
 [  2   8   0 212   0   0   1   3   1   0   0   3]
 [  4   7   6   0 206   1   2   0   2   5   2   1]
 [  2   3   2   0   1 230   0   1   0   4   5   0]
 [  1   8   0   6   0   0 256   3   1   3   2   0]
 [  3   8   0   0   0   2   0 217   0   0   5   7]
 [  3   2   1   0   1   0   0   0 245   0  10   0]
 [  2   9   0   0   3   0   0   2   4 242   1   0]
 [  1   0   0   1   0   5   0   1   4   0 229   2]
 [  1   2   0   2   0   1   0   3   1   1   1 218]]
INFO:tensorflow:Step 18000: Validation accuracy = 91.1% (N=2994)
```

Fonte: Autoria própria.

Assim, no *log* do último passo pode-se perceber que a acurácia ao fim do treinamento foi de 89,0%, ou seja, no último passo, a rede foi capaz de classificar corretamente 89,0% das amostras de entrada. Já na matriz de confusão[18] do processo de validação, realizado com um conjunto de 2994 amostras (dado por *N*), é possível obter várias informações a respeito das classificações da rede.

Por exemplo, é possível concluir que, como há poucos (e pequenos) números fora da diagonal principal, a rede classificou muito bem as amostras, distinguindo completamente os momentos de silêncio. Esta conclusão é corroborada pelas estatísticas da matriz de confusão: a acurácia da etapa de validação foi de 91,1%.

Em seguida, na Figura 57, é possível conferir a matriz de confusão e as estatísticas da etapa de testes da rede.

[18] A primeira linha da matriz de confusão representa o rótulo "*_unknown_*", a segunda representa o rótulo "*silence*", a terceira representa o rótulo "*zero*" e assim por diante, até a última, que representa o rótulo "*nine*". As colunas possuem a mesma representação das linhas.

Figura 57 – Matriz de confusão do processo de testes da rede.

```
INFO:tensorflow:set_size=3064
INFO:tensorflow:Confusion Matrix:
 [[256   0   0   0   0   0   0   0   0   0   0   0]
 [  0 190   9   4   5  10   2   6   6   8   2  14]
 [  1   9 225   0   6   1   0   1   2   4   0   1]
 [  0   9   0 219   0   0   0   5   0   2   2  11]
 [  0   8   0   0 236   1   4   1   2   3   9   0]
 [  1   7   0   0   2 235   0   1   3   2  15   1]
 [  1   8   1   4   1   0 228   8   1   1   0   0]
 [  1   4   0   1   0   0   1 258   1   0   2   3]
 [  1   0   0   0   0   1   0   1 232   1   7   1]
 [  0   5   1   0   0   0   0   2   0 227   1   3]
 [  0   2   1   1   2   0   0   1  11   1 237   1]
 [  0   5   0   0   0   0   0   8   1   0   3 242]]
INFO:tensorflow:Final test accuracy = 90.9% (N=3064)
```

Fonte: Autoria própria.

Dessa forma, é possível verificar que, nos testes, a rede também obteve altas taxas de acerto em suas classificações. Para uma amostra de 3064 elementos, verifica-se pela diagonal principal que o modelo obteve uma acurácia de 90,9%. Assim, é possível concluir que o treinamento gerou uma rede capaz de generalizar com uma boa qualidade as classificações dos números.

Por conseguinte, para gerar o modelo congelado da rede foi executado o *script freeze.py*, como apresentado na Figura 58. Com o comando, foi criado o arquivo binário "*my_frozen_graph.pb*" no caminho "*/tmp*" utilizando o último passo (18000) do treinamento realizado.

Figura 58 – Execução do *script freeze.py*.

```
# python3 freeze.py \
--start_checkpoint=/tmp/speech_commands_train/conv.ckpt-18000 \
--output_file=/tmp/my_frozen_graph.pb
```

Fonte: Adaptado de TensorFlow, 2017b, online.

Após a geração do modelo congelado, foi necessário gerar o arquivo de áudio contínuo. Assim, foi utilizado o *script generate_streaming_test.py*, como pode ser visto na Figura 59. O programa gerou o arquivo de som e o arquivo de texto com as informações do momento da pronúncia de cada palavra.

Figura 59 – Execução do *script generate_streaming_test_wav.py*.

```
# bazel run generate_streaming_test_wav
```

Fonte: Adaptado de TensorFlow, 2017b, online.

Em seguida, com o arquivo de áudio gerado, foi utilizado o programa *test_streaing_accuracy.cc*, que gerou estatísticas de precisão do modelo congelado. A chamada ao programa pode ser conferida na Figura 60 (os resultados da saída foram escritos em um arquivo de texto).

Figura 60 – Execução do programa *test_streaming_accuracy.cc*.

```
# bazel run test_streaming_accuracy -- \
--graph=/tmp/my_frozen_graph.pb \
--labels=/tmp/speech_commands_train/conv_labels.txt \
--wav=/tmp/speech_commands_train/streaming_test.wav \
--ground_truth=/tmp/speech_commands_train/streaming_test_labels.txt \
--verbose 2> /root/saída_test_streaming.txt
```

Fonte: Adaptado de TensorFlow, 2017b, online.

Assim, com a execução do programa, o modelo congelado apresentou as seguintes estatísticas de precisão:

- Acertos: 96,5%.
- Erros: 3,5%.
- Falsos positivos: 0%.

Por fim, a realização dos testes com o áudio contínuo, que representa de uma forma mais próxima os sons do mundo real, foram realizados vinte testes de classificação com arquivos de som específicos (sem ruídos e silêncio) utilizando o *script label_wav.py*, cuja chamada para testar um arquivo de som da palavra "*left*" pode ser visto na Figura 61.

Figura 61 – Execução do *script label_wav.py*.

```
# python3 label_wav.py \
--graph=/tmp/my_frozen_graph.pb \
--labels=/tmp/speech_commands_train/conv_labels.txt \
--wav=/tmp/speech_dataset/left/a5d485dc_nohash_0.wav
```

Fonte: Adaptado de TensorFlow, 2017b, online.

Na chamada à ferramenta, o parâmetro “*graph*” indica o caminho do grafo do modelo congelado, “*labels”* representa o caminho do arquivo de texto com os rótulos das palavras para as quais a rede foi treinada e “*wav”* indica o arquivo de som que servirá de entrada.

Dos vinte testes realizados com o programa, dez utilizaram sons relativos aos números e os outros dez utilizaram palavras que não estavam no treinamento da rede. Assim, nas Tabelas 3 e 4, estão representados os testes realizados para as palavras de zero a nove. Já nas Tabelas 5 e 6, estão representados os testes realizados para as palavras desconhecidas à rede.

Tabela 3 – Testes de reconhecimento dos sons de “*zero*” a “*four*”.

	RÓTULOS CORRETOS								
	zero (%)		**one (%)**		**two (%)**		**three (%)**		**four (%)**	
PREVISÕES	zero	96,77	one	98,13	two	86,03	three	99,09	four	93,96
	unknown	1,48	unknown	1,65	seven	7,10	unknown	0,78	unknown	2,78
	nine	1,39	four	0,15	unknown	2,70	eight	0,08	two	0,85
	eight	0,15	nine	0,06	zero	1,96	six	0,05	one	0,57
	three	0,10	five	0,01	four	1,70	nine	0,00	zero	0,51
	six	0,05	zero	0,00	six	0,22	zero	0,00	six	0,48
	one	0,05	eight	0,00	eight	0,11	five	0,00	seven	0,33
	seven	0,01	three	0,00	five	0,08	two	0,00	five	0,21
	two	0,00	seven	0,00	three	0,04	seven	0,00	three	0,15
	five	0,00	six	0,00	one	0,03	four	0,00	eight	0,06

Fonte: Autoria própria.

Tabela 4 – Testes de reconhecimento dos sons de “*five*” a “*nine*”.

	RÓTULOS CORRETOS								
	five (%)		**six (%)**		**seven (%)**		**eight (%)**		**nine (%)**	
PREVISÕES	five	78,82	six	96,12	seven	96,76	eight	89,54	nine	98,84
	unknown	12,18	seven	1,38	six	1,55	six	8,65	five	0,64
	seven	4,90	unknown	1,13	unknown	0,99	three	0,92	unknown	0,29
	nine	2,64	eight	0,54	five	0,22	unknown	0,86	one	0,21
	six	1,14	five	0,33	eight	0,16	two	0,01	three	0,02
	eight	0,11	three	0,15	two	0,16	five	0,01	zero	0,00
	four	0,07	two	0,11	one	0,05	zero	0,00	six	0,00
	one	0,05	four	0,08	nine	0,04	nine	0,00	seven	0,00
	zero	0,05	nine	0,08	zero	0,02	four	0,00	eight	0,00
	two	0,02	zero	0,06	four	0,02	one	0,00	four	0,00

Fonte: Autoria própria.

Tabela 5 – Testes de reconhecimento com sons desconhecidos (a).

	RÓTULOS CORRETOS									
	bed (%)		**bird (%)**		**cat (%)**		**dog (%)**		**down (%)**	
PREVISÕES	seven	71,75	unknown	75,94	unknown	65,97	unknown	98,12	unknown	63,74
	unknown	17,62	three	9,52	zero	13,43	nine	1,49	nine	13,55
	five	3,17	zero	8,79	six	12,03	zero	0,13	five	5,04
	nine	2,24	six	1,94	seven	2,23	five	0,12	two	3,85
	one	1,15	nine	1,71	two	2,09	four	0,10	four	3,75
	four	1,09	five	0,95	eight	1,56	one	0,02	zero	3,56
	zero	1,00	four	0,67	nine	1,37	six	0,01	seven	2,70
	six	0,83	seven	0,40	four	0,78	two	0,01	eight	1,27
	two	0,55	one	0,04	five	0,33	seven	0,01	one	0,98
	three	0,37	eight	0,03	one	0,14	eight	0,00	silence	0,67

Fonte: Autoria própria.

Tabela 6 – Testes de reconhecimento com sons desconhecidos (b).

	RÓTULOS CORRETOS									
	go (%)		**happy (%)**		**house (%)**		**left (%)**		**marvin (%)**	
PREVISÕES	unknown	83,30	unknown	55,85	unknown	89,28	unknown	92,42	one	56,45
	five	6,14	seven	18,18	five	4,44	nine	4,80	unknown	23,31
	nine	4,71	six	12,49	four	1,55	one	2,46	four	16,29
	zero	2,13	eight	9,59	six	1,23	three	0,19	five	1,57
	four	0,99	five	1,27	nine	0,93	five	0,07	nine	1,49
	two	0,93	three	0,85	seven	0,87	four	0,03	zero	0,37
	seven	0,73	two	0,61	eight	0,57	zero	0,02	three	0,28
	six	0,56	nine	0,47	two	0,56	eight	0,01	six	0,10
	eight	0,27	zero	0,41	zero	0,38	six	0,00	eight	0,09
	one	0,19	four	0,23	one	0,11	seven	0,00	two	0,03

Fonte: Autoria própria.

Dessa forma, com os testes realizados com a ferramenta *label_wav.py*, verificou-se que a rede reconheceu corretamente todos os dez números de zero a nove e classificou incorretamente duas das dez palavras desconhecidas. Ou seja, dos vintes testes, a rede classificou corretamente 90% dos casos.

Por fim, com os altos níveis de acerto das classificações da rede, tanto no teste com o arquivo de som contínuo quanto nos testes com os arquivos de som específicos, é possível concluir que o modelo de reconhecimento de fala gerado atinge um nível de “compreensão” satisfatório das palavras do treinamento.

CONSIDERAÇÕES FINAIS

As Redes Neurais Artificiais possuem grande relevância na atualidade, sendo aplicadas com sucesso na resolução de diversos tipos de problema, como no reconhecimento de fala, onde é possível construir poderosos modelos preditivos, com níveis de precisão no estado da arte e com grande adaptabilidade aos ruídos do ambiente.

Sendo assim, como o desenvolvimento completo do código de um modelo de RNA pode ser oneroso e dispendioso, nos últimos anos surgiram sistemas de AM que simplificam o desenvolvimento de modelos de RNAs. Dessa forma, tendo em vista a importância das RNAs na resolução de problemas e o recente surgimento destes sistemas de AM, o objetivo deste trabalho foi explicar a construção de uma RNA utilizando o *framework* de Aprendizado de Máquina *TensorFlow*, visando responder a seguinte questão central: “Como utilizar o *framework TensorFlow* no desenvolvimento de uma RNA”?

Para responder a essa questão, durante o trabalho foram apresentados os principais conceitos dos fenômenos físicos do som e da voz humana, das Redes Neurais Artificiais e do *framework TensorFlow*, para, em seguida, explicar as etapas do desenvolvimento e o funcionamento do código da RNA utilizada.

A rede neural utilizada foi a *cnn-trad-fpool3* – utilizada para classificação de comandos de voz –, que no trabalho foi treinada para reconhecer os sons em inglês dos números de zero a nove. Assim, foram abordados o funcionamento teórico, o código da implementação e as etapas da execução das ferramentas de *software* com o código da rede.

No tópico onde foram abordados os conceitos do som e da voz humana, na perspectiva de responder o questionamento “Como é produzido o som da voz humana”? foram definidos os fundamentos do som e da produção da voz humana, onde verificou-se que o som é uma onda longitudinal que se propaga em um meio, geralmente o ar. Já os sons da fala humana são produzidos pelas modificações vibratórias ocasionadas pela ação do aparelho fonador sobre a corrente de ar vinda dos pulmões.

Na abordagem da teoria das RNAs, o conteúdo foi desenvolvido visando responder a pergunta “O que são as Redes Neurais Artificiais e como podem ser

utilizadas na resolução de problemas”? Com isso, verificou-se que as redes neurais são baseadas no funcionamento do cérebro humano, que é composto por bilhões de neurônios que funcionam de forma paralela. Além disso, foi possível compreender os principais conceitos da área, tais como: neurônios artificiais, funções de ativação, algoritmos de treinamento, arquiteturas de rede, etc.

Já no tópico onde desenvolveu-se o conteúdo relacionado ao *framework TensorFlow*, objetivou-se responder a questão: “O que é, como instalar e como funciona o *TensorFlow*”? Assim, explicou-se os procedimentos para a instalação do *framework*, além dos principais conceitos e métodos deste.

No tópico onde foram desenvolvidos os princípios da rede *cnn-trad-fpool3*, tentou-se responder o questionamento “Como o *framework TensorFlow* foi utilizado no desenvolvimento da RNA para reconhecimento de fala”? Assim, verificou-se que a rede que possui basicamente três módulos: um de extração de características, que transforma as entradas de áudio em uma impressão digital MFCC; um da rede neural profunda, que é composta por duas camadas ocultas, uma camada de *linear low-rank*, uma camada profunda *feedforward* completamente conectada e uma camada com a função *softmax;* e o de tratamento posterior, que toma uma decisão de classificação de acordo com a pontuação de saída da rede.

Com isso, foi possível compreender o funcionamento básico da rede neural profunda *cnn-trad-fpool3*, com seus principais processos: convolução, *max-pooling*, função de custo *cross-entropy*, etc.

Por conseguinte, no tópico onde foram explicadas as etapas necessárias para a execução das ferramentas de *software* que implementam e realizam testes de precisão na rede, verificou-se que o treinamento e as validações para o reconhecimento dos dez comandos de voz levaram 31 horas, 17 minutos e 10 segundos, e os testes levaram 1 min e 13 segundos.

Na execução dos testes de precisão, verificou-se que a rede obteve uma acurácia de 89% em seu treinamento, de 91,1% em sua validação e de 90,9% em seus testes. Para os testes de precisão realizados com um áudio contínuo de 10 minutos, com várias palavras, a rede obteve 96,5% de acertos, 3,5% de erros e 0% de falsos positivos em suas classificações. Já para os testes realizados com vinte palavras isoladas (sem ruído e silêncio), a rede classificou corretamente 90% das amostras.

Assim, com estas estatísticas, foi possível denotar o alto nível de precisão e generalização do modelo desenvolvido, que estaria pronto para ser utilizado em uma aplicação final, como por exemplo, um aplicativo de reconhecimento de voz para *smartphones*.

Dessa forma, no trabalho foram explicadas todas as etapas para construção de uma RNA com o *framework TensorFlow*. Com isso, percebeu-se o grande número de métodos de AM pré-construídos e o alto nível de abstração que a ferramenta disponibiliza. Estas características permitiram o desenvolvimento de uma robusta RNA (em razoavelmente poucas linhas), o que permitiu concluir que, com o *framework* é possível construir poderosos modelos preditivos de forma simplificada.

Por conseguinte, é importante elencar as realizações pessoais e as dificuldades encontradas durante a realização do trabalho. Como realizações pessoais, pode-se citar o conhecimento obtido, tanto com os fascinantes campos de IA e das RNAs em si, quanto com as ferramentas e tecnologias utilizadas para o desenvolvimento do trabalho, tais como: *GitHub*, *Bitbucket*, *Limarka*, *LaTEX*, *Bibtex*, *Docker*, *git, VirtualBox* e *Markdown*.

Outra realização pessoal foi o conhecimento dos seguintes fatos relacionados a empresa *Google*: a qualidade de seus sistemas – onde vários são *open source*, disponíveis para estudo e alterações – e de suas pesquisas científicas – onde a maioria são abertas, disponíveis gratuitamente para leitura, o que é muito relevante, tendo em vista que pesquisas da área de IA geralmente são caras.

Já sobre os obstáculos encontrados no desenvolvimento do trabalho, pode-se citar o (curto) tempo disponível para a compreensão dos conceitos das RNAs e do *framework TensorFlow*, e a baixa capacidade de processamento da CPU utilizada para o treinamento da rede. Com a utilização de uma GPU de qualidade, por exemplo, o tempo de treinamento reduziria bastante.

Além disso, o objetivo inicial do trabalho era treinar a rede para reconhecer os números de zero a nove em português, mas como houve uma grande dificuldade em encontrar um corpus oral gratuito e aberto para uso em português brasileiro, foi necessário utilizar um com palavras em inglês (pois são mais simples de se encontrar).

Outra dificuldade encontrada foi que, pelo fato de ainda ser muito novo no mercado (surgiu em 2015), o *TensorFlow* ainda possui muitos *bugs* e bibliotecas em desenvolvimento. Assim, houveram várias situações onde o código da rede utilizava

bibliotecas que ainda não estavam completas na versão mais recente (1.3) do sistema. Este problema foi contornado utilizando-se uma imagem *Docker* mais atualizada com as bibliotecas necessárias.

Por fim, dois possíveis trabalhos futuros são o desenvolvimento de um corpus oral gratuito no idioma português brasileiro, colaborativo, *online* e aberto para uso e a análise da utilização do *TensorFlow* e de outros *frameworks* para a implementação de outras estruturas de RNAs de reconhecimento de fala, com os treinamentos destas realizados em uma GPU de qualidade.

REFERÊNCIAS

ABADI, M. et al. *TensorFlow: Large-scale machine learning on heterogeneous distributed systems*. 2015. Disponível em: <goo.gl/jzqEYX>. Acesso em: 19 ago. 2017.

AGHDAM, H. H.; HERAVI, E. J. *Guide to Convolutional Neural Networks*: *A practical application to traffic-sign detection and classification*. 2. ed. [S.l.]: Springer International Publishing, 2017. ISBN 9783319575506. DOI 10.1007/978-3-319-57550-6.

AIY PROJECTS. *Open speech recording*. 2017. Disponível em: <goo.gl/ykhqoo>. Acesso em: 03 ago. 2017.

AL-RFOU, R. et al. *Theano: For fast computation of mathematical expressions*. arXiv e-prints, abs/1605.02688, 2016. Disponível em: <goo.gl/JhdCun>. Acesso em: 15 set. 2017.

ALEKSANDER, I.; MORTON, H. *An introduction to neural computing*. 1. ed. London: Chapman and Hall, 1990. ISBN 9780412377808.

AMAZON INC. *Amazon Alexa*. 2017. Disponível em: <goo.gl/8bTesS>. Acesso em: 20 jun. 2017.

APPLE INC. *iOS Siri*. 2017. Disponível em: <goo.gl/pZQ3XH>. Acesso em: 18 fev. 2017.

ARBIB, M. A. *Brains, Machines and Mathematics*. 2. ed. New York: Springer-Verlag, 1987. ISBN 9781461291534.

BEAM, A. L. *Deep Learning 101 - Part 1* : *History and background*. 2017. Disponível em: <goo.gl/X9f2Vw>. Acesso em: 07 set. 2017.

BELLMAN, R. *An introduction to artificial intelligence*: *Can computers think?* Michigan: Boyd & Fraser, 1978. ISBN 9780878350667.

BOSER, B. E.; GUYON, I. M.; VAPNIK, V. N. A Training Algorithm for Optimal Margin Classifiers. In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. San Mateo: Morgan Kaufmann, 1992. (COLT ’92), p. 144-152. ISBN 089791497X. Disponível em: <goo.gl/UgpXnU>. Acesso em: 28 ago. 2017.

BRAGA, A. de P.; CARVALHO, A. C. P. de L. F.; LUDERMIR, T. B. *Redes neurais artificiais*: *Teoria e aplicações*. 1. ed. Rio de Janeiro: Livros Técnicos e Científicos - LTC, 2000. ISBN 9788521612186.

BRESOLIN, A. de A. *Reconhecimento de voz através de unidades menores do que a palavra, utilizando Wavelet Packet e SVM, em uma nova Estrutura Hierárquica de Decisão*. Tese (Doutorado) – Programa de Pós-Graduação em Engenharia Elétrica, Centro de Tecnologia, Universidade Federal do Rio Grande do

Norte, Natal, 2008. Disponível em: <goo.gl/t8NfcK>. Acesso em: 10 jun. 2017.

CAJAL, S. R. y. *Histologie du Système Nerveux de l'Homme & des Vertébrés*. 1. ed. Paris: Maloine, 1911.

CALLOU, D.; LEITE, Y. *Iniciação à fonética e à fonologia*. 11. ed. Rio de Janeiro: Zahar, 1990. ISBN 9788571100961.

CAVALIERE, R. S. *Pontos essenciais em fonética e fonologia*. 1. ed. Niterói: Nova Fronteira, 2011. ISBN 9788520933534.

CHOLLET, F. *Keras*: *The python deep learning library*. GitHub, 2015. Disponível em: <goo.gl/xDNbB1>. Acesso em: 02 jun. 2017.

CORTES, C.; VAPNIK, V. Support-vector networks. *Machine Learning*, Kluwer Academic Publishers, Boston, v. 20, n. 3, p. 273-297, 1995. ISSN 1573-0565. Disponível em: <goo.gl/HmNamL>. Acesso em : 01 jun. 2017.

CUNHA, C. F. da; CINTRA, L. F. L. *Breve gramática do português contemporâneo*. 1. ed. Lisboa: Edições J. Sá da Costa, 1985. ISBN 97292300506.

DAVIS, K.; BIDDULPH, R.; BALASHEK, S. Automatic recognition of spoken digits. *The Journal of the Acoustical Society of America*, v. 24, n. 6, p. 637-642, 1952. Disponível em: <goo.gl/mR7ETe>. Acesso em: 21 jun. 2017.

DEEPLEARNING4J DEVELOPMENT TEAM. *Deeplearning4j*: *Open-source distributed deep learning for the JVM*. 2017. Disponível em <goo.gl/vP3WYG>. Acesso em: 25 ago. 2017.

DENES, P. The design and operation of the mechanical speech recognizer at University College London. *Journal of the British Institution of Radio Engineers*, v. 19, n. 4, p. 219-229, 1959. Disponível em: <goo.gl/5MfGRg>. Acesso em: 10 jun. 2017.

DOCKER. *Get Docker CE for Debian*. Docker Docs. Disponível em <goo.gl/uj5aku>. Acesso em: 01 out. 2017.

FAGGIN, F. VLSI implementation of neural networks: Tutorial notes. In: *Proceedings of the International Joint Conference on Neural Networks*. Seattle: [s.n.], 1991.

FERGUS, R. *Neural Networks*. MLSS 2015 Summer School. Facebook AI Research. New York University, Courant Institute. Disponível em <goo.gl/TYpFok>. Acesso em 01 out. 2017.

FORGIE, J. M.; FORGIE, C. D. Results obtained from a vowel recognition computer program. *The Journal of the Acoustical Society of America*, v. 31, n. 11, p. 1480-1489, 1959. Disponível em: <goo.gl/z8ybMM>. Acesso em: 15 jun. 2017.

FRY, D. B. Theoretical aspects of mechanical speech recognition. *Journal of the British Institution of Radio Engineers*, v. 19, n. 4, p. 211-218, 1959. Disponível em:

<goo.gl/gHWWTT>. Acesso em: 10 jun. 2017.

GÉRON, A. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*: *Concepts, tools and techniques to build intelligent systems*. 1. ed. Sebastopol: O'Reilly Media, 2017. ISBN 9781491962244.

GOLLAPUDI, S. *Practical Machine Learning*. 1. ed. Mumbai: Packt Publishing, 2016. ISBN 9781784399689.

GOOGLE. *Cloud Speech API*. 2017. Disponível em: <goo.gl/d96198>. Acesso em: 17 fev. 2017.

GOOGLE. *Google Now*. 2017. Disponível em: <goo.gl/Bdu2L6>. Acesso em: 17 fev. 2017.

HAYKIN, S. *Redes Neurais*: *Princípios e prática*. 2. ed. Porto Alegre: Bookman, 2007. ISBN 9788573077186.

HEBB, D. O. *The organization of behavior: A neuropsychological theory*. 1. ed. New York: John Wiley & Sons, 1949.

HINTON, G. E.; SALAKHUTDINOV, R. R. Reducing the dimensionality of data with neural networks. *Science*, American Association for the Advancement of Science, v. 313, n. 5786, p. 504-507, 2006. ISSN 0036-8075. Disponível em: <goo.gl/xTuQFb>. Acesso em: 28 ago. 2017.

HISTORY OF COMPUTERS. *The Robots of Westinghouse*. 2012. Disponível em: <goo.gl/ zSMdFg>. Acesso em: 07 set. 2017.

HOPFIELD, J. J. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, National Academy Sciences, v. 79, n. 8, p. 2554-2558, 1982. Disponível em: <goo.gl/6isfu8>. Acesso em: 29 ago. 2017.

HUGO, M. *Uma interface de reconhecimento de voz para o sistema de gerenciamento de central de informação de fretes*. Dissertação (Mestrado) — Programa de Pós-Graduação em Engenharia de Produção, Centro Tecnológico, Universidade Federal de Santa Catarina, Florianópolis, 1995. Disponível em: <goo.gl/isHtJB>. Acesso em: 05 jun. 2017.

IDC. International Data Corporation (IDC). *The Digital Universe of Opportunities*: *Rich data and the increasing value of the internet of things*. 2014. Disponível em: <goo.gl/5QvdZV>. Acesso em: 25 ago. 2017.

JIA, Y. et al. Caffe: Convolutional architecture for fast feature embedding. In: *Proceedings of the 22Nd ACM International Conference on Multimedia*. New York: ACM, 2014. (MM' 14), p. 675-678. ISBN 9781450330633. Disponível em: <goo.gl/h9hc5k>. Acesso em: 28 ago. 2017.

KARN, U. *An Intuitive Explanation of Convolutional Neural Networks*. 2016.

Disponível em: <goo.gl/pm6F1C>. Acesso em: 28 set. 2017.

KOVÁCS, Z. L. *Redes neurais artificiais*: *Fundamentos e aplicaçoes, um texto básico*. 4. ed. São Paulo: Editora Livraria da Física, 2006. ISBN 8588325144.

KUBALA, F. et al. Continuous speech recognition results of the BYBLOS system on the darpa 1000-word resource management database. In: IEEE (Ed.). *ICASSP-88., International Conference on Acoustics, Speech and Signal Processing*. IEEE, 1988. v. 1, p. 291-294. ISSN 1520-6149. Disponível em: <goo.gl/VFVTPu>. Acesso em: 12 fev. 2017.

KURZWEIL, R. *The Age of Intelligent Machines*. Cambridge: MIT Press., 1990. ISBN 9780262111263.

LEE, K.-F. *Automatic Speech Recognition*: *The development of the SPHINX system*. 4. ed. Massachussetts: Springer Science & Business Media, 1999. ISSN 0893-3405. ISBN 0898382963.

MARTIN, T. B.; NELSON, A. L.; ZADELL, H. J. *Speech recognition by feature-abstraction*. [S.l.], 1964. Disponível em: <goo.gl/g1gpZt>.

MARTINS, J. A. *Avaliação de diferentes técnicas para reconhecimento da fala*. Tese (Doutorado) – Programa de Pós-Graduação em Engenharia Elétrica, Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas, Campinas, 1997. Disponível em: <goo.gl/CagiMG>. Acesso em: 10 fev. 2017.

MATOS, D. *Big Data + Deep Learning = Google TensorFlow*. 2016. Disponível em: <goo.gl/ZT8J7h>. Acesso em: 22 ago. 2017.

MCCLURE, N. *TensorFlow Machine Learning Cookbook*. 1. ed. [S.l.]: Packt Publishing, 2017. ISBN 978-1-78646-216-9.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, v. 5, n. 4, p. 115-133, 1943. ISSN 1522-9602. Disponível em: <goo.gl/XEr6sq>. Acesso em 01 set. 2017.

MENDEL, J. M.; MCLAREN, R. W. Reinforcement-learning control and pattern recognition systems: Theory and applications. In: MENDEL, J. M.; FU, K. S. (Ed.). *Adaptive, Learning, and Pattern Recognition Systems*. New York: Elsevier Science, 1970, (Mathematics in Science and Engineering, v. 66). p. 287-318. ISBN 9780080955759. Disponível em: <goo.gl/EQMp2j>. Acesso em: 08 ago. 2017.

MESQUITA, D. *Classificando textos com Redes Neurais e TensorFlow*. Medium. 2017. Disponível em: <goo.gl/qM8iAe>. Acesso em: 10 set. 2017.

MICROSOFT CORPORATION. *O que é a Cortana?* 2017. Disponível em: <goo.gl/YK3JnU>. Acesso em: 19 jun. 2017.

MICROSOFT RESEARCH. *Microsoft Cognitive Toolkit*: *Open source large-scale deep learning toolkit*. 2017. Disponível em: <goo.gl/JjhaNn>. Acesso em: 25 ago. 2017.

MINSKY, M. L. Steps toward artificial intelligence. *Proceedings of the IRE*, IEEE, v. 49, 1, p. 8-30, 1961. ISSN 0096-8390. Disponível em: <goo.gl/JMBmnp>. Acesso em: 26 ago. 2017.

MINSKY, M. L. *Computation*: *finite and infinite machines*. 1. ed. Englewood Cliffs: Prentice-Hall, 1967.

MINSKY, M. L.; PAPERT, S. *Perceptron*: An introduction to computational geometry. 1. ed. Cambridge: MIT Press., 1969. ISBN 9780262130431.

NIELSEN M. A. *Neural Networks and Deep Learning*. 1. ed. [S.I.]: Determination Press, 2015. Disponível em <goo.gl/e4iGVH>. Acesso em: 07 out. 2017.

OLSON, H. F.; BELAR, H. Phonetic typewriter. *The Journal of the Acoustical Society of America*, v. 28, n. 6, p. 1072-1081, 1956. Disponível em: <goo.gl/WcWs76>. Acesso em: 12 jun. 2017.

O'SHEA, K.; NASH, R. *An introduction to convolutional neural networks*. arXiv e-prints, abs/1511.08458, 2015.

PIERREL, J.-M. *Dialogue oral homme-machine*: *Connaissances linguistiques, stratégies et architectures des systèmes*. 1. ed. Paris: Hermès Science Publications, 1987. ISBN 9782866011055.

POVEY, D. et al. The kaldi speech recognition toolkit. In: *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*. IEEE Signal Processing Society, 2011. Disponível em: <goo.gl/efvCSb>. Acesso em: 15 set. 2017.

RABINER, L.; JUANG, B.-H. *Fundamentals of speech recognition*. 1. ed. New Jersey: PTR Prentice Hall, 1993. ISBN 0132858266.

REDDY, D. R. *An Approach to Computer Speech Recognition by Direct Analysis of the Speech Wave*. [S.l.], 1966. v. 40, n. C549, 1273-1273 p.

REZENDE, S. O. *Sistemas inteligentes*: Fundamentos e Aplicações. 1. ed. Barueri: Manole, 2005. ISBN 8520416837.

RICHERT, W.; COELHO, L. P. *Building Machine Learning Systems with Python*. 1 ed. Birmingham: Packt Publishing. 2013. ISBN 9781782161400.

ROBERTA, C. et al. *Sistema nervoso*. 2016. Disponível em: <goo.gl/y8PKWD>. Acesso em: 7 set. 2017.

ROSENBLATT, F. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, American Psychological Association, v. 65, n. 6, p. 386-408, 1958. Disponível em: <goo.gl/UMc1xZ>. Acesso em: 26 ago. 2017.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. Learning representations by

backpropagating errors. *Nature*, v. 323, n. 1, p. 533-536, 1986. Disponível em: <goo.gl/vdYVjW>.

RUMELHART, D. E.; MCCLELLAND, J. L.; PDP RESEARCH GROUP. *Parallel Distributed Processing*: Explorations in the microstructure of cognition. Cambridge: MIT Press., 1986. ISSN 1555-5216. ISBN 9780262680530.

RUSSEL, S.; NORVIG, P. *Inteligência artificial*: *Uma abordagem moderna*. 3. ed. Porto Alegre: Elsevier, 2013. ISBN 9788535237016.

SAINATH, T. N.; PARADA, C. Convolutional neural networks for small-footprint keyword spotting. In: *Sixteenth Annual Conference of the International Speech Communication Association*. Dresden: INTERSPEECH, 2015. p. 1478-1482. ISSN 1990-9770. Disponível em: <goo.gl/asxiXD>. Acesso em: 22 set. 2017.

SAINATH, T.N; KINGSBURY, B.; SINDHWANI, V.; ARISOY, E.; RAMABHADRAN, B. Low-rank matrix factorization for Deep Neural Network training with high-dimensional output targets. In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Vancouver: [S.l], 2013. p. 6655-6659. ISSN 1520-6149. Disponível em <goo.gl/hygqtv>. Acesso em: 09 ago. 2017.

SAKAI, T.; DOSHITA, S. The phonetic typewriter: Information processing. In: *International Federation for Information Processing (IFIP) Congress*. Munich: Amsterdam, North-Holland Pub. Co., 1962. v. 445, p. 445-450.

SAKOE, H.; CHIBA, S. Dynamic programming algorithm optimization for spoken word recognition. *IEEE transactions on acoustics, speech, and signal processing*, IEEE, v. 26, n. 1, p. 43-49, 1978. ISSN 0096-3518. Disponível em: <goo.gl/u8vHyB>. Acesso em: 17. jul. 2017.

SILVA, I. N. da; SPATTI, D. H.; FLAUZINO, R. A. *Redes neurais artificiais para engenharia e ciências aplicadas*: *Curso prático*. São Paulo: Artliber, 2010. ISBN 9788588098534.

SUZUKI, J.; NAKATA, K. Recognition of japanese vowels-preliminary to the recognition of speech. *Journal of the Radio Research Laboratory*, v. 37, n. 8, p. 193-212, 1961.

TECHGENIX. *How Nvidia video games changed the face of AI*. 2017. Disponível em: <goo.gl/J5MioS>. Acesso em: 7 set. 2017.

TENSORFLOW TEAM. *TensorFlow*. 2017. Disponível em: <goo.gl/zPaGrU>. Acesso em: 01 de jun. 2017.

TENSORFLOW TEAM. Tensorflow speech commands example. Medium. 2017. Disponível em: <goo.gl/1SU1b5>. Acesso em: 01 de jun. 2017.

TING, K. M. Confusion Matrix. In: SAMMUT, C.; WEBB, G. I. (Orgs.) *Encyclopedia of Machine Learning*. 1 ed. New York: Springer, 2011. ISBN 9780387303688. DOI 10.1007/978-0387-30164-8.

TOKUI, S. et al. Chainer: A next-generation open source framework for deep learning. In: *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*. [s.n.], 2015. v. 5. Disponível em: <goo.gl/RRL4UY>. Acesso em: 15 set. 2017.

VAPNIK, V. N. *The Nature of Statistical Learning Theory*. 1. ed. New York: Springer-Verlag, 1995. ISBN 0387945598.

VAPNIK, V. N. *Statistical Learning Theory*. 1. ed. New York: Wiley, 1998. ISBN 9780471030034.

VELICHKO, V. M.; ZAGORUYKO, N. G. Automatic recognition of 200 words. *International Journal of Man-Machine Studies*, Elsevier, v. 2, n. 3, p. 223-234, 1970. ISSN 0020-7373. Disponível em: <goo.gl/wtRAQK>. Acesso em: 17 jul. 2017.

VINTSYUK, T. K. *Speech discrimination by dynamic programming*. *Cybernetics and Systems Analysis*, v. 4, n. 1, p. 52-57, 1968. ISSN 1573-8337. Disponível em: <goo.gl/CF2XDg>. Acesso em: 25 jun. 2017.

WARDEN, P. *Speech Commands: A public dataset for single-word speech recognition*. 2017. Disponível em <goo.gl/Qg55Hj>. Acesso em: 11 set. 2017.

WIDROW, B.; HOFF, M. E. Adaptive switching circuits. *1960 IRE WESCON Convention Record*, IRE, p. 96-104, 1960. Disponível em: <goo.gl/8eJG4d>.

WU, J. *Introduction to Convolutional Neural Networks*. 2017. Disponível em: <goo.gl/pm6F1C>. Acesso em: 28 set. 2017.

YOUNG, H. D. et al. *Sears e Zemansky Física II* : *Ótica e fisica moderna*. 10. ed. São Paulo: Pearson Addison Wesley, 2003. ISBN 85886390303.

ZACCONE, G. *Getting started with TensorFlow*. 1. ed. Birmingham: Packt Publishing, 2016. ISBN 9781786468574.

APÊNDICE A

Neste apêndice é apresentado de forma integral o código da ferramenta de *software train.py*.

```
# Copyright 2017 The TensorFlow Authors. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# ==============================================================================
r"""Simple speech recognition to spot a limited number of keywords.

This is a self-contained example script that will train a very basic audio
recognition model in TensorFlow. It downloads the necessary training data and
runs with reasonable defaults to train within a few hours even only using a
CPU. For more information, please see
https://www.tensorflow.org/tutorials/audio_recognition.

It is intended as an introduction to using neural networks for audio
recognition, and is not a full speech recognition system. For more advanced
speech systems, I recommend looking into Kaldi. This network uses a keyword
detection style to spot discrete words from a small vocabulary, consisting of
"yes", "no", "up", "down", "left", "right", "on", "off", "stop", and "go".

To run the training process, use:
bazel run tensorflow/examples/speech_commands:train

This will write out checkpoints to /tmp/speech_commands_train/, and will
download over 1GB of open source training data, so you'll need enough free
space and a good internet connection. The default data is a collection of
thousands of one-second .wav files, each containing one spoken word. This data
set is collected from
https://aiyprojects.withgoogle.com/open_speech_recording,
please consider contributing to help improve this and other models!

As training progresses, it will print out its accuracy metrics, which should
rise above 90% by the end. Once it's complete, you can run the freeze script
to get a binary GraphDef that you can easily deploy on mobile applications.

If you want to train on your own data, you'll need to create .wavs with your
recordings, all at a consistent length, and then arrange them into subfolders
organized by label. For example, here's a possible file structure:

my_wavs >
  up >
```

```
    audio_0.wav
    audio_1.wav
  down >
    audio_2.wav
    audio_3.wav
  other>
    audio_4.wav
    audio_5.wav

You'll also need to tell the script what labels to look for, using the
`--wanted_words` argument. In this case, 'up,down' might be what you want, and
the audio in the 'other' folder would be used to train an 'unknown' category.

To pull this all together, you'd run:

bazel run tensorflow/examples/speech_commands:train -- \
--data_dir=my_wavs --wanted_words=up,down

"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import argparse
import os.path
import sys

import numpy as np
from six.moves import xrange  # pylint: disable=redefined-builtin
import tensorflow as tf

import input_data
import models
from tensorflow.python.platform import gfile
import datetime
FLAGS = None

print ("HORARIO - INICIO GERAL: " + str(datetime.datetime.now()))
def main(_):
  # We want to see all the logging messages for this tutorial.
  tf.logging.set_verbosity(tf.logging.INFO)

  # Start a new TensorFlow session.
  sess = tf.InteractiveSession()

  # Begin by making sure we have the training data we need. If you already
  # have training data of your own, use `--data_url= ` on the command line
  # to avoid downloading.
  model_settings = models.prepare_model_settings(
      len(input_data.prepare_words_list(FLAGS.wanted_words.split(','))),
      FLAGS.sample_rate, FLAGS.clip_duration_ms, FLAGS.window_size_ms,
      FLAGS.window_stride_ms, FLAGS.dct_coefficient_count)
  audio_processor = input_data.AudioProcessor(
      FLAGS.data_url, FLAGS.data_dir, FLAGS.silence_percentage,
      FLAGS.unknown_percentage,
      FLAGS.wanted_words.split(','), FLAGS.validation_percentage,
      FLAGS.testing_percentage, model_settings)
  fingerprint_size = model_settings['fingerprint_size']
```

```
label_count = model_settings['label_count']
time_shift_samples = int((FLAGS.time_shift_ms * FLAGS.sample_rate) / 1000)
# Figure out the learning rates for each training phase. Since it's often
# effective to have high learning rates at the start of training, followed
# by lower levels towards the end, the number of steps and learning rates
# can be specified as comma-separated lists to define the rate at each
# stage. For example --how_many_training_steps=10000,3000 –
# learning_rate=0.001,0.0001 will run 13,000 training loops in total, with
# a rate of 0.001 for the first 10,000, and 0.0001 for the final 3,000.
training_steps_list=list(map(int,FLAGS.how_many_training_steps.split(',')))
learning_rates_list = list(map(float, FLAGS.learning_rate.split(',')))
if len(training_steps_list) != len(learning_rates_list):
  raise Exception(
      '--how_many_training_steps and --learning_rate must be equal length '
      'lists, but are %d and %d long instead' % (len(training_steps_list),
                                                 len(learning_rates_list)))

fingerprint_input = tf.placeholder(
    tf.float32, [None, fingerprint_size], name='fingerprint_input')

logits, dropout_prob = models.create_model(
    fingerprint_input,
    model_settings,
    FLAGS.model_architecture,
    is_training=True)

# Define loss and optimizer
ground_truth_input = tf.placeholder(
    tf.float32, [None, label_count], name='groundtruth_input')

# Optionally we can add runtime checks to spot when NaNs or other symptoms
# of numerical errors start occurring during training.
control_dependencies = []
if FLAGS.check_nans:
  checks = tf.add_check_numerics_ops()
  control_dependencies = [checks]
# Create the back propagation and training evaluation machinery in the
# graph.
with tf.name_scope('cross_entropy'):
  cross_entropy_mean = tf.reduce_mean(
      tf.nn.softmax_cross_entropy_with_logits(
          labels=ground_truth_input, logits=logits))
tf.summary.scalar('cross_entropy', cross_entropy_mean)
with tf.name_scope('train'), tf.control_dependencies(control_dependencies):
  learning_rate_input = tf.placeholder(
      tf.float32, [], name='learning_rate_input')
train_step = tf.train.GradientDescentOptimizer(
    learning_rate_input).minimize(cross_entropy_mean)
predicted_indices = tf.argmax(logits, 1)
expected_indices = tf.argmax(ground_truth_input, 1)
correct_prediction = tf.equal(predicted_indices, expected_indices)
confusion_matrix = tf.confusion_matrix(expected_indices, predicted_indices)
evaluation_step = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
tf.summary.scalar('accuracy', evaluation_step)

global_step = tf.contrib.framework.get_or_create_global_step()
increment_global_step = tf.assign(global_step, global_step + 1)

```

```
166   saver = tf.train.Saver(tf.global_variables())
167
168   # Merge all the summaries and write them out to /tmp/retrain_logs (default)
169   merged_summaries = tf.summary.merge_all()
170   train_writer = tf.summary.FileWriter(FLAGS.summaries_dir + '/train',
171                                        sess.graph)
172   validation_writer = tf.summary.FileWriter(FLAGS.summaries_dir+'/validation')
173
174   tf.global_variables_initializer().run()
175
176   start_step = 1
177
178   if FLAGS.start_checkpoint:
179     models.load_variables_from_checkpoint(sess, FLAGS.start_checkpoint)
180     start_step = global_step.eval(session=sess)
181
182   tf.logging.info('Training from step: %d ', start_step)
183
184   # Save graph.pbtxt.
185   tf.train.write_graph(sess.graph_def, FLAGS.train_dir,
186                        FLAGS.model_architecture + '.pbtxt')
187
188   # Save list of words.
189   with gfile.GFile(
190     os.path.join(FLAGS.train_dir, FLAGS.model_architecture + '_labels.txt'),
191         'w') as f:
192     f.write('\n'.join(audio_processor.words_list))
193   print("HORARIO INICIO - TREINAMENTO E VALIDACOES: " + str(datetime.datetime.now()))
194   # Training loop.
195   training_steps_max = np.sum(training_steps_list)
196   for training_step in xrange(start_step, training_steps_max + 1):
197     # Figure out what the current learning rate is.
198     training_steps_sum = 0
199     for i in range(len(training_steps_list)):
200       training_steps_sum += training_steps_list[i]
201       if training_step <= training_steps_sum:
202         learning_rate_value = learning_rates_list[i]
203         break
204     # Pull the audio samples we'll use for training.
205     train_fingerprints, train_ground_truth = audio_processor.get_data(
206         FLAGS.batch_size, 0, model_settings, FLAGS.background_frequency,
207         FLAGS.background_volume, time_shift_samples, 'training', sess)
208     # Run the graph with this batch of training data.
209     train_summary, train_accuracy, cross_entropy_value, _, _ = sess.run(
210         [
211             merged_summaries, evaluation_step, cross_entropy_mean, train_step,
212             increment_global_step
213         ],
214         feed_dict={
215             fingerprint_input: train_fingerprints,
216             ground_truth_input: train_ground_truth,
217             learning_rate_input: learning_rate_value,
218             dropout_prob: 0.5
219         })
220     train_writer.add_summary(train_summary, training_step)
221     tf.logging.info('Step #%d: rate %f, accuracy %.1f%%, cross entropy %f' %
222                     (training_step, learning_rate_value, train_accuracy * 100,
223                      cross_entropy_value))
```

```
224     is_last_step = (training_step == training_steps_max)
225     if (training_step % FLAGS.eval_step_interval) == 0 or is_last_step:
226       set_size = audio_processor.set_size('validation')
227       total_accuracy = 0
228       total_conf_matrix = None
229       for i in xrange(0, set_size, FLAGS.batch_size):
230         validation_fingerprints, validation_ground_truth = (
231             audio_processor.get_data(FLAGS.batch_size, i, model_settings, 0.0,
232                                      0.0, 0, 'validation', sess))
233         # Run a validation step and capture training summaries for TensorBoard
234         # with the `merged` op.
235         validation_summary, validation_accuracy, conf_matrix = sess.run(
236             [merged_summaries, evaluation_step, confusion_matrix],
237             feed_dict={
238                 fingerprint_input: validation_fingerprints,
239                 ground_truth_input: validation_ground_truth,
240                 dropout_prob: 1.0
241             })
242         validation_writer.add_summary(validation_summary, training_step)
243         batch_size = min(FLAGS.batch_size, set_size - i)
244         total_accuracy += (validation_accuracy * batch_size) / set_size
245         if total_conf_matrix is None:
246           total_conf_matrix = conf_matrix
247         else:
248           total_conf_matrix += conf_matrix
249       tf.logging.info('Confusion Matrix:\n %s' % (total_conf_matrix))
250       tf.logging.info('Step %d: Validation accuracy = %.1f%% (N=%d)' %
251                       (training_step, total_accuracy * 100, set_size))
252
253     # Save the model checkpoint periodically.
254     if (training_step % FLAGS.save_step_interval == 0 or
255         training_step == training_steps_max):
256       checkpoint_path=os.path.join(FLAGS.train_dir,FLAGS.model_architecture + '.ckpt')
257       tf.logging.info('Saving to "%s-%d"', checkpoint_path, training_step)
258       saver.save(sess, checkpoint_path, global_step=training_step)
259   print("FIM - TREINAMENTO E VALIDACOES: " + str(datetime.datetime.now()))
260   print (" HORARIO INICIO - TESTES: " + str(datetime.datetime.now()))
261   set_size = audio_processor.set_size('testing')
262   tf.logging.info('set_size=%d', set_size)
263   total_accuracy = 0
264   total_conf_matrix = None
265   for i in xrange(0, set_size, FLAGS.batch_size):
266     test_fingerprints, test_ground_truth = audio_processor.get_data(
267         FLAGS.batch_size, i, model_settings, 0.0, 0.0, 0, 'testing', sess)
268     test_accuracy, conf_matrix = sess.run(
269         [evaluation_step, confusion_matrix],
270         feed_dict={
271             fingerprint_input: test_fingerprints,
272             ground_truth_input: test_ground_truth,
273             dropout_prob: 1.0
274         })
275     batch_size = min(FLAGS.batch_size, set_size - i)
276     total_accuracy += (test_accuracy * batch_size) / set_size
277     if total_conf_matrix is None:
278       total_conf_matrix = conf_matrix
279     else:
280       total_conf_matrix += conf_matrix
281   print (" HORARIO FIM - TESTES: " + str(datetime.datetime.now()))
```

```
282   tf.logging.info('Confusion Matrix:\n %s' % (total_conf_matrix))
283   tf.logging.info('Final test accuracy = %.1f%% (N=%d)' % (total_accuracy *
284                                                            100, set_size))
285   print ("HORARIO - FIM GERAL: " + str(datetime.datetime.now()))
286 if __name__ == '__main__':
287   parser = argparse.ArgumentParser()
288   parser.add_argument(
289     '--data_url',
290      type=str,
291      # pylint: disable=line-too-long
292      default='http://download.tensorflow.org/data/speech_commands_v0.01.tar.gz',
293      # pylint: enable=line-too-long
294      help='Location of speech training data archive on the web.')
295   parser.add_argument(
296     '--data_dir',
297     type=str,
298     default='/tmp/speech_dataset/',
299     help="""\
300     Where to download the speech training data to.
301     """)
302   parser.add_argument(
303     '--background_volume',
304     type=float,
305     default=0.1,
306     help="""\
307     How loud the background noise should be, between 0 and 1.
308     """)
309   parser.add_argument(
310     '--background_frequency',
311     type=float,
312     default=0.8,
313     help="""\
314     How many of the training samples have background noise mixed in.
315     """)
316   parser.add_argument(
317     '--silence_percentage',
318     type=float,
319     default=10.0,
320     help="""\
321     How much of the training data should be silence.
322     """)
323   parser.add_argument(
324     '--unknown_percentage',
325     type=float,
326     default=10.0,
327     help="""\
328     How much of the training data should be unknown words.
329     """)
330   parser.add_argument(
331     '--time_shift_ms',
332     type=float,
333     default=100.0,
334     help="""\
335     Range to randomly shift the training audio by in time.
336     """)
337   parser.add_argument(
338     '--testing_percentage',
339     type=int,
```

```
340     default=10,
341     help='What percentage of wavs to use as a test set.')
342   parser.add_argument(
343     '--validation_percentage',
344     type=int,
345     default=10,
346     help='What percentage of wavs to use as a validation set.')
347   parser.add_argument(
348     '--sample_rate',
349     type=int,
350     default=16000,
351     help='Expected sample rate of the wavs',)
352   parser.add_argument(
353     '--clip_duration_ms',
354     type=int,
355     default=1000,
356     help='Expected duration in milliseconds of the wavs',)
357   parser.add_argument(
358     '--window_size_ms',
359     type=float,
360     default=30.0,
361     help='How long each spectrogram timeslice is',)
362   parser.add_argument(
363     '--window_stride_ms',
364     type=float,
365     default=10.0,
366     help='How long each spectrogram timeslice is',)
367   parser.add_argument(
368     '--dct_coefficient_count',
369     type=int,
370     default=40,
371     help='How many bins to use for the MFCC fingerprint',)
372   parser.add_argument(
373     '--how_many_training_steps',
374     type=str,
375     default='15000,3000',
376     help='How many training loops to run',)
377   parser.add_argument(
378     '--eval_step_interval',
379     type=int,
380     default=400,
381     help='How often to evaluate the training results.')
382   parser.add_argument(
383     '--learning_rate',
384     type=str,
385     default='0.001,0.0001',
386     help='How large a learning rate to use when training.')
387   parser.add_argument(
388     '--batch_size',
389     type=int,
390     default=100,
391     help='How many items to train with at once',)
392   parser.add_argument(
393   '--summaries_dir',
394     type=str,
395     default='/tmp/retrain_logs',
396     help='Where to save summary logs for TensorBoard.')
397   parser.add_argument(
```

```
398     '--wanted_words',
399     type=str,
400     default='zero,one,two,three,four,five,six,seven,eight,nine',
401     help='Words to use (others will be added to an unknown label)',)
402   parser.add_argument(
403     '--train_dir',
404     type=str,
405     default='/tmp/speech_commands_train',
406     help='Directory to write event logs and checkpoint.')
407   parser.add_argument(
408     '--save_step_interval',
409     type=int,
410     default=100,
411     help='Save model checkpoint every save_steps.')
412   parser.add_argument(
413     '--start_checkpoint',
414     type=str,
415     default='',
416     help='If specified, restore this pretrained model before any training.')
417   parser.add_argument(
418     '--model_architecture',
419     type=str,
420     default='conv',
421     help='What model architecture to use')
422   parser.add_argument(
423     '--check_nans',
424     type=bool,
425     default=False,
426     help='Whether to check for invalid numbers during processing')
427
428 FLAGS, unparsed = parser.parse_known_args()
429 tf.app.run(main=main, argv=[sys.argv[0]] + unparsed
```