Unioeste - Universidade Estadual do Oeste do Paraná
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
Colegiado de Ciência da Computação
Curso de Bacharelado em Ciência da Computação

Um Estudo Comparativo Entre as Bibliotecas Google TensorFlow e MATLAB® *Deep Learning Toolbox* Sobre a Construção de Redes Neurais Profundas

João Pedro Silveira

CASCAVEL
2018

JOÃO PEDRO SILVEIRA

UM ESTUDO COMPARATIVO ENTRE AS BIBLIOTECAS GOOGLE TENSORFLOW E MATLAB® *DEEP LEARNING TOOLBOX* SOBRE A CONSTRUÇÃO DE REDES NEURAIS PROFUNDAS

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação, do Centro de Ciências Exatas e Tecnológicas da Universidade Estadual do Oeste do Paraná - Campus de Cascavel

Orientador: Prof. Josué Pereira de Castro

CASCAVEL
2018

JOÃO PEDRO SILVEIRA

UM ESTUDO COMPARATIVO ENTRE AS BIBLIOTECAS GOOGLE TENSORFLOW E MATLAB® *DEEP LEARNING TOOLBOX* SOBRE A CONSTRUÇÃO DE REDES NEURAIS PROFUNDAS

Monografia apresentada como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação, pela Universidade Estadual do Oeste do Paraná, Campus de Cascavel, aprovada pela Comissão formada pelos professores:

Prof. Josué Pereira de Castro
Colegiado de Ciência da Computação,
UNIOESTE

Prof. Adriana Postal
Colegiado de Ciência da Computação,
UNIOESTE

Prof. Suzan Kelly Borges Piovesan
Colegiado de Ciência da Computação, UTFPR

Cascavel, 5 de dezembro de 2018

Lista de Figuras

Lista de Abreviaturas e Siglas

API	*Application Programming Interface* – Interface de Programação de Aplicações
AVX2	Intel® *Advanced Vector Extensions* 2 – Extensões Avançadas de Vetores da Intel® 2
CPU	*Central Processing Unit* – Unidade Central de Processamento
GPU	*Graphics Processing Unit* – Unidade de Processamento Gráfico
IA	Inteligência Artificial
RNA	Rede Neural Artificial
SGD	*Stochastic Gradient Descent* – Gradiente Descendente Estocástico
TPU	*Tensor Processing Unit* - Unidade de Processamento de Tensor

Lista de Símbolos

x	Um sinal de entrada de um neurônio
w	O peso de uma entrada de um neurônio
x_i	O i-ésimo conjunto de sinais de entrada da rede neural artificial
y_i	Um conjunto de sinais de saída associado com o i-ésimo conjunto de sinais de entrada
$\hat{y}_i$	Um conjunto de sinais de saída estimados pela RNA para o i-ésimo conjunto de entrada
$f(s)$	Função de ativação

Sumário

Resumo

Este trabalho busca comparar as bibliotecas Google TensorFlow e MATLAB® *deep learning toolbox* para criação de sistemas de RNAs. Foram desenvolvidos dois testes, uma RNA *feedforward* para classificar espécies de flores da base de dados Iris e uma CNN para reconhecimento de dígitos manuscritos do MNIST[1]. O MATLAB® apresentou um desempenho superior para o treinamento dos modelos e para predição de grande quantidade de dados. TensorFlow apresentou uma modelagem de RNAs mais simplificada, intuitiva e flexível.

Palavras-chave: TensorFlow, Aprendizado de Máquina, Redes Neurais Artificiais Profundas

[1] *Modified National Institute of Standards and Techonogy database*

Capítulo 1

Introdução

A área de Inteligência Artificial (IA) vêm crescendo em vários aspectos. O número de artigos pertinentes à área publicados anualmente aumentou mais que nove vezes comparando os anos de 1996 e 2018 (COLUMBUS, 2018). Investiu-se em 2018 seis vezes mais em *startups* relacionadas à IA do que nos anos 2000 (COLUMBUS, 2018). Os sistemas inteligentes estão cada vez mais eficientes e próximos da população. A área conta com diversas técnicas para resolver uma variedade de problemas como: navegação autônoma, detecção de fraudes bancárias e reconhecimento de voz.

Redes Neurais Artificiais (RNAs) é uma técnica promissora de IA que busca simular redes neurais humanas em um computador. Esta técnica adquiriu grande notoriedade na área devido a sua eficiência, capacidade de aprendizado e vasta aplicabilidade. Devido ao seu sucesso, muitas ferramentas foram criadas para desenvolver sistemas inteligentes com estas RNAs. Python (Python Core Team, 2015) é a linguagem de programação que tem liderado o ranking de linguagens preferidas para desenvolvimento de sistemas de RNAs (Developer Economics, 2018). A linguagem chegou nesta posição devido a existência de diversas bibliotecas e *frameworks*, como TensorFlow (ABADI et al., 2015), Keras (CHOLLET et al., 2015) e Theano (Theano Development Team, 2016), que facilitam o processo de desenvolvimento.

TensorFlow é uma biblioteca *open source* de cálculo numérico computacional de alta performance originalmente desenvolvida por engenheiros da Google. A biblioteca ganhou notoriedade e chegou a ser declarada como o repositório nº 1 no Github. Alguns pontos que a levaram a esta posição são: suporte para múltiplas plataformas, alta escalabilidade e comunidade numerosa e ativa (BHATIA, 2017).

1.1 Objetivo

Este trabalho tem o objetivo de comparar as bibliotecas Google TensorFlow e MATLAB® *deep learning toolbox*, identificando arquiteturas de RNAs que podem ser construídas por uma ou ambas as bibliotecas, quais algoritmos de treinamento são disponilizados por cada uma, como devem ser processadas/codificadas as saídas e entradas de dados da rede e quais são as plataformas que cada biblioteca suporta. Pretende-se estudar também quais outras facilidades são disponibilizadas por elas para o desenvolvedor.

1.1.1 Objetivos Específicos

- Estudar as arquiteturas disponíveis nas bibliotecas, identificando suas principais características como algoritmo de treinamento e funções de ativação.
- Estudar as possíveis formas de entrada e saída permitidas para cada arquitetura.
- Identificar as plataformas de execução suportadas.
- Testar as capacidades de cada arquitetura com relação aos seguintes quesitos:
  - Facilidade de implementação
  - Manutenibilidade
  - Eficiência
  - Portabilidade

1.2 Organização do Trabalho

Os seguintes itens serão abordados neste trabalho

Redes Neurais Artificias

Contém uma curta introdução às RNAs. Apresenta um breve histórico, fundamentos, organização e categorias de RNAs.

RNAs *Feedforward*

Explica o funcionamento desta categoria de RNAs em detalhes mais específicos.

Recursos Utilizados

Aponta os recursos de *software* e *hardware* utilizados para o desenvolvimento dos testes, assim como apresenta as ferramentas TensorFlow e MATLAB®.

Desenvolvimento

Apresenta o processo que fundamentou o desenvolvimento dos testes, as bases de dados utilizadas e explica o desenvolvimento dos modelos construídos.

Resultados e Conclusões

Retrata os resultados alcançados neste trabalho e a conclusão atingida.

Trabalhos Futuros

Apresenta possíveis continuidades deste trabalho.

Capítulo 2

Redes Neurais Artificiais

2.1 Introdução

Criadas com o intuito de simular a inteligência humana, as redes neurais artificiais, RNAs ou somente redes neurais, buscam reproduzir a capacidade do cérebro humano de aprender alguma coisa baseado em experiências a que o indivíduo é submetido. Para isto foram desenvolvidos modelos matemáticos que buscam descrever o funcionamento do nosso cérebro.

2.2 O Cérebro Humano

O cérebro é o principal órgão que compõe o sistema nervoso humano, responsável por receber estímulos dos órgãos sensoriais, armazenar informações e acionar impulsos motores. Esse órgão é composto por múltiplos neurônios, estima-se que estão na casa dos 100 bilhões, e cada um deles se comunica com os outros por meio de conexões, ou sinapses, que chegam ao número de 10.000 por neurônio (CAMPOS; SAITO, 2004).

O neurônio é uma célula especializada de comportamento simples, ele recebe sinais de conexões específicas, aplica uma função apropriada no sinal acumulado a fim de formar uma resposta e, por fim, transmite esta resposta para outros neurônios relacionados (CARDOSO, 2001b). Sua estrutura pode ser separada em três partes básicas: dendritos, corpo celular, também conhecido como soma ou núcleo, e axônios, e são dispostas como mostrado na Figura 2.1.

O comportamento do neurônio, assim como sua estrutura, é simples. Estímulos eletroquímico de outros neurônios são recebidos pelos dendritos por meio das sinapses, e então o estímulo é levado até o núcleo em forma de sinal elétrico a fim de potencializar a membrana da célula, quando a membrana atinge o potencial de repouso, também conhecido como potencial

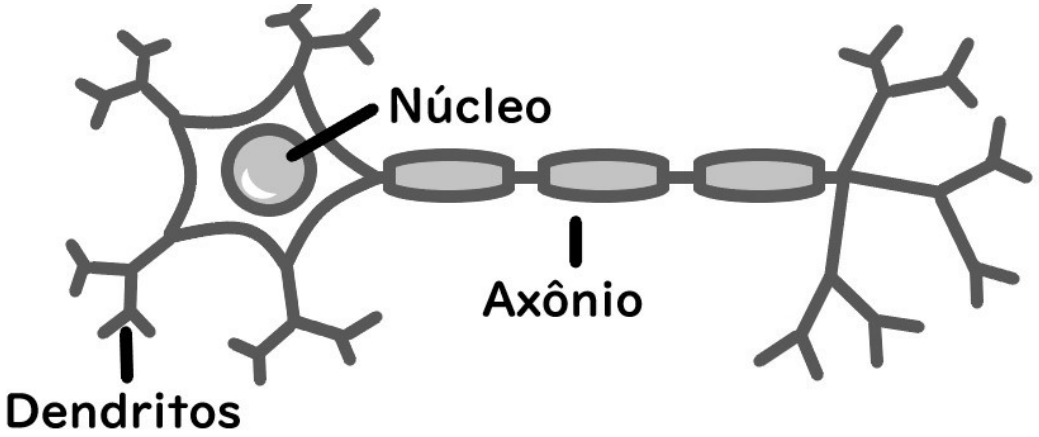


Figura 2.1: Estrutura básica de um neurônio

de ação ou limiar de ativação, que é o estímulo mínimo necessário para que o impulso nervoso de resposta seja disparado até os outros neurônios por meio do axônio (KOVACS, 1996; FAUSETT, 1994).

Sinapses químicas, assim como as do cérebro, podem ser separadas em duas categorias de acordo com o efeito causado no elemento que está recebendo sinal: sinapses excitatórias e sinapses inibitórias. Sinapses excitatórias aumentam o potencial da célula levando a um valor mais próximo do limiar, enquanto que as inibitórias diminuem o potencial contribuindo para o repouso da célula (CARDOSO, 2001a).

2.3 O Neurônio Artificial

Para que fosse possível mimetizar certos comportamentos inteligentes do ser humano, era necessário que um modelo matemático de neurônio fosse criado. Em 1943 Warren McCulloch e Walter Pitts foram responsáveis por criar o primeiro modelo de neurônio artificial (HAYKIN, 2001). O modelo desenvolvido por eles possui um conjunto de sinais de entrada, um conjunto de pesos associado a cada um deles, um somador, uma função de ativação e por fim um sinal de saída.

Os sinais de entrada são denotados sequencialmente de x_1 até x_n, da mesma maneira são os pesos indo de w_1 até w_n, onde cada sinal x está associado com um peso w. O somador Σ é a entidade encarregada de multiplicar cada par (x_i, w_i) e somar os resultados, e após, transmitir este valor para a função de ativação $f(s)$. A função então definirá qual será o valor de saída y,

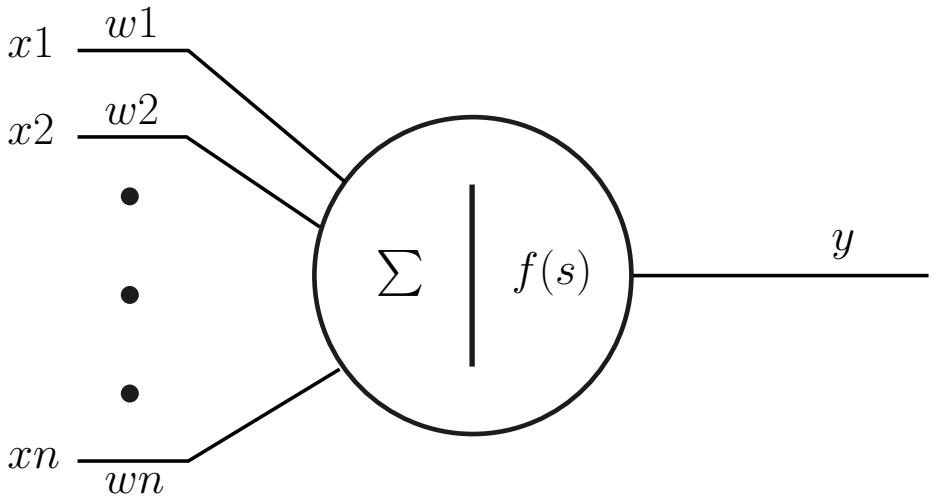


Figura 2.2: Estrutura de um neurônio artificial

que pode ser somado a um *bias*. *Bias* é um valor que serve para aumentar o grau de liberdade da função de ativação. A organização de um neurônio artificial pode ser vista na Figura 2.2.

2.3.1 Funções de Ativação

Existem vários tipos de funções de ativação. Dependendo do algoritmo de treinamento da rede neural, as funções passíveis de serem utilizadas nos neurônios dependem do algoritmo de treinamento, visto que treinamento baseado em gradiente exige que a função seja diferenciável. As funções mais comuns são as seguintes:

- Função ReLU[2]. Definida por:

$$f(s) = \begin{cases} 0 & \text{se } s < 0 \\ s & \text{se } s \geqslant 0 \end{cases}$$

- Função degrau. Definida por:

$$f(s) = \begin{cases} 0 & \text{se } s < 0 \\ 1 & \text{se } s \geqslant 0 \end{cases}$$

- Função sigmoide. Definida por:

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

  onde a é o coeficiente de inclinação da função.

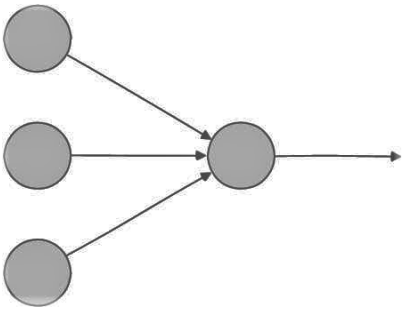
[2] *Rectified Linear Unit* - Unidade Linear Retificada

2.4 Arquiteturas de RNA

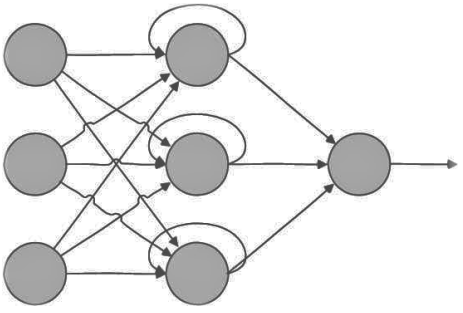
A arquitetura diz respeito a organização da rede neural, sobre quantas camadas à compõem, quantos neurônios existem em cada camada e sobre como eles estão conectados (SILVA, 2010). Arquiteturas com estes parâmetros definidos também são denotadas como modelos de RNA.

Redes neurais são constituídas de uma camada de entrada e uma de saída, podendo possuir camadas ocultas entre as de entrada e saída. A camada de entrada não é levada em consideração quando se conta o número de camadas da rede, visto que ela não processa dados. As camadas, com exceção da de entrada, podem possuir realimentação de sinal, onde sinais de entrada provêm da própria camada ou de camadas posteriores (FAUSETT, 1994).

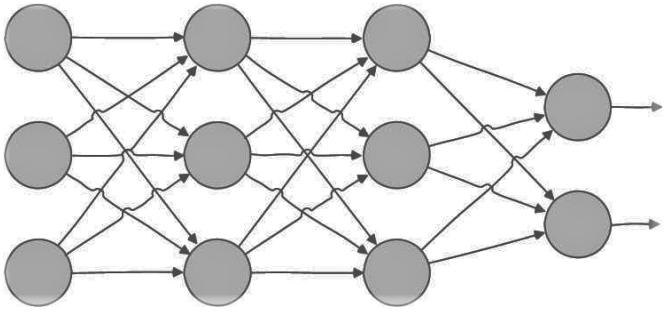
RNAs podem ser classificadas segundo o número de camadas que a constituem e pela existência de realimentação. Quando a rede é constituída de múltiplas camadas ela é dita multicamada, caso contrário é definida como rede de camada única. Redes com realimentação são chamadas recorrentes, sem realimentação são ditas *feedforward* (HAYKIN, 2001). A Figura 2.3 mostra como estas arquiteturas podem ser dispostas.



(a) RNA de camada única



(b) RNA multicamada com realimentação



(c) RNA multicamada

Figura 2.3: Arquiteturas básicas de Redes Neurais Artificiais

2.5 Treinamento de RNAs

O treinamento é a fase em que a rede tentará extrair conhecimento de uma coleção de dados de entrada. Existem três categorias de treinamento: supervisionado, não supervisionado e por reforço. Cada treinamento tem suas particularidades a respeito dos dados e categorias de problemas que podem ser solucionados por cada abordagem (HAYKIN, 2001).

2.5.1 Treinamento Supervisionado

Este tipo de treinamento acontece quando se deseja fazer predições sobre dados ou classificá-los. O processo ensinará a rede sobre os padrões que ocorrem nos dados de entrada e levam a uma determinada saída. O treinamento é dito supervisionado porque os dados de saída são conhecidos. O processo usa esses valores para ensinar a rede baseando-se nos erros cometidos por ela (SILVA, 2010).

O treinamento exige que os dados estejam separados em duas categorias: entrada e saída. Após o treinamento a rede poderá estimar uma saída para entradas desconhecidas. Redes que se encaixam nesta categoria são utilizadas comumente para problemas de classificação ou estimação de valores, como separar animais por espécies ou predizer preços da bolsa de valores de uma empresa.

2.5.2 Treinamento não Supervisionado

Ocorre quando não há conhecimento prévio sobre a classificação dos dados. Pode ser usado para estruturar os dados, extrair relações entre eles, ou reduzir a dimensionalidade dos dados mantendo as informações mais relevantes. Nesta classe de treinamento os dados são particionados em grupos baseado na similaridade entre eles, e procura-se encontrar o particionamento ótimo, onde as não-similaridades são mínimas (SILVA, 2010).

2.5.3 Treinamento por Reforço

Este tipo de treinamento é utilizado em controle de agentes. Um agente é uma entidade que interage com um ambiente e recebe recompensas por ações tomadas (FERBER, 1999). Ações boas tem recompensas igualmente boas, ações ruins tem recompensas ruins. Este treinamento controlará as ações do agente com o objetivo de otimizar a recompensa recebida (SILVA, 2010).

2.6 RNAs Profundas

Deep Learning é um campo derivado do Aprendizado de Máquina, que por sua vez deriva da IA, que definiu o termo RNA profunda. O campo classifica RNAs como profundas ou rasas segundo a quantia de camadas ocultas em sua arquitetura. O número necessário de camadas ocultas para considerar uma rede profunda não é fixo, o valor pode variar dependendo da complexidade do problema que a rede busca resolver (GOODFELLOW; BENGIO; COURVILLE, 2016).

Para a área, a saída de uma camada oculta pode ser vista como uma nova representação dos dados de entrada. A perspectiva da *Deep Learning* em relação à profundidade das redes é que ela permite que a rede aprenda programas de múltiplos passos, e cada camada é vista como um estado da memória do computador após a execução de um conjunto de instruções (GOODFELLOW; BENGIO; COURVILLE, 2016).

Camadas ocultas permitem uma maior acurácia em problemas mais complexos, fato comprovado pelo teorema da aproximação universal apresentado por Hornik (1991), por esta razão, redes profundas buscam aumentar o número de camadas ocultas a fim de atingir o máximo de eficiência. Entretanto existem variáveis que inviabilizam o aumento indiscriminado de camadas. Modelos mais profundos aprendem mais lentamente que modelos mais rasos, exigem uma quantia maior de dados de treinamento, são mais custosos computacionalmente e suscetíveis à *overfitting* (GOODFELLOW; BENGIO; COURVILLE, 2016).

2.7 RNAs *Feedforward*

Utilizadas comumente como classificadoras, esta categoria de RNA consegue extrair características dos dados de entrada e associá-las à uma categoria. Este capítulo se refere as redes *feedforward* para classificação, com ou sem camadas ocultas, e com treinamento supervisionado. Serão abordadas especificidades sobre a arquitetura e dados utilizados por esta classe de RNAs.

2.7.1 Dados

Os dados apresentados à rede devem ser um par x, y onde os valores contidos no conjunto x caracterizam uma instância pertencente às classes descritas em y. Durante o período de treinamento da rede deve-se apresentar dados diversos de todas as classes distintas de y, para que ela possa extrair características que descrevem cada classe (SILVA, 2010).

A rede treinada classificará dados novos, ou dados antigos, em uma das categorias contidas em y. Caso seja necessário incorporar uma nova classe, a rede deve ser alterada de modo à suportar a inclusão, comumente resulta em adicionar um neurônio na camada de saída, e então, a rede resultante é treinada novamente com dados novos e antigos. Nesta situação é possível reaproveitar parcialmente, ou totalmente, os pesos antigos na nova rede neural, técnica conhecida como *transfer learning* (PAN; YANG et al., 2010).

2.7.2 Treinamento

O treinamento é a fase de aprendizado da rede, onde um algoritmo de treinamento é responsável por otimizar a acurácia do modelo de RNA. O módulo de treinamento é composto por dois itens: função de perda e otimizador. A função de perda, comumente denotada função de custo na IA, calcula a magnitude do erro cometido por uma predição da rede. O otimizador, como seu nome indica, otimiza a performance da rede, isto implica em recalcular os pesos sinápticos dos neurônios a fim de minimizar o valor da função de perda (GOODFELLOW; BENGIO; COURVILLE, 2016).

O processo realizado pelo módulo de treinamento a fim de ensinar a rede neural descrito pelos seguintes passos de acordo com (HAYKIN, 2001):

- Dado um conjunto de entrada x predizer $\hat{y}$.
- Calcular o erro ϵ cometido pela predição $\hat{y}$ com o valor real y.
- Recalcular os pesos da rede neural proporcionalmente ao erro ϵ.

Este processo pode ser realizado de dois modos de acordo com o momento em que o otimizador recalculará os pesos. O processo pode ocorrer iterativamente a cada entrada ou por grupos de entradas, ou *batches* (SILVA, 2010).

O treinamento por *batches* separará os dados em vários grupos de n itens, caso o número total de dados não seja divisível por n um grupo ficará com menos itens ou será preenchido com itens de outros grupos. O módulo de treinamento acumulará os n erros ϵ de cada predição $\hat{y}$ realizada com conjuntos x pertencentes ao grupo e, com o valor de erro final ϵ_f, recalculará os pesos da rede neural.

Função de Perda

Esta função provê para o otimizador uma medida de erro ϵ, considerando os valores de $\hat{y}$ e y, que deve ser minimizada. Existem vários métodos de calcular o valor de ϵ e cada um deles influencia o resultado do treinamento (HAYKIN, 2001). Alguns métodos são:

- Erro Quadrático Médio
$$EQM = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$
- Erro Absoluto Médio
$$EMA = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$
- Logaritmo do cosseno hiperbólico
$$LCH = \frac{1}{n}\sum_{i=1}^{n}\log(\arccos(y_i - \hat{y}_i))$$
- *Binary Cross Entropy*, entropia cruzada binária
$$BCE = \frac{1}{n}\sum_{i=1}^{n} -(y\log(\hat{y}_i) + (1-y)\log(1-\hat{y}_i))$$

Otimizadores

Existem vários otimizadores propostos, cada um com seus pontos fortes e fracos. A eficiência de um otimizador é afetada por características dos dados de treinamento, assim como a função de perda escolhida. A fim de encontrar uma combinação de otimizador e função de perda que tragam resultados satisfatórios, é necessário realizar testes com diferentes combinações (GOODFELLOW; BENGIO; COURVILLE, 2016). Alguns métodos são:

- Adadelta (ZEILER, 2012)

- Adam (KINGMA; BA, 2014)
- Adagrad (DUCHI; HAZAN; SINGER, 2011)
- AdaMax (KINGMA; BA, 2014)
- AMSGrad (REDDI; KALE; KUMAR, 2018)
- Nadam (DOZAT, 2016)
- *Nesterov accelerated gradient* (NESTEROV, 1983)
- RMSProp (HINTON; SRIVASTAVA; SWERSKY, 2014)
- *Stochastic Gradient Descent* (ROBBINS; MONRO, 1985)

2.7.3 Redes Neurais Convolucionais

A configuração das redes neurais convolucionais (CNN - *Convolutional Neural Network*) foi inspirada no cortex visual, área do cérebro responsável pela visão (HEARTY, 2016). CNNs são um tipo especial de RNAs de multicamadas, apresentando diferenças em sua arquitetura. Estas redes foram propostas com o objetivo reconhecer padrões visuais em imagens com mínimo pré-processamento. Apresentam grande habilidade em reconhecer padrões mesmo que as imagens sofram distorções ou transformação geométrica simples (LECUN, 2010).

Esta categoria de RNAs recebe esta nomenclatura devido ao fato de existirem camadas em sua arquitetura que aplicam matrizes convolucionais, ou filtros, na imagem de entrada a fim de extrair características visuais, devido a este fato elas são denominadas camadas convolucionais, o processo de convolução é explicado no Apêndice B. CNNs podem também incluir em sua arquitetura, além de camadas utilizadas em uma RNA comum, camadas de *pooling* e normalização. A Figura 2.4 mostra uma arquitetura básica de uma CNN.

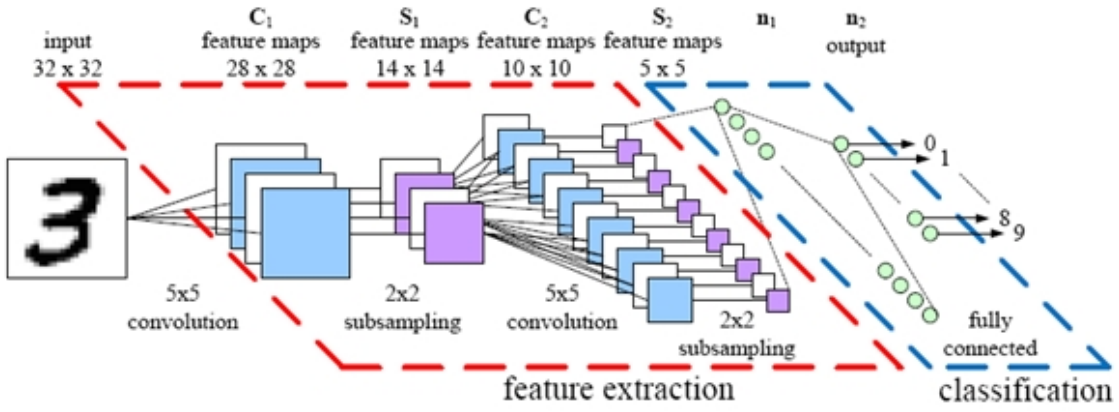


Figura 2.4: Estrutura de uma CNN (WARD et al., 2011)

Camada Convolucional

As camadas convolucionais desempenham a função de extrair características e encontrar padrões nas imagens de entrada. Cada camada possiu n filtros, que são os neurônios da camada, os quais são utilizados para convolucionar a imagem de entrada gerando a mesma quantia de imagens resultantes, estas são denotadas *feature maps* - mapa de características (KALUZA, 2016). É possível definir parâmetros utilizados pela convolução como *padding* e *stride*.

Os pesos de cada neurônio são os valores do filtro convolucional. Os neurônios de uma camada convolucional, assim como os convencionais, também possuem um valor de *bias*. Os pesos de um neurônio são compartilhados de acordo com o processo de convolução, esta partilha se deve ao fato que um mesmo peso multiplica mais de um valor de entrada. A Figura 2.5 mostra a relação de partilha para uma imagem de entrada 3x3 convolucionada por um filtro 2x2, resultando em uma imagem também 2x2. As conexões entre os valores de entrada e saída são os pesos utilizados pelas operações de convolução.

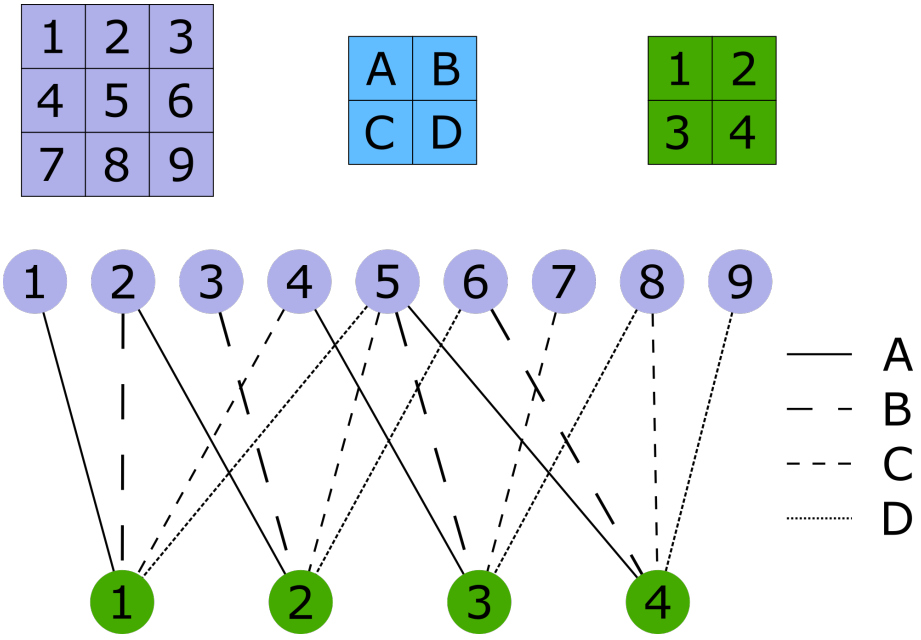


Figura 2.5: Representação dos pesos compartilhados de uma *feature map*

Camada de *Pooling*

Também chamada de camada de *subsampling*, sua função é reduzir o tamanho da imagem de entrada (HEARTY, 2016). Seu funcionamento é muito parecido com o processo de convolução, é definida uma região MxN que será transladada pela imagem de acordo com os valores do *stride*. O processo de *pooling* selecionará um valor para representar a região atual. Algumas estratégias são: selecionar o valor máximo da região atual, calcular a média dos valores ou calcular a mediana. A Figura 2.6 mostra o processo do *pooling* em uma imagem 4x4 com região e *stride* 2x2 selecionando o valor máximo, conhecido comumente como *maxpooling*.

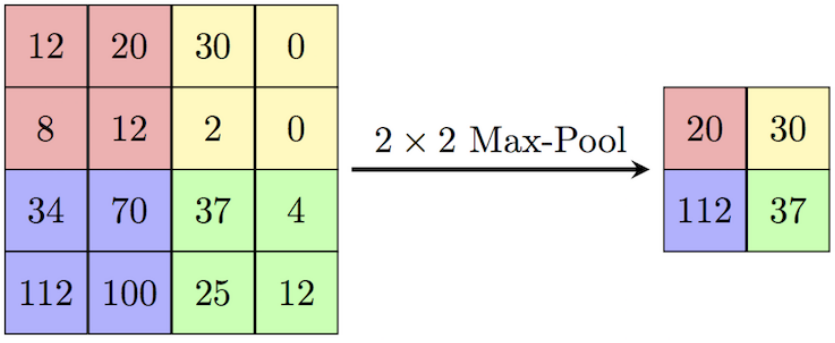


Figura 2.6: Exemplo de *Max Pooling* com uma região e *stride* de 2x2 em uma imagem 4x4[3]

[3]Imagem retirada de: https://computersciencewiki.org/index.php/Max-pooling_/_Pooling

Camada de Normalização

Esta camada normaliza o sinal de entrada. O objetivo de uma camada de normalização é evitar o deslocamento interno da covariável como descrito por Ioffe e Szegedy (2015). O deslocamento interno da covariável é a mudança da distribuição das ativações de uma RNA por causa de mudança nos parâmetros de entrada. A normalização permite que o modelo alcance maior acurácia em menos iterações de treinamento.

Treinamento

As CNNs utilizam uma versão adaptada do algoritmo de *backpropagation* para realizar o treinamento. Dentre as 3 camadas específicas de uma CNN, as de *pooling* e normalização não passam pelo processo de treinamento, somente a camada convolucional possui a habilidade de aprender. A modificação feita no algoritmo se deve ao fato de existirem pesos compartilhados, o que resulta em um cálculo adaptado do gradiente de erro. O gradiente de erro de um valor de entrada será a soma dos gradientes partilhados pelos pesos que multiplicam esta entrada (Theano Development Team, 2013). Então o gradiente resultante é utilizado por um algoritmo de treinamento convencional.

Capítulo 3

Recursos Utilizados

Será apresentado neste capítulo uma breve descrição sobre as bibliotecas utilizadas, suas respectivas versões, assim como outras informações pertinentes. Também será descrita a máquina utilizada para a realização de testes, especificando seus recursos de *hardware* e *software*.

3.1 TensorFlow

TensorFlow é uma biblioteca de cálculo numérico de alta performance desenvolvida pela Google (ABADI et al., 2015). Sua primeira versão foi lançada em novembro de 2015 (TensorFlow, 2018f) como sucessora da antiga DistBelief (OREMUS, 2015). Seu grande suporte para desenvolvimento de sistemas de aprendizado de máquina e *deep learning*, aliado com a sua flexibilidade de executar em máquinas heterogêneas, ou distribuídas, atraiu interesse de grandes empresas como Intel®, AMD e Coca-Cola® (ABADI et al., 2015), que passaram a utilizar a biblioteca em seus negócios.

A biblioteca é passível de ser usada em uma gama de sistemas operacionais, porém oficialmente provê suporte somente para os seguintes sistemas operacionais (TensorFlow, 2018b):

- macOS - versão 10.12.6 ou superior
- Ubuntu - versão 16.04 ou superior
- Windows - versão 7 ou superior
- Raspbian - versão 9.0 ou superior
- Android - versão não especificada
- iOS - versão não especificada

A biblioteca foi desenvolvida nas seguintes linguagens de programação: Python, C++, Java, Javascript, Go e Swift. Destas linguagens somente Python cumpre as promessas de estabilidade da API (*Application Programming Interface* - Interface de Programação de Aplicações). Além das linguagens utilizadas pela equipe da biblioteca, a API pode ser usada em outras linguagens por meio de *bindings*, vínculos, desenvolvidos pela comunidade (TensorFlow, 2018a).

TensorFlow provê suporte para o desenvolvimento de redes neurais convolucionais, *feed-forward* e recorrentes, disponibiliza também modelos pré-treinados para a elaboração ágil de soluções. A Google também ofereçe a ferramenta TensorBoard para visualização dos modelos e treinamento das RNAs (TensorFlow, 2018g), ademais, é possível visualizar informações sobre consumo de memória e tempo de computação de tensors e operações.

É possível para o usuário executar aplicações em CPUs (*Central Processing Unit* - Unidade Central de Processamento), GPUs (*Graphics Processing Unit* - Unidade de Processamento Gráfico), TPUs (*Tensor Processing Unit* - Unidade de Processamento de Tensor), *clusters* de servidores, celulares e dispositivos embarcados (ABADI et al., 2015).

3.1.1 Modo de Operação

Um dos modos de funcionamento do TensorFlow consiste na criação de grafos de computação, este modo também é conhecido como API de baixo nível, onde os vértices do grafo podem ser divididos em dois tipos: Operações ou Tensors – tensors são uma generalização de vetores, matrizes e estruturas de múltiplas dimensões. Internamente o TensorFlow representa tensors como um vetor n-dimensional. Operações são vértices que descrevem uma operação que consome e produz tensors (TensorFlow, 2018c). A Figura 3.1 mostra como pode ser construído o grafo de computação para a fórmula de Bhaskara, onde as raízes da função são $Divide$ e $Divide\_1$.

Em baixo nível não há abstração de componentes das RNAs como camadas e modelo, porém utilidades como funções de ativação e otimizadores existem. Camadas da rede devem ser definidas por matrizes e o modelo pelo fluxo de execução, devido a este fato o entendimento e manutenção de sistemas feitos com esta API não são triviais, porém são eficientes e os modelos de RNAs flexíveis.

Além do modo de baixo nível, existem duas APIs de alto nível onde o fluxo de execução

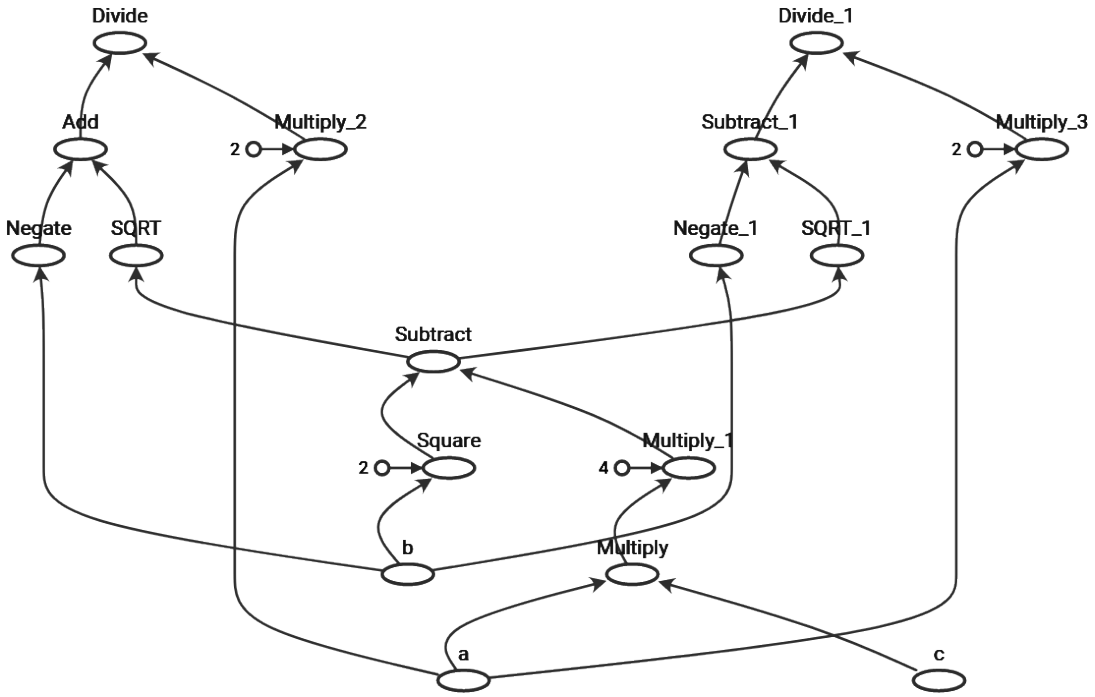


Figura 3.1: Exemplo de grafo de computação do TensorFlow

não é visível para o usuário: Keras e Estimators. Keras é uma biblioteca independente que foi incorporada ao TensorFlow em novembro de 2017 quando este lançou a versão 1.4.0, seu foco está na rápida prototipação de modelos (TensorFlow, 2018e). Estimators foi introduzida na biblioteca quando esta lançou a versão 1.1.0, esta API é usada para treinamento em grande escala e para ambientes de produção (TensorFlow, 2018d). Ambos modos discretizam componentes das RNAs para objetos, facilitando o entendimento e manutenção (TensorFlow, 2018h).

Dentre os três modos de operação Keras é o mais simplificado para construção de sistemas de RNAs. Ainda que o processo de modelagem de RNAs do Estimators não seja tão diferente do Keras – inclusive é possível importar modelos desenvolvidos no Keras, eliminando a necessidade da construção de modelos na API – Estimators exige que os dados de entrada sejam modelados coluna a coluna para um objeto nativo da biblioteca a fim de serem utilizados pelo modelo durante o treinamento ou produção.

3.2 MATLAB®

MATLAB® é uma ferramenta proprietária e uma linguagem de programação de alto nível assim como ambiente interativo para computação numérica, visualização e programação.

MATLAB® permite ao usuário analisar dados, desenvolver algoritmos e criar modelos e aplicações. A linguagem, ferramentas e funções matemáticas nativas permitem a exploração de múltiplas maneiras de resolver um problema mais rapidamente que planilhas ou linguagens de programação tradicionais, como C/C++ ou Java™(MathWorks, 2012). Sua história começou a mais de 60 anos, entretanto, sua primeira versão foi desenvolvida somente no final da década de 70 (MathWorks, 2004), MATLAB® hoje é um software renomado com mais de 3 milhões de usuários em mais de 180 países (MathWorks, 2018a).

As funcionalidades do MATLAB® são modularizadas em diversas *toolboxes*, cada uma delas provê utilidades para resolver problemas de áreas específicas como processamento de imagens, estatística e *Deep Learning*. A *toolbox* de *Deep Learning* oferece suporte para o planejamento e o desenvolvimento de modelos de redes neurais artificiais, nativamente possui facilidades para criar redes convolucionais, recorrentes, *feedforward* ou redes com treinamento não supervisionado. Para acelerar o treinamento em um grande conjunto de dados, os cálculos e dados da aplicação podem ser distribuídos em processadores *multi-core*, GPUs ou utilizar processamento de servidores *cloud* e *clusters* (MathWorks, 2018b).

A *toolbox* de *deep learning* apresenta uma abordagem em alto nível para modelagem de redes neurais, componentes da rede são abstraídos para objetos facilitando o entendimento e manutenção – assemelha-se fortemente com a API do Keras. Não é visível para o usuário o funcionamento interno da *toolbox*. É possível realizar o treinamento em GPUs, CPUs, *cloud* ou *clusters*. É possível importar modelos desenvolvidos em outras bibliotecas como Caffe (JIA et al., 2014) e Keras, são disponibilizados também modelos pré-treinados para desenvolver soluções mais rapidamente.

São dispostas também rotinas para visualizar informações sobre a arquitetura de uma rede neural, progresso do treinamento e ativações de uma camada. O MATLAB® pode ser utilizado nos seguintes sistemas operacionais:

- Windows 7 *Service Pack* 1 e 10
- Windows Server 2012, 2012 R2 e Server 2016
- macOS - a partir da versão 10.12
- Ubuntu 14.04 LTS, 16.04 LTS e 18.04 LTS

- Debian 8 e 9
- Red Hat Enterprise Linux 6 (a partir da versão 6.7) e 7 (a partir da versão 7.3)
- SUSE Linux Entreprise Desktop 12 (a partir do *Service Pack* 2) e 15
- SUSE Linux Entreprise Server 12 (a partir do *Service Pack* 2) e 15

Além destes sistemas é possível executar aplicativos desenvolvidos no MATLAB® em celulares e dispositivos embarcados, para isto deve-se converter o código escrito para C/C++ por meio do MATLAB Coder™.

3.3 Recursos Utilizados

A Tabela 3.1 mostra a relação de *software* e versão utilizada. A execução dos testes foi realizado em uma máquina com as seguintes configurações de hardware e sistema operacional:

- Sistema Operacional: Windows 10 x64 compilação 17134
- Processador: Intel Core i3 7100 3.9 GHz, 2 núcleos físicos e 4 lógicos
- Placa gráfica: Intel HD Graphics 630
- Memória RAM: 8 Gb

Software	Versão
Python	3.6.6
TensorFlow	1.10.0[4]
MATLAB®	R2017b

Tabela 3.1: Relação de *software* e versão

3.4 Desenvolvimento

Neste capítulo será descrito o processo usado durante o desenvolvimento dos modelos desenvolvidos para os testes. Estes também serão aqui descritos e explicados.

[4]A versão utilizada foi compilada para suportar instruções AVX2 (Intel® *Advanced Vector Extensions* 2 - Extensões Avançadas de Vetores da Intel® 2), disponibilizada em: https://github.com/fo40225/tensorflow-windows-wheel/blob/master/1.10.0/py36/CPU/avx2/tensorflow-1.10.0-cp36-cp36m-win_amd64.whl

3.4.1 O Processo de Desenvolvimento das RNAs

Este é o processo usado como referência durante o desenvolvimento dos modelos dos testes. Nem todas as fases foram utilizadas devido a natureza dos dados utilizados e simplicidade do modelo.

Cada fase abordará o assunto pertinente a ela de maneira simples e breve, explicações mais detalhadas serão apresentadas nos capítulos posteriores quando forem necessárias.

Aquisição de Dados

Redes neurais foram criadas com o intuito de extrair conhecimento de uma coleção de dados. Tendo isto em vista, para que se possa treinar uma rede é vital a existência de um conjunto de dados. Quanto maior for esse conjunto, melhor será, visto que ajuda a resolver problemas como *overfitting*[5] e melhora o poder de generalização da rede.

Seleção de Dados

Após a aquisição dos dados, é recomendado selecionar somente os valores que são relevantes para o aprendizado da rede, tendo em vista que a complexidade de uma rede aumenta com o número de variáveis e dados não relacionados podem atrapalhar a aprendizagem. Por exemplo: para um sistema de empréstimos é mais relevante que dados como idade do indivíduos e seus respectivos salários sejam fornecidos ao invés de seus nomes ou etnias.

Limpeza de Dados

Com os dados selecionados, este é o momento de tratar problemas que os dados possam apresentar. Alguns problemas que podem ocorrer são:

- Dados em branco
- Dados Inconsistentes
- Dados Errôneos

Os problemas desta fase podem ser resolvidos seguindo uma das duas abordagens:

[5]Termo usado para descrever um modelo estatístico que se ajustou excessivamente ao conjunto de dados observados e, ao mesmo tempo, apresenta baixa eficiência para dados não observados.

– Pode-se alterar os dados que apresentam problema por um valor coerente, porém deve-se levar em conta que alterações podem interferir nos resultados da rede.

– Pode-se deletar o conjunto de dados que estão relacionados com a entrada que apresenta problema, fazendo com que esta alternativa possa vir a reduzir significantemente o número de amostras.

Codificação dos Dados Categóricos

Dados categóricos como marca, país ou espécie devem ser codificados para valores numéricos a fim que possam ser usados pela rede, isto inclui dados de saída. Existem vários codificadores como o ordinal, binário e *one-hot*. O codificador mais comumente utilizado em redes neurais é o *one-hot*, ele codifica as n categorias, ou classes, em n variáveis, cada variável indica uma classe, o valor 1 indica a pertinência àquela classe e o valor 0 indica a não pertinência. Outra maneira de se ver esta codificação é como uma matriz identidade onde as colunas indicam as classes e as linhas indicam a pertinência.

Definição da Saída da Rede Neural

Antes de definir a arquitetura da RNA é recomendável definir a organização da saída, tendo em vista que ela pode ser usada como referência na fase de definição da arquitetura. A organização da saída diz respeito sobre o número de saídas e o que elas codificam.

Definição da Arquitetura da Rede Neural

Nesta fase já é conhecido o que a rede recebe como entrada e o que ela deve retornar como saída, resta definir a arquitetura da rede neural que tornará isto possível. A arquitetura abordará em linhas básicas os seguintes itens:

- Número de camadas ocultas
- Número de neurônios em cada camada
- Conexões dos neurônios
- Função de ativação
- *Bias*

Seleção de Dados para Treinamento

Para redes que utilizam o treinamento supervisionado é interessante separar uma fração dos dados para avaliação de seu desempenho. Os dados são separados em dois grupos: conjunto de treinamento e conjunto de teste. O conjunto de treinamento é usado durante o treinamento e pode ser usado para avaliação. O conjunto de teste é usado somente para avaliação e não para treinamento. Comparando os resultados das avaliações dos dois conjuntos pode-se obter outras informações sobre a rede como a presença e nível de *overfitting*.

Treinamento

Neste estágio a RNA finalmente irá começar a aprender, a rede é alimentada com os dados de treinamento e tentará otimizar sua performance para estas entradas. Se a eficiência da rede não for satisfatória, a causa pode ser um destes problemas:

- A quantidade de dados foi insuficiente para o aprendizado da rede.
- A arquitetura da rede é muito simples ou muito complexa.
- O algoritmo de treinamento não tem uma compatibilidade muito boa com os dados.

Devido ao fato de não existir uma fórmula para descobrir os parâmetros ideais de uma rede neural, é necessário experimentar vários parâmetros empiricamente a fim de encontrar os valores que produzam os resultados mais satisfatórios.

3.5 Testes

Foram desenvolvidas duas RNAs *feedforward* para teste com o intuito de comparar o processo de desenvolvimento e desempenho das duas bibliotecas, MATLAB® e TensorFlow. Ambas as redes são classificadores multicamadas, ademais, uma é convolucional. Primeiramente foi idealizado um modelo de RNA para resolver o problema, então este modelo foi implementado em cada biblioteca medindo o tempo de treinamento e memória utilizada para armazenar o modelo. Cada exemplo será descrito em detalhadamente em suas respectivas sessões.

3.5.1 Iris

Introduzida por Fisher (1936), Iris é uma coleção de dados sobre flores do gênero iris. A coleção conta com 150 observações de flores, onde foram retiradas 4 medidas de cada uma: comprimento da sépala, largura da sépala, comprimento da pétala e largura da pétala – todas medidas em centímetros. As observações são separadas de maneira uniforme em 3 espécies: iris setosa, iris versicolor e iris virginica. A Tabela 3.2 mostra alguns valores retirados da coleção.

Sépala		Pétala		Espécie
comprimento	largura	comprimento	largura	
5.1	3.5	1.4	0.2	Iris-setosa
4.9	3.0	1.4	0.2	Iris-setosa
7.0	3.2	4.7	1.4	Iris-versicolor
6.4	3.2	4.5	1.5	Iris-versicolor
6.3	3.3	6.0	2.5	Iris-virginica
5.8	2.7	5.1	1.9	Iris-virginica

Tabela 3.2: Amostra de valores da coleção de dados Iris

A coleção de dados foi retirada de UC Irvine Machine Learning (2007). Os dados não apresentam valores em branco, uma espécie é separável linearmente das outras duas, contudo, não é possível separar linearmente as duas espécies restantes.

Neste teste procura-se construir uma RNA *feedforward* comum que aprenda a classificar corretamente flores das 3 espécies observadas, baseando-se nas medidas de suas sépalas e pétalas.

Tratamento de Dados

As medidas de pétalas e sépalas não passaram por nenhum tipo de tratamento, estes valores foram utilizados como constam na coleção. As espécies foram submetidas a um processo de codificação, visto que seus valores originais são textuais. A codificação utilizada foi a *one-hot encoding*, resultando em 3 valores. A Tabela 3.3 mostra alguns dados que passaram pela codificação.

Sépala		Pétala		Espécie		
comprimento	largura	comprimento	largura			
5.1	3.5	1.4	0.2	1	0	0
4.9	3.0	1.4	0.2	1	0	0
7.0	3.2	4.7	1.4	0	1	0
6.4	3.2	4.5	1.5	0	1	0
6.3	3.3	6.0	2.5	0	0	1
5.8	2.7	5.1	1.9	0	0	1

Tabela 3.3: Amostra de valores que sofreram codificação

Modelo de RNA Idealizado

Os dados nesta fase apresentam 4 variáveis como entrada e 3 como saída. A rede portanto apresentará esta mesma quantia em neurônios de entrada e saída. Foi testado empiricamente a melhor combinação de camadas ocultas e neurônios que melhoram a acurácia e não geram *overfitting*, resultando em uma camada oculta com 16 neurônios. A Figura 3.2 mostra como é organizado o modelo, somente as conexões com o primeiro neurônio da camada anterior estão presentes por motivos de simplicidade.

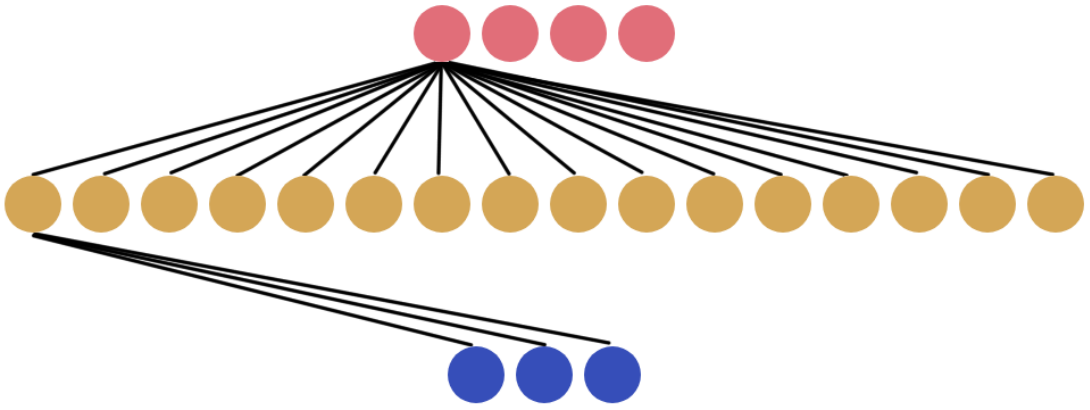


Figura 3.2: Modelo da RNA idealizada. Os neurônios superiores pertencem à camada de entrada, os inferiores à de saída e os do meio pertencem à camada oculta

A função de ativação escolhida para neurônios da camada oculta é a ReLU (*Rectified Linear Unit* – Unidade Linear Retificada) por apresentar melhores resultados. A função utilizada na camada de saída foi a softmax a fim de transformar os valores de saída em probabilidades, neste caso, a probabilidade de pertinência a certa espécie.

As camadas possuem conexão completa – ou densa – com as camadas anteriores, isto é, cada neurônio da camada oculta recebe os 4 valores de entrada e os neurônios da saída recebem

os 16 valores da camada oculta. Os pesos das conexões de todos os neurônios são inicializados com uma distribuição randômica uniforme com desvio de 0,05 e média 0. Os valores dos bias são inicializados em 0.

Treinamento

O algoritmo de treinamento utilizado foi o SGD (*Stochastic Gradient Descent* – Gradiente Descendente Estocástico), com taxa de aprendizado de 0,0005 e tamanho de *batch* 20. Foram realizadas 250 épocas de treinamento, onde os dados foram embaralhados em cada uma delas.

Código

Os Algoritmos A.1 e A.3 contém na íntegra o código desenvolvido, aqui será feita uma comparação sobre a construção do modelo idealizado nas bibliotecas. Primeiramente será comparada a codificação das classes das flores. O TensorFlow possui uma função *to_categorical* que codifica sequências numéricas para *one-hot*, o laço de repetição *for* do Algoritmo 3.1 transforma o campo texto das espécies em uma sequência numérica de 0 a 2 a fim de utilizá-la. MATLAB® oferece a função *categorical* que faz a mesma codificação porém não é limitada somente a dados numéricos, o Algoritmo 3.2 mostra o uso da função utilizando o campo em sua forma textual.

Algoritmo 3.1: Codificação *One-Hot* em Python

```
for index, clss in enumerate(np.unique(y)):
    y[y == clss] = index

y = to_categorical(y)
```

Algoritmo 3.2: Codificação *One-Hot* em MATLAB®

```
y = categorical(table2array(dataset(:, 5))');
```

Com os dados preparados, agora é construído o modelo da rede idealizada. Em TensorFlow é criado um modelo que dispões as camadas de maneira sequencial e nele é adicionado as camadas oculta e de saída, a primeira camada adicionada à rede deve definir as dimensões da entrada, o processo é mostrado pelo Algoritmo 3.3.

MATLAB® não possui um objeto que represente o modelo, por isto as camadas são adicionadas em um vetor, como é exposto no Algoritmo 3.4. Devido a simplicidade de configuração das camadas em MATLAB®, as funções de ativação são discretizadas como uma camada. Os pesos e *bias* das camadas são inicializados em 0 e outros métodos de inicialização devem ser implementados pelo usuário e, diferentemente do TensorFlow, a última camada deve informar se o propósito da rede é classificação ou regressão.

Algoritmo 3.3: Construção do Modelo em Python

```
model = Sequential()
model.add(Dense(16, activation='relu', kernel_initializer='
    random_normal', input_shape=(4,)))
model.add(Dense(3, activation='softmax', kernel_initializer='
    random_normal'))
```

Algoritmo 3.4: Construção do Modelo em MATLAB®

```
layers = [...
    sequenceInputLayer(4)
    fullyConnectedLayer(16)
    reluLayer
    fullyConnectedLayer(3)
    softmaxLayer
    classificationLayer];
layers(2).Weights = randn([16 4]) * 0.05;
layers(4).Weights = randn([3 16]) * 0.05;
```

Os Algoritmos 3.5 e 3.6 realizam o treinamento descrito, existem duas diferenças entre eles. TensorFlow necessita da especificação da função de perda que mede o erro cometido pela rede e uma especificação de métrica de desempenho. O MATLAB® constrói um objeto de opções de treinamento devido a falta de um objeto do modelo da rede.

Algoritmo 3.5: Treinamento em Python

```
model.compile(SGD(0.0005), loss='categorical_crossentropy',
    metrics=['accuracy'])
model.fit(x, y, epochs=250, batch_size=20, verbose=0)
```

Algoritmo 3.6: Treinamento em MATLAB®

```
opt = trainingOptions('sgdm',...
    'Verbose', false,...
    'Momentum', 0,...
    'MaxEpochs', 250,...
    'MiniBatchSize', 20,...
    'InitialLearnRate', 0.0005,...
    'ExecutionEnvironment', 'cpu',...
    'Shuffle', 'every-epoch');
net = trainNetwork(x, y, layers, opt);
```

3.5.2 MNIST

MNIST é um banco de dados de dígitos manuscritos o qual é um subconjunto retirado do banco de dados disponibilizado pelo NIST (*National Institute of Standards and Technology*). Os dígitos passaram por normalização de tamanho e foram centralizadas na imagem (LECUN..., 2013). Todas as imagens são em escala de cinza e possuem o tamanho de 28x28 – 28 pixels de largura e altura. Os dados estão separados em 60.000 imagens para treino e 10.000 para teste, o número de amostras por dígito são expostas na Tabela 3.4 e uma amostra das imagens é apresentada na Figura 3.3. Cada imagem tem associada à ela um valor, que é o dígito representado pela imagem.

O objetivo deste exemplo é comparar a construção de uma CNN nas duas bibliotecas, assim como criar um modelo que efetivamente classifique corretamente os dígitos manuscritos.

	0	1	2	3	4	5	6	7	8	9
Treino	5923	6742	5958	6131	5842	5421	5918	6265	5851	5949
Teste	980	1135	1032	1010	982	892	958	1028	974	1009
Total	6903	7877	6990	7141	6824	6313	6876	7293	6825	6958

Tabela 3.4: Número de amostras de cada dígito nos conjuntos de Treino e Teste

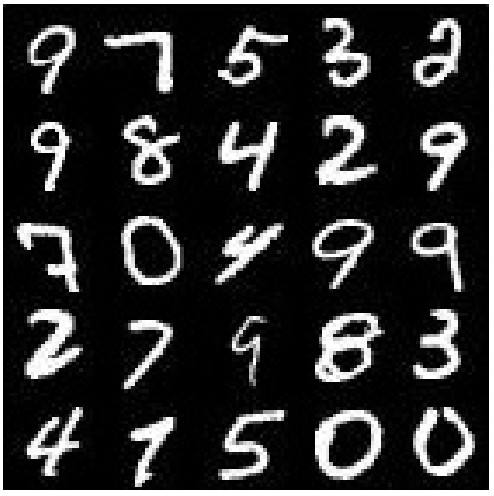


Figura 3.3: Amostra de imagens do retirada do MNIST

Tratamento de Dados

Os dados já passaram por uma regularização realizada pelo autor do banco de dados, consequentemente as imagens não requerem nenhum processamento a fim de serem utilizadas. Todavia, os pixels das imagens foram normalizados para beneficiar-se das mesmas propriedades exploradas por Ioffe e Szegedy (2015).

Os valores associados às imagens passarão pela codificação *one hot*, para que a rede retorne as probabilidades da imagem de entrada ser classificada como cada dígito.

Modelo de CNN Idealizado

A entrada da rede será uma imagem 28x28x1 – 28 pixels de largura e altura e um canal de cor – e a saída será 10 valores que são a probabilidade associada a cada dígito. O modelo constará com 2 camadas convolucionais, 2 camadas de normalização, 1 camada de *max pooling*, 1 camada oculta e as camadas de entrada e saída, todas elas sendo densamente conectadas. As camadas, com exceção das camadas de normalização que estão após as convolucionais, estão dispostas como mostrado na Figura 3.4.

Os *bias* de todas as camadas e os filtros convolucionais serão inicializados em zero. As camadas oculta e de saída terão a inicialização dos pesos por distribuição randômica uniforme. A Tabela 3.5 mostra como serão definidas e dispostas as camadas do modelo.

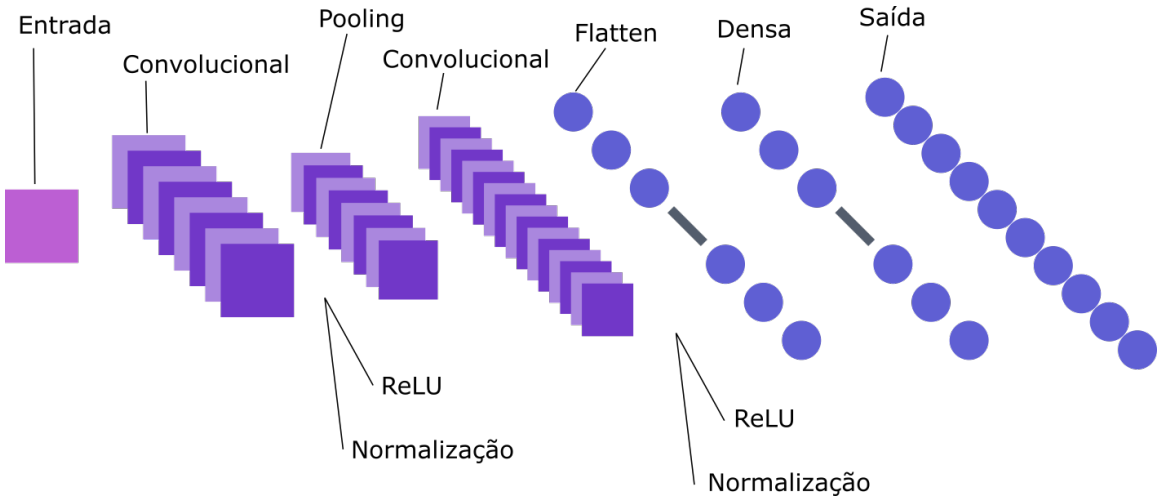


Figura 3.4: Modelo da CNN desenvolvida

Camada	Tipo	Descrição
1	Entrada de Imagem	Define que as imagens de entrada tem dimensões 28x28x1
2	Convolucional 2D	Camada convolucional com 8 *features* e um filtro de tamanho 3x3
3	Normalização	Aplica normalização nos valores
4	ReLU	Aplica ativação ReLU nos valores
5	*Max Pooling*	Aplica *pooling* na imagem com área 2x2
6	Convolucional 2D	Camada convolucional com 16 *features* e um filtro de tamanho 5x5
7	Normalização	Aplica normalização nos valores
8	ReLU	Aplica ativação ReLU nos valores
9	*Flatten*[6]	Transforma a imagem 3D em um vetor 1D
10	Oculta	Camada oculta com 64 neurônios, ativação ReLU
11	Saída	Camada de saída com 10 neurônios, ativação softmax

Tabela 3.5: Descrição das camadas da rede idealizada

Treinamento

O modelo treinará com as 60.000 imagens por 5 épocas e com tamanho do *batch* 500. O algoritmo de treinamento que será utilizado é o SGD com taxa de aprendizado de 0.01 e momento de 0.9.

[6] *Flatten* ou nivelamento, é uma camada que transforma a imagem 3D (largura x altura x canais) em um vetor 1d que será a entrada dos neurônios da próxima camada.

Código

Os Algoritmos A.2 e A.4 contém o código completo desenvolvido. Aqui será apresentada uma comparação sobre a construção do modelo nas duas bibliotecas com foco nas camadas específicas de uma CNN.

TensorFlow provê acesso à alguns bancos de dados por meio da biblioteca, MNIST é um destes. O Algoritmo 3.7 mostra como foi realizado o processo de normalização das imagens, codificação da saída e formatação dos dados de entrada. As imagens de entrada podem ser formatadas de duas maneiras: canais x largura x altura ou largura x altura x canais. A primeira é denotada *channel first* (canal por primeiro) e a segunda *channel last* (canal por último). A formatação é dependente da máquina utilizada.

Algoritmo 3.7: Leitura e Tratamento de dados em TensorFlow

```
from tensorflow.keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

if image_data_format() == 'channels_last':
    x_train = x_train.reshape(x_train.shape[0], x_train.shape
        [1], x_train.shape[2], 1).astype('float32') / 255.0
else:
    x_train = x_train.reshape(x_train.shape[0], 1, x_train.
        shape[1], x_train.shape[2]).astype('float32') / 255.0
y_train = to_categorical(y_train, 10)
```

Na versão utilizada do MATLAB® não está disponível esta facilidade dos bancos de dados. Contudo foram utilizadas duas funções disponibilizadas por UFLDL (2011) para leitura das imagens e valores associados à elas. A função de leitura de imagem normaliza as imagens em seu código. Em termos utilizados pelo TensorFlow, a formatação das imagens em MATLAB® é *channel last*. O Algoritmo 3.8 mostra o processo implementado.

Algoritmo 3.8: Leitura e Tratamento de dados em MATLAB®

```
images = loadMNISTImages('../train-images.idx3-ubyte');
labels = loadMNISTLabels('../train-labels.idx1-ubyte');
images = reshape(images, 28, 28, 1, []);
labels = categorical(labels);
```

O Algoritmo 3.9 contém o trecho do código responsável pela construção do modelo idealizado. Nele a função de ativação ReLU é utilizada como uma camada, assim como é feito em MATLAB®, mostrando a flexibilidade da construção de RNAs na bilioteca.

Algoritmo 3.9: Construção do Modelo Idealizado em TensorFlow

```
classifier = Sequential()

classifier.add(Conv2D(8, 3, input_shape=x_train.shape[1:],
    kernel_initializer='zeros'))
classifier.add(BatchNormalization())
classifier.add(Activation('relu'))

classifier.add(MaxPooling2D())

classifier.add(Conv2D(16, 5, kernel_initializer='zeros'))
classifier.add(BatchNormalization())
classifier.add(Activation('relu'))

classifier.add(Flatten())

classifier.add(Dense(64, activation='relu', kernel_initializer=
    'random_uniform'))

classifier.add(Dense(10, activation='softmax',
    kernel_initializer='random_uniform'))
```

O mesmo processo é exposto pelo Algoritmo 3.10 em MATLAB®. A maior diferença no processo de criação está na camada de *flatten*, que é inexistente. A biblioteca automaticamente redimensiona a entrada da camada oculta.

Algoritmo 3.10: Construção do Modelo Idealizado em MATLAB®

```
layers = [...
    imageInputLayer([28 28 1])

    convolution2dLayer(3,8)
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(5,16)
    batchNormalizationLayer
```

```
    reluLayer

    fullyConnectedLayer(64)
    reluLayer

    fullyConnectedLayer(10)
    softmaxLayer

    classificationLayer];

layers(9).Weights = randn([64 1296]) * 0.05;
layers(11).Weights = randn([10 64]) * 0.05;
```

O processo de treinamento não apresenta diferenças significativas do que foi realizado na sessão 3.5.1. Os algoritmos 3.11 e 3.12 mostram o código relacionado ao treinamento em TensorFlow e MATLAB®, respectivamente.

Algoritmo 3.11: Treinamento em TensorFlow

```
classifier.compile(SGD(lr=0.01, momentum=0.9), loss='
    categorical_crossentropy', metrics=['accuracy'])
classifier.fit(x_train, y_train, batch_size=500, epochs=5,
    verbose=0)
```

Algoritmo 3.12: Treinamento em MATLAB®

```
opt = trainingOptions('sgdm',...
    'Verbose', false,...
    'MaxEpochs', 1,...
    'ExecutionEnvironment', 'cpu',...
    'MiniBatchSize', 500);
net = trainNetwork(images, labels, layers, opt);
```

Capítulo 4

Considerações Finais

As sessões deste capítulo apresentarão as comparações realizadas assim como a conclusão alcançada levendo-se em consideração o desempenho de cada biblioteca.

4.1 Construção de Modelos de RNA

Apesar das bibliotecas possuírem uma interface muito semelhante para esta finalidade, TensorFlow apresenta duas características que lhe garante a vantagem. As características são: flexibilidade e organização. TensorFlow é mais flexível quanto a composição dos modelos, é possível construir o mesmo modelo de diversas formas distintas. Devido a biblioteca utilizar um objeto que representa um modelo de RNA, TensorFlow dispõe de uma melhor organização quando comparado com MATLAB®, onde o modelo é uma coleção de camadas.

TensorFlow também apresenta um conjunto maior de algoritmos de treinamento, fácil customização de inicialização de parâmetros de uma RNA e, controle sobre a medida de custo, ou erro, de um modelo. A inicialização de parâmetros deve ser realizada pelo desenvolvedor no MATLAB®, o que pode tornar a tarefa confusa e repetitiva.

MATLAB® tem suporte a RNAs com treinamento não supervisionado, facilidade que não se faz presente no TensorFlow.

4.2 Manutenção

As duas bibliotecas apresentam uma modelagem simples e abstraída, a manutenção para sistemas desenvolvidos com ambas é simples e intuitiva. A flexibilidade de construção de modelos

e fácil parametrização de componentes em TensorFlow é um ponto que o coloca na liderança no quesito manutenibilidade.

4.3 Tempo de Treinamento

Foi medido o tempo de treinamento dos dois modelos desenvolvidos neste trabalho nas duas bibliotecas. Os códigos foram executados 20 vezes, foi retirada a média dos tempos medidos e comparado entre as duas bibliotecas. Os Algoritmos A.1, A.2, A.3 e A.4 mostram como foi medido o tempo de treinamento, e a Tabela 4.1 mostra os valores.

	Iris	MNIST
TensorFlow	2,506	19220
MATLAB®	0,611	423,925
Razão de Tempo	4.1	45.338

Tabela 4.1: Duração em segundos dos treinamentos

O MATLAB® apresentou um desempenho muito acima do TensorFlow. Ele treinou 4 vezes mais rápido com o conjunto Iris e 45 vezes com o MNIST. MATLAB® se provou muito mais flexível quanto o ambiente de execução, conseguindo extrair o máximo de eficiência dos recursos disponibilizados na máquina.

4.4 Tempo de Predição

A medida do tempo de predição consta nas últimas 3 linhas de código dos algoritmos do Apêndice A. A base de dados MNIST contém 10.000 imagens para realização de testes, estas imagens foram utilizadas para a realização das predições. A base Iris não disponibiliza dados para realização de testes, por este motivo as 150 instâncias da base foram utilizadas para a realização das predições.

A Tabela 4.2 mostra a relação dos tempos das predições. Para a base MNIST, MATLAB® se mostrou aproximadamente 3 vezes mais rápida. Todavia, TensorFlow foi aproximadamente 2 vezes mais eficiente para a base Iris. Isto pode ser devido ao MATLAB® estar gastando mais tempo otimizando o processo do que a otimização pode ganhar.

	Iris	MNIST
TensorFlow	0,0013	8,665
MATLAB®	0,0028	3,0577
Razão de Tempo	0.464	2.8338

Tabela 4.2: Duração em segundos das predições

4.5 Análise Final

TensorFlow é uma boa ferramenta de prototipação, permite rápido desenvolvimento de modelos de forma flexível e clara, entretanto, não é uma alternativa viável em ambientes onde o tempo de execução é uma variável vital, fato reforçado pela Figura 4.1.

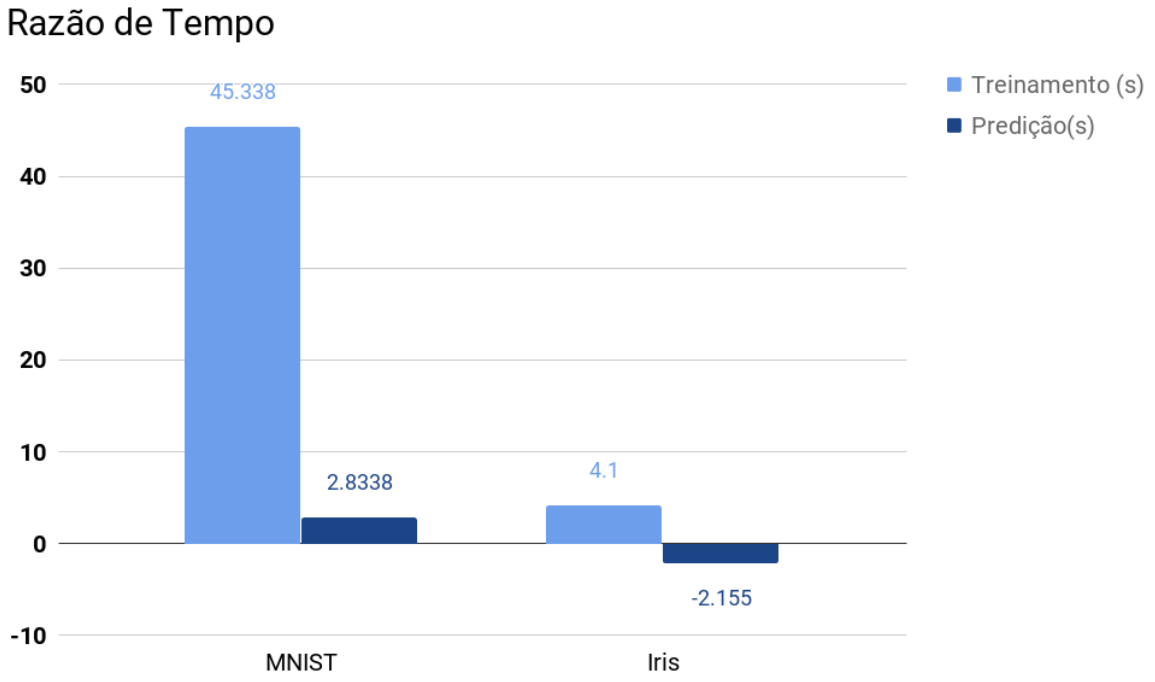


Figura 4.1: Gráfico da proporção de eficiência do MATLAB® em relação ao TensorFlow

MATLAB® garante suporte em uma maior gama de plataformas como mostrado na Tabela 4.3, entretanto TensorFlow possui suporte a mais ambientes de execução, como apresentado na Tabela 4.4.

MATLAB®	Plataforma	TensorFlow
✓	Windows	✓
✓	Windows Server	×
✓	macOS	✓
✓	Ubuntu	✓
✓	Debian	×
✓	Red Hat	×
✓	SUSE	×
×	Servidor Linux por Docker[7]	✓
×	Raspbian	✓
✓[8]	Android	✓
✓[8]	iOS	✓

Tabela 4.3: Relação de suporte de plataformas das ferramentas

MATLAB®	Plataforma	TensorFlow
✓	CPU	✓
✓	GPU	✓
×	TPU	✓
✓	*Cloud*	✓
✓	*Cluster*	✓
×	Android NN API	✓

Tabela 4.4: Relação de ambientes de execução das ferramentas

Deve-se levar em conta também que TensorFlow é uma ferramenta *open source* gratuita, enquanto MATLAB® é uma ferramenta proprietária de custo elevado.

4.6 Recomendações de Trabalhos Futuros

A partir dos resultados obtidos neste trabalho, apresentamos algumas sugestões para dar continuidade às pesquisas nesta área:

- Desenvolvimento de uma RNA recorrente, arquitetura não contemplada no escopo deste trabalho, para fins de comparação.
- Comparar a eficiência das bibliotecas com o uso de GPU, TPU, *cloud* ou *clusters*.
- Desenvolver os modelos utilizando a API de baixo nível do TensorFlow para verificar a existência de ganho em desempenho.

[7] Docker é um serviço de virtualização de máquinas disponível em: https://www.docker.com
[8] Suporte indireto, deve-se primeiro converter o código em MATLAB® para C, e então integrá-lo a um aplicativo

Apêndice A

Códigos Desenvolvidos

Este capítulo contém os códigos desenvolvidos para os testes. As funções *loadMNISTImages* e *loadMNISTLabels* utilizadas no Algoritmo A.4 foram retiradas de UFLDL (2011).

Algoritmo A.1: Classificador para a coleção de dados Iris em TensorFlow

```
import pandas as pd
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import SGD
import numpy as np
import time

dataset = pd.read_csv('iris.csv', header=None)

x = dataset.iloc[:, 0:4].values
y = dataset.iloc[:, 4].values

del dataset

for index, clss in enumerate(np.unique(y)):
    y[y == clss] = index

y = to_categorical(y)

model = Sequential()

model.add(Dense(16, activation='relu', kernel_initializer='
   random_normal', input_shape=(4,)))
model.add(Dense(3, activation='softmax', kernel_initializer='
   random_normal'))
```

```
model.compile(SGD(0.0005), loss='categorical_crossentropy',
    metrics=['accuracy'])

start = time.perf_counter()
model.fit(x, y, epochs=250, batch_size=20, verbose=0)
print(time.perf_counter() - start)

start = time.perf_counter()
model.predict(x)
print(time.perf_counter() - start)
```

Algoritmo A.2: Classificador de dígitos manuscritos em TensorFlow

```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, BatchNormalization,
    Activation, MaxPooling2D, Flatten, Dense
from tensorflow.keras.backend import image_data_format
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.optimizers import SGD
import time

(x_train, y_train), (x_test, y_test) = mnist.load_data()

if image_data_format() == 'channels_last':
    x_train = x_train.reshape(x_train.shape[0], x_train.shape
        [1], x_train.shape[2], 1).astype('float32') / 255.0
    x_test = x_test.reshape(x_test.shape[0], x_test.shape[1],
        x_test.shape[2], 1).astype('float32') / 255.0
else:
    x_train = x_train.reshape(x_train.shape[0], 1, x_train.
        shape[1], x_train.shape[2]).astype('float32') / 255.0
    x_test = x_test.reshape(x_test.shape[0], x_test.shape[1],
        x_test.shape[2], 1).astype('float32') / 255.0

y_train = to_categorical(y_train, 10)

classifier = Sequential()

classifier.add(Conv2D(8, 3, input_shape=x_train.shape[1:],
    kernel_initializer='zeros'))
classifier.add(BatchNormalization())
classifier.add(Activation('relu'))
```

```
classifier.add(MaxPooling2D())

classifier.add(Conv2D(16, 5, kernel_initializer='zeros'))
classifier.add(BatchNormalization())
classifier.add(Activation('relu'))

classifier.add(Flatten())

classifier.add(Dense(64, activation='relu', kernel_initializer=
    'random_uniform'))

classifier.add(Dense(10, activation='softmax',
    kernel_initializer='random_uniform'))

classifier.compile(SGD(lr=0.01, momentum=0.9), loss='
    categorical_crossentropy', metrics=['accuracy'])

start = time.perf_counter()
classifier.fit(x_train, y_train, batch_size=500, epochs=5,
    verbose=0)
print(time.perf_counter() - start)

start = time.perf_counter()
classifier.predict(x_test)
print(time.perf_counter() - start)
```

Algoritmo A.3: Classificador para a coleção de dados Iris em MATLAB®

```
dataset = readtable('../iris.csv');

x = table2array(dataset(:, 1:4))';
y = categorical(table2array(dataset(:, 5))');

clear dataset

layers = [...
    sequenceInputLayer(4)
    fullyConnectedLayer(16)
    reluLayer
    fullyConnectedLayer(3)
    softmaxLayer
    classificationLayer];

layers(2).Weights = randn([16 4]) * 0.05;
```

```
layers(4).Weights = randn([3 16]) * 0.05;

opt = trainingOptions('sgdm',...
    'Verbose', false,...
    'Momentum', 0,...
    'MaxEpochs', 250,...
    'MiniBatchSize', 20,...
    'InitialLearnRate', 0.0005,...
    'ExecutionEnvironment', 'cpu',...
    'Shuffle', 'every-epoch');

tic
net = trainNetwork(x, y, layers, opt);
toc

tic
ypred = predict(net, x);
toc
```

Algoritmo A.4: Classificador de dígitos manuscritos em MATLAB®

```
images = loadMNISTImages('../train-images.idx3-ubyte');
images_test = loadMNISTImages('../t10k-images.idx3-ubyte');
labels = loadMNISTLabels('../train-labels.idx1-ubyte');

images = reshape(images, 28, 28, 1, []);
images_test = reshape(images_test, 28, 28, 1, []);
labels = categorical(labels);

layers = [...
    imageInputLayer([28 28 1])

    convolution2dLayer(3,8)
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(5,16)
    batchNormalizationLayer
    reluLayer

    fullyConnectedLayer(64)
    reluLayer
```

```
    fullyConnectedLayer(10)
    softmaxLayer

    classificationLayer];

layers(9).Weights = randn([64 1296]) * 0.05;
layers(11).Weights = randn([10 64]) * 0.05;

opt = trainingOptions('sgdm',...
    'Verbose', false,...
    'MaxEpochs', 1,...
    'ExecutionEnvironment', 'cpu',...
    'MiniBatchSize', 500);

tic
net = trainNetwork(images, labels, layers, opt);
toc

tic
ypred = predict(net, images_test);
toc
```

Apêndice B

Filtros Convolucionais

Filtros convolucionais é um termo adotado em Aprendizado de Máquina para referir-se à técnica de Processamento de Imagem mais comumente conhecida como *kernel* ou máscara.

A técnica em processamento de imagem é utilizada para aplicar efeitos ou extrair informações de imagens. Exemplos de filtros são: detecção de bordas, borramento e realce de imagens. *Kernel*, filtro e máscara são termos utilizados para definir o conjunto de valores utilizado para aplicar o processo de convolução.

Este capítulo foi baseado nos materiais disponibilizados por Catarina (2018) e Hearty (2016). Será explicado somente o processo de convolução em imagens, visto que somente este foi utilizado neste trabalho.

B.1 Funcionamento

A convolução é um processo iterativo que recebe uma imagem como entrada e retorna uma matriz de valores, que pode também ser interpretada como uma imagem. O retorno pode ser menor ou ter o mesmo tamanho que a imagem de entrada, porém não maior. Existem três variáveis utilizadas durante o processo de convolução: o filtro, *padding* e *stride*.

O filtro, matriz, máscara ou *kernel* de convolução se referem aos valores utilizados para retirar informações da imagem. Existem diversos filtros para os mais diversos fins, para citar alguns: filtro de *blur*, de realçe e de bordas. O filtro é uma matriz de tamanho menor ou igual à imagem de entrada.

Como mencionado anteriormente, o processo de convolução pode resultar em uma imagem menor do que a de entrada, e para evitar essa redução de tamanho existe o *padding*. O *padding*

é um preenchimento inserido nas bordas da imagem para aumentandar o tamanho dela, assim, ao final da convolução, a imagem resultante manterá o seu tamanho.

O *stride* define o passo de avanço das iterações da convolução, este parâmetro pode ser mais facilmente entendido no exemplo na seção B.2.

B.2 Exemplo

Aqui será exposto uma parte do processo de convolução para uma imagem de tamanho 5x5. Será aplicado um *padding* com 0s na imagem. O processo contará com um *stride* de 2 e filtro 3x3. A imagem com *padding*, filtro e resultado da convolução são apresentados na Figura B.1.

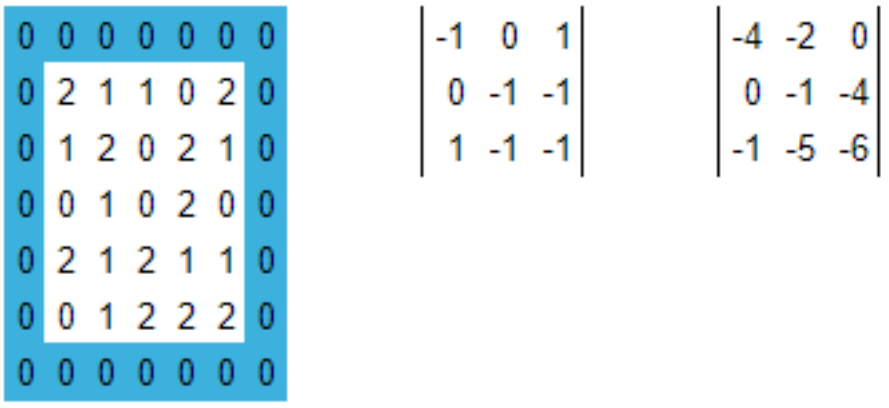


Figura B.1: Imagem de entrada com *padding*, Filtro de convolução e Resultado respectivamente

Ao início do processo o filtro de convolução é espelhado nos eixos x e y e então, é alinhado com a imagem como mostrado na Figura B.2. É realizada a soma do produto dos elementos em sobreposição, e este é o resultado da iteração atual da convolução.

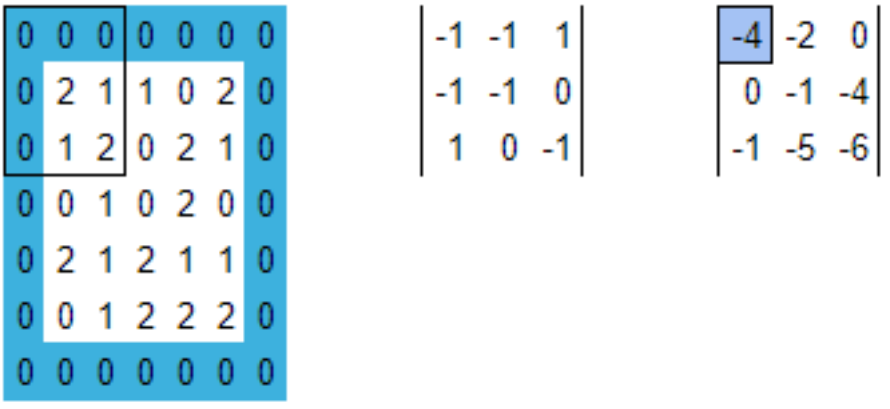


Figura B.2: Filtro espelhado e alinhado com a imagem e resultado da iteração em destaque

A convolução então avança uma iteração, o filtro será alinhado novamente com a imagem de acordo com o valor do *stride*. Neste exemplo o *stride* possui o valor 2 para linhas e colunas, portanto o próximo passo da convolução é como apresentado na Figura B.3.

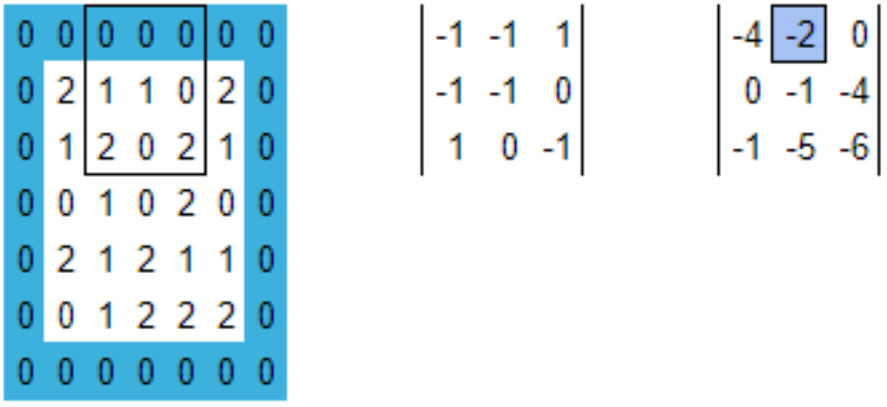


Figura B.3: Segunda iteração da convolução e resultado em destaque

Este processo se repetirá até alcançar o final da imagem. A Figura B.4 apresenta mais 3 iterações da convolução, note como o filtro é alinhado novamente à esquerda da imagem quando a iteração avança na vertical.

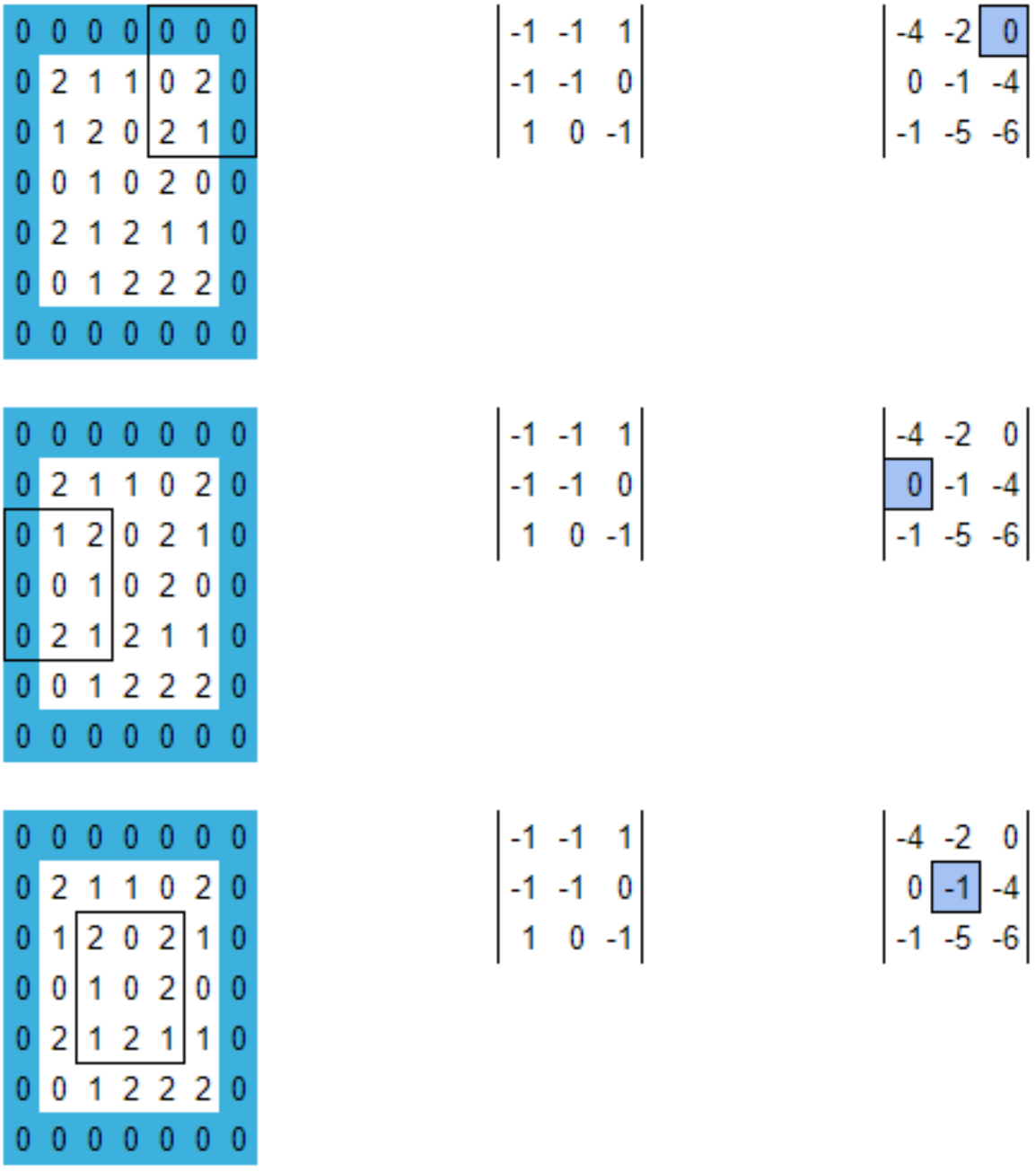


Figura B.4: As 3 próximas iterações da convolução

Referências Bibliográficas

ABADI, M. et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. Disponível em: <https://www.tensorflow.org/>. Acesso em: 20 ago 2018.

BHATIA, R. *Google's Tensorflow gaining popularity. Why?* 2017. Disponível em: <https://www.analyticsindiamag.com/googles-tensorflow-gaining-popularity/>. Acesso em: 24 out 2018.

CAMPOS, M.; SAITO, K. *Sistemas inteligentes em controle e automação de processos*. Rio de Janeiro (RJ: Ciência Moderna, 2004. ISBN 8573933089.

CARDOSO, S. *Comunicação entre as células nervosas*. 2001. Disponível em: <http://www.cerebromente.org.br/n12/fundamentos/neurotransmissores/neurotransmitters2_p.html>. Acesso em: 17 abr 2018.

CARDOSO, S. *Neurônios: Nossa Galáxia Interna*. 2001. Disponível em: <http://www.cerebromente.org.br/n07/fundamentos/neuron/rosto.htm>. Acesso em: 03 jul 2018.

CATARINA, A. S. *Processamento de Imagens Digitais*. 2018. Disponível em: <http://inf.unioeste.br/~adair/PID/Notas\\%20Aula/Aulas\\%20de\\%20PID\\%202018.pdf>. Acesso em: 31 out 2018.

CHOLLET, F. et al. *Keras*. 2015. Disponível em: <https://keras.io>. Acesso em: 20 ago 2018.

COLUMBUS, L. *10 Charts That Will Change Your Perspective On Artificial Intelligence's Growth*. 2018. Disponível em: <https://www.forbes.com/sites/louiscolumbus/2018/01/12/10-charts-that-will-change-your-perspective-on-artificial-intelligences-growth/>. Acesso em: 25 jul 2018.

Developer Economics. *What is the best programming language for Machine Learning?* 2018. Disponível em: <https://towardsdatascience.com/what-is-the-best-programming-language-for-machine-learning-a745c156d6b7>. Acesso em: 25 jul 2018.

DOZAT, T. Incorporating nesterov momentum into adam. 2016.

DUCHI, J.; HAZAN, E.; SINGER, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, v. 12, n. Jul, p. 2121–2159, 2011.

FAUSETT, L. *Fundamentals of neural networks: architectures, algorithms, and applications*. Englewood Cliffs, NJ Delhi Dorling Kindersley: Prentice-Hall, 1994. ISBN 0133341860.

FERBER, J. *Multi-agent systems: an introduction to distributed artificial intelligence*. Harlow: Addison-Wesley, 1999. ISBN 0201360489.

FISHER, R. A. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, Wiley Online Library, v. 7, n. 2, p. 179–188, 1936.

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

HAYKIN, S. *Redes neurais: princípios e prática*. Porto Alegre: Bookman, 2001. ISBN 8573077182.

HEARTY, J. *Advanced Machine Learning with Python*. [S.l.]: Packt Publishing, 2016.

HINTON, G.; SRIVASTAVA, N.; SWERSKY, K. *Rmsprop: Divide the gradient by a running average of its recent magnitude*. 2014. Disponível em: <http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf>. Acesso em: 20 jul 2018.

HORNIK, K. Approximation capabilities of multilayer feedforward networks. *Neural networks*, Elsevier, v. 4, n. 2, p. 251–257, 1991.

IOFFE, S.; SZEGEDY, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

JIA, Y. et al. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

KALUZA, B. *Machine Learning in Java*. [S.l.]: Packt Publishing - ebooks Account, 2016. ISBN 1784396583.

KINGMA, D. P.; BA, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

KOVACS, Z. *Redes neurais artificiais: fundamentos e aplicações, um texto básico*. São Paulo: Collegium Cognitio, 1996. ISBN 8586396028.

LECUN, Y. *LeNet-5, convolutional neural networks*. 2010. Disponível em: <http://yann.lecun.com/exdb/lenet/>. Acesso em: 16 out 2018.

LECUN, Yann and Cortes, Corinna and Burges, Christopher J.C. 2013. Disponível em: <http://yann.lecun.com/exdb/mnist/index.html>. Acesso em: 23 out 2018.

MathWorks. *The Origins of MATLAB*. 2004. Disponível em: <https://www.mathworks.com/company/newsletters/articles/the-origins-of-matlab.html>. Acesso em: 15 out 2018.

MathWorks. *MATLAB The Language of Technical Computing*. [S.l.], 2012. Disponível em: <https://www.mathworks.com/tagteam/73554_91199v02_overview.pdf>. Acesso em: 15 out 2018.

MathWorks. *Company Overview*. [S.l.], 2018. Disponível em: <https://www.mathworks.com/content/dam/mathworks/tag-team/Objects/c/company-fact-sheet-8282v18.pdf>. Acesso em: 15 out 2018.

MathWorks. *Deep Learning Toolbox*. 2018. Disponível em: <https://www.mathworks.com/products/deep-learning.html>. Acesso em: 15 out 2018.

NESTEROV, Y. A method for unconstrained convex minimization problem with the rate of convergence o (1/kˆ 2). In: *Doklady AN USSR*. [S.l.: s.n.], 1983. v. 269, p. 543–547.

OREMUS, W. *What Is ?TensorFlow,? and Why Is Google So Excited About It?* 2015. Disponível em: <http://www.slate.com/blogs/future_tense/2015/11/09/google_s_tensorflow_is_open_source_and_it_s_about_to_be_a_huge_huge_deal.html>. Acesso em: 20 ago 2018.

PAN, S. J.; YANG, Q. et al. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, Institute of Electrical and Electronics Engineers, Inc., 345 E. 47 th St. NY NY 10017-2394 USA, v. 22, n. 10, p. 1345–1359, 2010.

Python Core Team. *Python: A dynamic, open source programming language*. [S.l.], 2015. Disponível em: <https://www.python.org>. Acesso em: 25 jul 2018.

REDDI, S. J.; KALE, S.; KUMAR, S. On the convergence of adam and beyond. 2018.

ROBBINS, H.; MONRO, S. A stochastic approximation method. In: *Herbert Robbins Selected Papers*. [S.l.]: Springer, 1985. p. 102–109.

SILVA, I. N. da. *Redes Neurais Artificiais Para Engenharia e Cincias Aplicadas - Curso Pratico*. [S.l.]: ARTLIBER, 2010. ISBN 8588098539.

TensorFlow. *API Documentation*. 2018. Disponível em: <https://www.tensorflow.org/api_docs/>. Acesso em: 20 ago 2018.

TensorFlow. *Installing TensorFlow*. 2018. Disponível em: <https://www.tensorflow.org/install/>. Acesso em: 20 ago 2018.

TensorFlow. *Introduction*. 2018. Disponível em: <https://www.tensorflow.org/guide/low_level_intro>. Acesso em: 15 out 2018.

TensorFlow. *Release TensorFlow 1.1.0*. 2018. Disponível em: <https://github.com/tensorflow/tensorflow/releases/tag/v1.1.0>. Acesso em: 15 out 2018.

TensorFlow. *Release TensorFlow 1.4.0*. 2018. Disponível em: <https://github.com/tensorflow/tensorflow/releases/tag/v1.4.0>. Acesso em: 15 out 2018.

TensorFlow. *Releases - tensorflow/tensorflow - GitHub*. 2018. Disponível em: <https://github.com/tensorflow/tensorflow/releases?after=v0.8.0rc0>. Acesso em: 20 ago 2018.

TensorFlow. *TensorBoard: Visualizing Learning*. 2018. Disponível em: <https://www.tensorflow.org/guide/summaries_and_tensorboard>. Acesso em: 10 out 2018.

TensorFlow. *TensorFlow Guide*. 2018. Disponível em: <https://www.tensorflow.org/guide/>. Acesso em: 10 out 2018.

Theano Development Team. *Convolutional Neural Networks (LeNet)*. 2013. Disponível em: <http://deeplearning.net/tutorial/lenet.html>. Acesso em: 31 out 2018.

Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, maio 2016. Disponível em: <http://arxiv.org/abs/1605.02688>.

UC Irvine Machine Learning. *Iris Data Set*. 2007. Disponível em: <https://archive.ics.uci.edu/ml/datasets/iris>. Acesso em: 24 out 2018.

UFLDL. *Using the MNIST Dataset*. 2011. Disponível em: <http://ufldl.stanford.edu/wiki/index.php/Using_the_MNIST_Dataset>. Acesso em: 25 out 2018.

WARD, J. et al. Efficient mapping of the training of convolutional neural networks to a cuda-based cluster. *Eindhoven University of Technology, The Netherlands*, Citeseer, v. 12, 2011.

ZEILER, M. D. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.