UNIVERSIDADE FEDERAL DO PAMPA

LIZIANE ZERBIN GALLERT

CLASSIFICAÇÃO DE IMAGENS DE ULTRASSONOGRAFIA COM REDES NEURAIS CONVOLUCIONAIS EM AMBIENTE DE ALTO DESEMPENHO

Bagé
2018

LIZIANE ZERBIN GALLERT

CLASSIFICAÇÃO DE IMAGENS DE ULTRASSONOGRAFIA COM REDES NEURAIS CONVOLUCIONAIS EM AMBIENTE DE ALTO DESEMPENHO

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Engenharia de Computação como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Sandro da Silva Camargo
Coorientador: Leonardo Bidese de Pinho

Bagé
2018

G617C Gallert, Liziane Zerbin

Classificação de imagens de ultrassonografia com redes neurais convolucionais em ambiente de alto desempenho / Liziane Zerbin Gallert. – dezembro, 2018.

81 f.: il.

Trabalho de Conclusão de Curso (Graduação) – Universidade Federal do Pampa, Campus Bagé, Engenharia de Computação, 2018.

"Orientação: Sandro da Silva Camargo; Coorientação: Leonardo Bidese de Pinho".

1. Inteligência Artificial. 2. Diagnóstico por Imagem. 3. GPU. 4. Microsoft Cognitive Toolkit. I. Título.

LIZIANE ZERBIN GALLERT

CLASSIFICAÇÃO DE IMAGENS DE ULTRASSONOGRAFIA COM REDES NEURAIS CONVOLUCIONAIS EM AMBIENTE DE ALTO DESEMPENHO

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Engenharia de Computação como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação.

Trabalho de Conclusão de Curso defendido e aprovado em: 08 de Dezembro de 2018.

Banca examinadora:

Prof. Dr. Sandro da Silva Camargo
Orientador

Prof. Dr. Leonardo Bidese de Pinho
Coorientador

Prof. Dr. Milton Roberto Heinen
Universidade Federal do Pampa

Prof. Dr. Érico Marcelo Hoff do Amaral
Universidade Federal do Pampa

Aos amigos pelas incontáveis horas estudando, pelo incentivo e apoio constantes, possibilitando minha chegada até aqui.

“NÃO ENTRE EM PÂNICO!”

— (ADAMS, 2004)

RESUMO

Avanços na análise computadorizada de imagens médicas proporcionam o surgimento de métodos e técnicas que possam auxiliar no diagnóstico precoce de uma série de doenças. Atualmente esses diagnósticos se dão por meio de inspeção visual por parte de especialistas humanos. Este trabalho descreve o desenvolvimento de um modelo computacional, baseado em Redes Neurais Convolucionais (CNN), que objetiva auxiliar o médico radiologista na confirmação de um diagnóstico, por meio da identificação de massas em achados de imagens ultrassonográficas de fígado. Em geral, o processo de treinamento de uma rede neural exige um alto poder de processamento para dados densos. Quando se trata de processamento de imagens, esta questão é ainda mais realçada, visto que o número de entradas da rede é proporcional ao número de *pixels* da imagem de interesse, justificando o uso de processamento paralelo. É importante ressaltar que a classe de imagens médicas não pode permitir armazenamento com perdas de dados e exige precisão em sua identificação. O modelo de CNN proposto foi capaz de realizar a classificação dessas massas com uma precisão de 85%, para o conjunto de imagens analisadas.

Palavras-chave: Inteligência Artificial. Diagnóstico por Imagem. GPU. Microsoft Cognitive Toolkit.

ABSTRACT

Advances in the computerized analysis of medical images provide the emergence of methods and techniques that may assist in the early diagnosis of a number of diseases. Currently, these diagnoses are so due to visual inspection by human experts. This paper describes the development of a computational model, based on Convolutional Neural Network (CNN), that aims to assist the radiologist in the confirmation of a diagnosis by identifying masses in findings of ultrasonographic images of the liver. In general, the process of training a neural network requires a high processing power for dense data. When it comes to image processing, this issue is further enhanced, since the number of network inputs is proportional to the number of pixels of the image of interest, justifying the use of parallel processing. It is important to note that the medical imaging class does not allow lossy data storage and requires precission in its identification. The developed CNN model was able to perform the classification of these masses with an accuracy of 85% , for the set of analyzed images.

Keywords: Artificial neural network, Convolutional neural network, GPU, Ultrasound images.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CMY	*Cyan Magenta Yellow*
CNN	*Convolutional Neural Network*
CPU	*Central Processing Unit*
GIMP	*GNU Image Manipulation Program*
GPU	*Graphics Processing Unit*
GUI	*Graphical User Interface*
HSI	*Hue Saturation Intensity*
IDE	*Integrated Development Environment*
MLP	*Multilayer Perceptron*
PDI	Processamento Digital de Imagens
RGB	*Red Green Blue*
RNA	Rede Neural Artificial
ROC	*Receiver Operating Characteristic*

SUMÁRIO

1 INTRODUÇÃO

Imagens têm desempenhado um papel importante na sociedade, independentemente dos métodos utilizados para sua criação, transmissão e acesso (CORRÊA, 2013). Esta importância foi potencializada pela evolução das tecnologias de captura de imagens, com ênfase na passagem da era da fotografia analógica para a digital, assim como no aumento da capacidade de armazenamento dos dispositivos de captura. Dentre as áreas que passaram a explorar extensivamente o uso de imagens, a medicina exerce papel de destaque, principalmente em relação às atividades de diagnóstico por imagem (OLIVEIRA, 2009). Neste contexto, e em decorrência do desenvolvimento tecnológico, técnicas de ultrassonografia foram impulsionadas, transformando este método em um importante instrumento de investigação diagnóstica.

A ultrassonografia é um exame de diagnóstico muito utilizado na medicina moderna que tem como objetivo visualizar, em tempo real, qualquer órgão ou tecido do corpo (KEALY; MCALLISTER; GRAHAM, 2012). As imagens coletadas durante exames necessitam de tratamento, extração das características relevantes e identificação, a fim de gerar informações úteis para contribuir com o diagnóstico. Atualmente esses diagnósticos, salvo raras exceções, se dão por meio de inspeção visual por parte de especialistas humanos. Com o rápido crescimento do volume de imagens produzidas, sua análise por esses especialistas torna-se difícil, pois apresenta diversos problemas, entre os quais o custo associado a esse processo de análise e a falta de disponibilidade dos especialistas. Mesmo que a mão de obra e os recursos necessários estejam à disposição, a análise minuciosa das imagens coletadas torna-se difícil e cansativa, especialmente quando em grandes quantidades, tornando este processo suscetível a erro.

Dessa forma, ocorre a demanda por métodos que possibilitem o processamento, extração de características e análise da imagem não somente de maneira mais eficiente, mas que também mantenha um alto grau de confiabilidade, simulando ou atuando como auxiliar no trabalho dos especialistas humanos. Técnicas ou ferramentas de inteligência artificial surgem como opções na automatização desses métodos, em especial as Redes Neurais Artificiais (RNA) .

As CNNS (*Convolutional Neural Networks*), base deste estudo, aprimoram a generalização das RNAS, possuindo a habilidade de lidar com deformações e variações nos dados de entrada devido à combinação da sua arquitetura das idéias dos campos receptivos, compartilhamento de pesos e subamostragem temporal ou espacial (LECUN;

BENGIO, 2003). Sendo assim, as CNN são capazes de prever ou extrapolar dos padrões identificados para dados desconhecidos, viabilizando a análise e extração de conhecimento de grandes imagens, de modo a produzir saídas consistentes para entradas não apresentadas anteriormente. Como ponto negativo, o processo de treinamento demanda um tempo longo por ser computacionalmente custoso.

Objetivando reduzir o tempo computacional para executar o treinamento da rede neural, e a análise das imagens durante o processo de teste, assim como tirando proveito da característica distribuída natural das RNA, optou-se por transferir as partes de processamento intensivo da ferramenta da CPU (*Central Processing Unit*) para uma GPU (*Graphics Processing Unit*). Isto se deve a sua arquitetura paralela, conforme (NVIDIA, 2017a), que consiste em centenas ou milhares de núcleos menores e mais eficientes criados para lidar com múltiplas tarefas simultaneamente, potencializando a eficiência do processo de treinamento da rede neural implementada.

Por fim foram realizados testes simuladores, com consequente verificação da eficácia dos diagnósticos em imagens ultrassonográficas de fígado. Foram avaliados então a acurácia e o tempo de processamento total para a realização das análises propostas no decorrer da presente pesquisa, realizando uma comparação entre os tempos de treino e teste da rede obtidos na CPU e GPU.

Neste contexto, objetiva-se o desenvolvimento de um modelo computacional - baseado em CNN - para identificação de massas em achados de imagens ultrassonográficas de fígado.

1.1 Problema de pesquisa

Os objetivos deste trabalho são os seguintes:

1.1.1 Objetivo geral

Desenvolver um modelo de CNN para identificar massas em imagens de ultrassonografia de fígado, sobre um ambiente de alto desempenho.

1.1.2 Objetivos específicos

Uma vez definido o objetivo geral, elencou-se o seguinte conjunto de metas específicas da pesquisa que complementam e viabilizam o alcance do objetivo geral:

1. Estudar trabalhos correlatos e analisar o estado da arte;
2. Entender o funcionamento de uma rede neural convolucional e suas diferentes topologias;
3. Implementar e avaliar diferentes configurações de redes neurais convolucionais para classificação das imagens;
4. Otimizar os parâmetros da rede para atingir a melhor acurácia no problema em questão;

1.2 Trabalhos correlatos

Estudos demonstram o grande potencial das RNA em sistemas de apoio ao diagnóstico, principalmente nas aplicações de classificação de padrões utilizando imagens (MENESES et al., 2008; MARCOMINI, 2013; AMBRÓSIO, 2002). Ambrósio (2007) também propõe a utilização de modelos de RNA com a função de extração de atributos automatizada, apresentando resultados positivos. Esta Seção expõe as principais ideias desses autores, realizando uma síntese de seus trabalhos.

Ambrósio (2002) descreve um sistema de apoio diagnóstico baseado em uma RNA do tipo MLP (*Multilayer Perceptron*), que funciona como um identificador de padrões, classificando as lesões intersticiais pulmonares apresentandas entre determinadas patologias. Ao final do treinamento realizado, a RNA teve sua performance testada e apresentou um indice de acerto de $91,67\%$.

Ambrósio (2007) verifica a utilização de redes neurais auto-organizáveis como ferramenta de extração de atributos e redução de dimensionalidade de imagens radiográficas de tórax, objetivando a caracterização de lesões intersticiais de pulmão. Para tal implementou um algoritmo baseado nos mapas alto-organizáveis (SOM), com algumas variações, obtendo uma redução dos cerca de 3 milhões de *pixels* que compõem uma imagem, para 24 elementos. Já na classificação dos padrões encontrados entre determinadas patologias, utilizou-se uma RNA MLP, validada com a metodologia *leave-one-out*. Com um banco de dados contendo 79 exemplos de padrão linear, 37 exemplos de padrão nodu-

lar, 30 exemplos de padrão misto e 72 exemplos de padrão normal, o classificador obteve a média de $89,5\%$ de acerto, sendo 100% de classificação correta para o padrão linear, $67,5\%$ para o padrão nodular, $63,3\%$ para o padrão misto e 100% para o padrão normal.

Marcomini (2013) propôs uma metodologia, baseada em um classificador neural, para a detecção e caracterização automática de achados ultrassonográficos da mama. Os experimentos foram executados com imagens obtidas por simuladores e exames clínicos. Na fase de pré-processamento foram utilizados filtro de *wiener*, equalização e filtro da mediana para a minimização do ruído. Para a segmentação, foram testadas cinco diferentes técnicas a fim de determinar a representação mais concisa. Como resultado, a rede neural SOM mostrou-se como a mais relevante. Após a delimitação do objeto, foram definidas as características mais expressivas para a descrição morfológica do achado. A acurácia alcançada durante o treinamento em imagens simuladas foi de $94,2\%$. Para avaliar a generalização do modelo, foi efetuada a classificação com imagens desconhecidas ao sistema, sendo atingida uma acurácia de até 90%.

Meneses et al. (2008) relata a aplicação de uma RNA com arquitetura MLP, *afeed-forward*, com algoritmo de *backpropagation*, no reconhecimento de tecido ósseo. As imagens médicas foram adquiridas através do laboratório Elettra, em Trieste, na Itália. A qualidade dos resultados na tarefa da classificação foi verificada através de Curvas ROC (*Receiver Operating Characteristic*), uma ferramenta utilizada para medir e especificar problemas no desempenho do diagnóstico em medicina. Os valores de área variam de 0 a 1, sendo que 1 indica o melhor desempenho possível. Para a RNA empregada, obteve-se uma área sob a curva de 1, o que significa que a arquitetura e o treinamento da RNA se mostraram adequados para a tarefa de reconhecimento de tecido ósseo.

Saito (2007) objetiva demonstrar a viabilidade da otimização do tempo de processamento de imagens mamográficas, sem que ocorra perda de *pixels*, definindo requisitos e subsídios para aplicação da computação paralela e distribuída no processamento de imagens médicas em geral. Para tal, implementou-se as etapas do processamento de forma sequencial e paralela, utilizando a linguagem de programação *Java*. A base de dados construída possui imagens de diferentes tamanhos ($500KB$, $1MB$, $11MB$ e $21MB$), com resolução de contraste de $16bits$. Para os testes, utilizou-se a implementação do filtro da mediana utilizando o algoritmo de ordenação *shellsort* e o algoritmo *bubblesort* para todos os casos. Os resultados obtidos não foram favoráveis à execução paralela dos filtros utilizando-se imagens pequenas, porém foi possível observar que para processamentos intensos, como é o caso da utilização de janelas de filtro com tamanhos $7x7$, $9x9$ e $11x11$,

o uso do processamento paralelo é bastante vantajoso.

1.3 Organização deste trabalho

Este trabalho está organizado da seguinte forma: O Capítulo 2 apresenta os conceitos fundamentais para compreensão deste trabalho, incluindo as três principais vertentes desta pesquisa: o Processamento Digital de Imagens (PDI), as CNN e o ambiente de alto desempenho. No Capítulo 3, a metodologia do estudo é descrita, explanando detalhes envolvidos no desenvolvimento prático do trabalho bem como os materiais e métodos utilizados. Os resultados obtidos na modelagem da solução são apresentados e discutidos no Capítulo 4. As conclusões e perspectivas de trabalhos futuros utilizando o modelo de CNN implementado são exploradas no Capítulo 5.

2 FUNDAMENTAÇÃO TEÓRICA

Neste Capítulo constam os conceitos necessários sobre processamento de imagens, incluindo a apresentação do modelo de imagem digital e modelos de cores, apresentação dos fundamentos de imagens de ultrassom e as etapas do processamento digital de imagens. Também são apresentados os conceitos de aprendizado de máquina, redes neurais artificiais e convolucionais. Por fim são discutidos os detalhes de uso das redes neurais convolucionais para reconhecimento de imagens em um ambiente de alto desempenho.

2.1 Processamento de imagens

O PDI trata da manipulação de imagens digitais por um sistema computacional, onde suas entradas e saídas também correspondem a imagens. Outra saída possível desse processo de manipulação é um conjunto de atributos extraídos das imagens de entrada.

O interesse pelos métodos de processamento digital advém da necessidade da melhoria de imagens digitais para interpretação humana ou para percepção automática de máquinas e técnicas de inteligência artificial (GONZALEZ; WOODS, 2010).

2.1.1 Imagem digital

Gonzalez e Woods (2010) definem uma imagem como uma função bidimensional $f(x,y)$, em que x e y são coordenadas espaciais, e a amplitude de f em qualquer par de coordenadas (x,y) é chamada de intensidade ou cor da imagem nesse ponto. A matriz 1 descreve o conceito de uma imagem digital.

O processo de discretização da imagem digital no espaço e na amplitude recebem os nomes de amostragem e quantização, respectivamente. Isso ocorre na conversão da imagem analógica (contínua na variação espacial e nos níveis de cinza) em digital (MASCARENHAS; VELASCO, 1984). Esse processo ocorre, atualmente, nos próprios dispositivos de aquisição de imagem.

Quando x, y e os valores de intensidade de f são quantidades finitas e discretas, temos uma imagem digital (GONZALEZ; WOODS, 2010). A seguir consta a explanação

de uma matriz de m linhas e n colunas da digitalização de uma imagem.

$$f(x,y) = \begin{bmatrix} f(0,0) & f(0,1) & \cdots & f(0,n-1) \\ f(1,0) & f(1,1) & \cdots & f(1,n-1) \\ \vdots & \vdots & \vdots & \vdots \\ f(m-1,0) & f(m-1,1) & \cdots & f(m-1,n-1) \end{bmatrix} \tag{1}$$

Maiores valores de m e n implicam em uma imagem de maior resolução (FILHO; NETO, 1999). Os elementos dessa matriz finita, identificados por um par de índices (x,y), são conhecidos como *pixels* (elementos pictórios ou elementos de imagem) e correspondem à menor unidade que compõe uma imagem digital. Eles contêm os atributos de cor em cada ponto. A quantização faz com que cada um destes *pixels* assuma um valor inteiro, na faixa de 0 a $2n-1$ Filho e Neto (1999) sendo representados por números binários de modo a permitir seu armazenamento, transferência, impressão ou reprodução, bem como seu processamento por meios eletrônicos.

Quanto maior o valor de n, maior o número de níveis de cinza presentes na imagem digitalizada. De acordo com Gonzalez e Woods (2010), o número mínimo de *pixels* em uma imagem digital, para evitar a perda de qualidade, é dado por:

$$\beta = mnk \tag{2}$$

Onde m corresponde ao número de linhas da matriz; n corresponde ao número de colunas e $k \in \mathbb{N}^*$ (números inteiros positivos com exceção do zero) e é a potência que determina a faixa de representação da quantização da imagem (de 0 à $2^k - 1$).

2.1.2 Imagem monocromática

Para Filho e Neto (1999), uma imagem monocromática pode ser descrita matematicamente por uma função $f(x,y)$ da intensidade luminosa, sendo seu valor, em qualquer ponto de coordenadas espaciais (x,y), proporcional ao brilho (ou nível de cinza) da imagem naquele ponto:

$$f(x,y) = i(x,y)r(x,y) \tag{3}$$

Onde a função $i(x,y)$ representa a iluminância, ou seja, quantidade de luz que incide sobre o objeto. Já a função $r(x,y)$ caracteriza as propriedades de refletância, isto

é, a fração de luz incidente que o objeto vai transmitir ou refletir. Ambas as funções respeitam os seguintes intervalos:

$$0 < i(x,y) < \infty \tag{4}$$

e

$$0 < r(x,y) < 1 \tag{5}$$

Figura 1 – Representação da imagem monocromática e seu par de eixos (x,y).

Fonte: Adaptado de (FILHO; NETO, 1999).

A intensidade de uma imagem monocromática f nas coordenadas (x,y), representadas na Figura 1, será denominada nível de cinza (l) da imagem naquele ponto. Este valor estará no intervalo:

$$l_{min} < l < l_{mx} \tag{6}$$

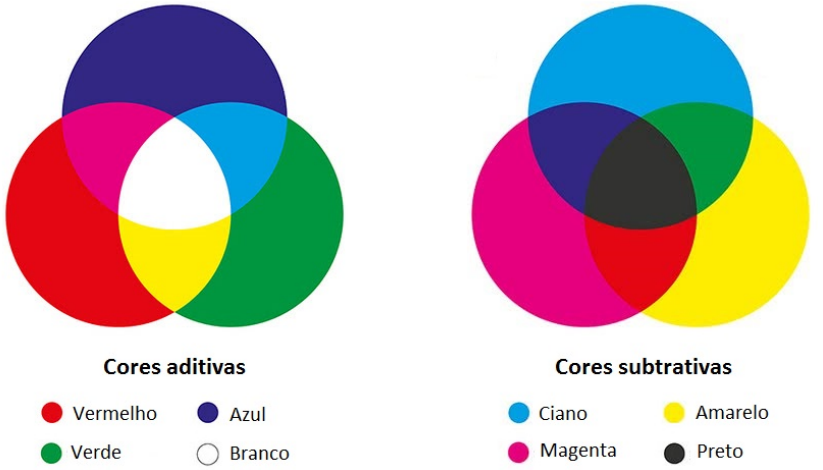
O intervalo $[l_{min}, l_{mx}]$ é denominado escala de cinza da imagem. É comum deslocar este intervalo numericamente para o intervalo dos inteiros $[0,w)$, onde $l = 0$ representa o *pixel* preto e $l = w - 1$ representa o *pixel* branco. Normalmente w é uma potência inteira positiva de 2.

2.1.3 Imagem colorida

A estrutura do olho humano nos permite ver todas as cores como combinações de três cores primárias aditivas, vermelho, verde e azul. Essas cores, quando adicionadas, produzem as cores secundárias subtrativas, denominadas magenta (vermelho mais azul), ciano (verde mais azul) e amarelo (vermelho mais verde). Cores secundárias também são conhecidas como cores primárias de pigmentos (GONZALEZ; WOODS, 2010).

Misturar três cores primárias, ou uma secundária com sua cor primária oposta, em intensidades corretas, produz a luz branca, como pode ser observado na Figura 3(a). Já uma mistura adequada das três secundárias, ou uma secundária com sua primária oposta, produz o preto, conforme a Figura 3(b).

Figura 2 – Diagramas de cores.(a) e (b)



(a) Cores primárias e suas combinações.

(b) Cores secundárias e suas combinações.

Fonte: Adaptado de (GONZALEZ; WOODS, 2010).

As quantidades das cores primárias necessárias na formação de qualquer cor são especificadas através de seus coeficientes tricromáticos, definidos como:

$$r = \frac{R}{(R+G+B)} \tag{7}$$

$$g = \frac{G}{(R+G+B)} \tag{8}$$

e

$$b = \frac{B}{(R+G+B)} \tag{9}$$

Onde as variáveis r e R representam a cor vermelha (*red*); g e G a cor verde (*green*) e b e B a cor azul (*blue*), ao passo que $r+g+b=1$.

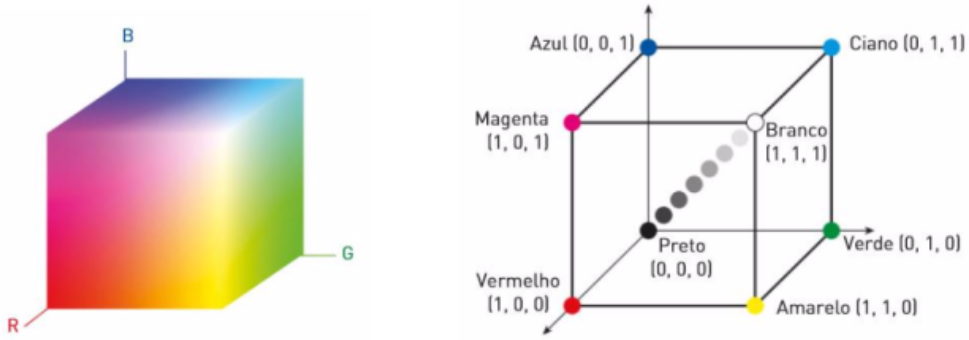
2.1.4 Modelos de cores

Modelos de cores objetivam propiciar uma forma de padronização de representação de cores. Essencialmente, um modelo de cores é uma especificação de um sistema de coordenadas e um subespaço dentro desse sistema no qual cada cor é representada por um único valor.

Atualmente, de acordo com Gonzalez e Woods (2010), os modelos de cor orientados a *hardware* mais utilizados são o RGB (*Red Green Blue*) para monitores coloridos e câmeras de vídeo em cores e o CMY (*Cyan Magenta Yellow*) para impressoras coloridas. Ainda segundo Gonzalez e Woods (2010), o modelo HSI (*Hue Saturation Intensity*) que corresponde estritamente à forma como os seres humanos descrevem e interpretam as cores.

O sistema RGB foi desenvolvido em conjunto com a tecnologia da televisão, cujas cores são criadas por misturas aditivas da luz Baldner et al. (2017), onde cada cor aparece em seus componentes espectrais primários; vermelho, verde e azul. Este modelo baseia-se em um sistema de coordenadas cartesianas. O subespaço de cores de interesse é o cubo apresentado na Figura 4(a), no qual os valores RGB primários estão em três vértices unidos na origem; as cores secundárias estão em outros três vértices; o preto está na origem e o branco está no vértice mais distante da origem. Ainda neste modelo, a escala de cinza são os pontos de valores RGB iguais e estendem-se do preto até o branco ao longo do segmento de reta que une esses dois pontos.

Figura 3 – Modelo do cubo de cores RGB.(a) e (b)



(a) Cubo de cores RGB. (b) Esquema de cores do cubo RGB.

Fonte: Adaptado de (GONZALEZ; WOODS, 2010).

Por conveniência, assume-se que todos os valores de cor foram normalizados, de forma que o cubo mostrado na Figura 4(b) é unitário, isto é, assume-se que os valores de R, G e B estejam no intervalo $[0, 1]$.

Imagens representadas no modelo de cores RGB consistem de três componentes de imagens, uma para cada cor primária. Quando alimentadas em um monitor RGB, essas três imagens se combinam na tela para produzir uma imagem de cores compostas (GONZALEZ; WOODS, 2010).

2.1.5 Imagem de ultrassom

A característica mais marcante na aquisição de imagens utilizando ondas sonoras é a sua capacidade de coletar dados em praticamente qualquer região, a qualquer momento, independentemente do clima ou das condições de iluminação do ambiente. Esse método de formação de imagens encontra aplicação em várias áreas, tais como exploração geológica, indústria e saúde.

Quanto à aplicação na área da saúde, as imagens de ultrassonografia são geradas utilizando o procedimento seguinte, descrito por Gonzalez e Woods (2010):

1. O sistema de ultrassom (um computador e uma sonda de ultrassom, consistindo em uma fonte, um receptor e um monitor) transmite pulsos sonoros de alta frequência (1 a 5 MHz) ao corpo;
2. As ondas sonoras percorrem o corpo e atingem uma fronteira entre tecidos (entre fluido e tecido mole ou entre tecido mole e osso). Algumas das ondas sonoras são

refletidas de volta à sonda, ao passo que outras continuam o percurso até atingir outra fronteira e serem refletidas;

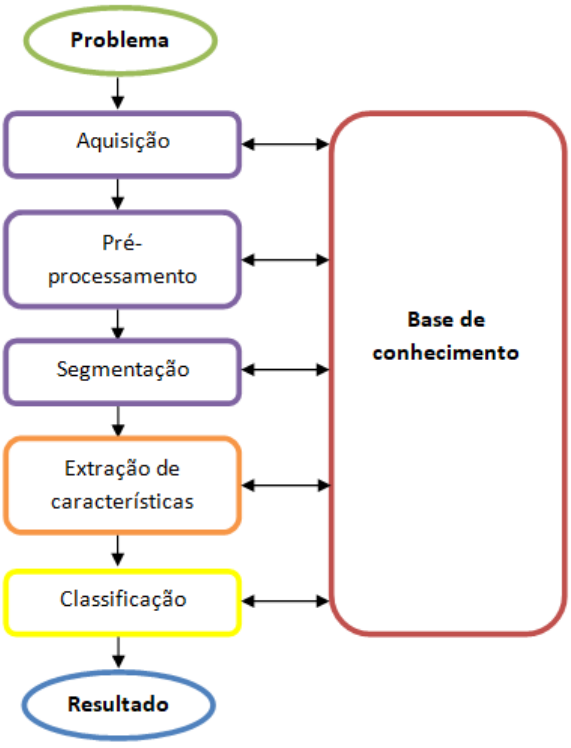
3. As ondas refletidas são captadas pela sonda e transmitidas ao computador;
4. A máquina calcula a distância da sonda até as fronteiras de tecido ou órgãos utilizando a velocidade do som no tecido ($1.540 m/s$) e o tempo de retorno de cada eco;
5. O sistema exibe as distâncias e as intensidades dos ecos na tela, formando uma imagem bidimensional.

Em uma imagem de ultrassonografia, milhões de pulsos e ecos são enviados e recebidos a cada segundo. A sonda pode ser deslocada ao longo da superfície do corpo e inclinada para obter várias vistas. Uma vez encontrada a vista de interesse, a imagem é salva e impressa, para sua análise posterior, por especialistas.

2.1.6 Etapas do processamento digital de imagens

A Figura 4 mostra um diagrama de blocos correspondente às principais etapas do PDI e foi adaptada de (FILHO; NETO, 1999). Esta Subseção discorre sobre essas etapas.

Figura 4 – PDI e suas principais etapas.



Fonte: Adaptado de (FILHO; NETO, 1999).

Após definido o domínio do problema, inicia-se o processo de aquisição de imagens, objetivando formar um banco de dados para processamento nas etapas seguintes. As imagens de interesse podem ser obtidas em bancos de imagens *online* ou produzidas para esse fim, com auxilio de uma ferramenta de captura de imagem.

O pré-processamento objetiva aprimorar a qualidade das imagens obtidas no passo anterior para as etapas subsequentes. A imagem resultante ao fim desse processo é uma imagem digital de melhor qualidade que a original. Já o processo de segmentação trata-se da divisão da imagem digital em suas unidades significativas, ou seja, nos objetos de interesse que a compõem.

A etapa de extração de características procura extrair das imagens resultantes da segmentação suas características relevantes, por meio de descritores que permitam caracterizar com precisão os objetos presentes na imagem. Estes descritores devem ser representados por uma estrutura de dados adequada ao algoritmo de reconhecimento utilizado. É importante observar que nessa etapa a entrada ainda é uma imagem, mas a saída é um conjunto de dados correspondentes aquela imagem.

Por fim, tem-se o reconhecimento e a interpretação (fases da etapa de classificação). Denomina-se reconhecimento o processo de atribuição de um rótulo a um objeto, baseado em suas características, traduzidas por seus descritores. A tarefa de interpretação, por outro lado, consiste em atribuir significado a um conjunto de objetos reconhecidos.

Todas essas tarefas das etapas descritas pressupõem a existência de um conhecimento sobre o problema a ser resolvido, armazenado em uma base de dados, cujo tamanho e complexidade variam enormemente conforme a aplicação. As etapas do PDI descritas nessa Subseção foram embasadas em Filho e Neto (1999).

2.2 Redes neurais artificiais

Haykin (2001) define uma rede neural, ou RNA, como um processador paralelamente distribuído constituído de unidades de processamento simples, que tem propensão para armazenar conhecimento experimental e torná-lo disponível para o uso. Ela se inspira no funcionamento do cérebro em dois aspectos:

1. O conhecimento é adquirido pela rede a partir de seu ambiente através de um processo de aprendizagem;
2. Forças de conexão entre neurônios, conhecidas como pesos sinápticos, são utiliza-

das para armazenar o conhecimento adquirido.

Para Rezende (2003) uma RNA é um modelo matemático que se assemelha às estruturas neurais biológicas e que tem capacidade computacional adquirida por meio de aprendizado e de generalização. O aprendizado é atingido não pelas modificações dos neurônios, mas pelas modificações dos pesos das interconexões, assumindo que:

1. O processamento da informação ocorre com auxílio de vários elementos chamados neurônios;
2. Os sinais são propagados de um elemento a outro através de conexões;
3. Cada conexão possui um peso associado que, em uma RNA típica, pondera o sinal transmitido;
4. Cada neurônio possui uma função de ativação (geralmente não-linear), que tem como argumento a soma ponderada dos sinais de entrada para determinar sua saída.

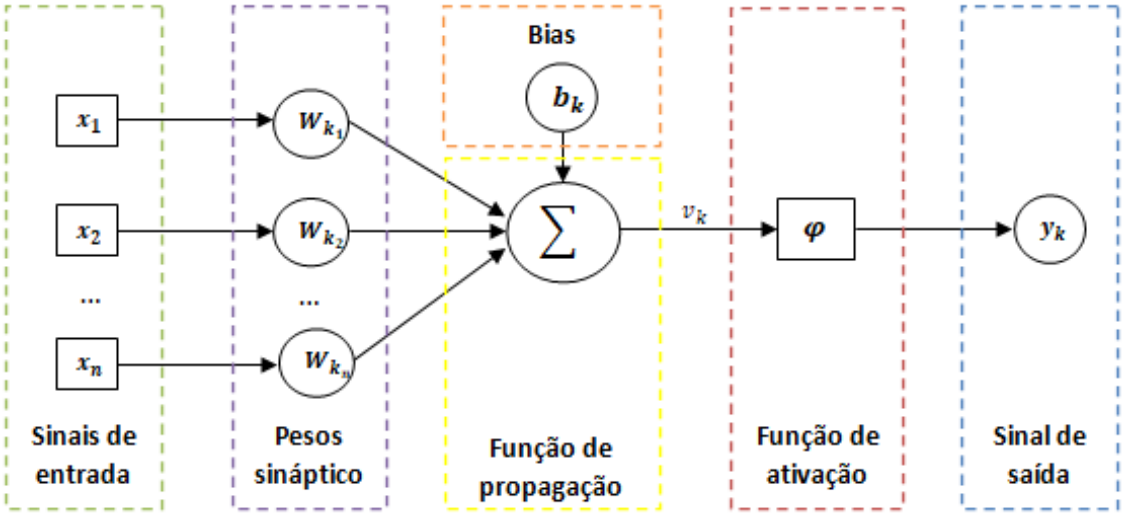
Em suma, o aprendizado em uma RNA consiste na realização de mudanças sistemáticas aos pesos sinápticos a fim de levar a resposta da RNA a valores aceitáveis de precisão. Logo, o conjunto de pesos das conexões pode ser visto como um sistema dinâmico, pois eles adaptam-se para codificar o conhecimento que se deseja aprender. A generalização de uma RNA está ligada à capacidade de dar respostas coerentes para dados não apresentados a ela previamente na etapa de treinamento.

2.2.1 O neurônio artificial

Um neurônio é a menor unidade de processamento de informação na operação de uma RNA. O diagrama de blocos da Figura 5 mostra o modelo de um neurônio onde, de acordo com Haykin (2001), pode-se identificar três elementos no modelo neural:

1. Um conjunto de sinapses caracterizadas por seus respectivos pesos. Especificamente, um sinal x_j na entrada da sinapse j conectada ao neurônio k é multiplicado pelo peso sináptico w_{kj};
2. Uma função de propagação, cuja função é somar os sinais de entrada, ponderados pelas respectivas sinapses do neurônio;
3. Uma função de ativação para restringir a amplitude da saída de um neurônio.

Figura 5 – Modelo não-linear de um neurônio artificial.



Fonte: Adaptado de (HAYKIN, 2001).

O modelo neuronal apresentado na Figura 5 inclui também uma bias b_k, com objetivo de aumentar ou diminuir a entrada da função de ativação. A partir disso, Haykin (2001) define matematicamente um neurônio k escrevendo o seguinte par de equações:

$$v_k = \sum_{j=1}^{n} w_{kj} x_j \tag{10}$$

e

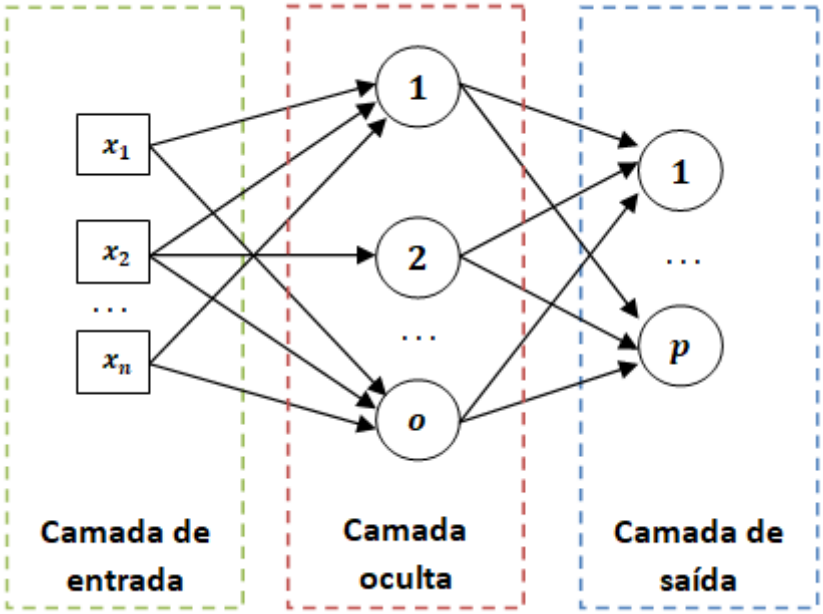
$$y_k = \varphi(v_k + b_k) \tag{11}$$

Onde x_j são os sinais de entrada; w_{kj} são os pesos sinápticos do neurônio k; v_k é o sinal resultante do somatório; b_k é a bia; φ é a função de ativação e por fim, y_k é o sinal de saída do neurônio k.

2.2.2 Topologia e arquitetura de rede

A topologia da rede refere-se às diferentes composições estruturais e quantidades de neurônios nas camadas que compõem uma RNA. Usualmente os neurônios da RNA distribuem-se em três camadas, como explicita a Figura 6.

Figura 6 – Ilustração da topologia de camadas de uma RNA típica.



Fonte: Autor (2017).

A camada de entrada é responsável pelo recebimento dos dados a serem analisados e suas respectivas associações com os pesos sinápticos de entrada. A camada intermediária ou oculta, composta de uma ou mais camadas de neurônios, extrai as informações associadas aos dados, sendo responsável pela maior parte do processamento deles (aprendizado). Já a camada de saída, recebe os dados das camadas anteriores e ativa a resposta mais adequada.

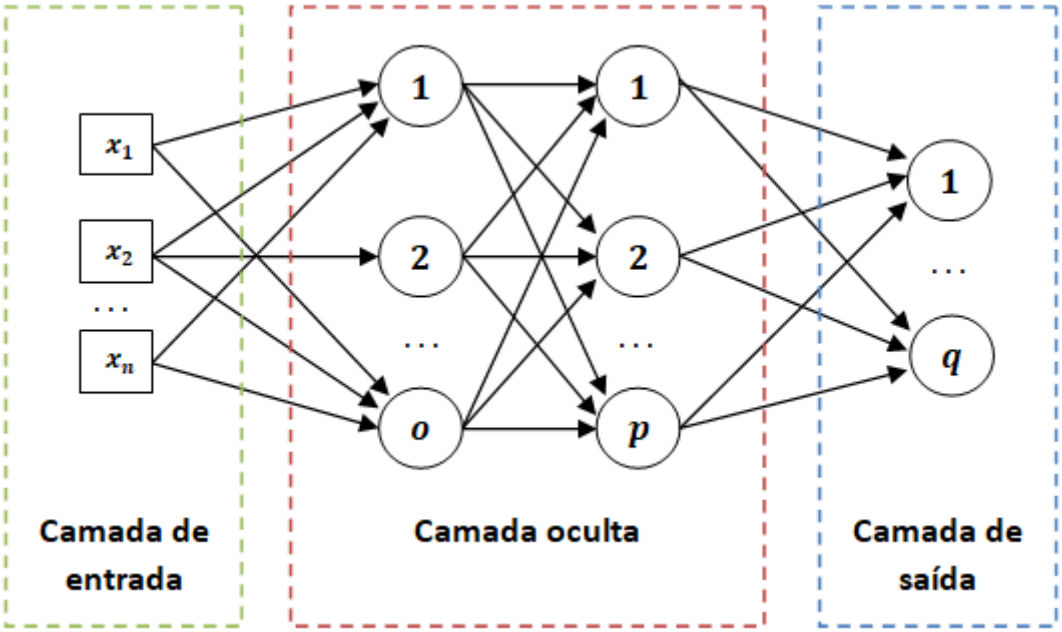
A arquitetura de uma RNA refere-se à disposição de seus neurônios, um em relação ao outro, seguindo as conexões sinápticas. Conjuntos de neurônios artificiais, organizados e conectados de várias formas, resultam em diferentes arquiteturas neurais, com características e aplicações bastante distintas (HAYKIN, 2001). Essas arquiteturas de rede podem ser classificadas quanto às seguintes características: número de camadas que possuem; conexão entre os neurônios e conectividade da rede. Em relação ao número de camadas, a RNA pode ser de camada única, existindo somente um neurônio entre qualquer entrada e qualquer saída da rede, ou de múltiplas camadas, onde existe mais de um neurônio entre qualquer entrada e qualquer saída.

A propriedade de conexão entre os neurônios pode ser do tipo alimentada a diante - a saída de um neurônio de uma determinada camada da RNA só pode ser usada como entrada de um neurônio pertencente a uma camada posterior a ela - ou retroalimentada - onde a saída de um neurônio de uma determinada camada da RNA é utilizada como entrada de um neurônio pertencente a uma camada anterior a ela.

Finalmente, RNA dividem-se em termos de conectividade em fracamente e totalmente conectadas. Uma rede diz-se fracamente conectada quando nem todos os neurônios de uma camada estão associados aos neurônios da camada subseqüente a ela. Em contrapartida, uma rede é totalmente conectada quando essa associação entre todos os neurônios de uma camada e os neurônios de sua camada seguinte existe.

Diferentes combinações das características citadas formam as arquiteturas de RNA utilizadas na solução de problemas de reconhecimento e generalização. Buscando exemplificar isso, a Figura 7 ilustra uma RNA de múltiplas camadas, alimentada a diante e totalmente conectada, conhecida como MLP.

Figura 7 – Ilustração da arquitetura de uma RNA.



Fonte: Autor (2017).

2.3 Aprendizado de máquina

De acordo com Mitchell (1997), um algoritmo aprende quando o seu desempenho melhora com a experiência em uma determinada tarefa. O aprendizado de máquina é uma subárea da inteligência artificial que explora a utilização de algoritmos e técnicas no aprendizado de informações e padrões sobre dados existentes, buscando realizar previsões sobre o comportamento dos dados ou o reconhecimento de novos dados introduzidos. Isso dá aos computadores a habilidade de aprender sem serem explicitamente programados.

Este tópico apresenta uma breve motivação das técnicas de aprendizagem de má-

quina utilizadas nessa pesquisa e conceitos relacionados a elas.

2.3.1 Paradigmas de aprendizado

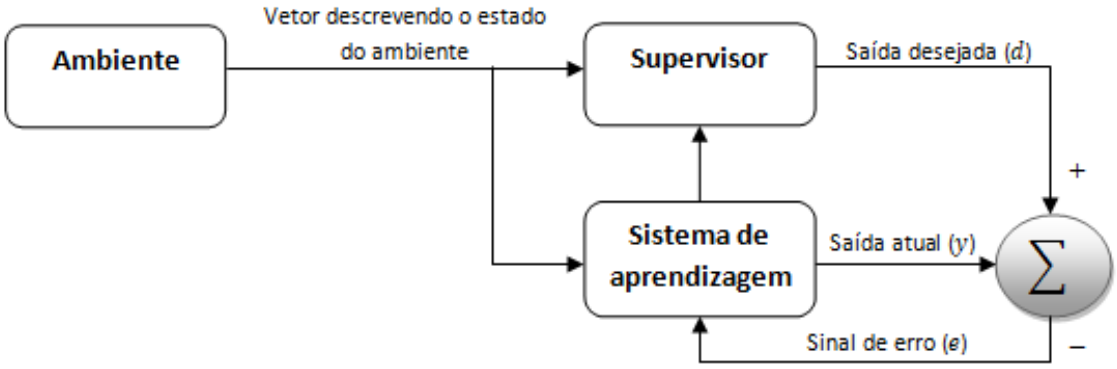
Uma vez definida, a RNA precisa de treinamento. O aprendizado de uma RNA é realizado por meio de processos iterativos de ajustes aplicados aos pesos sinápticos, o chamado treinamento. O aprendizado só ocorre quando a rede neural atinge uma solução generalizada para um determinado problema. Em síntese, treinar uma rede é ajustar a sua matriz de pesos sinápticos de forma que o vetor de saída coincida com um valor desejado para cada vetor de entrada (MIRANDA; FREITAS; FACCION, 2009).

De acordo com Haykin (2001), aprendizado é o processo pelo qual os parâmetros livres (conjunto de pesos sinápticos e bias) de uma rede neural são adaptados através de um mecanismo de apresentação de estímulos fornecidos pelo ambiente no qual a rede está inserida. Assim, a apresentação de um conjunto de dados de entrada à rede neural implica na alteração dos pesos sinápticos da rede que, consequentemente, geram um novo padrão de resposta do ambiente, para novos dados de entrada. Ainda segundo Haykin (2001), o tipo de treinamento (ou paradigma de aprendizado) é definido pela forma na qual os parâmetros são modificados. Existem basicamente três paradigmas de aprendizado: supervisionado, não-supervisionado e por reforço.

No paradigma de aprendizado não-supervisionado os dados de entrada não são rotulados, deixando a RNA livre para descobrir novos padrões nos dados (HAYKIN, 2001). Já o paradigma de aprendizado por reforço, ainda conforme Haykin (2001) enfatiza a aprendizagem através da interação direta com o ambiente, sem uma supervisão ou modelo completo do ambiente.

O aprendizado supervisionado trata-se de um paradigma de aprendizagem no qual um supervisor possui conhecimento sobre o ambiente em que a rede está inserida e apresenta exemplos de conjuntos (vetores) de entradas e saídas desejadas. Objetiva-se então aprender uma regra geral que mapeia as entradas as suas respectivas saídas. Para as redes com pesos, o aprendizado de uma RNA é realizado por meio de processos iterativos de ajustes aplicados aos pesos. A Figura 8 ilustra o esquema de funcionamento do aprendizado supervisionado.

Figura 8 – Diagrama de blocos do processo de aprendizagem supervisionada.



Fonte: Adaptado de (HAYKIN, 2001)

Seja t um índice de tempo discreto, marcando o intervalo de tempo do processo iterativo responsável pelo ajuste de parâmetros de um determinado neurônio k, constituinte da rede neural, o sinal de erro produzido a cada iteração dá-se por:

$$e_k(t) = d_k(t) - y_k(t) \tag{12}$$

Os pesos atribuídos para cada entrada são atualizados conforme o algoritmo de aprendizagem usado, para que este erro seja minimizado. Este processo é repetido para o conjunto amostral de treinamento até que o erro calculado alcance uma tolerância (estipulada previamente) desejada, ou que o parâmetro do número máximo de ciclos através dos exemplos de treinamento (épocas) seja atingido. Após a etapa de treinamento, espera-se que a RNA seja capaz de generalizar.

Para que a RNA chegue ao resultado esperado de generalização dos dados apresentados a ela, segue-se um conjunto de etapas, abaixo citadas:

1. Aquisição de informação: Constitui-se um banco com os dados a serem generalizados pela RNA;
2. Pré-processamento: Os dados coletados devem ser tratados para a análise eficiente da RNA. Esse tratamento pode se dar, na forma de normalização, suavização de ruídos, reamostragem, separação em diferentes conjuntos, dependendo do tipo de dados a serem analisados;
3. Treinamento: Aplicação da técnica de aprendizado supervisionado, ilustrada na Figura 8;
4. Teste: Verificação se o mapeamento de entradas e saídas computado pela RNA, para um conjunto amostral não utilizado no processo de treinamento, é correto (ou

dentro da tolerância).

2.4 Reconhecimento de padrões em imagens utilizando RNA

O processo para o reconhecimento de padrões em imagens utilizando RNA aqui descrito é embasado teoricamente em Haykin (2001). Primeiramente deve-se selecionar uma arquitetura apropriada para a RNA, nesse caso a arquitetura MLP, onde a camada de entrada contenha uma quantidade de neurônios correspondente a quantidade de *pixels* que a imagem de entrada possui.

Define-se, então, uma camada de saída com um número de neurônios correspondente a quantidade de respostas possíveis após o processo de análise. A seguir, um subconjunto de exemplos é utilizado para treinar a rede por meio de um algoritmo apropriado. Esta fase do projeto da rede é chamada de aprendizagem.

Por último, o desempenho de reconhecimento da rede treinada é testado com dados não apresentados anteriormente. Especificamente, uma imagem de entrada é apresentada para a rede, mas dessa vez não lhe é fornecido o resultado que correspondente a esta imagem em particular. O desempenho da rede é então estimado comparando-se o reconhecimento do resultado fornecido com a resposta real da imagem em questão. Essa segunda fase da operação da rede é a chamada generalização. Contudo, Fernandes (2013) apresenta alguns problemas enfrentados por esse modelo:

1. Imagens são representadas por grandes matrizes, normalmente com milhares de elementos (*pixels*). Cada *pixel* apresenta uma característica de entrada para a rede, originando entradas com uma grande quantidade de atributos. Isso torna o processo de reconhecimento computacionalmente custoso, especialmente por se tratar de uma rede completamente conectada;
2. A MLP necessitaria de muitos padrões de treino para tolerar variações de translação, rotação ou escala em uma imagem;
3. Padrões nos *pixels* de entrada não afetariam o treinamento de rede, ou seja, o MLP não consegue tirar proveito da disposição dos *pixels* no seu processo de aprendizagem.

Visando contornar esses problemas, surge uma variação do MLP denominada CNN. Para tal, as CNN fazem uso da teoria de campos receptivos locais, compartilhamento de pesos e subamostragem espacial em sua arquitetura (LECUN; BENGIO, 2003),

como explicita a Subseção 2.5.

2.5 Redes neurais convolucionais

Uma CNN é uma variação das redes MLP, tendo sido inspirada no processo biológico de processamentos de dados visuais (VARGAS; PAES; VASCONCELOS, 2016). Essa variação, segundo LeCun e Bengio (2003) é dada ao desenvolver uma estrutura especializada incorporando informação prévia no projeto da MLP. Para tal deve-se utilizar a combinação de duas técnicas (LECUN; BENGIO, 2003):

1. Restringir a escolha de pesos sinápticos através do uso de compartilhamento de pesos;
2. Restringir a arquitetura da rede pelo uso de conexões locais, conhecidas como campos receptivos.

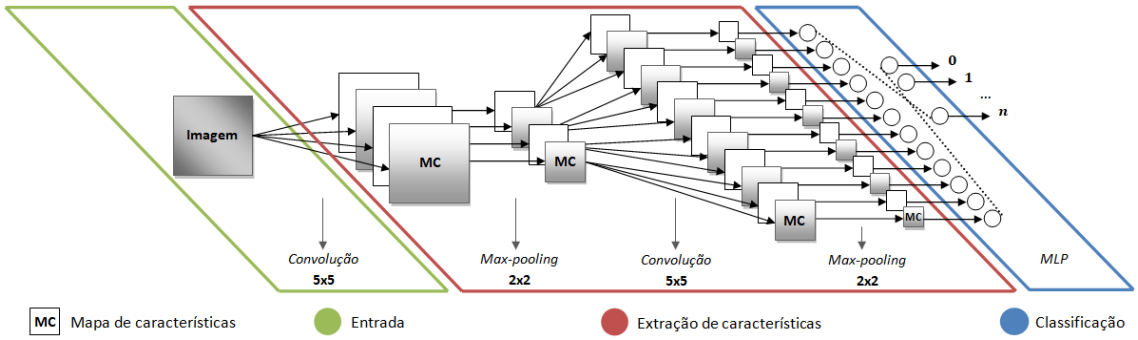
De acordo com Haykin (2001) a utilização dessas técnicas, particularmente a última, têm um benefício marginal vantajoso: o número de parâmetros livres de rede é reduzido significativamente. Isto torna o processo computacional envolvido menos custoso.

Uma CNN consiste em várias etapas com funções diferentes (VARGAS; PAES; VASCONCELOS, 2016). A Figura 9 ilustra as etapas de entrada, extração de características e classificação.

Para ser útil à etapa de entrada, uma imagem deve possuir boa qualidade (ausência de ruídos e sombras) e ter seu tamanho (número de *pixels*) conhecido. O número de *pixels* influencia de maneira direta a quantidade de estágios pertencentes à etapa de extração de características. Por sua vez, o número de etapas influenciará na quantidade de neurônios pertencentes à camada de entrada da etapa de classificação.

No exemplo da Figura 9, a etapa de extração de características possui dois estágios. Cada estágio é composto por uma camada convolutiva seguida de uma camada de *pooling*. Essas camadas e suas funções são descritas nas Seções 2.5.1 e 2.5.2, respectivamente.

Figura 9 – Exemplo de CNN com dois estágios e suas diferentes etapas.



Fonte: Adaptado de (LECUN; BENGIO, 2003).

A etapa de classificação diz respeito ao uso de um MLP no reconhecimento e interpretação dos dados extraídos da imagem de entrada, cuja saída corresponde à quantidade de respostas possíveis após o processo de análise.

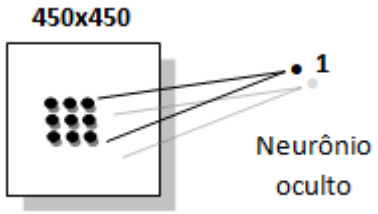
2.5.1 Camada convolutiva

O propósito principal da etapa convolucional em uma CNN é detectar características ou recursos visuais nas imagens (TORRES, 2016). Essa extração de características se dá na camada oculta do processo de convolução.

Primeiramente, cada *pixel* de uma imagem de interesse é atribuído a um neurônio na camada de entrada. Na CNN os neurônios da camada de entrada não são completamente conectados aos neurônios da camada oculta; a conexão se dá de maneira local e com dependência espacial na distribuição dos *pixels* (TORRES, 2016).

No exemplo da Figura 10, cada neurônio da camada oculta é conectado a uma pequena região de 3x3 (portanto com 9 neurônios) da camada de entrada. Essa janela convolutiva de tamanho 9x9 percorre toda a camada de 450x450 *pixels* que contém a imagem de entrada. Para cada posição da janela há um neurônio na camada oculta que processa sua informação.

Figura 10 – Exemplo de conexão de um neurônio da camada oculta com a camada de entrada.

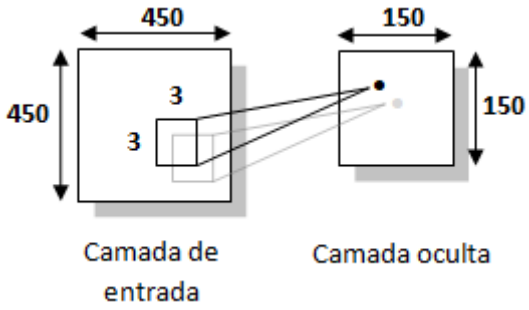


Fonte: Adaptado de (TORRES, 2016).

Supõe-se que a janela convolutiva está na zona superior esquerda da imagem, o que provê a informação necessária ao primeiro neurônio da camada oculta. A janela faz movimentos de avanço de 1 *pixel* de distância, onde em cada passo a nova janela se sobrepõem com a anterior, exceto na linha de *pixels* avançada. Quanto maior o tamanho do passo, maior será a redução sofrida pela imagem. O avanço pode variar de implementação para implementação. Esse processo se repetirá até que o espaço da camada de entrada tenha sido recorrido.

Observa-se na Figura 11 que uma imagem de entrada com 450x450 *pixels* e uma janela convolutiva de 3x3 nos define um mapa de características com 150x150 neurônios no primeiro nível da camada oculta.

Figura 11 – Exemplo de convolução para uma janela convolutiva de 3x3.



Fonte: Adaptado de (TORRES, 2016).

Para conectar cada neurônio da camada oculta com os 9 neurônios que lhe correspondem na camada de entrada, precisamos de um valor de *bias* b e uma matriz de pesos W de tamanho 3x3. A matriz e a *bias* são compartilhadas entre todas os neurônios da camada oculta, nesse caso, para os 150x150 (22500) neurônios. Esse compartilhamento restringe a escolha de pesos sinápticos, reduzindo de maneira drástica o número de parâmetros da CNN, pois de outra forma seriam gerados 150x150x3x3 (202500) neurônios.

Uma matriz e uma *bias* definem um *kernel* (ou *filter*) (TORRES, 2016). Ainda segundo Torres (2016), um *kernel* detecta somente certa característica relevante na ima-

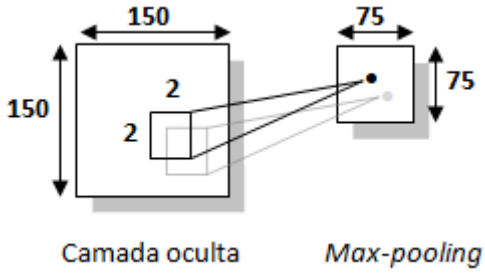
gem. Portanto, é necessário um *kernel* para cada característica que deve ser reconhecida. Isso significa que uma camada convolucional em uma CNN consiste em vários *kernels*.

2.5.2 Camada de *Pooling*

É usual que a camada de convolução seja seguida por uma camada de *pooling*, também conhecida como redução ou subamostragem. A camada de *pooling* simplesmente condensa a saída da camada convolucional - através de conexões locais - e cria uma versão compactada da informação, que servirá de entrada para a próxima camada de convolução (TORRES, 2016). Esse processo representa o campo receptivo.

Conforme Torres (2016), existem vários métodos para comprimir informação, porém o mais habitual é conhecido como *max-pooling*, que consiste em reduzir as informações apenas mantendo o valor máximo na região *kxk* considerada, como explicita a Figura 12.

Figura 12 – Exemplo de *max-pooling* para um $k = 2$.



Fonte: Adaptado de (TORRES, 2016).

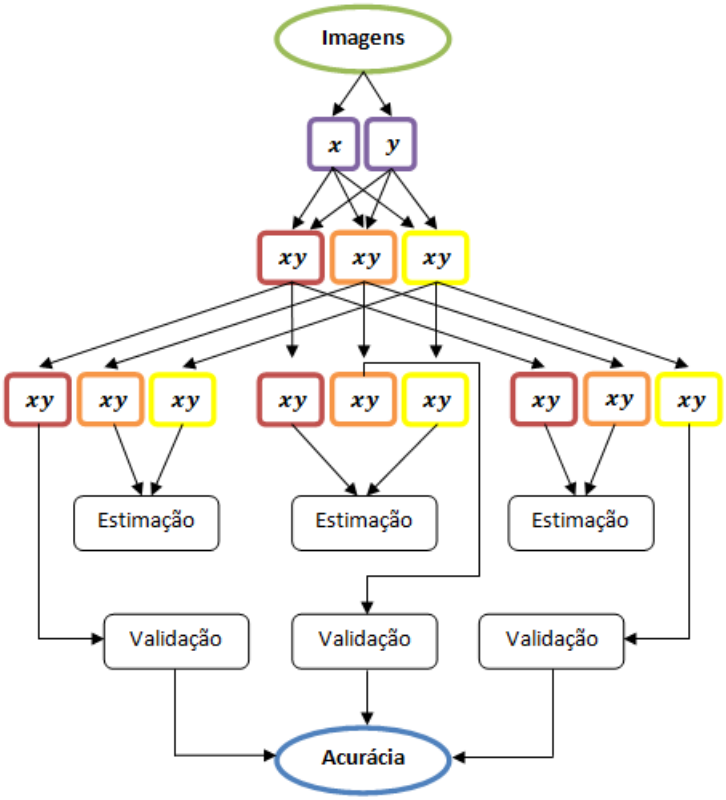
O *max-pooling* atua como uma forma de identificar se uma determinada característica encontra-se em qualquer lugar da imagem. Como a camada convolucional contém mais de um *kernel* e, cada *kernel* gera um mapa de características diferente, aplica-se então o *max-pooling* separadamente em cada mapa de características.

2.6 Validação do modelo

Kohavi (1995) define validação cruzada como uma técnica para avaliar a capacidade de generalização de um modelo, a partir de um conjunto de dados. Pretende-se dar confiabilidade ao resultado obtido, avaliando a capacidade de generalização do protótipo desenvolvido, através da aplicação da técnica estatística de validação cruzada $k - fold$.

Nessa técnica, o conjunto de dados (imagens), é igualmente dividido em k subconjuntos. O treino da CNN é feito concatenando $k-1$ subconjuntos e classificando o subconjunto restante. As fases de treino e teste são repetidas k vezes, com uma permutação circular dos subconjuntos. A acurácia, ou taxa de acerto, será então, calculada usando a média das acurácias de cada fase. A Figura 13 exemplifica o esquema de particionamento e execução do método $k-fold$, para um $k=3$, onde x representa o subconjunto de imagens de ultrassom que possuem a característica a ser reconhecida pela CNN e y o subconjunto que não possui.

Figura 13 – Exemplo do esquema de particionamento e execução do método $k-fold$ com $k=3$.



Fonte: Autor (2017).

Conforme (WITTEN, 2005), testes extensivos em numerosos bancos de dados - com diferentes técnicas de aprendizagem - tem mostrado que 10 é o número ideal de *folds* para obter-se a melhor estimativa de erro e será, portanto, utilizado nesta pesquisa.

2.7 Computação de alto desempenho

A computação paralela parte do princípio de que grandes problemas geralmente podem ser divididos em problemas menores. Esses problemas menores, por sua vez, normalmente correspondem a cálculos e podem ser resolvidos de forma simultânea (ALMASI; GOTTLIEB, 1989).

O paralelismo na resolução de problemas propicia uma vantagem no tempo de execução sobre um problema processado de forma sequencial. Essa Subseção discorre sobre o paralelismo relacionado ao uso de GPU.

2.7.1 Computação acelerada por placa de vídeo

Conforme NVIDIA (2017b), uma CPU possui em sua arquitetura alguns núcleos otimizados para o processamento serial sequencial, enquanto uma placa de vídeo tem uma arquitetura paralela que consiste em centenas de núcleos menores e mais eficientes criados para lidar com múltiplas tarefas simultaneamente.

Uma placa de vídeo é composta por diversos circuitos eletrônicos, entre os quais se encontra um processador dedicado especialmente para a renderização de gráficos em tempo real. Esse tipo de processador é chamado de GPU.

Computação acelerada por placas de vídeo é o uso de uma GPU juntamente com uma CPU para acelerar aplicativos de *deep learning*, análise, processamento e engenharia (NVIDIA, 2017b). Esse tipo de computação libera porções do aplicativo com uso intenso de computação para a placa de vídeo, enquanto o restante do código ainda é executado na CPU.

3 METODOLOGIA

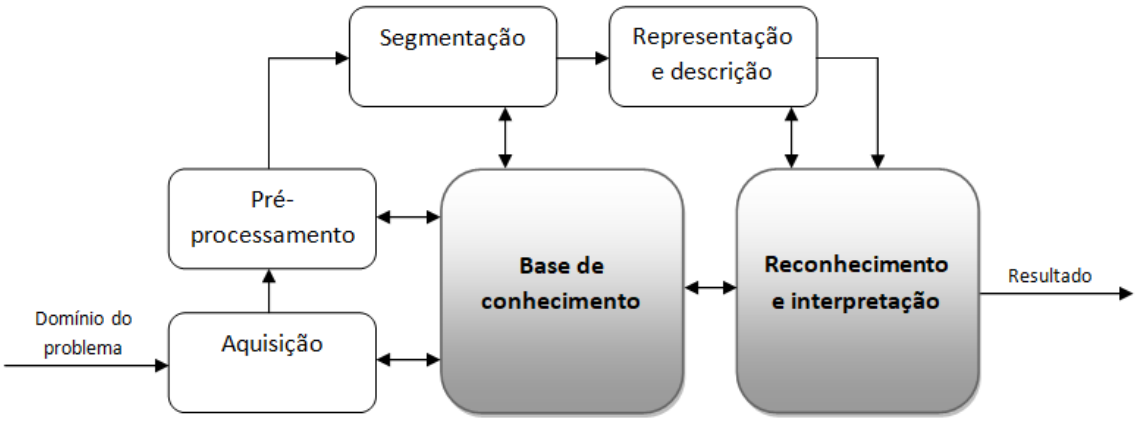
O método científico consiste em um conjunto de abordagens, técnicas e processos utilizados pela ciência para formular e resolver problemas de aquisição objetiva do conhecimento, de uma maneira sistemática (RODRIGUES, 2007). A metodologia é uma preocupação instrumental que compreende os passos teóricos e práticos realizados durante uma pesquisa científica, garantindo que qualquer pesquisador que repita a investigação, nas mesmas circunstâncias, obterá um resultado semelhante.

De acordo com Gil (1994), pesquisas científicas podem ser classificadas de acordo com a finalidade da pesquisa (básica ou aplicada), a forma de abordagem do problema (quantitativa ou qualitativa), sob a perspectiva dos objetivos do estudo (exploratória, descritiva ou explicativa), e a partir dos procedimentos técnicos a serem adotados (pesquisa bibliográfica, documental, experimental, levantamento, estudo de caso, entre outros).

Dito isto, a presente pesquisa classifica-se como de caráter aplicado, quanto à finalidade, objetivando a construção de um modelo, dirigido à solução do problema de pesquisa. Em termos de avaliação de resultado, apresenta característica quantitativa, uma vez que serão avaliados a precisão dos diagnósticos obtidos e o tempo de processamento total da solução. Em relação ao objeto de estudo - CNN - possui aspecto exploratório, proporcionando maior familiaridade com o tema. Por fim, os procedimentos técnicos adotados caracterizam a pesquisa como de cunho bibliográfico, com ênfase em artigos correlatos, experimental e estudo de caso.

Esta pesquisa é norteada pela metodologia ilustrada na Figura 14, adaptada de Gonzalez e Woods (2010), que a descreve como sendo uma representação dos passos fudamentais do PDI. Estas etapas, realizadas sobre o banco de dados, são descritas nas seções subsequentes. Também no decorrer deste capítulo definem-se os processos de engenharia de *software* e de validação do modelo desenvolvido.

Figura 14 – Fluxograma simplificado da metodologia proposta.

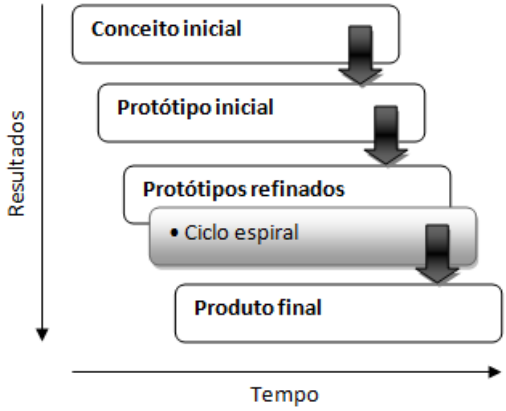


Fonte: Adaptado de Gonzalez e Woods (2010).

3.1 Processo de engenharia de *software*

A fim de alcançar um modelo de CNN que cumpra os objetivos propostos, definiu-se que o processo de engenharia de *software* empregado no seu desenvolvimento será uma variante do modelo em espiral, o modelo de prototipagem evolutiva, cujo ciclo de vida é ilustrado na Figura 15.

Figura 15 – O modelo de ciclo de vida de prototipagem evolutiva.



Fonte: Adaptado de (FILHO, 2009).

Conforme Filho (2009), neste modelo a espiral de desenvolvimento não é usado para desenvolver o produto completo, mas para construir uma série de versões provisórias que são chamadas de protótipos. Os protótipos cobrem cada vez mais aspectos, até que se atinja o produto desejado. O modelo de rede neural proposto precisará sofrer alterações para chegar ao usuário final - neste caso o profissional radiologista - como por exemplo a criação de uma interface amigável e sua expansão para identificação das massas encontradas. Neste contexto, foi descrito um documento de requisitos técnicos para execução

do modelo e posterior alteração do mesmo, disponível no Apêndice A.

3.2 Desenvolvimento

Esta Subseção explicita a implementação do código fonte da CNN proposta e as ferramentas envolvidas no seu desenvolvimento. Essa Subseção também detalha as decisões de projeto tomadas e enfatiza os métodos relacionados ao treinamento.

3.2.1 Ambiente de desenvolvimento

No que diz respeito ao ambiente de desenvolvimento, optou-se pela utilização da interface *Anaconda Navigator*, uma GUI (*Graphical User Interface*) que permite o desenvolvimento de aplicações e o gerenciamento de pacotes pertencente a elas, sem a necessidade de uso de linhas de comando (ANACONDA, 2017). Isto se deu porquê integrado a GUI, tem-se o *framework* CNTK (*Microsoft Cognitive Toolkit*), que possui suporte para as linguagens de programação *C++* , *C#*, *Python* e a linguagem de descrição *BrainScript*. Esse *toolkit* é conhecido por implementar o treinamento de CNN com grande eficiência para dados de fala, imagem e texto segundo dados da NVIDIA (2017c).

Para o desenvolvimento do código fonte, utilizou-se a ferramenta Spyder (*Scientific Python Development Environment*), disponível na interface *Anaconda Navigator*. O *Spyder* é um IDE (*Integrated Development Environment*) especializado na linguagem de programação *Python*, que permite edição, teste e depuração de código fonte (SPYDER, 2017). O restante das decisões de projeto que envolvem *software* e *hardware* utilizadas podem ser vistas na Tabela 1.

Software	**Especificação técnica**
Sistema operacional	*Windows* 10 Pro (64 *bit*)
Linguagem de programação	*Python* (versão 3.5.2)
Framework	CNTK 2.5.1 para GPU
GUI	Anaconda3 (versão 4.1.1)
IDE	*Spyder* (Integrado ao Anaconda)

***Hardware* (GPU)**	**Especificação técnica**
Modelo	GeForce GTX 1050 Ti
Cuda cores	768
Arquitetura da GPU	Pascal
Memória dedicada	16 GB

Hardware	**Especificação técnica**
Processador	Intel(R) Core(TM) i5 – 7400 CPU @$3,00GHz$
Memória RAM	4 GB

Tabela 1 – Especificações técnicas do *software* e *hardware* utilizados no desenvolvimento.
Fonte:Autor (2017).

Para mais detalhes, recomenda-se a analise do Apêndice A.

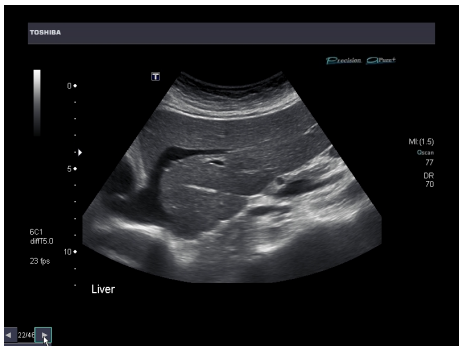
3.2.2 Processamento das imagens

Esta Subseção descreve as fases envolvidas no processamento das imagens que compõem o banco de dados deste estudo. Esta etapa é realizada antes da classificação das imagens pela rede neural e de acordo com a metodologia proposta na Figura 14.

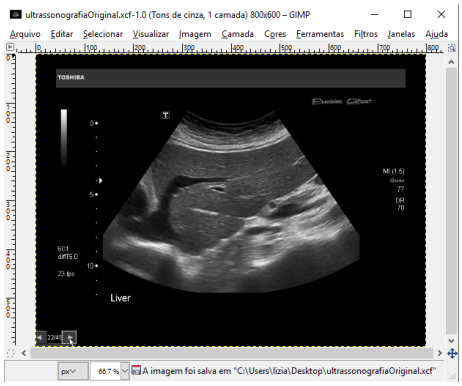
1. **Aquisição das imagens**: As imagens de ultrassom de fígados utilizadas foram obtidas através da plataforma *flickr*, um *site* de gerenciamento e compartilhamento de fotos e vídeo (FLICKR, 2017), como apresenta a Figura 17(a). Essas imagens encontram-se disponíveis para *download* nos álbuns de uma clínica especializada em ultrassonografia computadorizada e constituem o banco de dados, onde receberam números de 0 a 119. Por conveniência definiu-se que as imagens com nomes pares possuem massas e as impares não possuem (ver Figura 17).
2. **Pré-processamento**: Com o auxílio do editor gráfico *open source* GIMP (*GNU Image Manipulation Program*), as imagens obtidas tiveram seu canal de cores alterado de RGB para monocromático. Foram removidos logos e rótulos que pudessem afetar os resultados. Este processo pode ser observado na Figura 17(b).

3. **Segmentação**: Durante o processo de segmentação, ainda com auxílio da ferramenta GIMP, o tamanho em *pixels* das imagens foi redefinido, extraindo das imagens apenas a área de interesse (legendas e logos foram retirados). Essa redefinição ocorreu de forma que todas as imagens possuíssem o mesmo tamanho após o processo de edição (450x450), sem que houvesse perda de *pixels* de interesse para a aplicação, gerando a imagem de interesse (Figura 17(c)).
4. **Extração de características (representação e descrição)**: Nessa etapa, desenvolveu-se o código disponibilizado no Apêndice B, utilizando a linguagem de programação *Python*. O código visa extrair os valores de *pixel* das imagens resultantes do processo de segmentação, caracterizando com precisão as formas presentes nelas. Como essas imagens possuem um tamanho correspondente a 450x450 *pixels*, um vetor com 202500 posições é extraído para cada imagem do banco de dados.

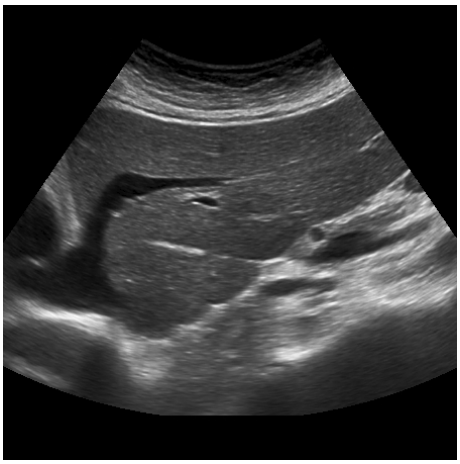
Figura 16 – Imagem ultrassonográfica durante suas etapas de processamento. (a), (b) e (c).



(a) Aquisição: Imagem original.



(b) Pré-processamento: Imagem monocromática.



(c) Segmentação: Imagem de interesse.

Fonte: Autor (2018).

A função que realiza essa extração é apresentada na Figura 17, onde *label*, *path* e *name* são entradas definidas pelo usuário e correspondem a saída esperada pela CNN, o caminho onde estão localizadas as imagens e nome do arquivo que será gerado, respectivamente.

Figura 17 – Função que carrega a imagem e transforma ela em um *array* de *pixels*.

```
def load_image(path):
    for f in os.listdir(path):
        if f.endswith('.jpg'):
            #carregar a imagem monocromática
            image = Image.open(path+'\\' + f).convert('L')
            #ler os valores dos pixels
            image_data = [image.getpixel((x, y)) for x in range(image.
                width) for y in range(image.height)]
            #transformar em array
            features = array(image_data)
            #decisão do label da imagem a ser gravada
            if ((count \% 2) == 0):
                label=label01
            else:
                label=label10
            #decisão do nome do arquivo a ser gravado
            if count < image_amount:
                name = 'Train.txt'
                print("Train")
            else:
                name = 'Test.txt'
                print("Test")
```

Fonte: Autor (2018).

Uma vez extraídos, os valores de *pixel* são gravados no campo *features* de um documento de texto, utilizando a função apresentada na Figura 18. O campo *name* define se o arquivo de texto em questão será o de teste ou treino da CNN e é passado como parametro pela função *load_image(path)*. A saída desta função é ilustrada na Figura 20(a), que corresponde ao arquivo de treino da rede neural, e Figura 20(b) ao de teste. Já o campo *label* deste documento, corresponde a saída que a imagem deve gerar na última camada de neurônios do modelo implementado de rede neural (onde 10 apresenta a característica a ser identificada e 01 não apresenta).

Para a aplicação da técnica estatística de validação cruzada $k-fold$ (ver Subeção 2.6) com um $k = 10$, as imagens foram separadas em 10 subconjuntos com 12 imagens cada (totalizando as 120 imagens do banco de dados). Destes, 9 subconjuntos foram gravados de maneira sequencial no arquivo de teste e o subconjunto restante foi gravado no arquivo de treino. Este processo ocorreu 10 vezes com uma permutação circular dos

subconjuntos, gerando 10 arquivos de treino distintos e seus respectivos arquivos de teste.

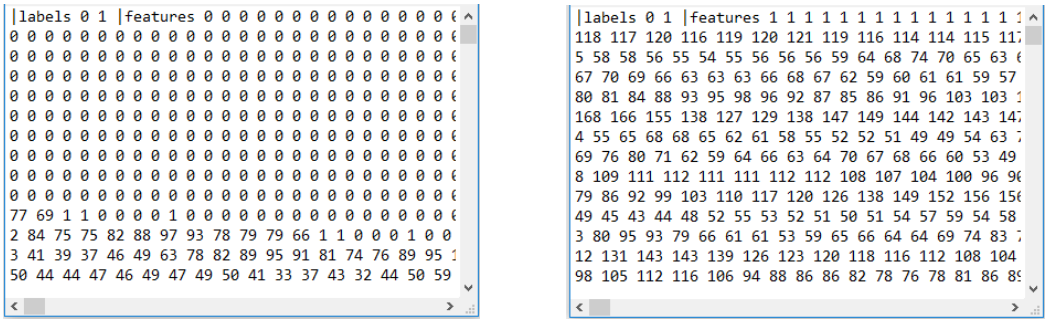
O valor da variável *image_amount* (ver Figura 17) será dada pelo número de imagens que constituem um subconjunto multiplicado pelo número de subconjuntos de treino, ou seja $12 * 9 = 108$, definindo em qual arquivo as imagens de um determinado subconjunto serão gravadas e dependendo diretamente do k definido. Neste caso 108 imagens serão gravadas no arquivo de treino e 12 no de teste.

Figura 18 – Função que recebe o *array* de *pixels* e o *label* da imagem, gravando-os em um arquivo com extensão .txt.

```
def write_txt(features, label, name):
    #abrir arquivo em formato .txt
    file = open(name,"a")
    #escrever o label da imagem
    file.write("|labels "+label)
    #escrever o array de pixels
    file.write(" |features ")
    for item in features:
        file.write("\%s " \% item)
    file.write("\n")
```

Fonte: Autor (2018).

Figura 19 – Imagem representada por seus valores de *pixel* e seu respectivo *label*.(a) e (b)



(a) Arquivo de treino gerado. (b) Arquivo de teste gerado.

Fonte: Autor (2018).

3.2.3 Classificação: reconhecimento e interpretação

Esta etapa foi realizada utilizando uma CNN, implementada em linguagem de programação *Python* com auxílio do *framework* CNTK. O modelo desenvolvido pode ser

executado tanto em um CPU quanto em uma GPU sem prejuízo a identificação das massas nas imagens de ultrassom de fígados utilizadas nos testes. Essa seleção entre *hardwares* é feita no trecho de código apresentado na Figura 20.

Figura 20 – Trecho de código que seleciona a GPU quando disponível.

```
if 'TEST_DEVICE' in os.environ:
    if os.environ['TEST_DEVICE'] == 'gpu':
        C.device.try_set_default_device(C.device.gpu(0))
    else:
        C.device.try_set_default_device(C.device.cpu())
```

Fonte: Autor (2018).

Ressalta-se que estes são trechos de maior destaque nas funções implementadas e, para uma maior compreensão do código desenvolvido, recomenda-se a análise do Apêndice C.

A função *create_reader()* é responsável por deserializar os arquivos de treino e teste gerados pelo código anterior (ver Subseção 3.2.2), utilizando-os como parâmetros de entrada padrão da CNTK, instanciada como *C*. Os parâmetros recebidos por essa função são *path*, *is_training*, *input_dim* e *num_label_classes*, que correspondem a localização dos arquivos de treino e teste, uma *flag* indicando o processo de treinamento, tamanho do vetor de *pixels* da imagem (neste caso 202500) e quantidade de possíveis saídas (possui massa ou não).

Figura 21 – Trecho da função que lê os arquivos .txt de treino e teste.

```
def create_reader(path, is_training, input_dim, num_label_classes):

    ctf = C.io.CTFDeserializer(path, C.io.StreamDefs(
          labels=C.io.StreamDef(field='labels', shape=num_label_classes
              , is_sparse=False),
          features=C.io.StreamDef(field='features', shape=input_dim,
              is_sparse=False)))
          ...
```

Fonte: Autor (2018).

O modelo de CNN é definido na função *create_model()*. Nos trechos ilustrados pela Figura 22, destacam-se a definição de uma camada convolutiva seguida de *max-pooling*. *C. layers.Convolution2D()* retorna uma função que aceita um argumento e aplica a operação de convolução sobre ele. Esse retorno é utilizado como argumento do retorno

da *C. layers.MaxPooling()* que, por sua vez, é uma função que aceita um argumento e aplica a operação de *max-pooling* sobre ele.

Após a definição das camadas de convolução e *max-pooling*, o retorno da última camada de *max-pooling* é definido como atributo da *C. layers.Dense()*. A *C. layers.Dense()* cria uma MPL, cujo número de saídas é passado como paramêtro (*num_output_classes*).

Figura 22 – Trecho da função que cria o modelo de CNN.

```
def create_model(features):
    with C.layers.default_options(init=C.glorot_uniform(), activation=C
        .relu, bias=True):
            h = features
            #definição da primeira etapa convolutiva
            h = C.layers.Convolution2D(filter_shape=(5,5), num_filters
                =2, strides=(2,2), pad=True, name='first_conv')(h)
            #definição da primeira etapa de max-pooling
            h = C.layers.MaxPooling(filter_shape=(2,2), strides=(2,2),
                name='first_maxpooling')(h)
            #adição de mais camadas convolutivas e de max-pooling...
            r = C.layers.Dense(num_output_classes, activation=None,
                name='classify')(h)
            return r
```

Fonte: Autor (2018).

Tencionando medir o quanto a função inferida pela rede se ajusta aos exemplos de treinamento, definiu-se uma função de perda, que mede a diferença (ou erro) que a rede comete no decorrer do processo de reconhecimento como pode ser observado na Figura 23. Para tal é realizada uma operação de *cross entropy* entre os *labels* e a probabilidade de predição da rede instanciada utilizando a função de ativação *softmax*.

Uma vez que estaremos usando a operação *softmax* combinada com a função de perda durante o treinamento, a camada densa final não necessita de uma função de ativação associada a ela.

Figura 23 – Função que define o critério de calculo de erro e perda.

```
def create_criterion_function(model, labels):
    loss = C.cross_entropy_with_softmax(model, labels)
    errs = C.classification_error(model, labels)
    return loss, errs
```

Fonte: Autor (2018).

Durante o processo de treinamento, chama-se a função de calculo de perda, cujo retorno servirá como um dos parâmentros de *C.Trainer()*, uma classe implementada pela CNTK que contém as funções de treinamento da rede neural (Figura 24).

Figura 24 – Trecho da função *train_test()* responsável pela inicialização da instância da classe de treinamento da ferramenta CNTK.

```
def train_test(train_reader, test_reader, model_func,
    num_sweeps_to_train_with=10):
    #abrir arquivo em formato .txt
    file = open(file_log,"a")
    #inicializar a função modelo
    model = model_func(x/255) #x é dividido por 255 para escalar os
        valores de pixel
    loss, label_error = create_criterion_function(model, y)
    #instanciar o objeto de treinamento
    learning_rate = 0.2
    lr_schedule = C.learning_rate_schedule(learning_rate, C.UnitType.
        minibatch)
    learner = C.sgd(z.parameters, lr_schedule)
    trainer = C.Trainer(z, (loss, label_error), [learner])
    ...
```

Fonte: Autor (2018).

Os exemplos utilizados para treinar a rede neural consistiram tanto de exemplos positivos como de exemplos negativos. Exemplos positivos são relativos aos dados de treinamento de entrada que contém o alvo de interesse (massas). Para atenuar o problema de reconhecimentos falsos ocasionais, foram incluídos nos dados de treinamento exemplos negativos (sem massa), para ensinar a rede a não confundir alterações com o alvo.

O arquivo de treino (Figura 20(a)) possui 108 imagens, sendo 54 de exemplos positivos e 54 de exemplos negativos, gravados de forma intercalada. O código separa os arquivos em 9 lotes de 12 imagens como explicita a Figura 25. Define-se então o mapa de entrada da rede neural, onde *y* corresponde aos *labels* e *x* as *features* do arquivo de treino.

Figura 25 – Trecho da função *train_test()* responsável pela inicialização da instância da classe de treinamento da ferramenta CNTK.

```
def train_test(train_reader, test_reader, model_func,
    num_sweeps_to_train_with=10):
    ...
    #inicializar os parâmetros de treinamento
    minibatch_size = 12 #quantidade de imagens por lote de treinamento
    num_samples_per_sweep = 108
    num_minibatches_to_train = (num_samples_per_sweep *
        num_sweeps_to_train_with) / minibatch_size
    #definição do mapa de entrada entre as características e os labels
        do arquivo de treino
    input_map={
        y  : train_reader.streams.labels,
        x  : train_reader.streams.features
    }
    ...
```

Fonte: Autor (2018).

Na Figura 26, a variável *data* recebe a cada iteração da estrutura de repetição *for* um novo lote de imagens lidas do mapa de entrada da rede neural. *Data*, por sua vez, se torna o parâmentro da função de treinamento implementada pela CNTK dentro da classe *Trainer()*.

Figura 26 – Trecho da função *train_test()* responsável pela execução do treinamento do modelo de CNN.

```
def train_test(train_reader, test_reader, model_func,
    num_sweeps_to_train_with=10):
    ...
    for i in range(0, int(num_minibatches_to_train)):
      #ler um lote de features e labels do arquivo de treinamento
      data=train_reader.next_minibatch(minibatch_size, input_map=
          input_map)
      trainer.train_minibatch(data)
    ...
```

Fonte: Autor (2018).

Quando todos as imagens do arquivo de treino tiverem sido classificados pela rede, termina-se uma época de treinamento. Foram executadas ao todo 10 épocas para cada arquivo de treino (*num_sweeps_to_train_with=10*) e uma época para o arquivo de teste. Cabe reforçar que são 10 arquivos distintos de treino e seus respectivos arquivos de teste.

Ao final da execução do processo de treinamento, se dá inicio ao processo de teste.

O arquivo de teste possui 12 imagens, sendo 6 de exemplos positivos e 6 de exemplos negativos, também gravados de maneira intercalada. O código separa define essas imagens como um lote e então, de forma análoga ao processo de treinamento, define-se um mapa de entrada para a rede neural utilizando o arquivo de teste (Figura 20(b)), como observa-se na Figura 27.

Figura 27 – Trecho da função *train_test()* responsável pela inicialização da instância da classe de teste da ferramenta CNTK.

```
def train_test(train_reader, test_reader, model_func,
    num_sweeps_to_train_with=10):
    ...
    #definição do mapa de entrada entre as características e labels do
        arquivo de teste
    test_input_map = {
        y  : test_reader.streams.labels,
        x  : test_reader.streams.features
    }

    #inicializar os parâmetros de teste
    test_minibatch_size = 12 #quantidade de imagens por lote de
        treinamento
    num_samples = 12 #quantidade de imagens totais no arquivo de
        treinamento
    num_minibatches_to_test = num_samples // test_minibatch_size  #
        quantidade de lotes de imagens usados no treinamento
    ...
```

Fonte: Autor (2018).

Na Figura 28, a variável *data* recebe o lote de imagens lidas do mapa de entrada da rede neural. *Data*, por sua vez, se torna o parâmetro da função de teste implementada pela CNTK dentro da classe *Trainer()*, que retorna para cada lote o valor de erro cometido pela rede no processo de teste. Enquanto isso, *test_result* é o somatório dos erros parciais do lote usado no treinamento.

Figura 28 – Trecho da função *train_test()* responsável pela execução do teste do modelo de CNN.

```
def train_test(train_reader, test_reader, model_func,
    num_sweeps_to_train_with=10):
    ...
    for i in range(num_minibatches_to_test):
        data = test_reader.next_minibatch(test_minibatch_size, input_map=
            test_input_map)
        eval_error = trainer.test_minibatch(data)
        test_result = test_result + eval_error
    ...
```

Fonte: Autor (2018).

Após a execução do processo de treinamento, é efetuado um calculo do erro médio de avaliação do modelo de CNN e ele é mostrado ao usuário no terminal. Por fim, tem-se a função *do_train_test*, explicitada na Figura 29, que realiza a chamada das funções de criação do modelo de CNN (*create_model()*), e leitura (*create_reader()*) dos arquivos de treino e teste (atribuidos à *reader_train* e *reader_test* respectivamente) que, posteriormente, são utilizados como parâmetros de entrada para *trein_test()*, dando inicio a mais um processo de treino e teste do modelo.

Figura 29 – Função principal, onde encontram-se as chamadas das demais funções na ordem de execução.

```
def do_train_test():
    global z
    z = create_model(x) #criação do modelo
    reader_train = create_reader(train_file, True, input_dim,
        num_output_classes) #função que lê os arquivos .txt
    reader_test = create_reader(test_file, False, input_dim,
        num_output_classes)
    train_test(reader_train, reader_test, z) #função que treina e testa
        a rede neural
```

Fonte: Autor (2018).

Este processo de execução é repetido para cada um dos 10 arquivos de treino e seus respectivos arquivos de teste gerados conforme a Subseção 3.2.2 e foi denominado "passo". Ao final deste processo obtêm-se um arquivo de texto onde estão gravados os dados obtidos para cada passo de treinamento; erro associado a cada lote do treino e erro relacionado ao lote de teste. Este arquivo pode ser observado na Figura 30. A acurácia então foi calculada através de uma média aritmética dos erros encontrados em cada uma

das execuções dos lotes de testes.

Figura 30 – Arquivo de saída gerado para cada passo.

```
| Minibatch | Loss   | Error   |
------------------------------
|  0        | 0.7031| 58.33% |
|  1        | 0.6184| 25.00% |
|  2        | 0.8770| 50.00% |
|  3        | 0.7027| 58.33% |
|  4        | 0.6910| 33.33% |
|  5        | 0.6998| 50.00% |
|  6        | 0.7092| 75.00% |
|  7        | 0.6925| 41.67% |
|  8        | 0.6764| 33.33% |
------------------------------

--> Training took 1415 sec
--> Test took 467 sec
--> Average test error: 50.00%
```

Fonte: Autor (2018).

4 RESULTADOS

A função de *pooling* reduz a dimensionalidade dos dados na rede. Essa redução é importante pois aumenta a agilidade no treinamento, mas principalmente porque cria invariância espacial. Aplicou-se a função *max-pooling* para realizar essa tarefa com um janela de 2x2, diminuindo a entrada de dados da próxima camada pela metade. Este tamanho de janela foi escolhido pois as imagens do banco de dados já são relativamente pequenas e uma janela maior acarretaria em uma grande limitação da quantidade de camadas aplicadas.

Para definição do tamanho de janela convolutiva foram testados os tamanhos 3x3 e 5x5; ambos utilizando duas camadas de convolução seguidas de *max-pooling* e passo de deslocamento 1. A janela de tamanho 3x3, além de obter um erro médio de classificação menor, possibilita a adição de mais camadas de convolução seguidas de *max-pooling* sobre a imagem de entrada. É importante ressaltar que as imagens possuem apenas 450x450 *pixels* de tamanho e que, tanto o processo convolutivo, quando o de *max-pooling* causam reduções significativas sobre elas e é preciso levar em consideração seu limite físico. Utilizou-se então uma janela convolutiva de 3x3 com um passo de deslocamento de 1 (ver Subseção 2.5.1).

Uma vez definidas as janelas convolutivas e de *max-pooling*, variou-se a quantidade de suas camadas aplicadas as imagens antes de servirem de entrada para a MLP classificatória. Para cada quantidade de camadas de convolução seguida de *max-pooling*, o processo de treino e teste com o método $k-fold$ foi repetido, tanto na CPU quanto na GPU, gerando a série de testes descritos na Tabela 2. Os resultados destes testes foram discriminados nas Tabelas 3 até 9.

Testes realizados

Código	***Hardware***	**Camadas**	**Tabela de resultados**
CPU1C	CPU	1	3
CPU2C	CPU	2	4
CPU3C	CPU	3	5
GPU1C	GPU	1	7
GPU2C	GPU	2	8
GPU3C	GPU	3	9

Tabela 2 – Testes realizados.

Fonte:Autor (2018).

A Tabela 3 explicita os resultados obtidos para o teste CPU1C. Esse teste obteve

um erro médio de 50%, ou seja, uma acurácia de 50%. O erro do subconjunto de teste foi de $41,666\%$

Teste CPU1C

K	Tempo de treino (s)	Tempo de teste (s)	Erro ($\%$)
0	3127	348	58,333
1	3129	355	41,666
2	3121	350	50,000
3	3128	355	58,333
4	3129	352	41,666
5	3121	353	41,666
6	3127	349	58,333
7	3130	352	50,000
8	3126	353	58,333
9	3122	351	41,666
Média	3126	351,8	50,000

Tabela 3 – Resultados obtidos para o teste CPU1C.
Fonte:Autor (2018).

A Tabela 4 ilustra os resultados obtidos para o teste CPU2C. Nota-se uma queda no erro médio de 50% do teste CPU1C para $34,165\%$ na execução dos testes, obtendo uma acurácia média de $65,835\%$. O erro do subconjunto de teste foi de $25,000\%$, ou seja, $75,000\%$ de acurácia.

Teste CPU2C

K	Tempo de treino (s)	Tempo de teste (s)	Erro ($\%$)
0	5349	594	41,666
1	5350	593	25,000
2	5352	595	33,333
3	5350	597	33,333
4	5346	592	33,333
5	5347	598	41,666
6	5349	593	33,333
7	5345	600	41,666
8	5348	597	33,333
9	5349	594	25,000
Média	5348,5	595,3	34,165

Tabela 4 – Resultados obtidos para o teste CPU2C.
Fonte:Autor (2018).

A Tabela 5 descreve os resultados obtidos para o teste CPU3C, onde se atingiu um erro no subconjunto de teste de 15%. Isso acarreta em uma acurácia média de 85%. Já o erro obtido no subconjunto de teste foi de $8,333\%$, ou seja, uma acurácia de $91,666\%$.

Teste CPU3C

K	**Tempo de treino (*s*)**	**Tempo de teste (*s*)**	**Erro (%)**
0	7725	858	8,333
1	7747	865	8,333
2	7738	859	25,000
3	7724	861	16,666
4	7750	867	8,333
5	7722	868	25,000
6	7769	858	25,000
7	7781	852	8,333
8	7716	863	16,666
9	7763	857	8,333
Média	7743,5	860,8	15,000

Tabela 5 – Resultados obtidos para o teste CPU3C.
Fonte:Autor (2018).

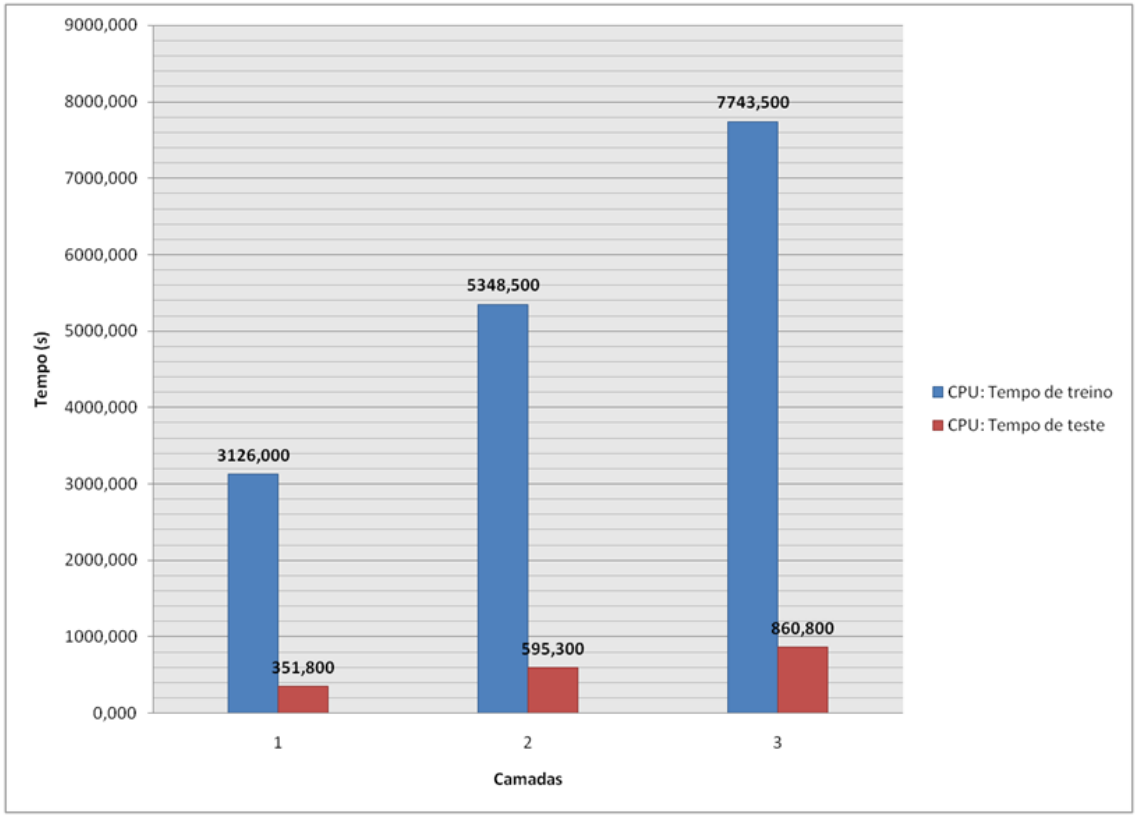
As médias alcançadas para a execução dos testes em uma CPU foram então dispostas na Tabela 6, juntamente com as médias de tempo de treino e teste em segundos (*s*). Buscando facilitar a visualização dos dados, estes resultados também podem ser examinados no Gráfico 31.

Médias da CPU

Camadas	Tempo de treino (s)	Tempo de teste (s)	*Error* (%)
1	3126,000	351,800	50,000
2	5348,500	595,300	34,165
3	7743,500	860,800	15,000

Tabela 6 – Média dos resultados obtidos utilizando a CPU.
Fonte:Autor (2018).

Figura 31 – Gráfico comparativo dos tempos de treino e teste da CPU para as diferentes quantidades de camadas.



Fonte: Autor (2018).

Analogamente ao teste CPU1C, a Tabela 7 expõem os resultados obtidos para o teste GPU1C e obtém um erro médio esperado de 50%. Realizando uma analise nos tempos de execução, nota-se que a GPU obteve tempos médios menores tanto para o processo de treino ($293,3s$) quanto no processo de teste ($34,6s$) em relação a CPU ($3126s$ e $351,8s$, respectivamente).

Teste GPU1C

K	Tempo de treino (s)	Tempo de teste (s)	Erro (%)
0	292	32	58,333
1	298	35	41,666
2	291	39	50,000
3	299	33	58,333
4	291	34	41,666
5	293	34	41,666
6	290	33	58,333
7	293	36	50,000
8	292	38	58,333
9	294	32	41,666
Média	293,3	34,6	50,000

Tabela 7 – Resultados obtidos para o teste GPU1C.
Fonte:Autor (2018).

Assim como no teste anterior a GPU possui vantagem nos tempos de execução de treino e teste, a média deles sendo $496,7s$ e $55,7s$ (Tabela 8), respectivamente, enquanto a CPU possui uma média de $5348,5s$ e $7595,3s$.

Teste GPU2C

K	Tempo de treino (s)	Tempo de teste (s)	Erro (%)
0	498	54	41,666
1	496	56	25,000
2	495	57	33,333
3	497	54	33,333
4	496	56	33,333
5	499	55	41,666
6	498	58	33,333
7	496	56	41,666
8	497	57	33,333
9	495	54	25,000
Média	496,7	55,7	34,165

Tabela 8 – Resultados obtidos para o teste GPU2C.
Fonte:Autor (2018).

Assim como nos testes anteriores a GPU apresentou melhor desempenho em ter-

mos de tempo de execução sobre a CPU. Os tempos obtidos podem ser observados na Tabela 9 e a média deles na Tabela 10.

Teste CPU3C

K	**Tempo de treino (*s*)**	**Tempo de teste (*s*)**	**Erro (%)**
0	719	79	8,333
1	716	84	8,333
2	711	80	25,000
3	712	81	16,666
4	713	82	8,333
5	712	80	25,000
6	720	83	25,000
7	714	79	8,333
8	713	82	16,666
9	715	83	8,333
Média	714,5	81,3	15,000

Tabela 9 – Resultados obtidos para o teste GPU3C.
Fonte:Autor (2018).

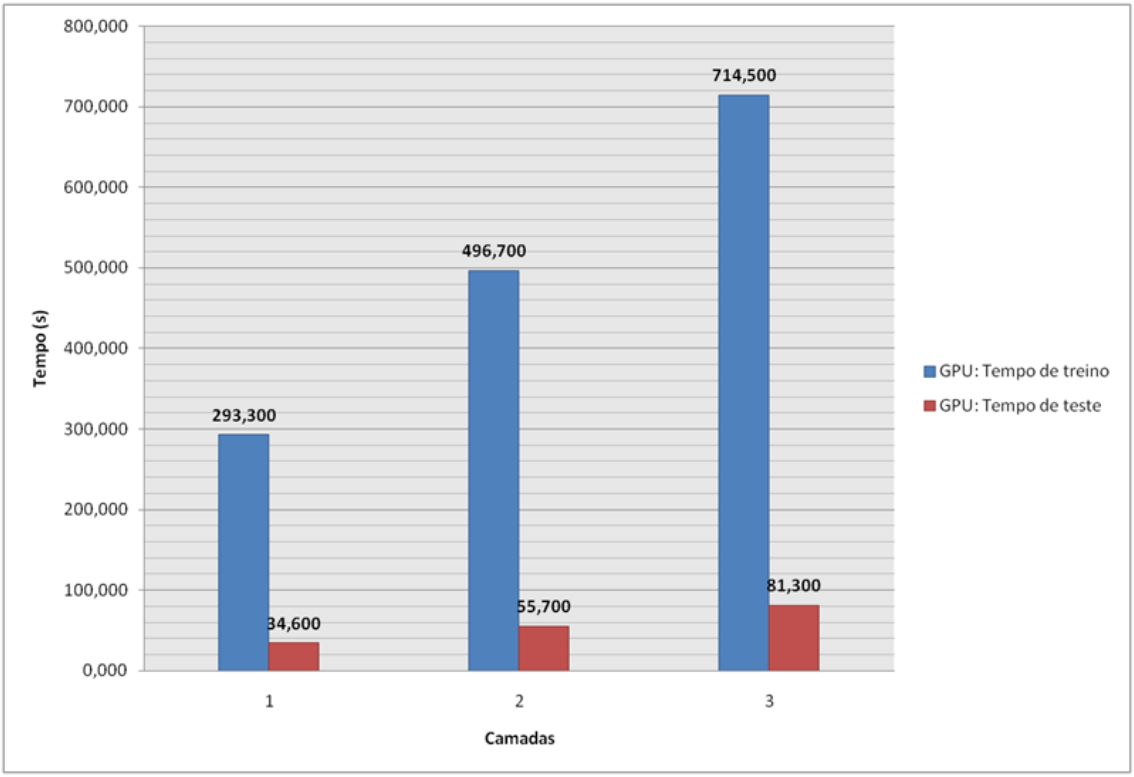
Por fim, a Tabela 10 apresenta as médias do tempos de execução de treino e teste obtidos na GPU assim como o erro médio obtido para cada teste aplicado. Objetivando uma melhor visualização dos dados obtidos, gerou-se o gráfico ilustrado na Figura 32.

Médias da GPU

Camadas	Tempo de treino (s)	Tempo de teste (s)	*Error* (%)
1	293,300	34,600	50,000
2	496,700	55,700	34,165
3	714,500	81,300	15,000

Tabela 10 – Média dos resultados obtidos utilizando a GPU.
Fonte:Autor (2018).

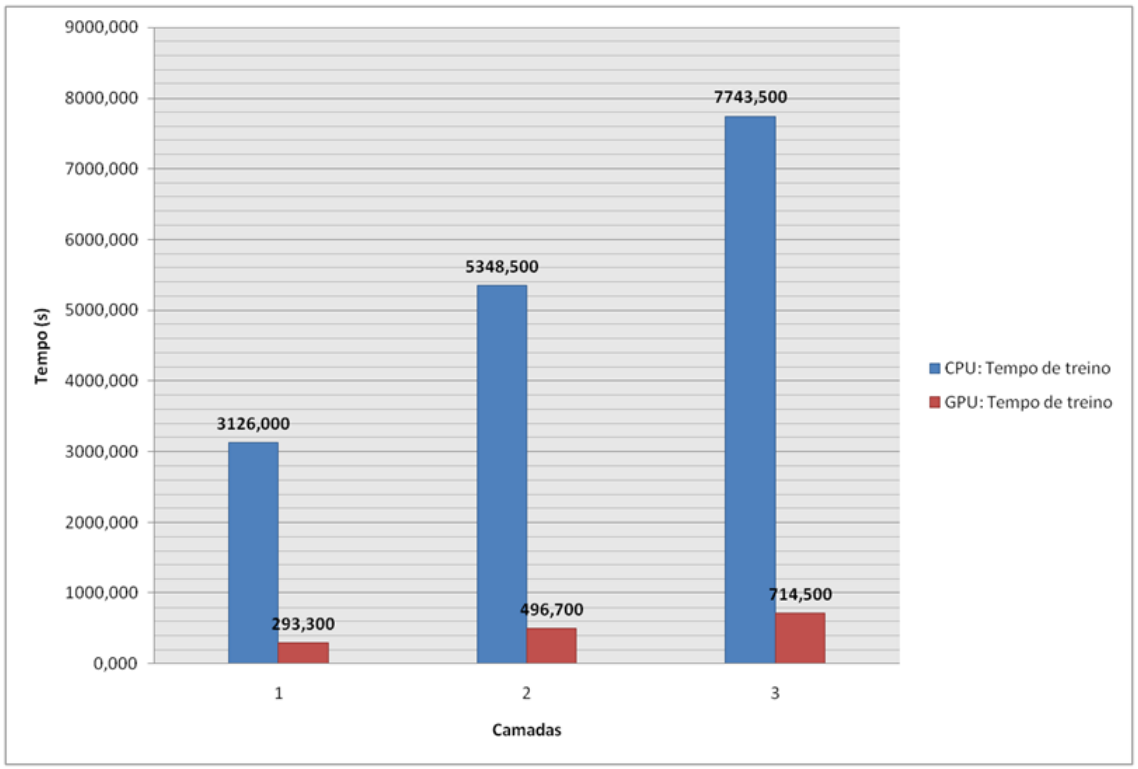
Figura 32 – Gráfico comparativo dos tempos de treino e teste da GPU para as diferentes quantidades de camadas.



Fonte: Autor (2018).

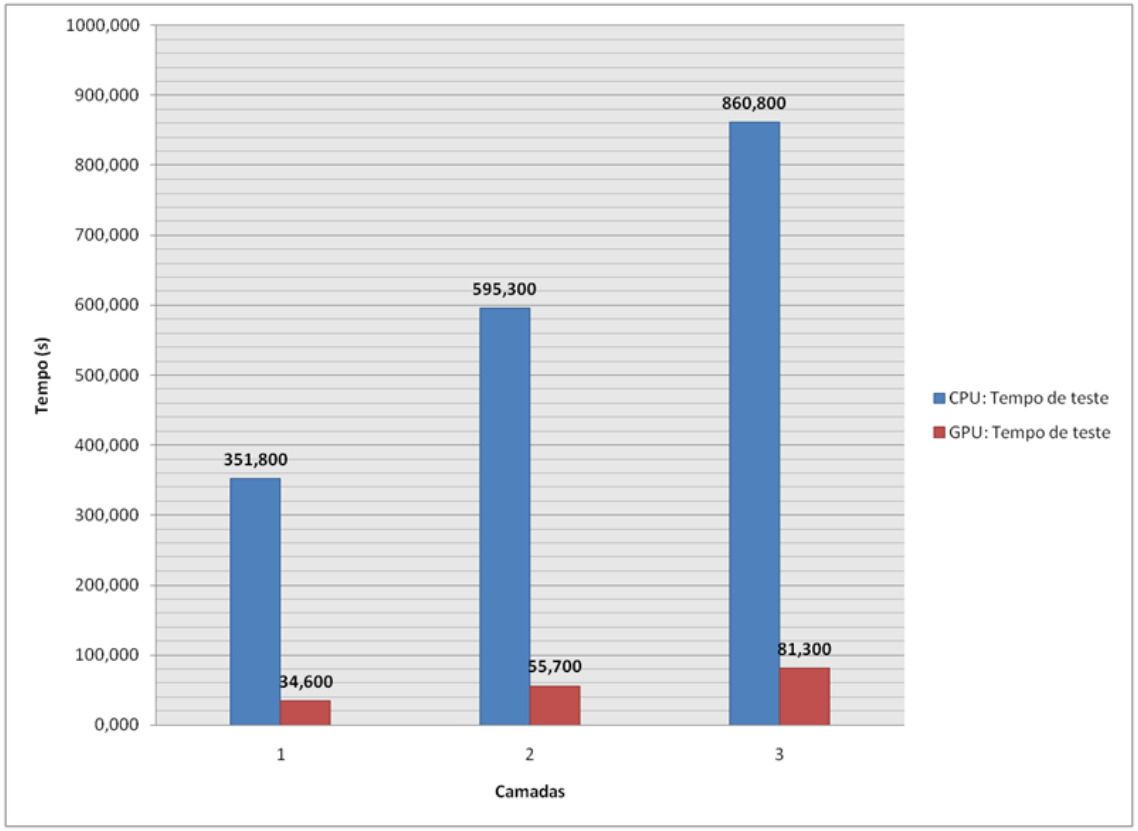
Comparando a Tabela 6 e a Tabela 10 observa-se que o percentual de erro na identificação de massas e não massas - para cada camada adicionada - é igual. Os dados apresentados nessas tabelas foram dispostos na Figura 33, que compara o tempo de treino entre a CPU e a GPU, e na Figura 34, que compara seus tempos de teste.

Figura 33 – Gráfico comparativo dos tempos de treino entre a CPU e a GPU.



Fonte: Autor (2018).

Figura 34 – Gráfico comparativo dos tempos de teste entre a CPU e a GPU.



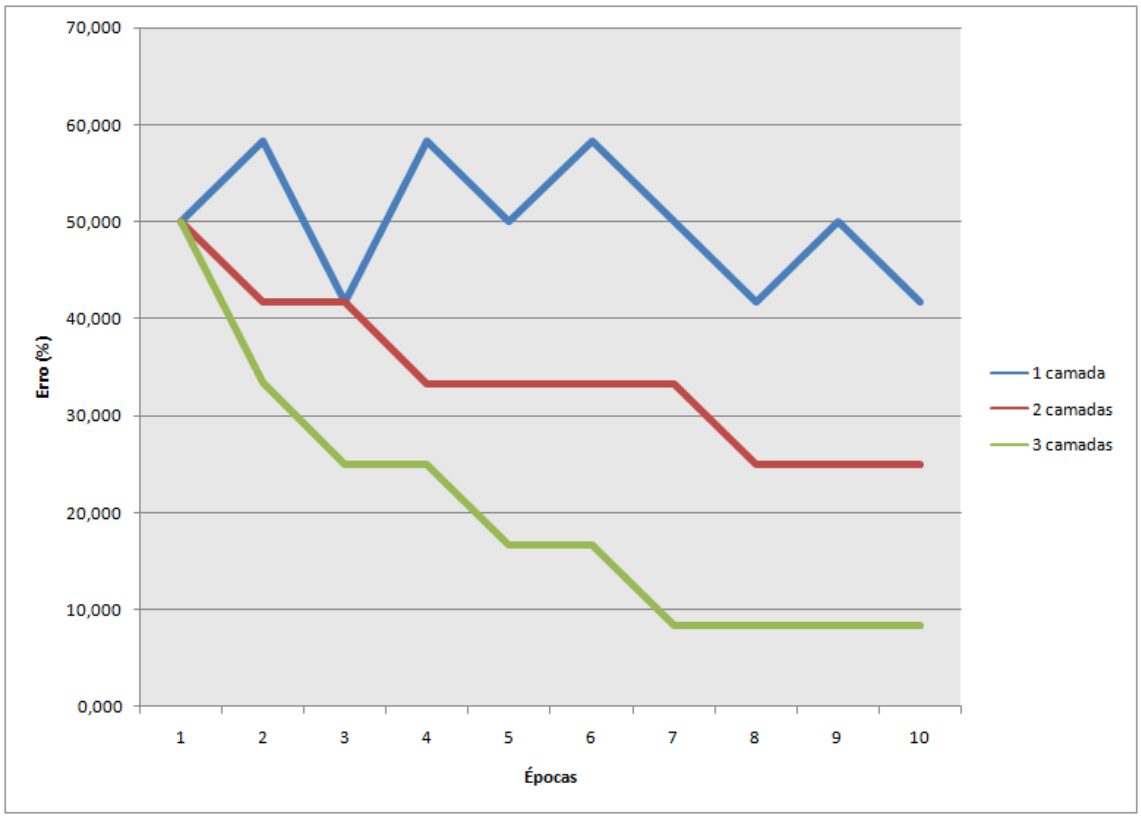
Fonte: Autor (2018).

A rede neural formada pelo modelo de CNN com apenas uma camada convolutiva seguida de *max-pooling* é constituida por 22500 neurônios de entrada e 2 de saída (possui massa ou não possui massa) e com 197346 neurônios no total. Isso ocorre porque a imagem de entrada possui um tamanho de 450*x*450 *pixels* e ao final da primeira camada de convolução temos uma redução para 150*x*150 *pixels* que, por sua vez, servem de entrada para o *max-pooling* onde a redução espacial gera uma imagem com 75*x*75 *pixels*. A saída da função de *max-pooling* serve então de entrada para a rede neural.

O modelo de CNN com apenas duas camadas convolutivas seguidas de *max-pooling* possui 5625 neurônios de entrada e 2 de saída com 12634 neurônios no total. Os 75*x*75 *pixels* que, no modelo anterior, serviram de entrada para a rede neural agora viram parâmetros para a próxima camada convolutiva, cuja saída possui 25*x*25 *pixels*. Analogamente ao modelo anterior, os 25*x*25 *pixels* se tornam entrada para a função de *max-pooling*, cuja saída então serve de entrada a MLP. A CNN com três camadas convolutivas seguidas de *max-pooling* sofre o mesmo processo das redes anteriores, possuindo 10322 neurônios no total.

É possível então gerar um gráfico para acompanhar o aprendizado da rede para cada k do método $k - fold$ executado, este progresso é ilustrado na Figura 35.

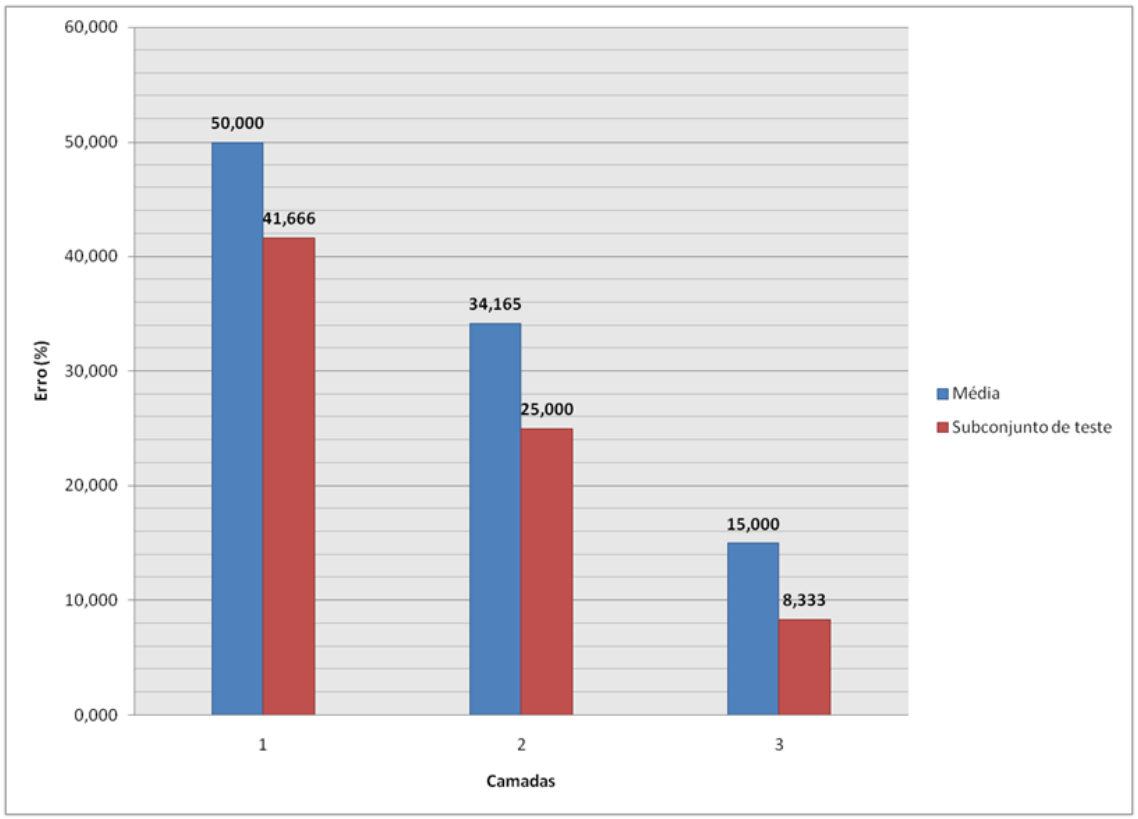
Figura 35 – Gráfico de aprendizado das redes neurais propostas para a execução do K=9.



Fonte: Autor (2018).

Para os modelos de CNN propostos, possuindo uma, duas e três camadas convolutivas seguidas de *max-pooling*, alcançou-se uma acurácia média de 50%, 65,835% e 85% respectivamente. Já para os subconjuntos de teste as acurácias encontradas correspondem a 58,666%, 75% e 91,666%. A Figura 36 explicita essa diferença de acurácia comparando os erros médios e os erros do subconjunto de teste para cada quantidade de camadas.

Figura 36 – Gráfico comparativo entre os erros médios e os erros obtidos para o subconjunto de teste.



Fonte: Autor (2018).

Uma rede neural com 58,666% de acurácia não é viável, pois age como se estivesse obtendo a saída de forma aleatória ao invés de aprender as características que levam a classificação das imagens em "possui massa"e "não possui massa".

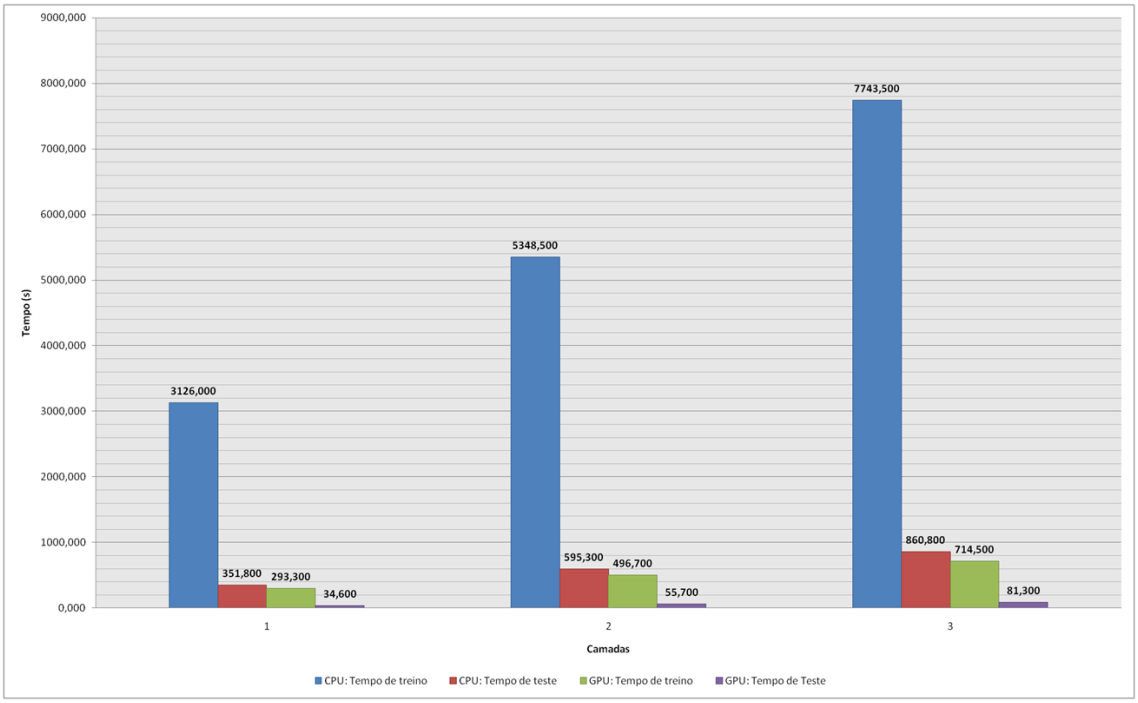
Já para a rede com 75% de acurácia nota-se que, o aumento de camadas contribuiu no aprendizado da rede neural. O valor obtido ainda não foi o desejável portanto repetiu-se o processo de treinamento com mais uma camada. Observou-se então uma melhoria na precisão para 91,666%. Devido as limitações de tamanho das imagens do banco de dados, não é possível aplicar mais uma camada de convolução e *max-pooling*.

Portanto conclui-se que, para o banco de dados utilizado, a capacidade de generalização do terceiro modelo proposto foi a melhor obtida, utilizando como métrica o

método de validação cruzada $k - fold$.

Fazendo uma análise nos tempos de execução, nota-se que o processo de treino levou em média $8,85$ vezes o tempo de teste das redes neurais implementadas. Também pode ser observado - tanto para o caso de treino quanto para o de teste - que a GPU obteve uma vantagem significativa sobre a CPU (cerca de $10,6$ vezes mais rápida), mesmo tratando-se de um banco de dados relativamente pequeno, constituído de 120 imagens com 202500 *pixels* cada. A Figura 37 explicita esta vantagem dos tempos de treino e teste obtidos para a GPU sobre a CPU.

Figura 37 – Gráfico comparativo dos tempos de treino e teste entre a CPU e a GPU.



Fonte: Autor (2018).

A partir disto afirma-se que, caso haja um aumento no tamanho das imagens do banco de dados ou na quantidade delas, o uso de um ambiente de alto desempenho como a GPU torna-se crucial na execução do processo de treinamento da rede neural.

5 CONSIDERAÇÕES FINAIS

De acordo com o Instituto Nacional do Câncer (2018), massas originárias no fígado podem ser hepatocarcinoma (também conhecido como carcinoma hepatocelular), colangiocarcinoma (que acomete os ductos biliares dentro do fígado), angiossarcoma (tumor no vaso sanguíneo) e, em crianças, o hepatoblastoma. O modelo de CNN desenvolvido foi capaz de realizar a identificação dessas massas em imagens ultrassonográficas de fígado com uma precisão média de 85%, para o conjunto de imagens analisados. É possível expandir esse modelo de CNN para que, além de classificar as massas e não massas, ele consiga identificá-las.

Para que a identificação ocorra, deve-se expandir o número de imagens no banco de dados e atribuir um *label* único a cada tipo de massa. Deve-se manter também um conjunto de imagens de fígados saudáveis com *label* próprio, objetivando evitar falsos reconhecimentos. Esse aumento no banco de dados levará a um aumento no tempo de treinamento e de teste da CNN, tornando a utilização de um ambiente de alta capacidade de processamento indispensável.

Uma vez que o processo de treino e teste seja concluído, é possível desenvolver uma interface gráfica que permita ao usuário submeter ultrassonografias de fígado para análise, obtendo-se uma ferramenta de auxílio diagnóstico, que pode ser utilizada para fins educacionais, colaborando com o treinamento e qualificação de estudantes da área médica.

Como trabalho futuro também propõem-se verificar o desempenho, em termos de tempo de execução, dos processos de treino e teste da rede neural utilizando outras arquiteturas de CPU e GPU.

REFERÊNCIAS

ADAMS, D. **O Guia do Mochileiro das Galáxias**. Rio de Janeiro, BR: Sextante, 2004. 180 p.

ALMASI, G. S.; GOTTLIEB, A. **Highly Parallel Computing**. California, USA: Benjamin-Cummings Publishing Co., Inc., 1989. ISBN 0-8053-0177-1.

AMBRÓSIO, P. E. Redes neurais artificiais no apoio ao diagnóstico diferencial de lesões intersticiais pulmonares. Universidade de São Paulo, São Paulo, BR, junho 2002.

AMBRÓSIO, P. E. Redes neurais auto-organizáveis na caracterização de lesões intersticiais de pulmão em radiografia de tórax. Universidade de São Paulo, São Paulo, BR, junho 2007.

ANACONDA. **Documentation: What is Anaconda Navigator?** 2017. [Online; acessado em 20-Maio-2017]. Disponível na Internet: <https://docs.continuum.io/anaconda/navigator/>.

BALDNER, F. et al. **Metrologia por Imagem**. Elsevier Editora Ltda, 2017. ISBN 9788535272598. Disponível na Internet: <https://books.google.com.br/books?id=Bq04DwAAQBAJ>.

CÂNCER, I. N. do. **Câncer de Fígado**. 2018. [Online; acessado em 11-Outubro-2018]. Disponível na Internet: <http://www.inca.gov.br/conteudo_view.asp?id=330>.

CORRÊA, J. R. A evolução da fotografia e uma análise da tecnologia digital. Universidade Federal de Viçosa, Minas Gerais, BR, maio 2013.

FERNANDES, B. J. T. Redes neurais com extração implícita de características para reconhecimento de padrões visuais. Universidade Federal de Pernambuco, Pernambuco, BR, julho 2013.

FILHO, O. M.; NETO, H. V. **Processamento Digital de Imagens**. Rio de Janeiro, BR: Brasaport, 1999. 624 p.

FILHO, W. de P. P. **Engenharia de Software Fundamentos, Métodos e Padrões**. Rio de Janeiro, BR: LTC, 2009. 1248 p.

FLICKR. **Sobre o Flickr**. 2017. [Online; acessado em 05-Junho-2017]. Disponível na Internet: <https://www.flickr.com/about>.

GIL, A. C. **Métodos e técnicas de pesquisa social**. São Paulo, BR: Atlas, 1994. 200 p.

GONZALEZ, R. C.; WOODS, R. E. **Processamento Digital de Imagens**. São Paulo, BR: Pearson, 2010. 624 p.

HAYKIN, S. **Redes Neurais: Princípios e prática**. São Paulo, BR: Bookman, 2001. 908 p.

KEALY, K. J.; MCALLISTER, H.; GRAHAM, J. **Radiologia e ultrassonografia do cão e do gato**. Rio de Janeiro, BR: Elvevier, 2012. 594 p.

KOHAVI, R. A study of cross-validation and bootstrap for accuracy estimation and model selection. In: ARBIB, M. A. (Ed.). **International joint Conference on artificial intelligence**. [S.l.: s.n.], 1995. v. 14, p. 1137–1145.

LECUN, Y.; BENGIO, Y. Convolutional networks for images, speech and neural networks. In: ARBIB, M. A. (Ed.). **The Handbook of Brain Theory and Neural Networks**. Cambridge, USA: M.I.T. Press, 2003. v. 2, p. 276–279.

MARCOMINI, K. D. Aplicação de modelos de redes neurais artificiais na segmentação de nódulos em imagens de ultrassonografia de mama. Universidade de São Paulo, São Paulo, BR, março 2013.

MASCARENHAS, N. A.; VELASCO, F. R. D. **Processamento Digital de Imagens**. São Paulo, BR: INPE, 1984. 630 p.

MENESES, A. A. de M. et al. Análise de imagens médicas através de sistemas computacionais inteligentes para apoio ao diagnóstico clínico. outubro-dezembro 2008.

MIRANDA, F. A.; FREITAS, S. R. C. de; FACCION, P. L. Integração e interpolação de dados de anomalias ar livre utilizando-se a técnica de rna e krigagem. In: **Boletim de Ciências Geodésicas**. [S.l.]: UFPR, 2009. p. 428–443.

MITCHELL, T. M. **Machine Learning**. Boston, USA: WCB/McGraw-Hill, 1997. 414 p. (McGraw-Hill Series in Computer Science, v. 2).

NVIDIA. **Computação Acelerada: Solucionando os desafios mais importantes do mundo**. 2017. [Online; acessado em 11-Maio-2017]. Disponível na Internet: <http://www.nvidia.com.br/object/what-is-gpu-computing-br.html>.

NVIDIA. **Deep Learning Frameworks**. 2017. [Online; acessado em 11-Maio-2017]. Disponível na Internet: <https://developer.nvidia.com/deep-learning-frameworks>.

NVIDIA. **O que é Computação Acelerada por Placas de Vídeo?** 2017. [Online; acessado em 11-Maio-2017]. Disponível na Internet: <www.nvidia.com.br/object/what-is-gpu-computing-br.html>.

OLIVEIRA, F. P. M. de. **Emparelhamento e Alinhamento de Estruturas em Visão Computacional: Aplicações em Imagens Médicas**. Tese de Doutorado — Faculdade de Engenharia da Universidade de Porto, 2009.

REZENDE, S. O. **Sistemas inteligentes**. São Paulo, BR: Manole, 2003. 526 p.

RODRIGUES, W. C. Metodologia científica. FAETEC/IST, Rio de Janeiro, BR, 2007.

SAITO, P. T. M. Otimização do processamento de imagens médicas utilizando a computação paralela. Centro Universitário Eurípides de Marília, São Paulo, BR, novembro 2007.

SPYDER. **Spyder - Documentation**. 2017. [Online; acessado em 05-Junho-2017]. Disponível na Internet: <https://pythonhosted.org/spyder/>.

TORRES, J. **Hello World en Tensor Flow - para iniciarse en la programación del Deep Learning**. Barcelona, ES: Undertile - BSC and UPC Barcelona Tech, 2016. 150 p.

VARGAS, A. C. G.; PAES, A.; VASCONCELOS, C. N. Um estudo sobre redes neurais convolucionais e sua aplicação em detecção de pedestres. Universidade Federal Fluminense, Rio de Janeiro, BR, 2016.

WITTEN, E. F. I. H. **Data mining : practical machine learning tools and techniques**. San Francisco, USA: Elsevier, 2005. 558 p.

APÊNDICE A — DOCUMENTO DE REQUISITOS

UNIVERSIDADE FEDERAL DO PAMPA

CURSO DE BACHARELADO EM ENGENHARIA DE COMPUTAÇÃO

DOCUMENTO DE REQUISITOS:

IDENTIFICAÇÃO DE MASSAS EM IMAGENS DE ULTRASSONOGRAFIA DE FÍGADO UTILIZANDO REDES NEURAIS CONVOLUCIONAIS EM AMBIENTE DE ALTO DESEMPENHO.

AUTOR: LIZIANE ZERBIN GALLERT

ORIENTADOR: SANDRO DA SILVA CAMARGO

COORIENTADOR: LEONARDO BIDESE DE PINHO

BAGÉ, 2018

1. Propósito do Documento

Este documento contém a especificação de requisitos funcionais e não funcionais de um modelo de CNN (*Convolutional Neural Network*), para identificação de massas em ultrassonografias de fígado.

2. Descrição Simplificada do Sistema

Implementou-se um modelo de rede neural baseado em CNN que realiza a análise de imagens ultrassonográficas de fígado, identificando a presença de massas ou não nas mesmas. A CNN pode ser executada tanto em uma máquina com CPU (*Central Process Unit*) quanto em um ambiente com alta capacidade de processamento que utilize GPU (*Graphics Processing Unit*). A validação da acurácia dos resultados obtidos acontece através da técnica estatística de validação cruzada *k-fold*.

3. Requisitos

Esta Seção detalha os requisitos funcionais e não funcionais definidos para a CNN.

3.1. Requisitos Funcionais

RF 1 – Arquivos de entrada

Descrição: As imagens precisam ser transformadas em *bitmaps* e gravadas em um arquivo com extensão ".txt", sendo antecedidas por *labels* com seus respectivos diagnósticos. Define-se que os *labels* serão "1 0" para imagens que possuem massa e "0 1" para imagens que não as possuem. Devem ser gravados um arquivo para o treino (Traint.txt) e outro para o teste (Test.txt) da rede neural. Para realização deste processo, desenvolveu-se um código *Python*.

RF 2 – Rede neural

Descrição: A rede neural é capaz de receber os arquivos de entrada gerados (treino e teste), onde se encontram os *bitmaps* das imagens e seus *labels*. A rede implementa um código de treinamento com base nas imagens do arquivo "Train.txt" que realiza ajustes nos valores de *bias* até que se atinja taxas de erro de identificação aceitáveis.

Logo após o treinamento, dá-se inicio o processo de teste da rede que deve ser executado tomando por entrada o arquivo "Test.txt". Este processo de treino e teste pode ser realizado tanto em uma CPU quanto em uma GPU sem perdas na identificação das massas.

RF 3 – Validação dos resultados

Descrição: A capacidade de generalização da CNN desenvolvida deve ser avaliada através da aplicação da técnica estatística de validação cruzada *k-fold*, utilizando um k com valor 10.

Nessa técnica, o conjunto de dados (imagens), é igualmente dividido em k subconjuntos. O treino da CNN é feito concatenando k-1 subconjuntos e classificando o subconjunto restante. As fases de treino e teste são repetidas k vezes, com uma permutação circular dos subconjuntos. A acurácia, ou taxa de acerto, será então, calculada usando a média das acurácias de cada fase.

RF 4 - Apresentação dos erros de identificação de massas

Descrição: Apresentar visualmente, em um arquivo chamado "Log.txt", os erros de identificação de massas e não massas nos processos de teste e treino, para cada k do método de validação *k-fold*.

RF 5 - Apresentação dos tempos de treino e teste

Descrição: Também constam no arquivo "Log.txt", os tempos de execução de teste e treino para cada k do método de validação *k-fold*.

RF 6 - Apresentação da rede neural

Descrição: Apresentar visualmente durante a execução da rede neural, a quantidade de neurônios e camadas utilizadas pela rede neural, assim como os resultados parciais de cada k do método de validação *k-fold*.

3.2. Requisitos Não Funcionais

RFN 1 – *Software*

Descrição: No que diz respeito ao ambiente de desenvolvimento do modelo de rede neural, utilizou-se a interface *Anaconda Navigator*, uma GUI (*Graphical User Interface*) que permite o desenvolvimento de aplicações e o gerenciamento de pacotes pertencente a elas. Integrado a GUI, tem-se o *framework* CNTK (Microsoft *Cognitive Toolkit*), que possui suporte para as linguagens de programação C++, C#, *Python* e a linguagem de descrição *BrainScript*.

Para o desenvolvimento do código fonte, utilizou-se a ferramenta *Spyder* (*Scientific Python Development Environment*), disponível na interface *Anaconda Navigator*. O *Spyder* é um IDE (*Integrated Development Environment*) especializado na linguagem de programação *Python*, definida com linguagem de desenvolvimento do modelo. O restante das decisões de projeto que envolvem *softwares* utilizados no desenvolvimento podem ser vistos na Tabela 1.

Tabela 1: Requisitos não funcionais de *software*.

Software	**Especificação técnica**
Sistema operacional	*Windows* (64 *bit*)
Linguagem de programação	*Python* (versão 3.5.2)
Framework	CNTK 2.5.1 para GPU
GUI	*Anaconda3* (versão 4.1.1)
IDE	*Spyder* (Integrado ao *Anaconda*)

Fonte: Autor (2018).

RFN 2 – *Hardware* CPU

Descrição: A Tabela 2 apresenta os requisitos mínimos da máquina que contém a CPU para a execução do modelo de CNN desenvolvido.

Tabela 2: Requisitos não funcionais de *hardware* para o uso de CPU.

Hardware	**Especificação técnica**
Processador	*Inter(R) Core(TM) i*5-7400 CPU *@3,00GHz*
Memória RAM	4 GB

Fonte: Autor (2018).

RFN 3 – *Hardware* GPU

Descrição: Os requisitos mínimos da máquina que contém a GPU para a execução do modelo de CNN desenvolvido podem ser observados na Tabela 3.

Tabela 3: Requisitos não funcionais de *hardware* para o uso de GPU.

Hardware	**Especificação técnica**
Modelo	*GeForce* GTX 1050 Ti
Cuda cores	768
Arquitetura	Pascal
Memória dedicada	16 GB

Fonte: Autor (2018).

APÊNDICE B — CÓDIGO FONTE EM *PYTHON*: CONVERSÃO DAS IMAGENS EM *BITMAPS*

```
# -*- coding: utf-8 -*-
"""
@author: liziane gallert
"""
from PIL import Image
from numpy import array
import os

#label da imagem: 1 0 -> possui a característica (imagens pares); 0 1
    -> não possui a característica (imagens impares)
label01 = str("0 1")
label10 = str("1 0")
#path caminho onde se encontram as imagens
path = '.\\Image\\00'
#tamanho do k relacionado ao método de treinamento k-n fold
k = 10
#quantidade de imagens no banco de dados
image_num = 120
n = image_num / k

#função que carrega a imagem do computador e transforam ela em um array
     de pixels
def load_image(path):
    count = 0
    image_amount = n * (k - 1)
    for f in os.listdir(path):
        print(f)
        if f.endswith('.jpg'):
            #carregar a imagem monocromatica
            image = Image.open(path+'\\' + f).convert('L')
            #ler os valores dos pixels
            image_data = [image.getpixel((x, y)) for x in range(image.
                width) for y in range(image.height)]
            #transformar em array
            features = array(image_data)
            #mostrar a imagem
```

```
34              #image.show()
                #decisão do label da imagem a ser gravada
36              if ((count \% 2) == 0):
                    label=label01
38              else:
                    label=label10
40              #decisão do nome do arquivo a ser gravado
                if count < image_amount:
42                  name = 'Train.txt'
                    print("Train")
44              else:
                    name = 'Test.txt'
46                  print("Test")
                count = count + 1
48              #chamada da função que grava o array de pixels e o label da
                    imagem em um arquivo com extenção.txt
                write_txt(features, label, name)
50


52  #função que recebe o array de pixels da imagem e o label da imagem,
        gravando eles em um arquivo com extenção .txt
    def write_txt(features, label, name):
54      #abrir arquivo em formato .txt
        file = open(name,"a")
56      #escrever o label da imagem
        file.write("|labels "+label)
58      #escrever o array de pixels
        file.write(" |features ")
60      for item in features:
            file.write("\%s " \% item)
62      file.write("\n")


64
    #chamada da função que carrega a imagem e transforma ela em um array de
        pixels
66  features = load_image(path)

68  print("Arquivos gravados!")
    #mostra o array de pixels da imagem que foi gravado
70  #print("Array (pixels): \n", features)
```

APÊNDICE C — CÓDIGO FONTE EM *PYTHON*: REDE NEURAL CONVOLUCIONAL

```
# -*- coding: utf-8 -*-
"""
@author: liziane gallert
"""
from __future__ import print_function #Use a function definition from
    future version (say 3.x from 2.7 interpreter)
import numpy as np
import os
import time

import cntk as C

if 'TEST_DEVICE' in os.environ:
    if os.environ['TEST_DEVICE'] == 'gpu':
        C.device.try_set_default_device(C.device.gpu(0))
    else:
        C.device.try_set_default_device(C.device.cpu())

np.random.seed(0)

#Definição do tamanho das entradas de dados
input_dim_model = (1, 450, 450)  #a imagem tem formato 450x450 pixels
    com 1 canal de cor (monocromática)
input_dim = 450*450              #tamanho do vetor de pixels da imagem
num_output_classes = 2           #quantidade de possíveis saídas (massa
    ou não massa)

x = C.input_variable(input_dim_model)
y = C.input_variable(num_output_classes)

#Definição dos nomes dos arquivos de entrada e saída de dados
file_train = "Train.txt" #nome do arquivo de treino gerado com o código
    anterior
file_test = "Test.txt"   #nome do arquivo de teste gerado com o código
    anterior
file_log = "Log.txt"     #arquivo onde ficarão salvos
```

```
32
#Função que lê os arquivos .txt de treino e teste
34 def create_reader(path, is_training, input_dim, num_label_classes):

36     ctf = C.io.CTFDeserializer(path, C.io.StreamDefs(
           labels=C.io.StreamDef(field='labels', shape=num_label_classes
               , is_sparse=False),
38         features=C.io.StreamDef(field='features', shape=input_dim,
               is_sparse=False)))

40     return C.io.MinibatchSource(ctf,
           randomize = is_training, max_sweeps = C.io.INFINITELY_REPEAT if
               is_training else 1)
42
data_found=False #flag que indica se os .txt de treino e teste foram
    encontrados
44
for data_dir in [os.path.join(".")]:
46
    train_file=os.path.join(data_dir, file_train)
48  test_file=os.path.join(data_dir, file_test)

50  if os.path.isfile(train_file) and os.path.isfile(test_file):
        data_found=True #se os dois arquivos forem encontrados, altera-
            se a flag
52      break

54 if not data_found:
    print("\n")
56  raise ValueError("--> Error: Train/Test file not found")

58 print("\n")
print("--> Data directory: {0}".format(data_dir))
60 print("\n")

62 #Função que cria o modelo de CNN
def create_model(features):
64  with C.layers.default_options(init=C.glorot_uniform(), activation=C
        .relu, bias=True):
            h = features
66          #definição da primeira etapa convolutiva e suas
```

```
                características
            h = C.layers.Convolution2D(filter_shape=(5,5), num_filters
                =8, strides=(2,2), pad=True, name='first_conv')(h)
            #definição da primeira etapa de max-pooling e suas
                características
            h = C.layers.MaxPooling(filter_shape=(5,5), strides=(2,2),
                name='first_maxpooling')(h)

            #adição de mais camadas convolutivas e de max-pooling de
                acordo com o teste realizado
            #...

            #defininição da etapa de aprendizado (multi-camadas)
            r = C.layers.Dense(num_output_classes, activation=None,
                name='classify')(h)

            #adição de mais camadas de neurônios
            #...

            return r #(features) -> (r)

z = create_model(x) #criação do modelos

print("\n")
print("--> First convolutional layer (filters or kernels, width, height
    ):", z.first_conv.shape)
print("--> First maxpooling layer (filters or kernels, width, height):"
    , z.first_maxpooling.shape)

#adição de mais prints conforme a quantidade de camadas convolutivas do
     teste
#...

print("\n")

C.logging.log_number_of_parameters(z) #número de parâmetros na rede
    neural

#Função que define o critério de cálculo de erro e perda
def create_criterion_function(model, labels):
    loss = C.cross_entropy_with_softmax(model, labels)
```

```
98      errs = C.classification_error(model, labels)
        return loss, errs #(model, labels) -> (loss, error metric)
100
    #Função que imprime os resultados parciais do treinamento
102 def print_training_progress(trainer, mb, frequency, verbose, header):
        #abrir arquivo em formato .txt
104     file = open(file_log,"a")

106     training_loss = "NA"
        eval_error = "NA"
108
        if mb\%frequency == 0:
110         training_loss = trainer.previous_minibatch_loss_average
            eval_error = trainer.previous_minibatch_evaluation_average
112         if header:
                print("\n")
114             print("| Minibatch | Loss   | Error   |")
                print("-----------------------------------")
116
                file.write("| Minibatch | Loss   | Error   |\n
                    -----------------------------------\n")
118         if verbose:
                print("| {0:2}         | {1:.4f}| {2:.2f}\% |".format(mb,
                    training_loss, eval_error*100))
120
                file.write("| {0:2}         | {1:.4f}| {2:.2f}\% |\n".format
                    (mb, training_loss, eval_error*100))
122
        return mb, training_loss, eval_error #(trainer, mb, frequency,
            verbose, header) -> (loss, error metric)
124
    def train_test(train_reader, test_reader, model_func,
        num_sweeps_to_train_with=1):
126     #abrir arquivo em formato .txt
        file = open(file_log,"a")
128
        #----------------------TRAIN
130
        #Inicializar a função modelo. x é a variável com as dimensões do
            modelo
132     model = model_func(x/255) #x é dividido por 255 para escalar os
```

```
    valores de pixel entre 0 e 1

loss, label_error = create_criterion_function(model, y) #chamada da
    função de cálculo de erro e perda

#Instanciar o objeto de treinamento
learning_rate = 0.2
lr_schedule = C.learning_rate_schedule(learning_rate, C.UnitType.
    minibatch)
learner = C.sgd(z.parameters, lr_schedule)
trainer = C.Trainer(z, (loss, label_error), [learner])

#Inicializar os parâmetros de treinamento
minibatch_size = 12 #quantidade de imagens carregadas por lote de
    treinamento
num_samples_per_sweep = 108 #quantidade de imagens totais no
    arquivo de treinamento
num_minibatches_to_train = (num_samples_per_sweep *
    num_sweeps_to_train_with) / minibatch_size #quantidade de lotes
    de imagens usados no treinamento

#Definição do mapa de entrada entre as características e os labels
    do arquivo de treino
input_map={
    y  : train_reader.streams.labels,
    x  : train_reader.streams.features
}

training_progress_output_freq = 1 #definição da frequência de
    impressão dos resultados parciais do treinamento
header = 1 #flag para impressão do cabeçalho dos resultados
    parciais do treinamento

start = time.time() #início da contagem de tempo de treinamento

for i in range(0, int(num_minibatches_to_train)):
    #Ler um lote de features e respectivos labels  do arquivo de
        treinamento especificado por minibatch_size
    data=train_reader.next_minibatch(minibatch_size, input_map=
        input_map)
    trainer.train_minibatch(data)
```

```
162         print_training_progress(trainer, i,
                training_progress_output_freq, 1, header) #chamada da
                função que imprime os resultados parciais do treinamento
            header = 0 #flag de impressão do cabeçalho dos resultados
                parciais do treinamento
164
        #Impressão do tempo de treinamento
166     print("\n")
        print("--> Training took {:.1f} sec".format(time.time() - start)) #
            fim da contagem de tempo de treinamento
168     file.write("-----------------------------------\n\n--> Training took
            {:.1f} sec".format(time.time() - start))

170     #---------------------TEST

172     #Definição do mapa de entrada entre as características e os labels
            do arquivo de teste
        test_input_map = {
174         y   : test_reader.streams.labels,
            x   : test_reader.streams.features
176     }

178     #Inicializar os parâmetros de teste
        test_minibatch_size = 12 #quantidade de imagens carregadas por lote
             de treinamento
180     num_samples = 12 #quantidade de imagens totais no arquivo de
            treinamento
        num_minibatches_to_test = num_samples // test_minibatch_size #
            quantidade de lotes de imagens usados no treinamento
182
        test_result = 0.0
184
        start = time.time() #início da contagem de tempo de teste
186
        for i in range(num_minibatches_to_test):
188         #print("trainer.test_minibatch(data) in i = {0},
                num_minibatches_to_test = {1}".format(i,
                num_minibatches_to_test))
            #Ler um lote de features e respectivos labels  do arquivo de
                treinamento especificado por test_minibatch_size
190         #Cada posição de dados no minibatch é uma imagem de 202500
```

```
            pixels (450*450)
        data = test_reader.next_minibatch(test_minibatch_size,
            input_map=test_input_map)
        eval_error = trainer.test_minibatch(data)
        test_result = test_result + eval_error

    #Impressão do tempo de teste
    print("\n")
    print("--> Test took {:.1f} sec".format(time.time() - start)) #fim
        da contagem de tempo de teste
    file.write("\n--> Test took {:.1f} sec".format(time.time() - start)
        )

    #Média de erros de avaliação de todos os lotes de teste
    print("\n")
    print("--> Average test error: {0:.2f}\%".format(test_result*100 /
        num_minibatches_to_test))
    file.write("\n--> Average test error: {0:.2f}\%\n\n".format(
        test_result*100 / num_minibatches_to_test))

#Função que chama as funções de criação do modelo
def do_train_test():
    global z
    z = create_model(x) #criação do modelo
    reader_train = create_reader(train_file, True, input_dim,
        num_output_classes) #chamada da função que lê os arquivos .txt
    reader_test = create_reader(test_file, False, input_dim,
        num_output_classes) #chamada da função que lê os arquivos .txt
    train_test(reader_train, reader_test, z) #chamada da função que
        treina e testa a rede neural

do_train_test() #chamada da função que chama as funções de criação do
    modelo da rede, leitura dos arquivos .txt de teste/treino e da
    função que executa o teste e o treinamento do modelo
```