

## Capítulo

# 7

## Redes Neurais Convolucionais com Tensorflow: Teoria e Prática

Flávio H. D. Araújo, Allan C. Carneiro, Romuere R. V. Silva, Fátima N. S. Medeiros e Daniela M. Ushizima

### *Abstract*

*Convolutional Neural Networks (CNNs) belong to a category of algorithms based on artificial neural networks that use convolution in at least one of their layers. CNNs have proven to be efficient in various tasks of image and video recognition, recommendation systems and natural language processing, however they need a large number of labeled samples for learning. Thus, in addition to introducing the main concepts and components of CNNs, this chapter also discusses how to use transfer learning techniques to accelerate CNN training process or extract features of databases that do not have sufficient images for the training. Finally, we present a tutorial in python on how to use the Tensorflow library to build and run a CNN to classify images.*

### *Resumo*

*Redes neurais convolucionais (CNNs) pertencem a uma categoria de algoritmos baseados em redes neurais artificiais que utilizam a convolução em pelo menos uma de suas camadas. As CNNs provaram ser eficientes em diversas tarefas de reconhecimento de imagens e vídeos, sistemas de recomendação e processamento de linguagem natural, porém necessitam de uma grande quantidade de amostras rotuladas para o aprendizado. Com isso, além de introduzir os principais conceitos e componentes das CNNs, nesse capítulo também será abordado como utilizar técnicas de transferência de aprendizado para acelerar o treinamento de uma CNN ou extrair características de bases que não possuem imagens suficientes para o treinamento. Por fim, será apresentado um tutorial em python de como utilizar a biblioteca TensorFlow para construir uma CNN e executar a rotina de classificação de imagens.*

### **7.1. Introdução**

As redes neurais convolucionais (*ConvNets* ou *CNNs*) se tornaram o novo padrão em visão computacional e são fáceis de treinar quando existe grande quantidade de amostras

rotuladas que representam as diferentes classes-alvo. Algumas das vantagens consistem em: (a) capacidade de extrair características relevantes através de aprendizado de transformações (*kernels*) e (b) depender de menor número de parâmetros de ajustes do que redes totalmente conectadas com o mesmo número de camadas ocultas. Como cada unidade de uma camada não é conectada com todas as unidades da camada seguinte, há menos pesos para serem atualizados, facilitando assim o treinamento.

As Figuras 7.1 e 7.2 apresentam exemplos de como as CNNs podem ser utilizadas. Na Figura 7.1, o aplicativo realiza sugestões de palavras relevantes que descrevem o cenário da imagem, como “ponte”, “via férrea” e “tênis” através de conhecimento sintetizado via CNNs. A Figura 7.2 mostra exemplos de uma CNN usada para reconhecimento de objetos, pessoas e animais em imagens do dia a dia.

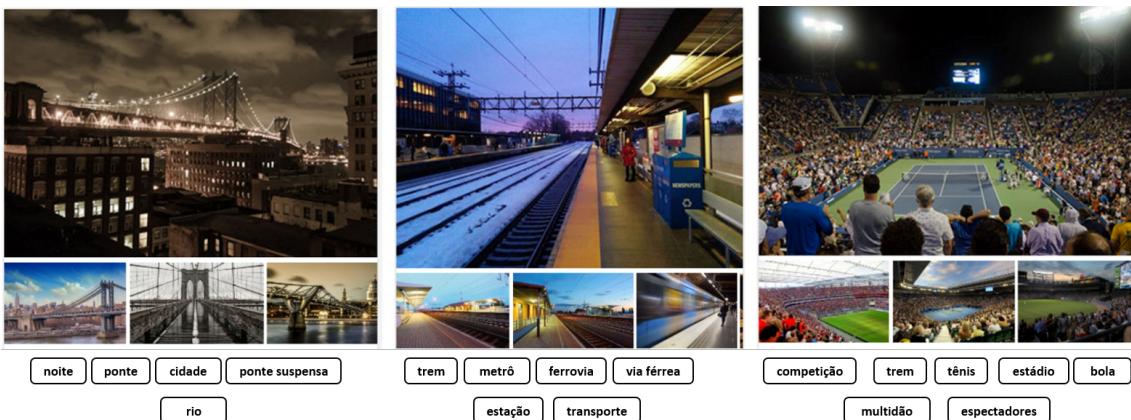


Figura 7.1: Exemplos de sugestões de palavras relevantes que descrevem a cena feita por uma *ConvNet*. Figura modificada de [Clarifai 2017]

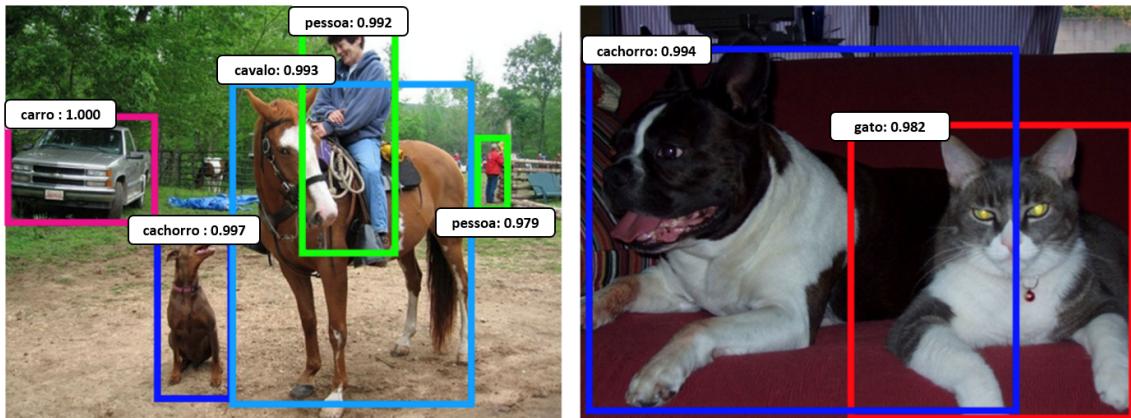


Figura 7.2: Exemplos de reconhecimento de objetos, pessoas e animais feitos por uma *ConvNet*. Modificado de [Ren et al. 2015].

Atualmente, existem diversas bibliotecas que providenciam implementações das principais operações utilizadas pelas CNNs, tais como: *Caffe* [Jia et al. 2014], *MatConvNet* [Vedaldi e Lenc 2015], *Theano* [Al-Rfou et al. 2016], *Torch* [Collobert et al. 2011] e *TensorFlow* [Abadi et al. 2015]. No entanto, a familiarização com essas bibliotecas e o

entendimento de como as CNNs trabalham com imagens não são tarefas triviais. Com isso, esse capítulo tem por objetivo introduzir os fundamentos básicos e principais componentes das CNNs, assim como ensinar a utilizar a biblioteca *TensorFlow* para construir uma CNN e utilizá-la para classificação de imagens. O *TensorFlow* é uma biblioteca de código aberto para computação numérica e aprendizado de máquina disponibilizado pelo *Google Brain Team*<sup>1</sup> em novembro de 2015. Essa biblioteca fornece implementação em múltiplas CPUs ou GPUs e é utilizada por diversas empresas como: Google, Airbnb, Uber, Dropbox, Intel, entre outras.

## 7.2. *LeNet* (1988)

A *LeNet* [Lecun et al. 1998], proposta por Yann LeCun em 1988, foi um dos primeiros projetos de CNNs, tendo a mesma auxiliado a impulsionar o campo de *Deep Learning*. Inicialmente, a *LeNet* foi utilizada para reconhecimento de caracteres, tais como código postal e dígitos numéricos. Novas arquiteturas foram propostas nos últimos anos como forma de melhoria da *LeNet*, embora as versões de CNN melhoradas compartilhem de conceitos fundamentais, descritos a seguir.

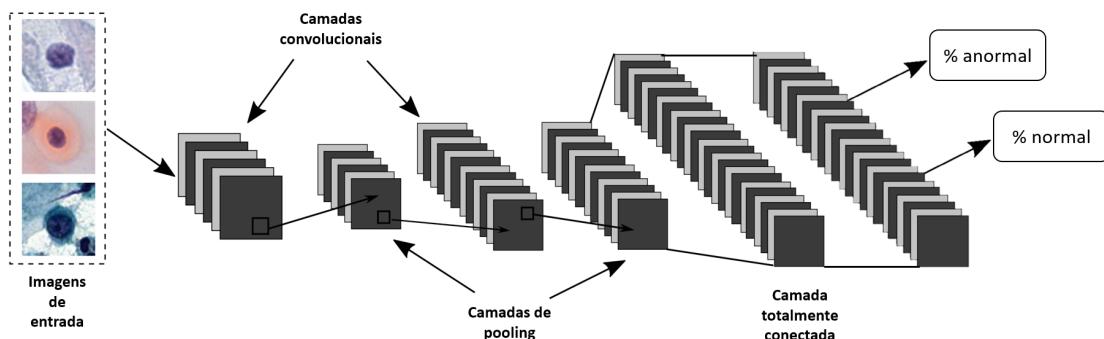


Figura 7.3: Ilustração da arquitetura de uma *LeNet* que classifica imagens de entradas em célula anormal ou célula normal e suas três principais camadas: convolucionais, de *pooling* e totalmente conectadas.

As CNNs são formadas por sequências de camadas e cada uma destas possui uma função específica na propagação do sinal de entrada. A Figura 7.3 ilustra a arquitetura de uma *LeNet* e suas três principais camadas: convolucionais, de *pooling* e totalmente conectadas. As camadas convolucionais são responsáveis por extrair atributos dos volumes de entradas. As camadas de *pooling* são responsáveis por reduzir a dimensionalidade do volume resultante após as camadas convolucionais e ajudam a tornar a representação invarianta a pequenas translações na entrada. As camadas totalmente conectadas são responsáveis pela propagação do sinal por meio da multiplicação ponto a ponto e o uso de uma função de ativação. A saída da CNN é a probabilidade da imagem de entrada pertencer a uma das classes para qual a rede foi treinada. Na seção seguinte, detalhamos o que ocorre em cada uma dessas camadas.

<sup>1</sup><https://research.google.com/teams/brain/>

### 7.2.1. Camada Convolucional

As camadas convolucionais consistem de um conjunto de filtros que recebem como entrada um arranjo 3D, também chamado de volume. Cada filtro possui dimensão reduzida, porém ele se estende por toda a profundidade do volume de entrada. Por exemplo, se a imagem for colorida, então ela possui 3 canais e o filtro da primeira camada convolucional terá tamanho  $5 \times 5 \times 3$  (5 pixels de altura e largura, e profundidade igual a 3). Automaticamente, durante o processo de treinamento da rede, esses filtros são ajustados para que sejam ativados em presença de características relevantes identificadas no volume de entrada, como orientação de bordas ou manchas de cores [Karpathy 2017]. A relevância é avaliada de tal forma que os resultados sejam optimizados em função de um conjunto de amostras previamente classificadas.

Cada um desses filtros dá origem a uma estrutura conectada localmente que percorre toda a extensão do volume de entrada. A somatória do produto ponto a ponto entre os valores de um filtro e cada posição do volume de entrada é uma operação conhecida como convolução, a qual é ilustrada na Figura 7.4. Os valores resultantes após a operação de convolução passam por uma função de ativação, e a mais comum é a função *ReLU* (Rectified Linear Units) [Krizhevsky et al. 2012]. Essa função pode ser calculada pela Equação 1.

$$f(x) = \max(0, x). \quad (1)$$

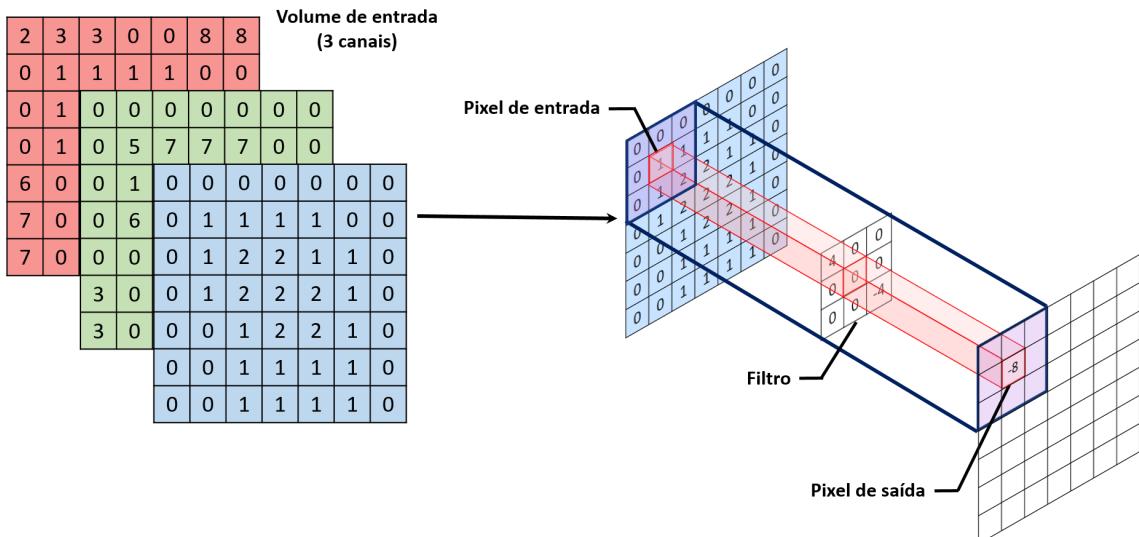


Figura 7.4: Ilustração da convolução entre um filtro 3x3 e o volume de entrada.

Existem três parâmetros que controlam o tamanho do volume resultante da camada convolucional: profundidade (*depth*), passo (*stride*) e *zero-padding* [Ujjwal 2016]. Note que a profundidade do volume resultante é igual ao número de filtros utilizados. Cada um desses filtros será responsável por extrair características diferentes no volume de entrada. Portanto, quanto maior o número de filtros maior o número de características extraídas, porém a complexidade computacional, relativa ao tempo e ao uso de memória, também será maior.

Enquanto que a profundidade do volume resultante depende somente do número de filtros utilizados, a altura e largura do volume resultante dependem do passo e do *zero-padding*. O parâmetro passo especifica o tamanho do salto na operação de convolução, como ilustrado na Figura 7.5. Quando o passo é igual a 1, o filtro salta somente uma posição por vez. Quando o passo é igual a 2, o filtro salta duas posições por vez. Quanto maior o valor do passo menor será a altura e largura do volume resultante, porém características importantes podem ser perdidas. Por esse motivo, é incomum se utilizar o valor de salto maior que 2.

A operação de *zero-padding* consiste em preencher com zeros a borda do volume de entrada. A grande vantagem dessa operação é poder controlar a altura e largura do volume resultante e fazer com que eles fiquem com os mesmos valores do volume de entrada.

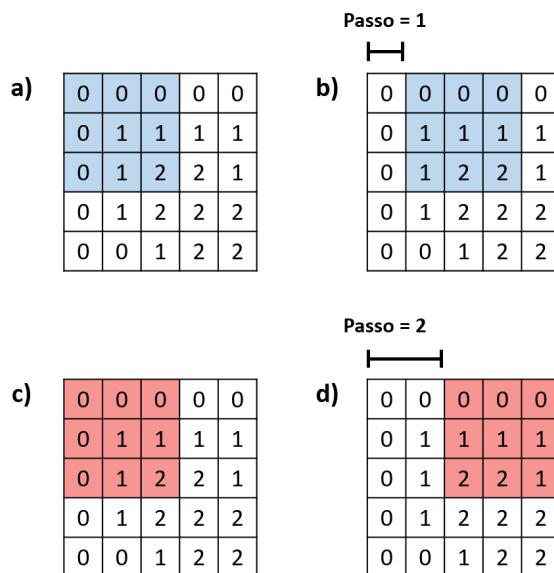


Figura 7.5: Ilustração de como o passo influencia o deslocamento de um filtro 3x3 em duas etapas sucessivas da convolução. As imagens em (a) e (b) correspondem a um passo unitário, enquanto as imagens em (c) e (d) a um passo igual a 2.

Com isso, é possível computar a altura (AC) e a largura (LC) do volume resultante de uma camada convolucional utilizando as Equações 2 e 3, respectivamente.

$$AC = \frac{A - F + 2P}{S} + 1, \quad (2)$$

$$LC = \frac{L - F + 2P}{S} + 1, \quad (3)$$

em que  $A$  e  $L$  correspondem, respectivamente, a altura e largura do volume de entrada,  $F$  é o tamanho dos filtros utilizados,  $S$  é o valor do passo, e  $P$  é o valor do *zero-padding*.

### 7.2.2. Camada de *Pooling*

Após uma camada convolucional, geralmente existe uma camada de *pooling*. O objetivo dessa camada é reduzir progressivamente a dimensão espacial do volume de entrada, consequentemente a redução diminui o custo computacional da rede e evita *overfitting* [Karpathy 2017]. Na operação de *pooling*, os valores pertencentes a uma determinada região do mapa de atributos, gerados pelas camadas convolucionais, são substituídos por alguma métrica dessa região. A forma mais comum de *pooling* consiste em substituir os valores de uma região pelo valor máximo [Goodfellow et al. 2016], como ilustra a Figura 7.6. Essa operação é conhecida como *max pooling* e é útil para eliminar valores desprezíveis, reduzindo a dimensão da representação dos dados e acelerando a computação necessária para as próximas camadas, além de criar uma invariância a pequenas mudanças e distorções locais. Outras funções de *pooling* comumente usadas são a média, a norma L2 e a média ponderada baseada na distância partindo do pixel central [Aggarwal et al. 2001].

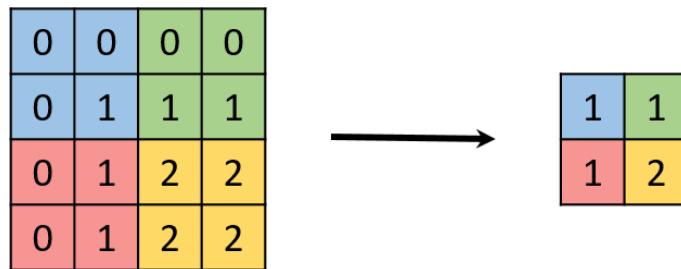


Figura 7.6: Aplicação de *max pooling* em uma imagem 4x4 utilizando um filtro 2x2. Além de reduzir o tamanho da imagem, consequentemente reduzindo o processamento para as próximas camadas, essa técnica também auxilia no tratamento de invariâncias locais. Modificada de [dos Santos Ferreira 2017].

A altura (AP) e a largura (LP) do volume resultante após a operação de *pooling* podem ser calculados pelas Equações 4 e 5, respectivamente.

$$AP = \frac{A - F}{S} + 1, \quad (4)$$

$$LP = \frac{L - F}{S} + 1, \quad (5)$$

onde as variáveis  $A$  e  $L$  correspondem respectivamente a altura e largura do volume de entrada, enquanto que  $F$  representa o tamanho da janela utilizada, e  $S$  é o valor do passo. Vale destacar que a profundidade do volume de entrada não é alterada pela operação de *pooling*.

### 7.2.3. Camada Totalmente Conectada

A saída das camadas convolucionais e de *pooling* representam as características extraídas da imagem de entrada. O objetivo das camadas totalmente conectadas é utilizar essas características para classificar a imagem em uma classe pré-determinada, como ilustrado

na Figura 7.7. As camadas totalmente conectadas são exatamente iguais a uma rede neural artificial convencional (*Multi Layer Perceptron* ou MLP) [Haykin et al. 2009] que usa a função de ativação *softmax* [Bishop 2006] na última camada (de saída).

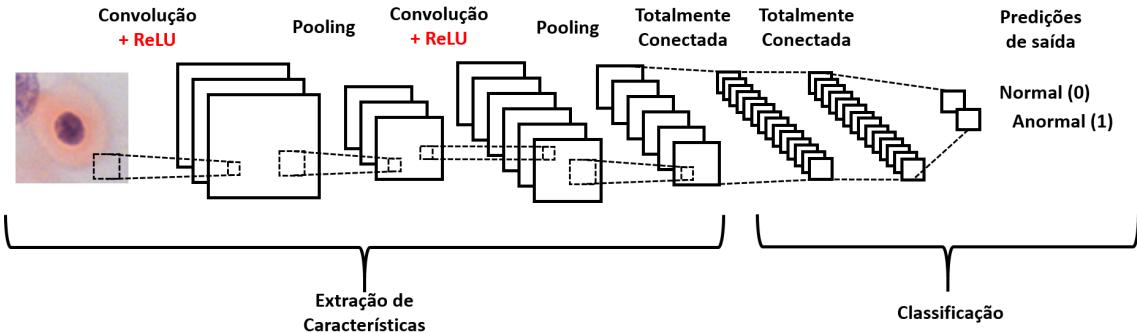


Figura 7.7: Ilustração da extração de características de uma imagem por uma CNN e sua posterior classificação.

Essas camadas são formadas por unidades de processamento conhecidas como neurônio, e o termo “totalmente conectado” significa que todos os neurônios da camada anterior estão conectados a todos os neurônios da camada seguinte.

Em termos matemáticos, um neurônio pode ser descrito pelas Equações 6 e 7:

$$u_k = \sum_{j=1}^m w_{kj}x_j, \quad (6)$$

$$y_k = \varphi(u_k + b_k), \quad (7)$$

em que  $x_1, x_2, \dots, x_m$  são os  $m$  sinais de entrada,  $w_{k1}, w_{k2}$  e  $w_{km}$  são os pesos sinápticos do neurônio  $k$ , e  $b_k$  corresponde ao viés ou *bias*, responsável por realizar o deslocamento da função de ativação definida por  $\varphi(\cdot)$ .

Como mencionado anteriormente, a última camada da rede utiliza *softmax* como função de ativação. Essa função recebe um vetor de valores como entrada e produz a distribuição probabilística da imagem de entrada pertencer a cada uma das classes na qual a rede foi treinada. Vale destacar que a soma de todas as probabilidades é igual a 1.

Uma técnica conhecida como *dropout* [Goodfellow et al. 2016] também é bastante utilizada entre as camadas totalmente conectadas para reduzir o tempo de treinamento e evitar *overfitting*. Essa técnica consiste em remover, aleatoriamente a cada iteração de treinamento, uma determinada porcentagem dos neurônios de uma camada, readicionando-os na iteração seguinte. Essa técnica também confere à rede a habilidade de aprender atributos mais robustos, uma vez que um neurônio não pode depender da presença específica de outros neurônios.

#### 7.2.4. Treinando a Rede com o *Backpropagation*.

Inicialmente, todos os valores dos filtros das camadas convolucionais e os pesos das camadas totalmente conectadas são inicializados de forma aleatória. Em seguida, esses valores

são ajustados de forma a otimizar a acurácia da classificação quando considerando a base de imagens utilizada no processo de treinamento.

A forma mais comum de treinamento de uma CNN é por meio do algoritmo *back-propagation* [Haykin et al. 2009]. O processo de treinamento da CNN usando esse algoritmo ocorre da seguinte forma:

- **Passo 1:** Todos os filtros e pesos da rede são inicializados com valores aleatórios;
- **Passo 2:** A rede recebe uma imagem de treino como entrada e realiza o processo de propagação (convoluçãoes, *ReLU* e *pooling*, e processo de propagação nas camadas totalmente conectadas). Após esse processo a rede obtém um valor de probabilidade para cada classe.
- **Passo 3:** É calculado o erro total obtido na camada de saída (somatório do erro de todas as classes):  $\text{Erro total} = \sum \frac{1}{2} (\text{probabilidade real} - \text{probabilidade obtida})^2$
- **Passo 4:** O algoritmo *Backpropagation* é utilizado para calcular os valores dos gradientes do erro. Em seguida, a técnica do gradiente descendente [Haykin et al. 2009] é utilizada para ajustar os valores dos filtros e pesos na proporção que eles contribuíram para o erro total. Devido ao ajuste realizado, o erro obtido pela rede é menor a cada vez que uma mesma imagem passa pela rede. Essa redução no erro significa que a rede está aprendendo a classificar corretamente imagens devido ao ajuste nos valores dos filtros e pesos. Em geral, os parâmetros: número e tamanho dos filtros nas camadas convolucionais, quantidade de camadas, dentre outros, são fixados antes do passo 1 e eles não sofrem alterações durante o processo de treinamento.
- **Passo 5:** Repete os passos 2-4 para todas as imagens do conjunto de treinamento.

Uma “época de treinamento” é o nome dado ao processo que considera todas as imagens do conjunto de treinamento durante os passos 2-4. O processo de treinamento da rede é repetido por consecutivas épocas até que uma das seguintes condições de parada seja satisfeita: a média do erro obtido pela rede durante a época atual seja menor que um limiar (por exemplo, utilizando o erro médio quadrático) ou um número máximo de épocas seja atingido.

Quando uma dessas duas condições de paradas for satisfeita, os valores dos filtros e pesos estarão calibrados para classificar as imagens do conjunto de treinamento otimamente. Caso o conjunto de treinamento seja abundante e variado o suficiente, a rede apresentará capacidade de generalização e conseguirá classificar corretamente novas imagens que não estavam presentes no processo de treinamento.

Vale destacar que detalhes da formulação matemática do algoritmo *Backpropagation* foram simplificados drasticamente visando rápida explicação do processo de treinamento. Para maiores detalhes consultar [Haykin et al. 2009], [Goodfellow et al. 2016] e [Karpathy 2014].

### 7.3. Outras Arquiteturas

Na seção anterior introduzimos a *LeNet* e as principais camadas que compõe esses algoritmos. Novas arquiteturas de CNNs que surgiram na última década são basicamente formadas pelas mesmas camadas presentes na *LeNet*, conforme discutido a seguir.

#### 7.3.1. *AlexNet* (2012)

Durante o período de 1990 a 2012, as CNNs estavam em um período de incubação. Com o aumento da quantidade de dados disponíveis e do poder computacional, por exemplo, através do uso de GPUs, as CNNs se tornaram cada vez mais eficientes. Em 2012, Krizhevsky e colaboradores projetaram a *AlexNet* [Krizhevsky et al. 2012] que consiste numa versão mais profunda da *LeNet*, ou seja, com mais camadas. Essa rede possuía 5 camadas convolucionais, camadas de *max-pooling* e três camadas totalmente conectadas com *dropout*.

A *AlexNet* foi utilizada para classificar imagens em 1000 possíveis categorias. Em 2012, esta rede ganhou o desafio de classificação ILSVRC 2012 (*ImageNet Large Scale Visual Recognition Challenge*) [Deng et al. 2009]. Vale destacar que este evento trata da mais famosa e maior competição em visão computacional do mundo. Neste evento, diversos times competem para conceber e implementar o melhor modelo para classificação, detecção e localização de objetos em imagens. O ILSVRC 2012 ficou marcado como o primeiro ano no qual uma CNN atingiu o primeiro lugar desse desafio, e com uma diferença bem significativa para o segundo colocado, com *top-5 erro* de 15.4% contra 26.2%. A rede *AlexNet* introduzida em Krizhevsky et al. (2012) constituiu um avanço significativo em relação às outras abordagens, tendo a mesma inspirado dezenas de outras redes convolucionais para reconhecimento de padrões em imagens, com o fato de que algumas redes continham até mesmo um número significativamente superior de camadas.

#### 7.3.2. *ZF Net* (2013)

Com o sucesso da *AlexNet* em 2012, foram submetidos diversos modelos baseados em CNN para o ILSVRC 2013. O vencedor dessa competição foi a rede construída por Zeiler e Fergus com *top-5 erro* de 11,2% [Zeiler e Fergus 2013]. A arquitetura da rede *ZF Net* consistiu de algumas modificações da *AlexNet* e diminuição do tamanho dos filtros e do passo na primeira camada. Porém, a maior contribuição de Zeiler e Fergus foi o detalhamento do modelo proposto e o desenvolvimento da técnica *DeConvNet* (Deconvolutional Network), que consistia numa forma de visualização dos mapas de características da rede.

A ideia básica dessa nova técnica é anexar a cada camada convolucional da rede a *DeConvNet*, que transforma os mapas de características em uma imagem de pixels. A técnica *DeConvNet* realiza operações reversas de *pooling* até que o tamanho original da imagem seja atingido. Por meio dessa operação, é possível verificar como cada camada convolucional “enxerga” a imagem de entrada e quais as regiões da imagem que estão ativando os filtros. A Figura 7.8 mostra exemplos de imagens de entrada e das regiões da imagem que estão ativando os filtros.

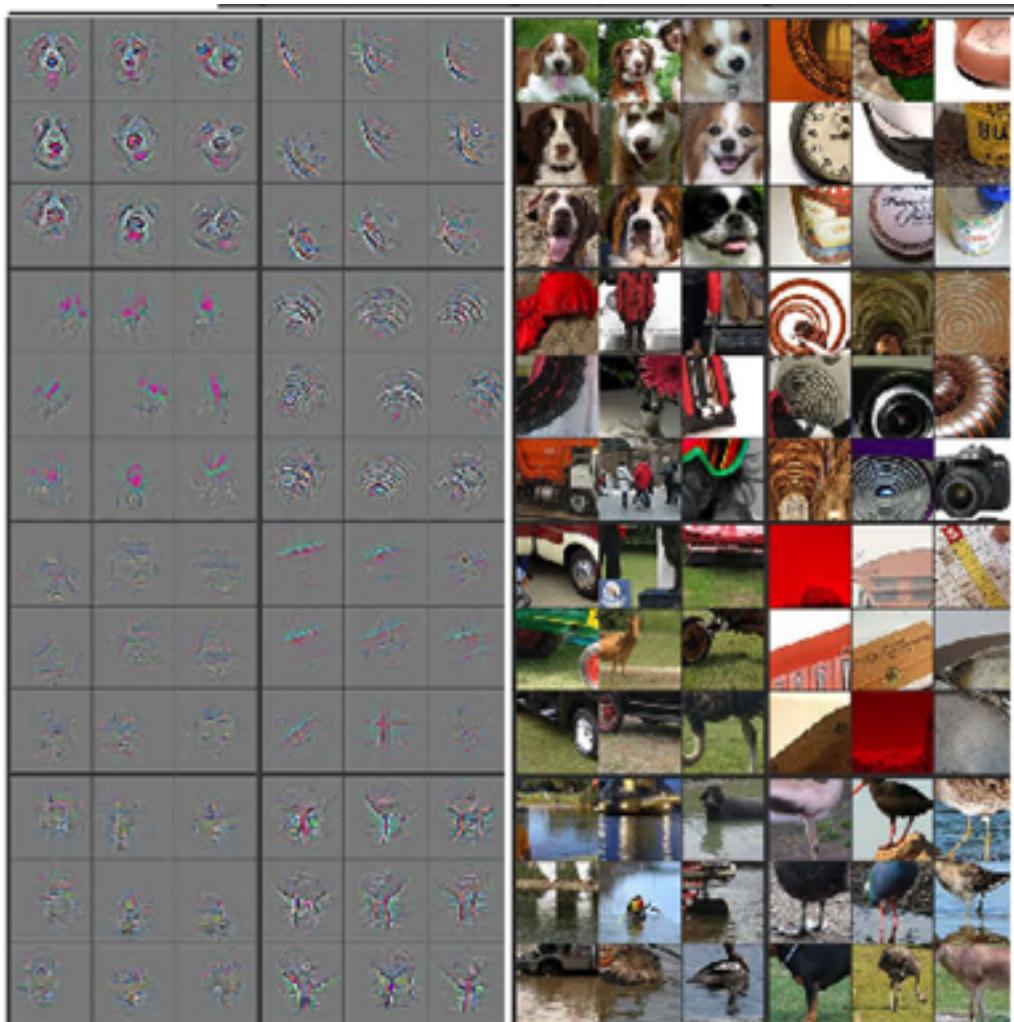


Figura 7.8: Exemplos de imagens de entrada e das regiões que ativam os filtros convolucionais. Modificada de [Deshpande 2016].

### 7.3.3. VGG (2014)

A rede VGG proposta por Simonyan e Zisserman (2014) [Simonyan e Zisserman 2014] foi a primeira a utilizar filtros pequenos ( $3 \times 3$ ) em cada camada convolucional. Isso era contrário aos princípios da *LeNet* e da *AlexNet*, onde filtros grandes ( $9 \times 9$  e  $11 \times 11$ ) eram usados para capturar características similares na imagem. A grande contribuição da VGG foi a ideia de que múltiplas convoluções  $3 \times 3$  em sequência podiam substituir efeitos de filtros de máscaras maiores ( $5 \times 5$  e  $7 \times 7$ ), e que resultavam em maior custo computacional. Com isso, foram testadas 6 arquiteturas diferentes, especificadas na Figura 7.9, e a arquitetura que apresentou o melhor desempenho foi a de rótulo *D*, que possuía 13 camadas convolucionais, 5 de *max-pooling* e 3 totalmente conectadas.

### 7.3.4. GoogLeNet e Inception (2014)

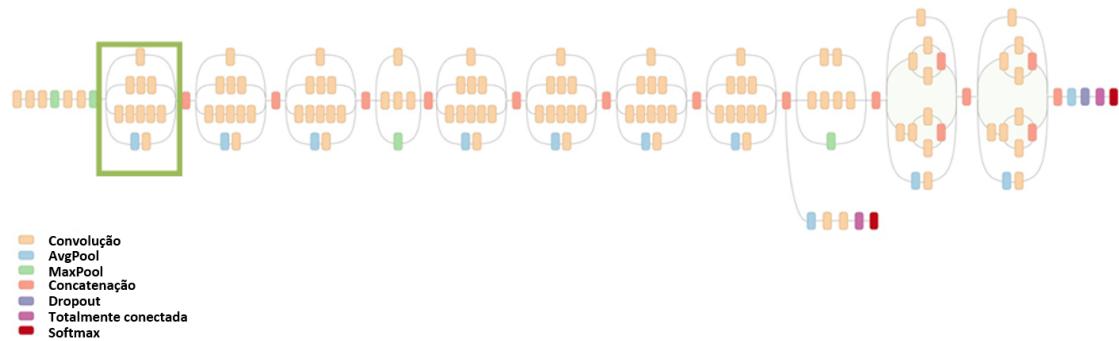
As CNNs se tornaram extremamente úteis em categorizar conteúdos de imagens e de frames de vídeos. Devido ao aumento da acurácia desses algoritmos, diversas empre-

Configuração da ConvNet					
A	A-LRN	B	C	D	E
11 camadas de peso	11 camadas de peso	13 camadas de peso	16 camadas de peso	16 camadas de peso	19 camadas de peso
Entrada (imagem RGB - 224x224)					
conv3-64	conv3-64 <b>LRN</b>	conv3-64 <b>conv3-64</b>	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 <b>conv3-128</b>	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 <b>conv1-256</b>	conv3-256 conv3-256 <b>conv3-256</b>	conv3-256 conv3-256 conv3-256 <b>conv3-256</b>
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 <b>conv1-512</b>	conv3-512 conv3-512 <b>conv3-512</b>	conv3-512 conv3-512 conv3-512 <b>conv3-512</b>
maxpool					
FC-4096					
FC-4096					
FC-1000					
softmax					

Figura 7.9: Especificação das 6 arquiteturas da VGG testadas, a D produziu o melhor resultado. Figura modificada de [Culurciello 2015]

sas, como a *Google*, estavam interessadas em diminuir a complexidade e melhorar a eficiência das arquiteturas existentes. Com isso, uma equipe de engenheiros da *Google* comandada por Christian Szegedy propuseram o modelo conhecido como *GoogLeNet* [Szegedy et al. 2015], ilustrada na Figura 7.10. Essa rede ganhou o ILSVRC 2014 com *top-5 erro* de 6,67%.

Analizando a Figura 7.10, é possível identificar que alguns blocos de camadas eram executados em paralelo. Esses módulos ficaram conhecidos como *Inception* (Figura 7.11) e consistiam de combinações paralelas de camadas com filtros convolucionais de



**A caixa verde indica uma região executada em paralelo da GoogLeNet**

Figura 7.10: Arquitetura da *GoogLeNet*. Modificada de [Deshpande 2016].

tamanho  $1 \times 1$ ,  $3 \times 3$  e  $5 \times 5$ . A grande vantagem do módulo *Inception* era o uso da convolução com filtros  $1 \times 1$  para reduzir o número de características no bloco paralelo antes de realizar as convoluções com os filtros maiores.

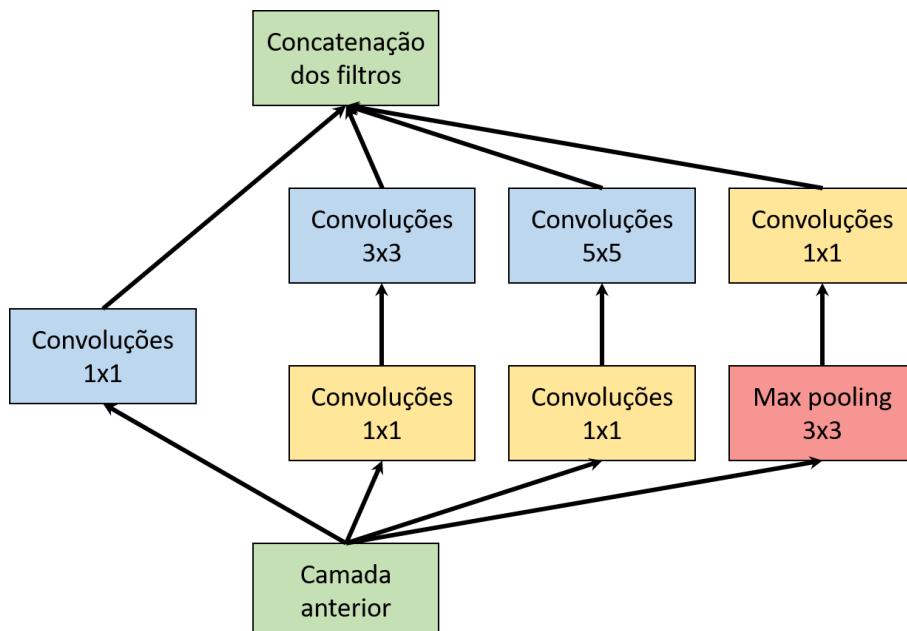


Figura 7.11: Arquitetura do módulo *Inception*. Figura modificada de [Culurciello 2015]

*GoogLeNet* foi o primeiro modelo que introduziu a ideia de que as camadas das CNNs não precisavam ser executadas sequencialmente. A sequência de módulos *Inception* mostraram que uma estrutura criativa de camadas pode melhorar o desempenho da rede e diminuir o custo computacional. No ano seguinte, a rede conhecida como *Inception-v4* [Szegedy et al. 2016], que consistia de alguns melhoramentos da *GoogLeNet*, alcançou um *top-5 erro* de 3,08%.

### 7.3.5. Microsoft e ResNet (2015)

A rede *ResNet* [He et al. 2016] foi proposta por pesquisadores da *Microsoft* e venceu o ILSVRC 2015 com *top-5 erro* de 3.6%. O desempenho dessa rede foi superior ao de seres humanos, que dependendo de suas habilidades e área de conhecimento, normalmente obtém *top-5 erro* entre 5 e 10%.

A *ResNet* era composta por 152 camadas e formada por blocos residuais (*Residual Blocks*). A ideia por trás dos blocos residuais é que uma entrada  $x$  passa por uma série de operações de convolução-*relu*-convolução. O resultado da operação  $F(x)$  é adicionado à entrada original  $x$ , como mostra a Equação 8 e como ilustrado na Figura 7.12.

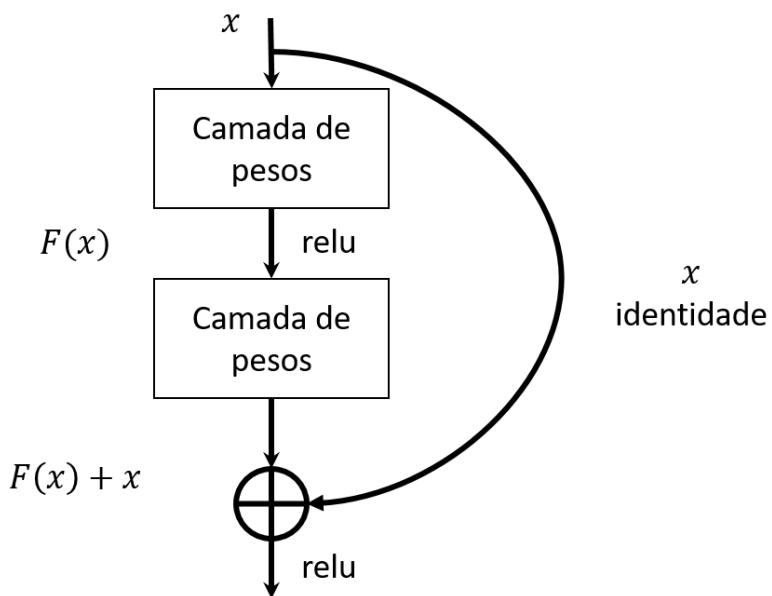


Figura 7.12: Arquitetura de um bloco residual. Figura modificada de [Deshpande 2016]

$$H(x) = F(x) + x. \quad (8)$$

Nas CNNs tradicionais, o resultado de processamento  $H(x)$  é igual ao  $F(x)$ . Com isso o espaço de saída é completamente alterado em relação ao espaço de entrada  $x$ . Com relação à *ResNet*, a função  $F(x)$  funciona somente como um termo de regularização. Portanto, o espaço de saída é somente uma alteração do espaço de entrada. Os criadores da *ResNet* acreditam que os mapas residuais são mais fáceis de serem otimizados por meio dessa alteração no espaço de entrada.

Vale destacar que a *ResNet* que ganhou o ILSVRC 2015 possuía 152 camadas, porém os criadores dessa rede testaram arquiteturas bem mais profundas, com até 1202 camadas. No entanto, as redes com mais de 152 camadas começaram a sofrer *overfitting* e com isso a capacidade de generalização era perdida.

## 7.4. Transferência de Aprendizado

Na prática, não é comum treinar uma CNN com inicializações aleatórias de pesos, pois para isso seria necessário uma grande quantidade de imagens e algumas semanas de treinamento utilizando múltiplas GPUs. Com isso, uma prática comum consiste em utilizar os pesos de uma rede já treinada para uma base muito grande, como a *ImageNet* que possui mais de 1 milhão de imagens e 1000 classes. Em seguida, esses pesos podem ser utilizados para inicializar e retreinar uma rede, ou mesmo para a extração de características de imagens [Karpathy 2015].

### 7.4.1. CNN como Extrator de Características

Para utilizar uma CNN já treinada como extrator de características, basta remover a última camada totalmente conectada da rede (a camada que computa a probabilidade da imagem de entrada pertencer a umas das classes predeterminadas) e utilizar a saída final da nova rede como características que descrevem a imagem de entrada.

As características extraídas das imagens da nova base podem ser utilizadas juntamente com um classificador que requeira menos dados para o treinamento que uma CNN, tais como uma Máquina de Vetor de Suporte (Support Vector Machine - SVM) [Cortes e Vapnik 1995], *Random Forest* (RF) [Breiman 2001], K Vizinhos Mais Próximos (K Nearest Neighbors - KNN) [Aha e Kibler 1991], dentre outros. Essa estratégia de extração de características é bastante utilizada para aplicações de imagens médicas [Zhu et al. 2015, van Ginneken et al. 2015], de materiais [Bell et al. 2014, Zhang et al. 2015], etc. Essa estratégia também é comum em aplicações de Recuperação de Imagens Baseada em Conteúdo (Content Based Image Retrieval - CBIR) [Liu et al. 2016, Wang e Wang 2016, Fu et al. 2016].

### 7.4.2. Fine-tuning uma CNN

A estratégia de *fine-tuning* consiste em dar continuidade ao treinamento de uma rede pretreinada utilizando o algoritmo *Backpropagation* e uma nova base de imagens. Em outras palavras, os pesos de todas as camadas de uma rede pretreinada, com exceção da última camada, são utilizados para a inicialização de uma nova CNN.

É possível fazer o *fine-tuning* de todas as camadas de uma CNN, ou somente das últimas camadas. Isso é motivado pelo fato que as primeiras camadas da rede contém extractores mais genéricos que podem ser utilizados para diferentes tarefas, como detectores de bordas e de cores, porém, as camadas mais profundas possuem detalhes específicos da base com a qual a rede foi originalmente treinada.

### 7.4.3. Escolhendo a Melhor Técnica de Transferência de Aprendizado

A escolha da técnica de transferência de aprendizado depende de diversos fatores, porém os dois principais são o tamanho da nova base de imagens e a similaridade com a base original. Por exemplo, provavelmente uma rede treinada com a base *ImageNet* não produzirá bons resultados quando utilizada para extrair características de uma base de imagens de microscopia.

Existem quatro cenários que devem ser considerados na escolha da técnica de

transferência de aprendizado utilizar e que são detalhados a seguir. Vale destacar que a quantidade de imagens necessária para considerar uma base grande é definida pela complexidade de diferenciação entre as imagens de suas classes. Por exemplo, a *ImageNet* possui cerca de 1.000 imagens por classes, porém com as operações de *augmentation* esse número pode aumentar consideravelmente.

**Nova base de imagens é pequena e similar a base original:** Sempre que a nova base de imagens é pequena não é uma boa ideia aplicar o *fine-tuning* à CNN, pois isso pode causar problemas de *overfitting*. Nesse caso, como a nova base é similar à base original, a melhor opção é remover a última camada da CNN pretreinada e utiliza-lá para extrair as características da nova base e treinar um outro classificador linear.

**Nova base de imagens é grande e similar a base original:** Como as bases de imagens são similares, é bem provável que a utilização da CNN pretreinada para a extração de características produza bons resultados. No entanto, como a nova base de imagens é grande, é improvável que ocorram problemas de *overfitting*, portanto, provavelmente o *fine-tuning* de toda a CNN produzirá melhores resultados.

**Nova base de imagens é pequena e muito diferente da base original:** Como a nova base de imagens é pequena, a melhor opção é utilizar a CNN pretreinada como extrator de características e treinar um classificador linear. No entanto, como as bases de imagens são bem diferentes, é melhor não utilizar as camadas finais da rede pretreinada, pois elas possuem informações específicas da base original. Nesse caso, a melhor opção é utilizar somente as primeiras camadas da rede para a extração de características.

**Nova base de imagens é grande e muito diferente da base original:** Neste caso a melhor opção é o *fine-tuning* de toda a CNN. Embora as bases de imagens sejam bem diferentes, é bem provável que o *fine-tuning* da CNN reduza bastante o tempo de treinamento em comparação com a inicialização aleatória dos pesos.

## 7.5. CNN com *Tensorflow*

Como mencionado anteriormente, o *TensorFlow* é uma biblioteca de código aberto para computação numérica e aprendizado de máquina. Ela foi disponibilizada pelo *Google Brain Team* em novembro de 2015 e atualmente já é uma das bibliotecas mais utilizadas para *deep learning*. O processo de instalação e integração com o *python* é bem simples e todos esses passos são explicados no site oficial do *TensorFlow*: <https://www.tensorflow.org>.

É extensa a lista de problemas que podem ser abordados com CNNs, porém nesse tutorial iremos utilizar imagens de células cervicais obtidas no exame do Papanicolau para exemplificar como é realizado o treinamento e a predição de uma CNN utilizando o *TensorFlow*.

### 7.5.1. Problema Abordado

O câncer cervical, ou câncer do colo do útero, é o quarto mais frequente entre mulheres, com aproximadamente 500.000 novos casos a cada ano. Com exceção do câncer de pele, o câncer de colo do útero é o que apresenta o maior potencial de prevenção e cura, desde que seja detectado precocemente. O exame do Papanicolau consiste no uso de um microscópio para a análise de amostras de células cervicais depositadas em uma lâmina. Nesse

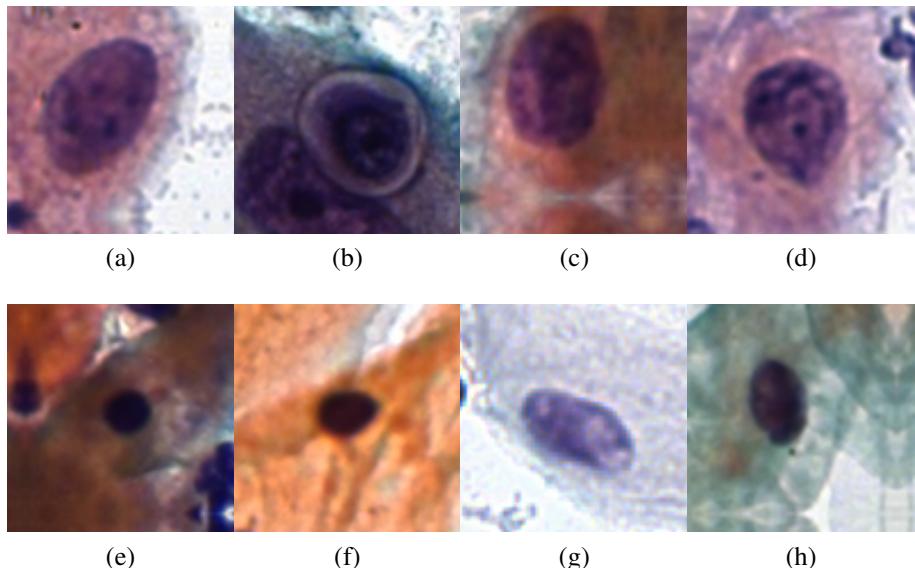


Figura 7.13: Exemplos de imagens de células cervicais utilizadas: (a), (b), (c) e (d) são células anormais; (e), (f), (g) e (h) são células normais.

exame, as imagens das células coletadas são analisadas por um citologista e este tenta encontrar células anormais. As principais anormalidades são identificadas pela análise das características de forma e textura do núcleo e do citoplasma das células. A automação do exame do Papanicolau pode ser feita pelo uso de técnicas de processamento de imagens para a identificação de células anormais. Nesse sentido, utilizaremos uma CNN para a classificação de imagens de células cervicais em normal ou anormal. A Figura 7.13 mostra exemplos de células normais e anormais.

### 7.5.2. Construindo a CNN

O Código Fonte 7.1 contém a declaração das principais funções necessárias para a construção da CNN.

```

1 #Pacotes principais
3 import numpy as np
from skimage.io import imread_collection
5 import tensorflow as tf

7   ''' weight_variable(shape)
9 - A entrada dessa função é uma lista no formato [batch, altura, largura,
  profundidade], na qual batch representa o número de imagens
  processadas de uma vez. Altura, largura e profundidade representam
  as dimensões do volume de entrada.

11 - O retorno dessa função são os valores de pesos inicializados de
  maneira aleatória seguindo uma distribuição normal com desvio
  padrão 0.1.
  ...
13 def weight_variable(shape):
```

```

15     initial = tf.truncated_normal(shape, stddev=0.1)
16     return tf.Variable(initial)

17     ''' bias_variable(shape)
18     - A entrada dessa função é o número de neurônios de uma camada.
19
20     - O retorno dessa função são os valores de bias inicializados com 0.1.
21     '''
22     def bias_variable(shape):
23         initial = tf.constant(0.1, shape=shape)
24         return tf.Variable(initial)

25     ''' conv2d(x, W)
26     - A entrada dessa função é o volume de entrada (x) e os pesos (W) da
27     camada, ambos no formato [batch, altura, largura, profundidade]. Os
28     pesos da camada são retornados na função weight_variable.
29
30     - O retorno dessa função é o volume de saída da camada após a operação
31     de convolução.
32
33     - A variável strides = [1, 1, 1, 1] representa que o passo (stride) da
34     convolução é igual a 1 em cada uma das dimensões.
35
36     - A variável padding='SAME' representa que a operação de zero padding
37     será utilizada para que o volume de saída tenha a mesma dimensão do
38     volume de entrada
39     '''
40     def conv2d(x, W):
41         return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

42
43     ''' max_pool_2x2(x)
44     - A entrada dessa função é o volume de entrada (x) da camada de pooling
45     no formato [batch, altura, largura, profundidade].
46
47     - O retorno dessa função é o volume de saída da camada após a operação
48     de max-pooling.
49
50     - A variável ksize = [1, 2, 2, 1] representa que o filtro utilizado na
51     operação de pooling tem tamanho 2x2 na altura e largura, e tamanho
52     1 na dimensão de batch e profundidade.
53
54     - A variável strides = [1, 2, 2, 1] representa que o passo (stride) da
55     operação de pooling é igual a 2 na altura e largura, e 1 na dimensão
56     de batch e profundidade.
57
58     - A variável padding='SAME' representa que a operação de zero padding
59     será utilizada para que o volume de saída tenha dimensão igual a [
60     batch, altura/2, largura/2, profundidade] do volume de entrada.
61     '''
62     def max_pool_2x2(x):
63         return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
64                               padding='SAME')

```

Código Fonte 7.1: Declarando as funções para a construção da CNN.

O Código Fonte 7.2 contém a construção da CNN utilizando as funções declaradas no Código Fonte 7.1.

```
1 #A variável x irá armazenar as imagens de entrada da rede. Na lista com
2 # parâmetros [None,3,10000], o None é utilizado porque não sabemos a
3 # quantidade de imagens de entrada. O 3 representa que as imagens
4 # possuem 3 canais. E o 10.000 representa a dimensão das imagens (100
5 # x100).
6 x = tf.placeholder(tf.float32, [None,3,10000])
7
8 #A variável y_ representa as classes das imagens de entrada. Na lista
9 # com parâmetros [None,2], o None é utilizado porque não sabemos a
10 # quantidade de imagens de entrada. O 2 representa a quantidade de
11 # classes que as imagens estão divididas.
12 y_ = tf.placeholder(tf.float32, [None, 2])
13
14 #A função tf.reshape redimensiona a variável x para o formato de
15 # entrada que o Tensorflow aceita.
16 x_image = tf.reshape(x, [-1,100,100,3])
17
18 #A variável W_conv1 irá armazenar os pesos da primeira camada
19 # convolucional, que terá 32 filtros de tamanho 5x5 e profundidade 3.
20 # O volume de entrada dessa camada tem dimensão [batch,100,100,3]. O
21 # volume de saída terá dimensão igual a [batch,100,100,32]
22 W_conv1 = weight_variable([5, 5, 3, 32])
23
24 #A variável b_conv1 irá armazenar os valores de bias para os 32 filtros
25 # da primeira camada convolucional.
26 b_conv1 = bias_variable([32])
27
28 #A função tf.nn.relu aplica a função de ativação Relu no volume de saí
29 # da da primeira camada convolucional. A variável h_conv1 irá
#     armazenar os valores resultante da primeira camada convolucional.
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
30
31 #A variável h_pool1 irá armazenar os valores resultantes após a operaçã
32 # o de max-pool. O volume de entrada dessa camada tem dimensão [batch
33 # ,100,100,32]. O volume de saída terá dimensão igual a [batch
34 # ,50,50,32]
35 h_pool1 = max_pool_2x2(h_conv1)
36
37 #A variável W_conv2 irá armazenar os pesos da segunda camada
38 # convolucional, que terá 64 filtros de tamanho 5x5 e profundidade
39 # 32. O volume de entrada dessa camada tem dimensão [batch,50,50,32].
40 # O volume de saída terá dimensão igual a [batch,50,50,64]
41 W_conv2 = weight_variable([5, 5, 32, 64])
42
43 #A variável b_conv2 irá armazenar os valores de bias para os 64 filtros
44 # da segunda camada convolucional.
45 b_conv2 = bias_variable([64])
46
47 #A função tf.nn.relu aplica a função de ativação Relu no volume de saí
48 # da da segunda camada convolucional. A variável h_conv2 irá
49 #     armazenar os valores resultante da segunda camada convolucional.
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
```

```

31 #A variável h_pool2 irá armazenar os valores resultantes após a operação de max-pool. O volume de entrada dessa camada tem dimensão [batch ,50,50,64]. O volume de saída terá dimensão igual a [batch ,25,25,64]
33 h_pool2 = max_pool_2x2(h_conv2)

35 #A variável W_fc1 irá armazenar os pesos da primeira camada totalmente conectada. O volume de entrada dessa camada tem dimensão [batch ,25,25,64]. Na lista com parâmetros [40000, 1024], o valor 40.000 é utilizado pois são  $25 \times 25 \times 64 = 40.000$  conexões. 1024 representa a quantidade de neurônios nessa camada.
37 W_fc1 = weight_variable([40000, 1024])

39 #A variável b_fc1 irá armazenar os valores de bias para os 1024 filtros da primeira camada totalmente conectada.
41 b_fc1 = bias_variable([1024])

43 #A função tf.reshape altera o formato do volume de saída da segunda camada de pooling para o formato de entrada da primeira camada totalmente conectada.
45 h_pool2_flat = tf.reshape(h_pool2, [-1, 40000])

47 #A função tf.nn.relu aplica a função de ativação Relu após a multiplicação ponto a ponto entre o volume de entrada e os pesos da primeira camada totalmente conectada.
49 h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

51 #A função tf.nn.dropout aplica o \textit{dropout} no volume resultante após a primeira camada totalmente conectada.
53 h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

55 #A variável W_fc2 conterá os pesos da segunda camada totalmente conectada. O volume de entrada dessa camada tem 1024 valores, referentes a quantidade de neurônios da camada anterior. O segundo parâmetro com valor 2 representa as duas classes que a rede será treinada.
57 W_fc2 = weight_variable([1024, 2])

59 #A variável b_fc2 conterá os valores de bias para os dois neurônios da segunda camada totalmente conectada.
61 b_fc2 = bias_variable([2])

63 #A função tf.matmul realiza a multiplicação ponto a ponto entre o volume de entrada e os pesos da segunda camada totalmente conectada . y_conv é a variável que contém a estrutura da rede.
65 y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2

```

Código Fonte 7.2: Construindo a CNN utilizando as funções declaradas no Código Fonte 7.1.

### 7.5.3. Treinando a CNN

O Código Fonte 7.3 mostra como são criadas as variáveis necessárias para o treinamento da CNN criada no Código Fonte 7.2.

```
1 #A função softmax_cross_entropy_with_logits utiliza a função cross-
  entropy para calcular o erro entre a saída gerada pela CNN de uma
  determinada entrada e a sua classe correspondente.
3 cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    y_conv, y_))

5 #A função tf.train.AdamOptimizer atualiza os filtros e pesos da CNN
  utilizando o backpropagation. A variável train_step será utilizada
  para realizar o treinamento da rede.
train_step = tf.train.AdamOptimizer(1e-5).minimize(cross_entropy)

7 #As duas próximas linhas são utilizadas para computar a predição da CNN
  e calcular a acurácia obtida.
9 correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Código Fonte 7.3: Criando as variáveis necessárias para o treinamento da CNN construída no Código Fonte 7.2.

O Código Fonte 7.4 mostra como é realizado a leitura das imagens e como é realizado o treinamento da CNN.

```
1 #A função read_images recebe o endereço da pasta que contém a base de
  dados e retorna dois vetores, um contendo as imagens e o outro
  contendo a classe de cada imagem.
3 def read_images(path):
    classes = glob.glob(path+'*')
    im_files = []
    size_classes = []
    for i in classes:
        name_images_per_class = glob.glob(i+'/*')
        im_files = im_files+name_images_per_class
        size_classes.append(len(name_images_per_class))
    labels = np.zeros((len(im_files),len(classes)))

13 ant = 0
14 for id_i,i in enumerate(size_classes):
15     labels[ant:ant+i,id_i] = 1
16     ant = i
17 collection = imread_collection(im_files)

19 data = []
20 for id_i,i in enumerate(collection):
21     data.append((i.reshape(3,-1)))
22 return np.asarray(data), np.asarray(labels)

25 #A variável path contém o endereço da base de imagens
path = '/Users/flavio/database/'
```

```

27 #A variável data irá receber as imagens presente na pasta especificada.
28     Já a variável labels irá receber a classe de cada uma das imagens.
29 data, labels = read_images(path)

31 #A variável batch_size representa o número de imagens que serão
32     processadas a cada passo de treinamento.
33 batch_size = 50

35 #A variável epochs representa o número de épocas de treinamento da rede
36     . Uma época acontece quando todas as imagens do conjunto de
37     treinamento passam pela rede e atualizam seus valores de pesos e
38     filtros.
39 epochs = 100

41 #A variável percent contém a porcentagem de imagens que serão
42     utilizadas para o treinamento.
43 percent = 0.9

45 #Os códigos das próximas 5 linhas estão apenas embaralhando a ordem das
46     imagens e dos labels.
47 data_size = len(data)
48 idx = np.arange(data_size)
49 random.shuffle(idx)
50 data = data[idx]
51 labels = labels[idx]

53 #Formando o conjunto de treinamento com a porcentagem de imagens
54     especificado na variável percent.
55 train = (data[0:np.int(data_size*percent),:,:], labels[0:np.int(
56         data_size*percent),:])

58 #Formando o conjunto de teste com as imagens que não foram utilizadas
59     no treinamento.
60 test = (data[np.int(data_size*(1-percent)):,:,:], labels[np.int(
61         data_size*(1-percent)),:,:])

63 #A variável train_size contém o tamanho do conjunto de treinamento.
64 train_size = len(train[0])

67 #Até aqui apenas criamos as variáveis que irão realizar as operações do
68     Tensorflow, porém é necessário criar uma sessão para que elas
69     posam ser executadas.
70 sess = tf.InteractiveSession()

73 #É necessário inicializar todas as variáveis
74 tf.initialize_all_variables().run()

77 #Laço para repetir o processo de treinamento pelo número de épocas
78     especificado.
79 for n in range(epochs):
80     #Laço para dividir o conjunto de treinamento em sub conjuntos com o
81     #tamanho especificado na variável batch_size. Cada iteração desse laço
82     #representa um batch.
83     for i in range(int(np.ceil(train_size/batch_size))):

```

```

#As próximas seis linhas de código dividem o conjunto de
treinamento nos batchs.
67    if (i*batch_size+batch_size <= train_size):
68        batch = (train[0][i*batch_size:i*batch_size+batch_size],
69                  train[1][i*batch_size:i*batch_size+batch_size])
70    else:
71        batch = (train[0][i*batch_size:], 
72                  train[1][i*batch_size:])
73
74    #Chamando a função de treinamento da rede com o valor de
75    #dropout igual a 0.5.
76    train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob:
77                             0.5})
78
79    #Exibindo a acurácia obtida utilizando o conjunto de
80    #treinamento a cada 5 iterações.
81    if(n%5 == 0):
82        print("Epoca %d, acuracia do treinamento = %g%(n,
83              train_accuracy))

```

Código Fonte 7.4: Leitura das imagens e treinamento da CNN.

Vale destacar que o treinamento da rede foi realizado por 100 épocas e a Figura 7.14 apresenta os valores de acurácia no decorrer das épocas de treinamento.

```

Epoca 0, acuracia do treinamento = 0.464286
Epoca 5, acuracia do treinamento = 0.535714
Epoca 10, acuracia do treinamento = 0.607143
Epoca 15, acuracia do treinamento = 0.678571
Epoca 20, acuracia do treinamento = 0.821429
Epoca 25, acuracia do treinamento = 0.75
Epoca 30, acuracia do treinamento = 0.821429
Epoca 35, acuracia do treinamento = 0.785714
Epoca 40, acuracia do treinamento = 0.821429
Epoca 45, acuracia do treinamento = 0.821429
Epoca 50, acuracia do treinamento = 0.821429
Epoca 55, acuracia do treinamento = 0.821429
Epoca 60, acuracia do treinamento = 0.857143
Epoca 65, acuracia do treinamento = 0.892857
Epoca 70, acuracia do treinamento = 0.892857
Epoca 75, acuracia do treinamento = 0.928571
Epoca 80, acuracia do treinamento = 0.928571
Epoca 85, acuracia do treinamento = 0.928571
Epoca 90, acuracia do treinamento = 0.928571
Epoca 95, acuracia do treinamento = 0.857143
Epoca 100, acuracia do treinamento = 0.964286

```

Figura 7.14: Valores de acurácia obtidos a cada 5 épocas utilizando o conjunto de treinamento.

#### 7.5.4. Classificação com a CNN

O Código Fonte 7.5 mostra como utilizar a CNN para classificar o conjunto de teste e computar a acurácia. Vale destacar que a acurácia obtida foi de aproximadamente 92%.

```
2 #Para computar a acurácia obtida pela CNN basta chamar a variável  
    accuracy criada anteriormente, e especificarmos as imagens de  
    entrada e as classes correspondentes. A variável keep_prob que  
    representa o dropout recebe o valor 1, pois essa operação é  
    utilizada somente no treinamento.  
acuracia = accuracy.eval(feed_dict={x: test[0][:], y_: test[1][:],  
    keep_prob: 1.0})  
4 print("Acuracia = ", acuracia)
```

Código Fonte 7.5: Usando a CNN para classificar o conjunto de teste e computar a acurácia obtida.

### Referências

- [Abadi et al. 2015] Abadi, M. et al. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. <http://tensorflow.org/>. Software disponível em tensorflow.org.
- [Aggarwal et al. 2001] Aggarwal, C. C., Hinneburg, A., e Keim, D. A. (2001). On the surprising behavior of distance metrics in high dimensional space. Em *Lecture Notes in Computer Science*, páginas 420–434. Springer.
- [Aha e Kibler 1991] Aha, D. e Kibler, D. (1991). Instance-based learning algorithms. *Machine Learning*, 6:37–66.
- [Al-Rfou et al. 2016] Al-Rfou, R. et al. (2016). Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688.
- [Bell et al. 2014] Bell, S., Upchurch, P., Snavely, N., e Bala, K. (2014). Material recognition in the wild with the materials in context database. Em *2014 Computer Vision and Pattern Recognition (CVPR)*.
- [Bishop 2006] Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.
- [Breiman 2001] Breiman, L. (2001). Random forests. *Mach. Learn.*, 45(1):5–32.
- [Clarifai 2017] Clarifai (2017). Artificial intelligence with a vision. <https://www.clarifai.com>.
- [Collobert et al. 2011] Collobert, R., Kavukcuoglu, K., e Farabet, C. (2011). Torch7: A matlab-like environment for machine learning. Em *BigLearn, NIPS Workshop*.
- [Cortes e Vapnik 1995] Cortes, C. e Vapnik, V. (1995). Support-vector networks. Em *Machine Learning*, páginas 273–297.

- [Culurciello 2015] Culurciello, E. (2015). Neural network architectures. <https://medium.com/towards-data-science/neural-network-architectures-156e5bad51ba>.
- [Deng et al. 2009] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., e Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. Em *CVPR09*.
- [Deshpande 2016] Deshpande, A. (2016). The 9 deep learning papers you need to know about (understanding cnns). <https://adethpande3.github.io/adethpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html>.
- [dos Santos Ferreira 2017] dos Santos Ferreira, A. (2017). Redes neurais convolucionais profundas na detecção de plantas daninhas em lavoura de soja. Dissertação de mestrado, Universidade Federal de Mato Grosso do Sul.
- [Fu et al. 2016] Fu, R., Li, B., Gao, Y., e Wang, P. (2016). Content-based image retrieval based on cnn and svm. Em *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*, páginas 638–642.
- [Goodfellow et al. 2016] Goodfellow, I., Bengio, Y., e Courville, A. (2016). *Deep learning (adaptive computation and machine learning series)*. Cambridge: The MIT Press.
- [Haykin et al. 2009] Haykin, S. S., Haykin, S. S., Haykin, S. S., e Haykin, S. S. (2009). *Neural networks and learning machines*, volume 3. Pearson Upper Saddle River, NJ, USA:.
- [He et al. 2016] He, K., Zhang, X., Ren, S., e Sun, J. (2016). Deep residual learning for image recognition. Em *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, páginas 770–778.
- [Jia et al. 2014] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., e Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*.
- [Karpathy 2014] Karpathy, A. (2014). Convolutional neural networks. <http://andrew.gibiansky.com/blog/machine-learning/convolutional-neural-networks/>.
- [Karpathy 2015] Karpathy, A. (2015). Transfer learning and fine-tuning convolutional neural networkss. <http://cs231n.github.io/transfer-learning/>.
- [Karpathy 2017] Karpathy, A. (2017). Convolutional neural networks for visual recognition. <http://cs231n.github.io/convolutional-networks/>.
- [Krizhevsky et al. 2012] Krizhevsky, A., Sutskever, I., e Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. Em Pereira, F., Burges, C. J. C., Bottou, L., e Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, páginas 1097–1105. Curran Associates, Inc.

- [Lecun et al. 1998] Lecun, Y., Bottou, L., Bengio, Y., e Haffner, P. (1998). Gradient-based learning applied to document recognition. Em *Proceedings of the IEEE*, páginas 2278–2324.
- [Liu et al. 2016] Liu, X., Tizhoosh, H. R., e Kofman, J. (2016). Generating binary tags for fast medical image retrieval based on convolutional nets and radon transform. Em *2016 International Joint Conference on Neural Networks (IJCNN)*, páginas 2872–2878.
- [Ren et al. 2015] Ren, S., He, K., Girshick, R., e Sun, J. (2015). Faster r-cnn: Towards real-time object detection with region proposal networks. Em *Advances in Neural Information Processing Systems 28*, páginas 91–99.
- [Simonyan e Zisserman 2014] Simonyan, K. e Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. Em *CVPR14*.
- [Szegedy et al. 2016] Szegedy, C., Ioffe, S., e Vanhoucke, V. (2016). Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261.
- [Szegedy et al. 2015] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., e Rabinovich, A. (2015). Going deeper with convolutions. Em *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
- [Ujjwal 2016] Ujjwal, K. (2016). An intuitive explanation of convolutional neural networks. <https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/>.
- [van Ginneken et al. 2015] van Ginneken, B., Setio, A. A. A., Jacobs, C., e Ciompi, F. (2015). Off-the-shelf convolutional neural network features for pulmonary nodule detection in computed tomography scans. Em *2015 IEEE 12th International Symposium on Biomedical Imaging (ISBI)*, páginas 286–289.
- [Vedaldi e Lenc 2015] Vedaldi, A. e Lenc, K. (2015). Matconvnet – convolutional neural networks for matlab. Em *Proceeding of the ACM Int. Conf. on Multimedia*.
- [Wang e Wang 2016] Wang, L. e Wang, X. (2016). Model and metric choice of image retrieval system based on deep learning. Em *2016 9th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*, páginas 390–395.
- [Zeiler e Fergus 2013] Zeiler, M. D. e Fergus, R. (2013). Visualizing and understanding convolutional networks. Em *CVPR13*.
- [Zhang et al. 2015] Zhang, Y., Ozay, M., Liu, X., e Okatani, T. (2015). Integrating deep features for material recognition.
- [Zhu et al. 2015] Zhu, R., Zhang, R., e Xue, D. (2015). Lesion detection of endoscopy images based on convolutional neural network features. Em *2015 8th International Congress on Image and Signal Processing (CISP)*, páginas 372–376.