INSTITUTO POLITÉCNICO DE BRAGANÇA
Escola Superior de Tecnologia e de Gestão

Obtenção de Modelos de *Deep Learning* para a Classificação Automática de Leucócitos

Romeu Ferreira Beato, nº7943

Dissertação

Mestrado em Tecnologia Biomédica

sob a orientação do **Professor Doutor Pedro João Soares Rodrigues**

Dezembro de 2018

AGRADECIMENTOS

Um agradecimento especial à minha família pelo apoio incondicional e pela paciência.

Ao Professor Doutor Pedro João Soares Rodrigues pela sua orientação, apoio, disponibilidade, pela total colaboração no solucionar de problemas e dúvidas ao longo da realização deste trabalho, e por permitir e mediar o acesso à máquina do Instituto Politécnico de Bragança, destinada a este tipo de utilização, possibilitando a realização de testes que de outra forma seriam muito mais demorados e/ou dispendiosos.

Resumo

A quantidade de leucócitos presente no sangue providencia informação pertinente relativa ao estado do sistema imunitário, permitindo avaliar potenciais riscos para a saúde. Estes corpos celulares são classificados em 5 categorias: linfócitos, monócitos, neutrófilos, eosinófilos e basófilos com base em características morfológicas e fisiológicas.

Neste trabalho é feita classificação de imagens de leucócitos utilizando arquiteturas de redes neuronais vencedoras do concurso anual de *software* ILSVRC.

A classificação dos leucócitos é feita recorrendo a redes pré-treinadas e às mesmas redes treinadas de raiz, com o intuito de selecionar as que conseguem melhor desempenho para a tarefa pretendida. As categorias utilizadas são: eosinófilos, linfócitos, monócitos e neutrófilos. Possivelmente devido à menor prevalência no sangue, foi difícil conseguir imagens de basófilos e leucócitos de banda em número suficiente pelo que estes foram excluídos do estudo.

A análise dos resultados obtidos é feita tendo em conta o número de *epochs* de treino necessárias, as técnicas de regularização utilizadas, o tempo de treino e a percentagem de acerto na classificação de imagens num *dataset* de teste. Os melhores resultados de classificação, na ordem dos 98% sugerem que é possível, considerando um pré-processamento competente, treinar redes como a DenseNet com 169 ou 201 camadas em cerca de 100 *epochs*, para classificar leucócitos em imagens de microscopia. Dado o carácter superficial da análise, este projeto constitui uma base para uma eventual automatização do processo de classificação e contagem de leucócitos.

Abstract

The amount of leukocytes present in the blood provides relevant information regarding the state of the immune system, allowing the assessment of potential health risks. These cell bodies are classified into 5 categories: lymphocytes, monocytes, neutrophils, eosinophils and basophils based on morphological and physiological characteristics.

In this work we classify leukocyte images using the neural network architectures that won the annual ILSVRC competition. The classification of leukocytes is made using pre-trained networks and the same networks trained from scratch, in order to select the ones that achieve the best performance for the intended task.

The categories used are: eosinophils, lymphocytes, monocytes and neutrophils. Possibly due to the low prevalence in the blood, it was difficult to obtain enough images of basophils and band leukocytes, so that they were excluded from the study. The analysis of the results obtained is done taking into account the amount of training required, the regularization techniques used, the training time and the percentage of correct image classification in a test dataset.

The best classification results, on the order of 98%, suggest that it is possible, considering a competent preprocessing, to train a network, like the DenseNet with 169 or 201 layers, in about 100 *epochs*, to classify leukocytes in microscopy images. Given the superficial nature of the analysis, this project provides a basis for an eventual automation of the leukocyte count and classification process.

Conteúdo

Lista de Tabelas **vi**

Lista de Figuras **vii**

1 Introdução **1**
1.1 Motivação 3
1.2 Estrutura 4

2 Contagem e identificação de leucócitos: Estado da arte **5**
2.1 Identificação de leucócitos 5
2.2 Contagem e classificação de células sanguíneas 6

3 Redes Neuronais **15**
3.1 Arquiteturas de redes neuronais 16
3.2 Treino 17
3.3 Funções de ativação 18
3.3.1 Função *step* e o perceptrão 18
3.3.2 Função logística e o neurónio sigmoide 20
3.3.3 Função tangente hiperbólica 21
3.3.4 Função *ReLU* 21
3.3.5 Função *Softmax* 22
3.4 *Feedforward* 23
3.5 *Backpropagation* 23
3.6 Métodos de otimização 23
3.6.1 Stochastic Gradient Descent (SGD) 24
3.6.2 Nesterov Accelerated Gradient (NAG) 24
3.6.3 Adam 25
3.6.4 RMSprop 25
3.7 *Overfitting* 25
3.8 *Dropout* 26
3.9 *Batch normalization* 27
3.10 Data Augmentation 28
3.11 *Fine-Tunning* 29

4 Redes Neuronais Convolucionais **30**
4.1 Tensores 30
4.2 Primeira camada: convolução 30
4.3 Agrupamento (*Pooling*) 32

4.4 Camadas mais profundas das CNN . . . 33
4.5 Camada Totalmente Conectada . . . 33
4.6 Arquiteturas de CNN . . . 34
4.6.1 LeNet-5 (1998) . . . 34
4.6.2 AlexNet (2012) . . . 34
4.6.3 ZFNet(2013) . . . 35
4.6.4 GoogLeNet/Inception(2014) . . . 36
4.6.5 VGGNet (2014) . . . 37
4.6.6 ResNet (2015) . . . 38

5 Implementação **40**
5.1 Frameworks, linguagens informáticas e equipamento utilizado . . . 40
5.2 Pré-processamento de dados . . . 41
5.3 Modelos de NN utilizados . . . 43
5.3.1 VGG16 e VGG19 . . . 43
5.3.2 Inception V3 . . . 43
5.3.3 Xception . . . 44
5.3.4 ResNet 50 . . . 44
5.3.5 Densenet 121, Densenet 169 e Densenet 201 . . . 45
5.4 Resultados . . . 46
5.4.1 Modelos com *Transfer Learning* . . . 47
5.4.2 Modelos treinados de raíz . . . 49
5.4.3 Modelos com *Transfer Learning vs.* Modelos treinados de raíz . . . 50
5.4.4 Treino dos modelos com *Transfer Learning* durante 5000 *epochs* . . . 53

6 Conclusões **55**

Bibliografia **59**

A Treino das redes por 100 *epochs* **A1**
A.1 Transfer Learning (100 epochs): *accuracies*, *losses* . . . A1
A.2 Treino de raiz (100 epochs): *accuracies*, *losses* . . . A4
A.3 Transfer Learning (5000 epochs): *accuracies*, *losses* . . . A7

B *Losses* de validação em modelos com *Transfer Learning*: 100 epochs **A10**

C Percentagens de acerto nas categorias **A11**
C.1 Percentagens de acerto em cada categoria. . . . A11

D Transfer Learning (100 *vs.* 5000 *epochs*) **A13**

E *Accuracy* de treino e validação em modelos com e sem *T. Learning* (100 *epochs*) **A15**

F Código Python para os modelos com *Transfer Learning* **A16**

G Código Python para os modelos treinados de raiz **A20**

Lista de Tabelas

5.1 Tabela das percentagens de acerto na classificação de leucócitos obtidas por cada uma das redes (treinadas de raiz) no *dataset* de teste, após o treino de 100 *epochs*. . 51
5.2 Tabela das percentagens de acerto na classificação de leucócitos obtidas por cada uma das redes (treinadas com *transfer learning*) no *dataset* de teste, após o treino de 100 *epochs*. 52

Lista de Figuras

1.1 Tipos de leucócitos: neutrófilos (1), bandas (2), linfócitos (3), monócitos (4), eosinófilos (5) e basófilos (6). 2

2.1 Dispositivo de contagem sanguínea de Cramer 7
2.2 Hematócrito de Hayem. 8
2.3 Aparelho de Gower destinado à contagem de células sanguíneas. 8
2.4 Hematócrito de Malassez. 9
2.5 Hematócrito de Alferow 9
2.6 Bürker. 10
2.7 Processo de segmentação de leucócitos. 13

3.1 Rede neuronal com duas camadas escondidas. 16
3.2 Exemplo ilustrativo da consequência de uma taxa de aprendizagem elevada onde os saltos são demasiado grandes inviabilizando a minimização da perda. 18
3.3 Rede neuronal de perceptrões. 19
3.4 Função *step*. 20
3.5 Função *sigmoide*. 21
3.6 Função *tangente hiperbólica*. 21
3.7 Função *ReLU*. 22
3.8 Adaptação de modelos polinomiais de ordem 2, 10, 16 e 20 à equação $y = \sin(x/3) + v$, com *overfitting* nos modelos de ordem 16 e 20. 26
3.9 Modelo de rede neuronal. Esquerda: NN com duas camadas escondidas. Direita: Aplicação de *dropout* à NN anterior com exclusão das unidades assinaladas com *X*. 27
3.10 Exemplos de transformações tradicionais, realizadas numa imagem. 29

4.1 Local de aplicação de um filtro. 31
4.2 Visualização do campo recetor; Representação matricial do campo recetor; Representação matricial do filtro. 31
4.3 Exemplo da visualização de um campo recetor, da representação em pixeis deste campo e representação matricial de um filtro. 32
4.4 Visualização de filtros de um caso prático. 32
4.5 Arquitetura da CNN com alternância de camadas convolutivas e de agrupamento. As camadas de *pooling* podem implementar operações de subamostragem ou agrupamento máximo. 33
4.6 Rede neuronal LeNet. 34
4.7 Esquema da arquitetura da rede neuronal AlexNet (2012). 35
4.8 Esquema da arquitetura da rede neuronal ZFNet (2013). 36
4.9 Módulo de *Inception* original usado na *GoogLeNet* 37

4.10 Visualização da arquitetura da VGG 38
4.11 Módulo residual da ResNet conforme originalmente proposto pelos autores da rede. 39
4.12 Representação gráfica da evolução dos erros de classificação obtidos pelos vencedores (e segundo classificado de 2014) do ILSVRC entre 2010 e 2015. 39

5.1 Imagem original de um eosinófilo (640×480). 42
5.2 Imagem do eosinófilo segmentada (209×201). 42
5.3 Imagem de neutrófilo segmentada. 42
5.4 Imagem de neutrófilo originada por *Data Augmentation* 42
5.5 Imagem de eosinófilo segmentada de forma a obter a área pertinente da imagem original. 43
5.6 Imagem de eosinófilo segmentada com o Paint3D, com eliminação do *background*. 43
5.7 Tabela 1 de *Very Deep Convolutional Networks for Large Scale Image Recognition*, Simonyan and Zisserman (2014). 44
5.8 A arquitetura Xception: Os dados passam pelo fluxo de entrada, depois pelo fluxo médio que é repetido oito vezes e finalmente pelo fluxo de saída. 45
5.9 Arquitetura da *DenseNet*. 46
5.10 *Loss* registada no *dataset* de treino pelo conjunto de modelos treinados com *Transfer Learning* ao longo de 100 epochs. 48
5.11 *Accuracies* registadas pelo conjunto de modelos no *dataset* de treino com *Transfer Learning* ao longo de 100 epochs. 49
5.12 *Loss* registada pelo conjunto de modelos treinados de raiz no *dataset* de treino com 100 epochs. 50
5.13 *Accuracy* registada pelo conjunto de modelos treinados de raiz no *dataset* de treino com 100 epochs. 51
5.14 *Losses* de treino e validação médias do conjunto de modelos treinados de raíz ou com *transfer learning* nos *datasets* de treino e validação, durante 100 epochs. . . . 52
5.15 Percentagens de acerto em imagens do *dataset* de teste obtidas com modelos de *transfer learning* treinados em 100 e 5000 epochs 54

A.1 *Train and validation*, *accuracies* and *losses* nos modelos treinados de raiz em 100 *epochs* A3
A.2 *Train and validation*, *accuracies* and *losses* nos modelos treinados de raiz em 100 *epochs* A6
A.3 *Train and validation*, *Accuracies* and *losses* nos modelos treinados de raiz em 5000 *epochs*. A9

B.1 *Losses* registadas pelo conjunto de modelos no *dataset* de validação com *Transfer Learning* ao longo de 100 epochs. A10

C.1 Percentagens de acerto em imagens do dataset de teste obtidas com modelos de *transfer learning* em 100 epochs A11
C.2 Percentagens de acerto em imagens do dataset de teste obtidas com modelos treinados de raiz em 100 epochs A11
C.3 Percentagens de acerto em imagens do dataset de teste obtidas com modelos de transfer learning em 5000 epochs A12

D.1 *Precision*, *recall*, *score* e *confusion matrix* dos modelos treinados com t*ransfer learning* em 100 e 5000 *epochs* A14

E.1 *Accuracies* de treino e validação médias do conjunto de modelos treinados de raíz ou com *transfer learning* nos *datasets* de treino e validação, durante 100 epochs. . A15

Capítulo 1

Introdução

A perda de uma quantidade substancial de sangue ou a suspeita da existência de patologias exigem regularmente a necessidade de realização de análises sanguíneas onde, entre outras coisas, se procura obter a contagem do número total (aproximado) de leucócitos existentes na corrente sanguínea. A quantidade de leucócitos, também conhecidos por glóbulos brancos, providencia informação pertinente relativa ao estado do nosso sistema imunitário permitindo avaliar potenciais riscos para a saúde. Uma mudança significativa no número de leucócitos, relativamente a valores de referência, é normalmente sinal de que o corpo se encontra afetado por algum tipo de antigénio. Além disso, a variação num tipo de glóbulo branco específico está geralmente correlacionada com um tipo específico de antigénio.

Os glóbulos brancos são classificados em 5 categorias: linfócitos, monócitos, neutrófilos, eosinófilos e basófilos. Existe também a designação de banda para uma forma específica do núcleo. A Figura 1.1 apresenta exemplos destas categorias.

Os neutrófilos, basófilos e eosinófilos possuem núcleos com vários lobos nucleares. Estes são diferenciados com base na cor do citoplasma, tamanho e cor do núcleo. A contagem de leucócitos fornece informação sobre variadas patologias e ajuda na monitorização da recuperação de pacientes após o início de tratamentos. A contagem diferencial do sangue indica que tipo de células sanguíneas se encontram mais afetadas. O número normal de leucócitos varia entre 4500 e 10000 células por microlitro (dependendo do género e idade do indivíduo), com uma composição de 50-70% de neutrófilos, 25-30% de linfócitos, 0.4-1% de eosinófilos e 1-3% de monócitos. Uma contagem de leucócitos elevada, superior a 30000 células por microlitro, não é indicativa de nenhuma doença específica mas é sinónimo de infeção, doença sistémica, inflamação, alergia, leucemia ou lesão tecidual provocada por queimaduras. A contagem de leucócitos aumenta com a ingestão de determinados fármacos como antibióticos, com o stress ou no caso de fumadores. Além disso, com o aumento do número de leucócitos acresce o risco de mortalidade por razões de ordem cardiovascular. Por outro lado, um número baixo de leucócitos pode ser indicativo de infeções virais, baixa imunidade e problemas na medula óssea. Valores abaixo de *2500 células*/µL são motivo para alerta e indicam um elevado risco de sepsis. De acordo com os procedimentos convencionais, lâminas contendo amostras de sangue são mergulhadas em solução de Lisman (baseada numa solução de azul de metileno, que permite diferenciar leucócitos, parasitas da malária e tripanosomas) sendo posteriormente analisadas ao microscópio, onde a contagem e classificação são feitas por um patologista [Ghosh et al., 2011].

A classificação dos leucócitos toma também parte importante na identificação de fenómenos corporais, associados à presença de um tipo específico de leucócito. Este conhecimento é ainda algo limitado pelo que se seguem ideias gerais do que se pensa serem as

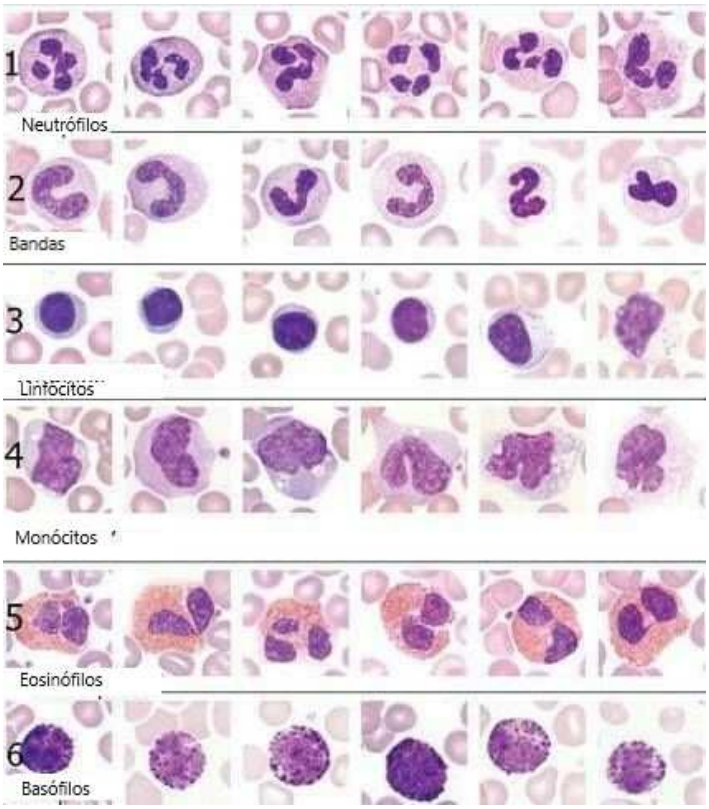


Figura 1.1: Tipos de leucócitos: neutrófilos (1), bandas (2), linfócitos (3), monócitos (4), eosinófilos (5) e basófilos (6).

Fonte:[leu, 2018].

principais funções dos vários tipos de leucócitos.

Os neutrófilos têm como principal função fagocitar bactérias e outros microrganismos e exibem grande mobilidade. Os eosinófilos são responsáveis pelo ataque a invasores de grande tamanho libertando proteínas tóxicas para os destruir. Os basófilos libertam histamina, que é importante nas inflamações e respostas alérgicas facilitando a saída de anticorpos e neutrófilos para locais onde são necessários. Os basófilos produzem também heparina, com propriedades anticoagulantes. Os monócitos, produzidos na medula óssea, migram para os tecidos onde, após se transformarem em macrófagos, fagocitam microrganismos e células mortas. Os linfócitos, podem ser do subtipo *B*, *T* ou *NK* (*natural killer*) exercem funções específicas no combate a infeções e ao cancro [Moraes, 2017].

A evolução das técnicas de contagem e identificação dos leucócitos (e das células sanguíneas em geral) teve início em meados do século XIX com a utilização de tubos capilares e lâminas. Ao longo de anos foram aparecendo variados tipos de dispositivos concebidos para o contagem de células sanguíneas, o que viria mais tarde a possibilitar a sua classificação.

A classificação de células sanguíneas sempre foi feita por especialistas humanos até que na década de 60 do século passado surgiu a possibilidade desta ser feita informaticamente. Primeiramente através da utilização de métodos óticos e de impedância e posteriormente através de algoritmos desenvolvidos especificamente para o efeito a partir de imagens de microscopia, no âmbito da visão por computador e, sobretudo nas últimas duas décadas, com recurso a redes neuronais. Como consequência do nível do poder de computação atingido nesta década, tornou-se possível a implementação de poderosos algoritmos concebidos há

várias décadas [Donnelly, 2017].

Em 1950 Alan Turing criou o ”Teste Turing”para determinar se um computador possui inteligência. Para passar este teste, o computador teria que ser capaz de convencer um humano de que também seria humano. Em 1952 Arthur Samuel escreveu o primeiro programa de aprendizagem máquina. O programa era um jogo de damas, e o computador *IBM*[1] melhorava o seu desempenho à medida que ia praticando, estudando quais os movimentos que levavam a estratégias vencedoras e incorporando-os nos seu programa. Em 1957, Frank Rosenblatt concebeu a primeira rede neuronal para computadores - o perceptrão -, que simulava (vagamente) os processos do cérebro humano. Em 1967 foi escrito o algoritmo *nearest neigbor*, permitindo que se começasse a usar um reconhecimento básico de padrões [Marr, 2017].

Apesar das origens do *Deep Learning* reportarem à década de 80 do século passado, é na corrente década que este se afirma como ramo do *Machine Learning* ou aprendizagem máquina, consistindo num sistema de computação paralela com um grande número de processadores simples interligados, organizados por camadas, possuindo, genericamente, uma camada de entrada, camadas escondidas e uma camada de saída. Dentro destas principais categorias existem variantes utilizadas em função do tratamento que se pretende dar à informação fornecida à rede. O processo de aprendizagem no contexto das redes neuronais pode ser visto com um problema de atualização da arquitetura da rede e dos pesos das ligações, de forma a que essa rede consiga desempenhar determinada tarefa [Nielsen, 2015].

No presente trabalho a proposta é a realização da classificação de imagens de leucócitos recorrendo a arquiteturas de rede vencedoras (com excepção de uma) do concurso anual de software ILSVRC (*ImageNet Large Scale Visual Recognition Challenge*) onde programas competem para detetar e classificar objetos e cenários [Russakovsky et al., 2015]. A identificação dos leucócitos é feita recorrendo a redes pré-treinadas e às mesmas redes treinadas de raiz, com o intuito de selecionar a melhor rede e melhor abordagem para a tarefa pretendida.

1.1 Motivação

A humanidade parece caminhar a passos largos para uma existência simbiótica com dispositivos eletrónicos. O objetivo primordial desta interação prende-se com a necessidade de, por um lado, nos permitir o acesso a mais informação e melhor qualidade de vida, e por outro, libertar mais tempo, em virtude da automatização de tarefas que até há bem pouco tempo careciam de intervenção ou supervisão humana. Neste sentido, a inteligência artificial, tem vindo a apresentar-se como uma boa solução para o desempenho de tarefas simples e automatizáveis.

O estudo dos mecanismos celulares e moleculares de interação entre leucócitos e os vários tecidos em várias condições inflamatórias reveste-se de extrema importância. O principal objetivo destes estudos é desenvolver estratégias terapêuticas eficazes no tratamento de doenças inflamatórias e auto-imunes. Atualmente, a classificação dos corpos celulares ainda tende a ser feita de forma visual. Este processo, além de demorado, leva à fadiga do observador, podendo resultar em análises incorretas. Neste contexto, o presente trabalho tem como objetivo fornecer uma ferramenta computacional para permitir a classificação de leucócitos de forma automática a partir de imagens obtidas em trabalhos de microscopia com

[1] *International Business Machines*

capacidade de abranger tipos de leucócitos que não são classificáveis pelos métodos atuais.

1.2 Estrutura

O capítulo um apresenta informações de caráter básico relativamente a alguns tipos de leucócitos, fazendo referência ao potencial do *Deep Learning* na classificação dos mesmos.

No capítulo dois são descritas, numa perspetiva histórica, algumas das técnicas desenvolvidas para a identificação e contagem de células sanguíneas e dos leucócitos em particular.

O capítulo três explica o que são redes neuronais, abordando diferentes funções de ativação, arquiteturas de redes, métodos de otimização, e fazendo a explicação de técnicas como *feedforward*, *backpropagation*, *overfitting*, *dropout*, *batch normalization*, *data augmentation* e *fine-tunning*.

No capítulo quatro fala-se especificamente de redes neuronais convolucionais, da noção de tensor e dos conceitos de convolução e *pooling*. Neste capítulo são apresentadas algumas redes neuronais conhecidas como a LeNet-5, AlexNet, ZFNet, GoogLeNet, VGGNet e ResNet.

O capítulo cinco descreve as *frameworks*, linguagens e equipamento utilizado. Aborda o pré-processamento de dados, os modelos utilizados e os resultados obtidos.

No capítulo seis são apresentadas as conclusões do trabalho.

Capítulo 2

Contagem e identificação de leucócitos: Estado da arte

2.1 Identificação de leucócitos

A biologia dos leucócitos continua a ser um campo muito dinâmico de investigação com novas células a serem descobertas regularmente.

Ilia Metchnikoff (1845-1916), vencedor de um Nobel da Medicina (juntamente com Paul Ehrlich) foi um biólogo microbiologista e anatomista ucraniano, que se distinguiu pelos estudos realizados em imunologia, incidindo sobretudo sobre o papel dos leucócitos na fagocitose de bactérias e seria, possivelmente, a figura mais proeminente no estudo dos leucócitos.

Maximow, nascido em São Petersburgo, introduziu a teoria unitarista da hematopoiese, sobre a qual se baseia o conceito moderno da origem e diferenciação de células sanguíneas, tendo em 1909 introduzido a noção de célula estaminal.

George Miller Sternberg (1838–1915), alegou ter sido o primeiro, em 1881, a sugerir que os glóbulos brancos poderiam ingerir e destruir bactérias.

Nas primeiras décadas do século XX, os esforços foram dedicados principalmente a identificar a origem dos macrófagos, com demonstrações da origem dos macrófagos teciduais feitas primeiramente *in vitro*.

Em 1914, a partir de células sanguíneas de um paciente com leucemia, Awrorow e Timofejewskij concluíram que o linfócito era uma célula estaminal, da qual surgiam, por exemplo, macrófagos. Dez anos depois, Sabin, Doan e Cunningham realizaram uma série de observações - com sangue de galinha - em que diferenciaram vários tipos de células sanguíneas por meio de corantes vitais e, com base nisso, tentaram agrupar as células do sangue mais de acordo com o que eles consideravam ser a sua origem.

Em 1925, Lewis consegue realizar observações semelhantes, finalmente realizadas em sangue humano. Muito antes disto já Metchnikoff reconhecia o papel fundamental da migração transleucocitária da corrente sanguínea, concluindo que a inflamação nos animais superiores era uma reação saudável do corpo. Durante processos inflamatórios, os leucócitos passam por aberturas especializadas entre as células endoteliais (via migração "para-celular"). No entanto, a transmigração de células circulantes foi relatada pela primeira vez por Augustus Volney Waller (1816-1870).

Nos seus livros, Metchnikoff também define claramente o papel dos neutrófilos, a que ele chamou de *"microphages"*:

"Nos vertebrados encontramos duas grandes categorias de corpúsculos brancos, dos

quais um se assemelha aos dos invertebrados, pois eles também possuem um único grande núcleo e um protoplasma ameboide. Estes são os macrófagos do sangue e da linfa, e estão intimamente ligados aos macrófagos de órgãos como o baço, as glândulas linfáticas e a medula óssea. Outro grupo de corpúsculos brancos nos vertebrados é composto por pequenas células ameboides, que se distinguem por terem um núcleo, que, embora único, é dividido em vários lobos. Estes são os micrófagos. A fagocitose é exibida não apenas pelos macrófagos, mas também, em alto grau, pelos micrófagos que se destacam como células defensivas por excelência contra microrganismos".

A primeira descrição de neutrófilos foi feita em 1865 por Max Johann Sigismund Schultze (1825–1874) que primeiro descreveu os quatro tipos diferentes de leucócitos do sangue correspondentes aos que são agora reconhecidos como linfócitos, monócitos, eosinófilos e o neutrófilos. Ehrlich, químico, começou sua carreira como histologista a corar seletivamente células e a identificar linfócitos e eosinófilos.

Em 1956, Hirsch caracterizou uma substância bactericida isolada de células polimorfonucleares, a que ele chamou de "fagocitina". A fagocitina, identificada em neutrófilos de coelho, era bactericida em bactérias Gram-negativas e positivas, mas não era bacteriolítica.

Dez anos antes, van Furth propusera uma nova classificação, explicando a linhagem dos macrófagos. Com Zanvil Cohn (1926-1993), van Furth identificou os monócitos do sangue como os precursores dos macrófagos teciduais e da medula óssea como fonte de monócitos. Em 1993, van Furth fez uma publicação onde descrevia a origem e a cinética dos fagócitos mononucleares e a natureza dos fatores hematopoiéticos envolvidos. O laboratório de Cohn foi um local de referência para estudar os macrófagos. Entre muitas descobertas importantes estava o papel desempenhado pelos grânulos de fagócitos específicos, os lisossomas, que descarregam os seus conteúdos no fagossoma que contém o microrganismo ingerido, levando à digestão do mesmo.

Precocemente, Elie Metchnikoff entendeu e definiu o papel dos monócitos/macrófagos e neutrófilos durante a inflamação e a imunidade inata. Num discurso pronunciado no *Institut Pasteur* em 29 de dezembro de 1890, disse:

O facto de que a invasão do organismo por micróbios induz, na maioria das vezes, por um lado, uma reação inflamatória associada a uma emigração de leucócitos, e por outro lado, a inclusão e destruição de invasores pelos fagócitos, leva-nos a admitir que o afluxo de fagócitos para a região invadida, e as suas propriedades bactericidas são mecanismos que servem para afastar ataques bacterianos e manter a integridade do organismo.

O que Metchnikoff descobriu no século XIX permanece verdadeiro no século XXI [Cavaillon, 2011].

2.2 Contagem e classificação de células sanguíneas

A contagem de células sanguíneas foi um dos primeiros métodos de investigação introduzidos para o estudo quantitativo do sangue, e durante muitos anos, provavelmente o mais requisitado. O crédito da primeira contagem de células sanguíneas é atribuído ao professor Karl Vierordt, da Universidade de Tübingen.

Na sua técnica original, apresentada numa série de 3 artigos, Vierordt colocava sangue em tubos capilares de diâmetro conhecido, cuja capacidade podia ser medida, e expelia uma quantidade desse sangue para uma lâmina contendo um esfregaço de albumina. Depois de seco era colocado um micrómetro sobre o vidro e as hemácias (também conhecidas por eritrócitos ou glóbulos vermelhos) eram contadas ao microscópio. O método viria a ser

melhorado com a realização de uma diluição com goma-arábica. Apesar deste método ter sido considerado bastante preciso revelou-se muito tedioso para implementação rotineira em termos clínicos.

Em 1855, Cramer introduziu vários princípios importantes na contagem dos elementos do sangue. Usando um volume de sangue conhecido numa diluição efetuou a contagem num espaço capilar de dimensões conhecidas, tendo também introduzido um micrómetro ocular quadrado para auxiliar na contagem das células. O princípio deste espaço capilar era o de que o conhecimento das suas dimensões permitia que este fosse preenchido com sangue diluído por atração capilar. Tratava-se de um método moderadamente bem sucedido que acabou por não ser melhorado pelo seu autor. A Figura 2.1 apresenta um esboço do dispositivo.

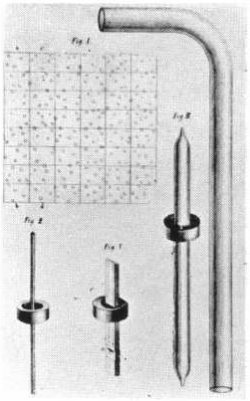


Figura 2.1: Dispositivo de contagem sanguínea de Cramer
Fonte:[Verso, 1964].

Em 1869 Pierre-Carl Joseph Potain contribuiu, entre outras coisas, com a invenção de uma pipeta de diluição, com o mesmo princípio das atuais, mas com dimensões e design diferente. Na mesma altura Louis-Charles Malassez concebia dois métodos de contagem de células. No primeiro método, o sangue contido numa pipeta de diluição era colocado em capilares de dimensão conhecida e secção elíptica que permitia a contagem numa dada porção do tubo.

Outro investigador, George Hayem, conhecido por valiosas contribuições para a medicina, como por exemplo, o reconhecimento das plaquetas como elementos independentes no sangue, concebeu uma câmara constituída por uma lâmina de espessura precisa, com um buraco de 1mm de diâmetro. A esta lâmina era junta uma outra de forma a resultar uma pequena calha circular de dimensões conhecidas. Quando uma gota de sangue era colocada na calha e aplicada uma lamela, a gota era convertida numa forma cilíndrica de líquido, de profundidade conhecida. À semelhança dos métodos anteriores, também neste era feita a utilização do micrómetro ocular quadrado para fazer a contagem das células. A Figura 2.2 apresenta um desenho do hematócrito de Hayem.

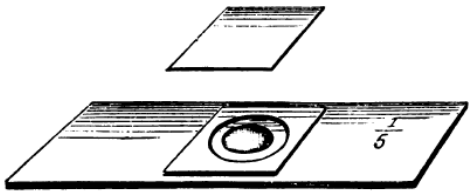


Figura 2.2: Hematócrito de Hayem.
Fonte:[Verso, 1964].

Gower, em 1877, pegaria no desenho inicial de Hayem e faria alterações importantes na câmara de contagem (Figura 2.3), tendo efetuado divisões no chão da câmara em quadrados com 1/10mm. A profundidade era de 1/5mm. Quando o sangue diluído era colocado na câmara e tapado, era muito mais fácil, após os corpúsculos terem assentado, efetuar a contagem.

Figura 2.3: Aparelho de Gower destinado à contagem de células sanguíneas.
Fonte:[Verso, 1964].

Richard Thoma (1847-1923) contribui também com alguns desenvolvimentos importantes. Para diluir o sangue usava uma pipeta baseada na de Potain, mas diferente em certos detalhes, sendo a de Thoma o modelo base de uma das pipetas mais usadas atualmente. A sua câmara de contagem consistia numa placa de vidro com um orifício de 11mm de diâmetro, em cujo centro existia uma placa de 5mm de diâmetro, criando o efeito de um fosso circundante. Esta placa continha 400 campos quadrados, divididos em 25 grupos de 16. Mais tarde ao verificar que uma contagem de leucócitos precisa não era possível numa diluição de 1:200, particularmente na presença de hemácias, criou uma pipeta para leucócitos na qual o sangue podia ser diluído nas proporções 1:10 e 1:20. Também introduziu o ácido acético como fluido diluente dos leucócitos que hemolisava as hemácias. Até esta altura era prática comum contar os leucócitos utilizando as preparações feitas para as hemácias, mas esta nova técnica viria a facilitar a contagem dos leucócitos.

Ao mesmo tempo que Thoma, Malassez introduzia o seu segundo instrumento, que possuía uma plataforma elevada, também rodeada de um fosso, com uma cobertura de vidro mantida num suporte articulado e três pontos de metal na lâmina, nos quais o suporte se apoiava e que determinavam a profundidade da gota de sangue diluído (Figura 2.4).

Sergei Alferow, procurando uniformizar a distribuição das células nas câmaras de contagem desenvolveu uma câmara com uma cobertura destacável mas passível de ser cheia por tração capilar (Figura 2.5). A plataforma de contagem era formada por dois fossos (6a), e a profundidade da célula determinada colocando a cobertura em quatro suportes de metal

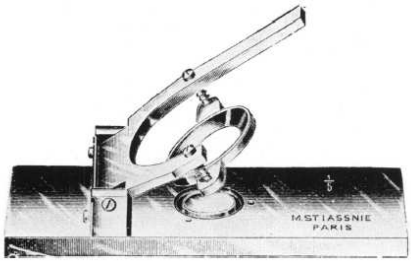


Figura 2.4: Hematócrito de Malassez.
Fonte:[Verso, 1964].

(6b). Quando a preparação estava pronta a cobertura era fixada com recurso a grampos. A imagem das células era então projetada numa placa de vidro com propriedades microfotográficas, possibilitando assim a marcação da posição de cada célula na placa para posterior contagem.

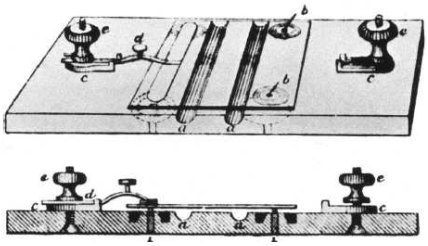


Figura 2.5: Hematócrito de Alferow
Fonte:[Verso, 1964].

Em 1896 George Oliver propunha um novo princípio para contagem de células sanguíneas, com um hematocímetro que funcionava com base no princípio turbidométrico, de que o sangue diluído, como suspensão opaca possibilitava a obtenção de determinadas propriedades óticas à luz da vela. O método nunca se tornou popular devido às variações nos tamanhos, formas e conteúdo de hemoglobina das células, que afetavam a estimativa.

Estas dificuldades estavam discriminadas num artigo de 1902, em que Robert Breuer descrevia as desvantagens da contagem de leucócitos utilizando a câmara de Thoma.

Strong e Seligman, em 1903, viriam a conseguir contar leucócitos diluindo 5ml de sangue na proporção 1:100 numa solução contendo violeta de metilo. Depois de dispostos 5mm^3 numa lâmina, secos, era montado um preparado com Bálsamo do Canadá. Os leucócitos eram contados em toda a área, sendo o cálculo realizado multiplicando o resultado da contagem pela diluição. O mesmo método é utilizável para contar hemácias com a realização de uma diluição de 100×.

Foram, entretanto, e ao longo de vários anos, realizadas modificações em várias das já conhecidas câmaras de contagem. Um dos desenvolvimentos mais importantes foi o aparecimento da câmara de Bürker, que consistia numa placa de base na qual foi colocada transversalmente uma peça de vidro de 25mm de comprimento e 5mm de largura com extremidades arredondadas. Esta peça de vidro era dividida em duas partes por meio de um canal de 1-5mm de profundidade. Um canal de 1-5mm de largura separava-a em cada lado de uma peça retangular de vidro de 21mm de comprimento, 5.5mm de largura e de espessura tal que quando a cobertura fosse colocada restasse um espaço com 0.1mm de profundidade (Figura 2.6). Uma das características que tornava esta câmara superior às anteriores era o facto

de poder ser enchida por capilaridade após a cobertura ter sido colocada. Mais tarde um número de fabricantes modificariam o hematócrito de Bürker construindo-o a partir de uma peça única de vidro. Seguiram-se pequenos melhoramentos à câmara de contagem de Bürker e às técnicas de pipetagem.

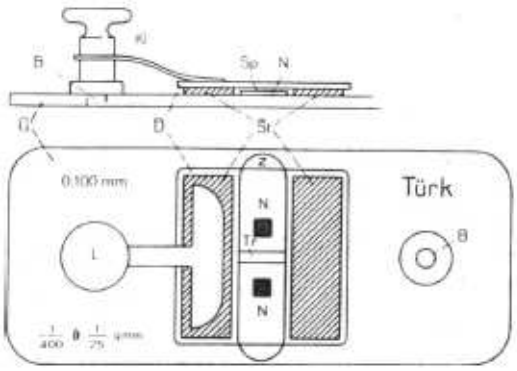


Figura 2.6: Bürker.
Fonte:[Verso, 1964].

Como resultado do sucesso de alguns métodos fotoelétricos pioneiros na contagem de bactérias em suspensão, ressurgiu o interesse na turbidimetria que voltaria a ser condicionada pelos erros causados pela não homogeneidade da suspensão ou pela ocorrência de hemólise.

As dificuldades inerentes aos métodos turbidimétricos foram ultrapassadas com a invenção do electro-hemoscópio criado por Loeschke e Wever. Este baseia-se no princípio de que a luz, ao passar por uma suspensão, é afetada pelo número de células bem como pelo tamanho destas, mas a quantidade de luz refratada depende unicamente do tamanho das células. Ao cobrir uma secção calculada da imagem de difração, a intensidade da luz que é difratada na superfície restante da célula fotográfica é, para todos os efeitos, independente do tamanho da partícula e pode ser medida fotoeletricamente.

Mais recentemente, foram desenvolvidos diferentes tipos de aparelhos para a contagem electrónica de partículas microscópicas (incluindo células sanguíneas) como dispositivos de varrimento de raios catódicos, dispositivos baseados na fraca condutividade das células, contagens baseadas em estimativas feitas a partir do hematócrito, entre outros [Verso, 1964].

A instrumentação para a classificação de leucócitos desenvolveu-se, sobretudo, ao longo de duas vias: características citoquímicas identificadas num sistema de fluxo e tentativas de imitar o globo ocular humano computacionalmente.

Em 1959, Perkin e Elmer iniciavam os seus esforços no sentido de desenvolver um microscópio automatizado ligado a computadores para a classificação automática de leucócitos. Apesar das limitações computacionais conseguiam já em 1966 reconhecer linfócitos, monócitos e neutrófilos com uma precisão de 90%. Este desenvolvimento levou ao aparecimento do dispositivo *Coulter diff 3*. Foram entretanto surgindo dispositivos semelhantes, tendo com alguns deles sido realizados estudos comparativos, como um publicado em 1983 no *Journal of Clinical Laboratory Automation.* Os resultados deste estudo indicavam que os aparelhos apresentavam precisão igual ou superior à dos técnicos de laboratório, com maior exatidão, e velocidade de análise superior na identificação de células anormais. Estes dispositivos exibiam tempos de análise inferiores a 2 minutos (ainda que a preparação do esfregaço tenda a demorar significativamente mais tempo). Uma das desvantagens deste tipo de dispositivos era (e é) o seu elevadíssimo custo, inviabilizando assim a sua difusão pelos hospitais e laboratórios em grande escala [Dutcher, 2016].

Os contadores atuais apresentam como principais metodologias o método ótico e o de impedância. Como contadores de referência podem referir-se o Sysmex XE-5000, o Advia 120 e o Coulter LH750.

Os aparelhos baseados no método ótico utilizam uma luz laser para análise e contagem das células por citometria de fluxo. Nestes contadores o sangue é aspirado e introduzido num fluxo contínuo, que faz com que as células se alinhem (uma a uma) no centro do fluxo, prevenindo desta forma a geração de pulsos anormais. Um laser semicondutor é emitido para o fluxo de células atravessando-o na perpendicular, existindo vários detetores colocados em diferentes posições de alinhamento relativamente ao laser. A radiação é convertida em pulsos elétricos, permitindo assim obter informação acerca das células. O número de pulsos indica o número de células e o ângulo de dispersão de luz fornece informação sobre o tamanho e complexidade das células (Figura [Braga, 2014].

Gert-Jan et al sugeriram num artigo de 2016, um novo teste de citometria de fluxo para a contagem de células em ascite. Com base no desempenho analítico, concluíram que a citometria de fluxo é adequada para a contagem de células no fluido ascítico para aferir a concentração de leucócitos e detetar patologias como peritonite bacteriana espontânea [Geijn et al., 2016].

O sistema de impedância elétrica aplicado à contagem celular tem por base o princípio de "Coulter". Nesta técnica é efetuada a medição de alterações na corrente elétrica durante a passagem das células sanguíneas (suspensas numa solução salina, isotónica e boa condutora), através de um pequeno orifício situado entre dois elétrodos. A fraca condutividade das células causa, à sua passagem, uma alteração do potencial entre os dois elétrodos, gerando um sinal elétrico proporcional ao volume da célula. Esta metodologia é utilizada essencialmente para contagem de hemácias e plaquetas com prévia lise de leucócitos [Braga, 2014].

Em 2012 Khan et al propuseram um método totalmente baseado em tecnologias de visão por computador, utilizável para a contagem de plaquetas, hemácias e leucócitos. Neste método é obtida uma imagem microscópica de uma amostra de sangue previamente preparada com técnicas de diluição tradicionais. É realizada a conversão da imagem para escala de tons de cinzento, com aumento do contraste e posteriormente efetuada uma conversão para imagem binária. Porém, antes da obtenção da imagem binária é realizada a análise do histograma e realizado o *thresholding* da imagem de forma a realçar a presença dos três componente sanguíneos, em três imagens diferentes (implicando a utilização de três *thsresholds* diferentes). As células (*foreground*) são assim separadas do fundo da imagem (*background*). São posteriormente implementadas técnicas para melhoria da imagem como, por exemplo, o filtro Gaussiano. É então possível contabilizar, recorrendo a um algoritmo concebido para o efeito, o número de células presentes na imagem [Khan et al., 2012].

De forma análoga à descrita no método anterior, Ghosh et al (2013), propõem no seu artigo *Automatic white blood cell measuring aid for medical diagnosis* um método com os seguintes passos:

1. Aquisição de imagem através da digitalização de amostras de sangue adquiridas com uma câmara CCD[1] montada num microscópio.

2. Melhoria da imagem com utilização de filtros como o Laplaciano aplicado separadamente aos componentes vermelho, verde e azul da imagem.

[1] *charge-coupled device*

3. Segmentação da imagem com conversão do espaço *RGB*[2] para *HSI*[3] de forma a isolar a componente *H*, que melhor permite identificar os glóbulos brancos. A imagem é posteriormente binarizada utilizando um algoritmo para encontrar o *threshold* de melhor realce das células pretendidas. É posteriormente implementado um segundo algoritmo para encontrar os contornos válidos na imagem. Implementa-se um terceiro algoritmo para separar os núcleos sobrepostos.

4. Deteção de objetos recorrendo a um quarto algoritmo para distinguir neutrófilos, basófilos e eosinófilos. A altura e largura do segmento de monócitos tem grande diferença, enquanto que, no caso de linfócitos, a diferença é menor. Usando esta informação, os monócitos e os linfócitos são identificados.

Este método efetua a distinção e contagem dos vários tipos de leucócitos. Trata-se de um método acessível e rápido e de relativamente fácil implementação [Ghosh et al., 2011].

Em 2014 Saeki et all publicaram um artigo onde descrevem um método de contagem digital integrada. Este método de contagem é realizado usando uma disposição de célula única fabricada num sensor de imagem semicondutor de óxido de metal. A matriz de célula única, construída usando uma matriz de microcavidades, pode capturar até 7.500 células individuais em microcavidades dispostas periodicamente num substrato metálico plano através da aplicação de uma pressão negativa. O método proposto para a contagem de células é baseado na imagem de sombra, que usa um padrão de difração de luz gerado pela matriz de microcavidades e células presas. Sob iluminação, as microcavidades ocupadas por células são visualizadas como padrões de sombra numa imagem registada pelo sensor semicondutor de óxido metálico complementar devido à atenuação da luz. A contagem de células é determinada pela enumeração dos padrões de sombra uniformes criados a partir de relações individuais com células únicas presas nas microcavidades, em formato digital. Todo o processo de contagem é implementado num único dispositivo integrado, miniaturizado, simples e rápido para o uso rotineiro em laboratório. Esta plataforma tem a vantagem de poder ser utilizada em concentrações de células extremamente baixas, isto é, 25 a 15000 células/mL [Saeki et al., 2014].

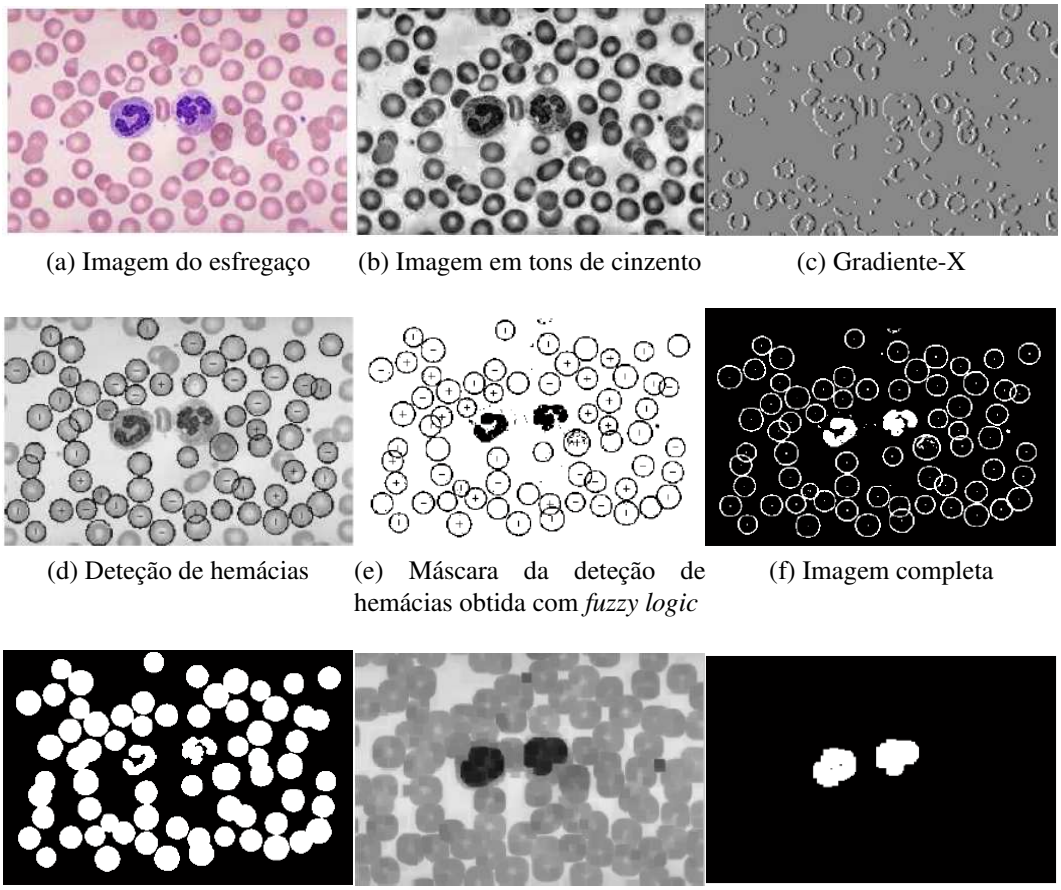
Em 2016, Bhagavathi e Thomas descrevem num artigo a aplicação de *fuzzy logic* na contagem de hemácias e leucócitos. A abordagem da *fuzzy logic* (lógica difusa) no processamento de imagens permite o uso de funções de pertença para definir o grau de pertença de um pixel a uma borda ou a uma região uniforme. Uma função de pertença não é mais que uma curva que fornece informação sobre a forma como cada ponto no espaço de entrada é mapeado para um grau de associação entre 0 e 1. Em *fuzzy logic*, uma simples função IF-THEN é usada para fazer as declarações condicionais.

A imagem RGB do esfregaço de sangue obtida (Figura 2.7 *(a)*) é convertida para imagem de escala de cinza (*b*) e as bordas são detetadas usando regras de *fuzzy logic* (*c*). O centro do círculo é encontrado e o raio é determinado para cada célula (*d*). Obtém-se então uma máscara (*d*) que é complementada (*f*) sendo preenchidas as circunferências obtidas (*g*).

Com base na presença de núcleos e nos valores de intensidade os leucócitos são diferenciados das hemácias. É aplicada erosão aos núcleos dos leucócitos (*h*) e estes são separados das hemácias. A imagem *i)*) mostra as máscara dos leucócitos extraída deste processo. A contagem é feita com recurso a um algoritmo específico e os resultados são exibidos na interface gráfica do usuário [Bhagavathi and Thomas Niba, 2016]. No entanto, neste artigo os

[2] *red, blue, green*

[3] *hue, saturation, lightness*



(a) Imagem do esfregaço (b) Imagem em tons de cinzento (c) Gradiente-X

(d) Deteção de hemácias (e) Máscara da deteção de hemácias obtida com *fuzzy logic* (f) Imagem completa

(g) Preenchimento de buracos (h) Erosão na imagem do esfregaço sanguíneo. (i) Máscara dos leucócitos

Figura 2.7: Processo de segmentação de leucócitos.
Fonte:[Bhagavathi and Thomas Niba, 2016].

autores não fazem referência à percentagem de acerto conseguida.

Em 2016, Zhao et al propuseram um sistema automático de deteção e classificação de glóbulos vermelhos a partir de imagens de sangue periférico. Em primeiro lugar, propõem um algoritmo para detetar leucócitos a partir das imagens obtidas ao microscópio com base na relação simples das cores vermelho e azul, e operações morfológicas. Em seguida, aplicaram um recurso de granularidade e *SVM* (*Support Vector Machine*) para distinguir eosinófilos e basófilos de outros glóbulos. Por fim, utilizaram redes neuronais de convolução para extrair automaticamente *features* de alto nível de leucócitos, e uma *Random Forest* para reconhecer os outros três tipos de leucócitos (neutrófilos, monócitos e linfócitos). A precisão de classificação média obtida foi de 92.8% [Zhao et al., 2017].

Em 2016 Falcón-Ruiz et al, fizeram um estudo que incidia sobre a possibilidade de utilizar características da forma do núcleo para classificar os glóbulos brancos no sangue periférico, em virtude da segmentação do núcleo ser muito mais fácil do que a segmentação de toda a célula, sobretudo em casos de toque, sobreposição e células impropriamente coradas ou danificadas. Os classificadores utilizados foram *Multi-Layer Perceptron*, *Support Vector Machine*, *k-Nearest Neighbors*, PART e C4.5. Os resultados demonstraram que utilizando apenas características do núcleo é possível obter precisões na ordem dos 96% [Falcón-Ruiz et al.,].

Em 2016 Bayramoglu et al, escreveram sobre a viabilidade do *transfer learning* no livro *"Transfer Learning for Cell Nuclei Classification in Histopathology Images"*. Neste livro os autores abordam o *transfer learning* na análise de imagens biomédicas, como forma de reduzir o esforço de rotulagem manual de dados, e analisam quantitativamente os desempenhos do *transfer learning* comparativamente à aprendizagem de raiz de redes neuronais para a classificação dos núcleos celulares. São avaliadas quatro arquiteturas diferentes de CNN (*Convolutional Neural Network*) treinadas em imagens naturais e imagens faciais. Os resultados mostraram que os seus modelos pré-treinados não apenas melhoraram o desempenho preditivo, como também exigiram menos tempo de treino [Bayramoglu and Heikkilä, 2016].

Em 2017 Othman et al abordaram também a problemática da identificação de glóbulos brancos. Os investigadores fizeram a classificação de cinco tipos de glóbulos brancos usando uma rede neuronal. Após a segmentação das células do sangue obtidas a partir de imagens microscópicas, as 16 características mais significativas dessas células foram fornecidas como entrada na rede. Utilizaram metade das 100 sub-imagens de leucócitos segmentadas para treinar a rede e a outra metade para teste. Os resultados fornecidos alegam uma precisão de classificação de 96% [Othman et al., 2017].

Em 2017 Dhruv Parthasarathy utilizou uma rede LeNet para efetuar a classificação de imagens de leucócitos em mononucleares e polionucleares, obtendo uma precisão de classificação de 98.6% [Parthasarathy, 2017].

Capítulo 3

Redes Neuronais

Os humanos são extraordinariamente competentes a retirar sentido daquilo que os olhos lhes mostram, mas praticamente todo este trabalho é realizado inconscientemente. A dificuldade de reconhecimento de padrões torna-se aparente quando tentamos escrever um programa de computador que reconheça leucócitos em imagens de microscopia. Quando procuramos um conjunto de regras precisas para o reconhecimento de formas, rapidamente nos perdemos numa imensidão de exceções, ressalvas e casos especiais. No entanto, as redes neuronais ou NN (*Neural Networks*) abordam este tipo de situações de forma diferente [Nielsen, 2015].

Inspiradas pelas redes neuronais biológicas, as NN são sistemas de computação paralela que consistem num grande número de processadores simples interligados. Estes modelos procuram fazer uso de princípios organizacionais que se acredita serem usados no cérebro humano. As NN podem ser vistas como grafos dirigidos em que neurónios artificiais são nodos (ou vértices), e as arestas são as ligações entre as entradas e saídas dos neurónios. Com base na sua arquitetura de base podem ser agrupados em duas categorias:

- *feedforward networks*, em que os grafos não possuem laços;
- *recurrent (feedback) networks*, em que os laços existem devido a ligações de feedback.

Na família mais comum das redes *feedforward* - as MLP (*multilayer perceptron*) - os neurónios estão organizados em camadas que têm ligações unidirecionais entre eles. Diferentes conectividades produzem diferentes comportamentos. Genericamente falando, as redes *feedforward* são estáticas, isto é, produzem apenas um tipo de valores de saída em vez de uma sequência de valores de uma dada entrada. As redes *feedforward* são redes sem memória no sentido em que a sua resposta é independente do estado anterior da rede. As redes recorrentes ou de *feedback*, por outro lado, são sistemas dinâmicos. Devido aos caminhos de *feedback*, as entradas de cada neurónio são modificadas, o que leva a que a rede neuronal entre num novo estado dependente dos estados passados.

Ainda que a definição precisa de inteligência seja difícil de formular, o processo de aprendizagem no contexto das redes neuronais pode ser visto como um problema de atualização da arquitetura da rede e dos pesos das ligações, de forma a que essa rede consiga desempenhar uma determinada tarefa. A rede aprende organizando os pesos das ligações a partir de padrões de treino disponíveis. O desempenho é então melhorado com o tempo através da atualização iterativa dos pesos da rede.

A capacidade de aprendizagem automática a partir de exemplos é o que torna as redes neuronais tão interessantes. Em vez de seguirem um conjunto de regras especificadas por humanos, as NN apreendem as regras subjacentes a uma coleção de exemplos representativos

de uma determinada categoria. Esta é uma das maiores vantagens das NN relativamente aos sistemas tradicionais.

Existem 3 paradigmas básicos de aprendizagem: supervisionada, não supervisionada e de reforço. Na aprendizagem supervisionada, é providenciada uma resposta correta para cada padrão de entrada. Os pesos das ligações são determinados de forma a permitirem que a rede produza respostas tão próximas quanto possível das respostas corretas conhecidas. A aprendizagem não supervisionada não requer uma informação associada ao padrão de entrada. Esta explora a estrutura da informação disponível, das correlações entre padrões e organiza estes padrões em categorias. A aprendizagem híbrida combina a aprendizagem supervisionada e não supervisionada.

A teoria da aprendizagem deve contemplar 3 problemas fundamentais associados à aprendizagem a partir de exemplos: a capacidade, a complexidade das amostras e a complexidade computacional. A capacidade diz respeito ao número de padrões que podem ser guardados, e quais funções e limites de decisão a rede pode formar. A complexidade das amostras diz respeito ao número de padrões de treino necessários para treinar a rede para garantir uma generalização válida. Demasiadas iterações de treino podem originar *overfitting*, o que faz com que a rede funcione bem com a base de treino, funcionando mal em padrões de teste independentes. A complexidade refere-se ao tempo necessário para um algoritmo de aprendizagem estimar a solução a partir dos padrões de treino [Jain et al., 1996]. Muitos dos algoritmos existentes possuem elevada complexidade computacional. Existem também quatro tipos básicos de regras de aprendizagem: *error-conection*, Boltzmann, Hebbian e aprendizagem competitiva, cuja definição se encontra fora do espectro deste trabalho.

3.1 Arquiteturas de redes neuronais

Algumas NN têm várias camadas escondidas. Por exemplo, a NN de quatro camadas da Figura 3.1 tem duas camadas escondidas:

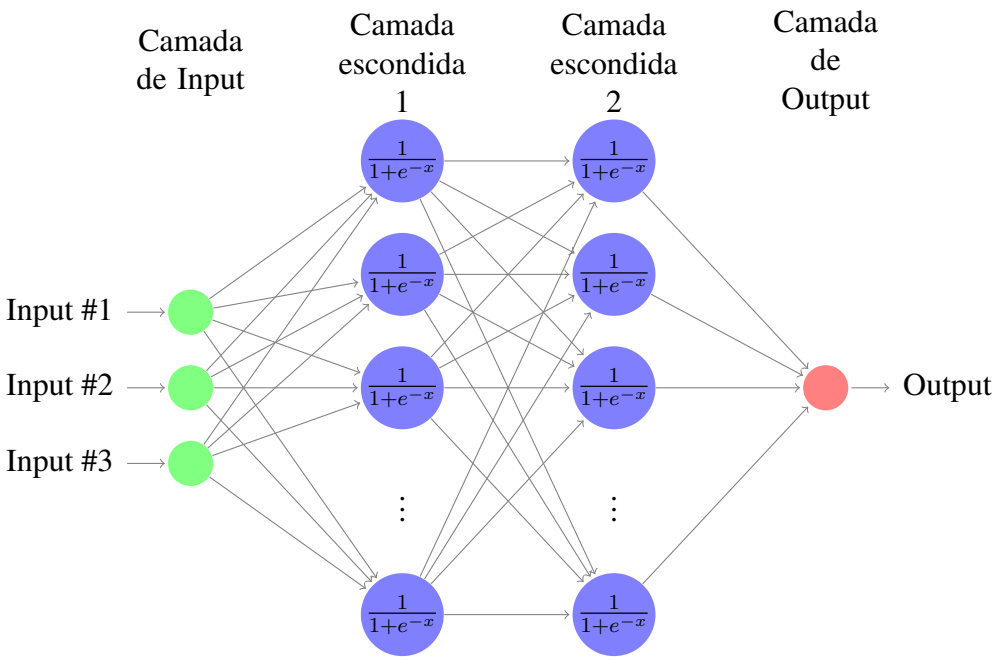


Figura 3.1: Rede neuronal com duas camadas escondidas.

De forma algo confusa, e por motivos de ordem histórica, estas redes de múltiplas camadas são por vezes chamadas de MLP, apesar de serem constituídas por neurónios sigmoides.

Ainda que a conceção das camadas de *input* e *output* de uma rede neuronal seja relativamente simples, pode ser muito difícil conceber as camadas escondidas. Em particular, não é possível resumir o processo de conceção das redes escondidas através de algumas regras simples. Ao invés, os investigadores na área das redes neuronais têm desenvolvido várias conceções heurísticas para as camadas escondidas, que ajudam os utilizadores a obter os resultados desejados das suas redes. Estas técnicas podem ser utilizadas para determinar a troca do número de camadas escondidas pelo tempo requerido para treinar a rede [Nielsen, 2015].

Embora as redes de uma ou duas camadas ocultas possam aprender "tudo", em teoria, o *deep learning*, que faculta a utilização de camadas escondidas, facilita uma representação mais complexa de padrões nos dados.

3.2 Treino

No início da implementação de uma CNN, os pesos ou os valores do filtro podem ser definidos aleatoriamente ou utilizando valores muito pequenos (mas distintos entre si). Nesta fase, os filtros das camadas iniciais não sabem procurar bordas e curvas, nem os das camadas mais altas sabem procurar por patas ou bicos. A ideia de fornecimento de uma imagem com um rótulo é o processo de treino pelo qual as CNN passam. Assim, temos um conjunto de treino que tem milhares de imagens das classes que se pretende identificar e cada uma das imagens tem um rótulo da respetiva categoria [Deshpande, 2016].

Dá entrada numa CNN uma imagem de treino de um dígito manuscrito, por exemplo uma matriz de $32 \times 32 \times 3$ que passa por toda a rede. No primeiro exemplo de treino, com todos os pesos ou valores de filtro inicializados aleatoriamente, a saída será provavelmente algo como [.1 .1 .1 .1 .1 .1 .1 .1 .1 .1]. Basicamente, origina uma saída que não dá preferência a qualquer número em particular. Digamos que a primeira imagem de treino inserida era um "3". O rótulo da imagem seria [0 0 0 1 0 0 0 0 0 0]. Aqui entra a função de perda, que pode ser definida de muitas maneiras diferentes, como o MSE (*mean square error*[1]) definido como:

$$E_{total} = \sum 1/2(target - output)^2 \tag{3.1}$$

Durante o treino o valor de cada peso, w, é atualizado de acordo com a direção negativa do gradiente de E, até que que E se torne suficientemente pequeno. Aqui, o parâmetro η é chamado de taxa de aprendizagem. Se utilizarmos apenas um exemplo de treino de cada vez para atualizar o w, então uma função de erro por amostra E_k dada por

$$E = \frac{1}{2}\sum_{j=1}^{m}(y_j(x_k, w) - d_{jk})^2 \tag{3.2}$$

é usada e w é atualizado como $w = w - \eta\frac{\partial E}{\partial w}$ [Zhang and Gupta, 2000].

A taxa de aprendizagem (*learning rate*) é um parâmetro escolhido pelo programador (podendo ser definido aleatoriamente). Uma elevada taxa de aprendizagem significa que são utilizadas etapas maiores nas atualizações de peso e, portanto, pode levar menos tempo para

[1] Erro quadrático médio

que o modelo converja num conjunto ideal de pesos. No entanto, uma taxa de aprendizagem muito alta pode resultar em saltos que são demasiado grandes, e não precisos o suficiente para alcançar o ponto ideal, como na Figura 3.2.

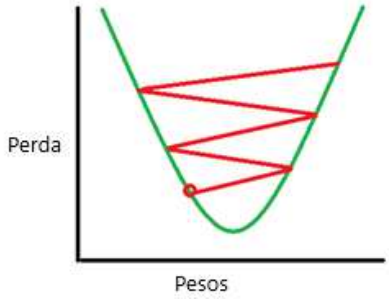


Figura 3.2: Exemplo ilustrativo da consequência de uma taxa de aprendizagem elevada onde os saltos são demasiado grandes inviabilizando a minimização da perda.
Fonte:[Deshpande, 2016].

Todo o processo descrito constitui uma iteração de treino. O programa irá repetir este processo para um número fixo de iterações para cada conjunto de imagens de treino (*batch* ou lote). Depois de terminar a atualização do parâmetro no último exemplo de treino, espera-se que a rede esteja treinada, significando isto que os pesos das camadas se encontram ajustados corretamente [Deshpande, 2016].

3.3 Funções de ativação

As redes neuronais podem usar vários tipos de funções de ativação. A seleção destas é uma consideração importante a fazer, umas vez que tem impacto direto no processamento da informação de *input*.

3.3.1 Função *step* e o perceptrão

Para perceber o que é uma rede neuronal é necessário compreender o modo de funcionamento dos seus componentes. Um dos componentes historicamente mais importantes é o perceptrão. Os perceptrões foram desenvolvidos nos anos 50 do século passado pelo cientista Frank Rosenblatt, inspirados pelo trabalho de Warren McCullock e Walter Pitts. Hoje é comum usar modelos semelhantes de neurónios artificiais que muito devem ao trabalho desenvolvido por Rosenblatt.

Um perceptrão recebe vários *inputs* binários, x_1, x_2,...,x_n, e produz um único output.

Para expressar a importância dos respetivos *inputs* e *outputs*, Rosenblatt introduziu pesos, w_1, w_2,...,w_n. Assim, o output de um neurónio, 0 ou 1, é determinado pelo facto de a soma ponderada $\sum_j w_j x_j$ ser maior ou menor do que um valor limiar (ou *threshold*). Tal como os pesos, o *threshold* é um número real e é um parâmetro do neurónio. Em termos algébricos temos:

$$output = \begin{cases} 0 & \text{se } \sum_j w_j x_j \leq 0, \\ 1 & \text{se } \sum_j w_j x_j > 0 \end{cases} \tag{3.3}$$

Um perceptrão pode ser visto como um dispositivo que toma decisões com base em evidências, consubstanciadas na variação dos pesos e em função de um *threshold*, sendo possível obter diferentes modelos de tomadas de decisão.

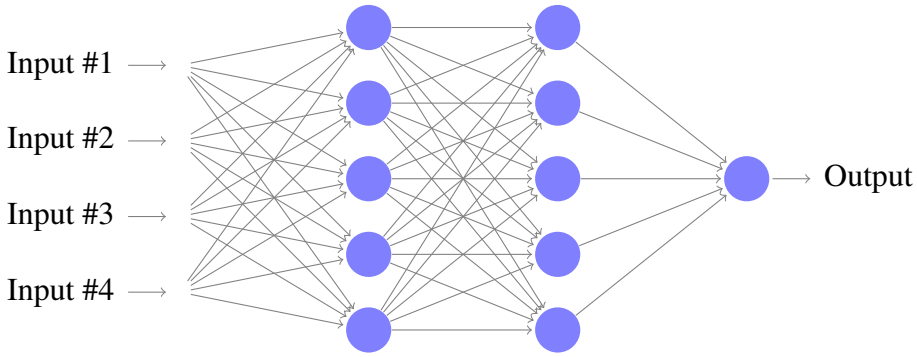


Figura 3.3: Rede neuronal de perceptrões.
Fonte:

No modelo representado na Figura 3.3 a primeira coluna (ou camada) de perceptrões toma decisões muito simples, pesando a informação dos *inputs*. Cada um dos perceptrões da segunda camada efetua uma decisão de nível mais complexo e abstrato, pesando os resultados obtidos da primeira camada. Uma terceira camada toma decisões ainda mais complexas. Desta forma, uma rede neuronal de várias camadas possibilita tomadas de decisão mais sofisticadas.

Os perceptrões também podem ser usados para calcular funções lógicas elementares geralmente associadas à computação, como as funções AND, OR, e NAND. Parta-se de um perceptrão com 2 inputs, cada um com peso -2, e um *b* total de 3. É possível verificar que o input 00 produz o output 1, dado que $(-2) * 0 + (-2) * 0 + 3 = 3$, é positivo. Cálculos similares mostram que os inputs 01 e 10 produzem o output 1. Mas o input 11 produz o output 0, uma vez que $(-2) * 1 + (-2) * 1 + 3 = -1$, é negativo. E assim o perceptrão implementa o operador lógico NAND. De facto, é possível usar redes de perceptrões para calcular qualquer função lógica. A razão pela qual o NAND é universal na computação deve-se à possibilidade de, com este, se conseguir qualquer operação computacional.

É importante pensar nos perceptrões de entrada como unidades especiais que se encontram definidas para produzir os valores desejados, x_1, x_2,...,x_n. As redes de perceptrões podem ser tão poderosas como qualquer outro dispositivo de computação, no entanto, é possível elaborar algoritmos de aprendizagem que conseguem afinar os pesos e *bias* de uma rede de neurónios artificiais automaticamente. Isto acontece em resposta ao estímulo exterior, sem intervenção direta do programador. Torna-se assim possível utilizar os neurónios artificiais de uma forma totalmente diferente dos operadores lógicos convencionais. Ao invés de elaborar um circuito de operadores NAND e outros, a rede neuronal consegue simplesmente aprender a resolver problemas que seriam extremamente difíceis de resolver com um circuito convencional [Nielsen, 2015].

Sumariamente, os percetrões fazem uso de uma função de ativação em degrau (*step*), uma função simples apresentada na Figura 3.4.

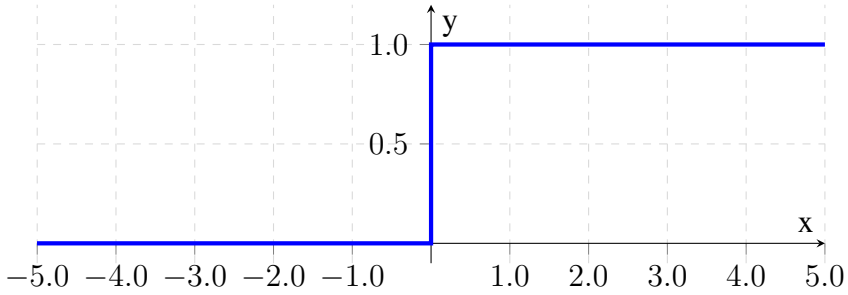


Figura 3.4: Função *step*.

A função apresenta um valor de y igual a 1 para inputs de x maiores ou iguais que 0, e y igual 0 para todos os valores negativos de x. As funções *step* são também chamadas de funções *threshold* porque devolvem 1 apenas para os valores acima de um determinado limite.

3.3.2 Função logística e o neurónio sigmoide

Supondo que os inputs de uma rede sejam constituídos pela informação crua de pixeis de um leucócito, pretende-se definir os pesos e *bias* de forma que a rede efetue a sua classificação corretamente.

O problema é que uma pequena alteração nos pesos ou *bias* de um único perceptrão da rede pode originar a alteração do output de 0 para 1, podendo consequentemente alterar o comportamento do resto da rede de forma mais ou menos imprevisível. É possível solucionar este problema através da introdução de um novo tipo de neurónio artificial, o neurónio sigmoide. Estes são similares aos perceptrões, mas modificados de forma a que pequenas alterações nos seus pesos e *bias* produzam apenas pequenas alterações no seu output. Tal como um perceptrão, o neurónio sigmoide tem *inputs* x_1, x_2,... Mas ao invés de apenas 0 e 1, os *inputs* podem assumir quaisquer valores entre 0 e 1. O *output* destes é $\sigma(w.x + b)$, onde σ é chamada de função sigmoide, definida por:

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}. \tag{3.4}$$

De forma mais explícita, o *output* do neurónio sigmoide com inputs x_1, x_2,..., pesos w_1, w_2,.., e *bias* b é:

$$\frac{1}{1 + \exp(-\sum_j w_j x_j - b)}. \tag{3.5}$$

Existem muitas similaridades entre neurónios sigmoides e perceptrões. Suponhamos que $z \equiv w \cdot x + b$ é um número grande, inteiro e positivo. Então $e^{-z} \approx 0$ e $\sigma(z) \approx 1$. Por outras palavras, o output do neurónio sigmoide é aproximadamente 1, tal como aconteceria para o perceptrão. Suponhamos que $z \equiv w \cdot x + b$ é um número muito negativo. Então $e^{-z} \to \infty$ e $\sigma(z) \approx 0$. É apenas quando $w \cdot x + b$ é de tamanho moderado que existe um desvio relativamente ao modelo com perceptrões. A função sigmoide é uma versão suavizada da função *step* (degrau).

A suavidade do σ implica que pequenas variações nos pesos e *bias* produzam pequenas alterações no output do neurónio. De facto, é possível aproximar o $\Delta output$ por:

$$\Delta\text{output} \approx \sum_j \frac{\partial\text{output}}{\partial w_j}\Delta w_j + \frac{\partial\text{output}}{\partial b}\Delta b, \tag{3.6}$$

onde a soma é relativa a todos os pesos, w_j, ∂output$/\partial w_j$ e ∂output$/\partial b$ denotam derivadas parciais do output com respeito a w_j e b, respetivamente. A equação significa simplesmente que o $\Delta output$ é uma função linear das mudanças ∂w_j e ∂b nos pesos e *bias*. Esta linearidade aumenta em muito a previsibilidade na utilização dos neurónios sigmoides. σ é comummente usado em trabalhos com redes neuronais e é possivelmente a função de ativação mais usada. A interpretação do *output* de um neurónio sigmoide deve ser feita em função da situação de utilização da rede neuronal. O *output* de valores entre 0 e 1 pode ser útil para representar a intensidade dos pixeis numa imagem de *input* de uma rede neuronal. Noutros casos, é possível efetuar classificações através da criação de convenções. Por exemplo um valor superior a 0.5 poderá ser indicativo de uma determinada categoria em detrimento de outra [Nielsen, 2015]. Os seus valores restringem-se ao intervalo [0,1] e o seu nome advém do formato gráfico obtido, visível na Figura 3.5.

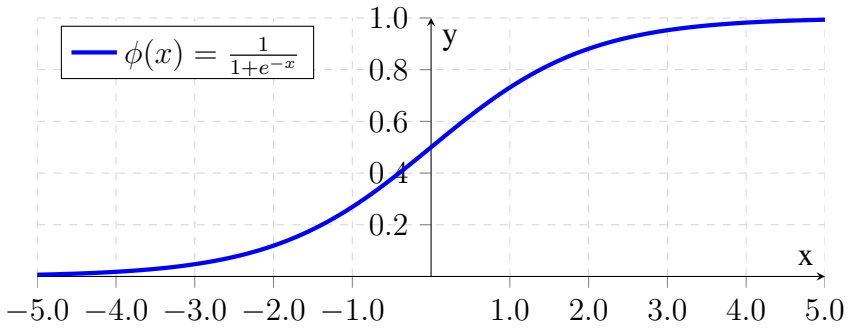


Figura 3.5: Função *sigmoide*.

3.3.3 Função tangente hiperbólica

A função tangente hiperbólica é uma das funções de ativação mais importantes. Encontra-se restringida ao intervalo [-1,1]. A função tangente hiperbólica (Figura 3.6) apresenta uma forma similar à função sigmoide, apresentando algumas vantagens sobre a sigmoide, dependendo da situação, como por exemplo a saturação mais rápida [Fausett and Fausett, 1994].

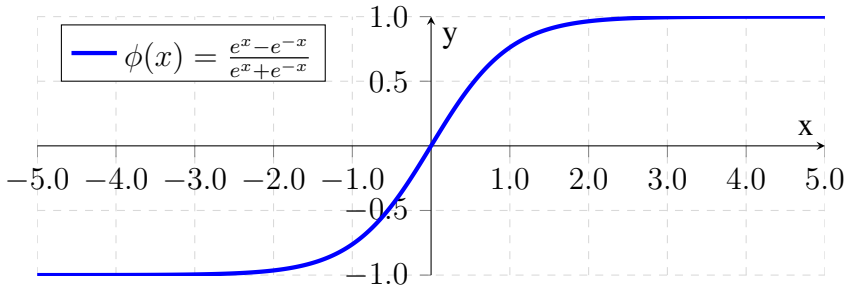


Figura 3.6: Função *tangente hiperbólica*.

3.3.4 Função *ReLU*

Em 2010 Teh e Hinton introduziram a *ReLU* (*Rectified Linear Unit*), que, apesar da sua simplicidade, constitui uma boa escolha para as camadas escondidas. A vantagem da *ReLU*

(representada na Figura 3.7) reside parcialmente no facto de ser linear e não saturada. Ao contrário das funções sigmoide e tangente hiperbólica, a *ReLU* não satura no domínio positivo, encontrando-se saturada no domínio negativo. Uma função de saturação tende a mover-se para um determinado valor [do Nascimento, 2016].

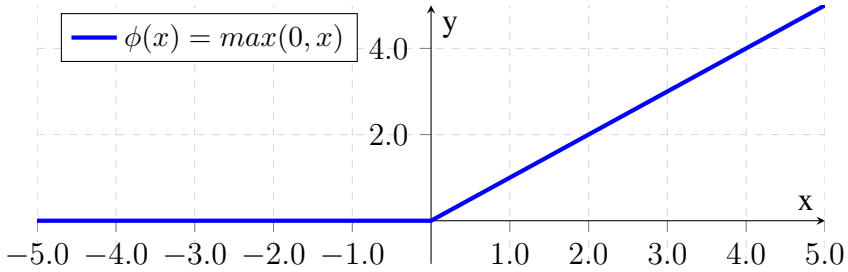


Figura 3.7: Função *ReLU*.

A literatura sobre neurociências indica que os neurónios corticais raramente estão no seu regime de saturação máxima e sugerem que a sua função de ativação pode ser aproximada por um retificador [Bush and Sejnowski, 1995].

Após a inicialização uniforme dos pesos, cerca de 50% dos valores de saída contínua das camadas escondidas são zeros reais, e esta fração pode aumentar facilmente com a regularização induzida por *sparsity* (relativa ao número de conexões entre um neurónio com outros, que se pretende minimizar), sendo biologicamente mais plausível que as funções de ativação anteriores [Glorot et al., 2011]. A função ReLU é habitualmente utilizada nas camadas escondidas, ao passo que a sigmoide e a *Softmax* se utilizam com frequência na camada de *output*.

3.3.5 Função *Softmax*

Juntamente com a função linear, a *softmax* é geralmente encontrada na camada de saída de uma rede neuronal. A função de ativação *softmax* força a saída da rede neuronal a representar a probabilidade de a entrada se inserir em cada uma das classes. Sem a *softmax*, os *outputs* são simplesmente valores numéricos, sendo o maior indicativo da classe vencedora. Quando inserimos informação na rede neuronal aplicando a função de ativação *softmax* obtém-se a probabilidade de o *input* poder ser categorizado de acordo com classes pré-definidas. A fórmula é a seguinte:

$$\phi_i = \frac{e^{z_i}}{\sum_{j \in group} e^{z_j}} \tag{3.7}$$

em que i representa o índice do nó de saída e j representa os índices de todos os nós no grupo ou nível. A variável z designa o vetor dos nós de saída. É importante notar que a função *softmax* é calculada de forma diferente das outras funções de ativação fornecidas. Ao usar a *softmax*, a saída de um único nó depende dos outros nós de saída. Na equação anterior, a saída dos outros nós de saída está contida na variável z, ao contrário das outras funções de ativação [Kloss, 2015].

3.4 *Feedforward*

Como vimos, as redes em que o *output* de um neurónio é utilizado como *input* do neurónio da camada seguinte são chamadas de redes neuronais *feedforward*. Isto significa que não existem laços ou ciclos na rede, a informação é sempre passada em frente e não existe realimentação [Nielsen, 2015].

Dados os *inputs* $x = [x_1, x_2, ..., x_n]^T$ e os pesos w, uma rede neuronal *feedforward* é usada para calcular os *outputs* $y = [y_1, y_2, ..., y_m]^T$ de uma rede *MLP*. No processo *feedforward*, os *inputs* externos alimentam os neurónios de *input* (primeira camada). Os *outputs* dos neurónios de *input* alimentam os neurónios escondidos da segunda camada, e assim sucessivamente, e finalmente os *outputs* da camada $L-1$ alimentam os neurónios da última camada (L) [Zhang and Gupta, 2000]. O cálculo é dado por

$$z_i^l = x_i, i = 1, 2, ..., N_1, N_1 = n \tag{3.8}$$

$$z_i^l = \sigma(\sum_{j=0}^{N_{l-1}} w_{ij}^l z_j^{l-1}), i = 1, 2, ..., N_l, l = 2, 3, ..., L. \tag{3.9}$$

Os *outputs* da NN são extraídos dos neurónios de *output* como

$$y_i = z_i^L, i = 1, 2, ..., N_L, N_L = M. \tag{3.10}$$

3.5 *Backpropagation*

O objetivo no desenvolvimento de modelos neuronais é encontrar um conjunto ótimo de pesos w, de forma a que $y = y(x, w)$ aproxime o comportamento original de um problema. Isto é conseguido através do processo de treino (ou seja, da otimização do espaço-w). Um conjunto de dados de treino é apresentado à rede neuronal, consistindo em pares de (x_k, d_k), $k = 1, 2, ..., P$, onde d_k é o *output* desejado no modelo para os *inputs* x_k, e P é o número total de exemplos de treino.

Durante o treino, a performance da rede neuronal é avaliada pelo cálculo da diferença entre os seus *outputs* e o *output* desejado para todos os exemplos de treino. A diferença, também conhecida como erro, é quantificada por

$$E = \frac{1}{2} \sum_{k \in T_r} \sum_{j=1}^{m} (y_j(x_k, w) - d_{jk})^2 \tag{3.11}$$

onde d_{jk} é o elemento j de $d_k, y_j(x_k, w)$ é o *output* j da rede neuronal para o *input* x_K e T_r é um índice do conjunto de treino. Os pesos w são ajustados durante o treino, de forma a que este erro seja minimizado. Em 1986, Rumelhart, Hinton, e Williams propuseram uma abordagem sistemática ao treino das NN. Uma das contribuições mais significativas do seu trabalho é o algoritmo de *backpropagation* [Zhang and Gupta, 2000].

3.6 Métodos de otimização

O termo otimização refere-se à procura de valores mínimos ou máximos para uma dada função, através de uma escolha sistemática de valores variáveis dentro de um conjunto viável.

Pretende-se portanto encontrar uma solução ótima para um problema, que redunde no melhor desempenho possível do sistema. Alguns dos métodos de otimização de NN mais utilizados são o SGD, NAG, Adam e RMSprop. Segue-se uma breve explicação das suas principais características.

3.6.1 Stochastic Gradient Descent (SGD)

No algoritmo de *backpropagation*, o gradiente deve ser calculado muitas vezes para ajustar os pesos da rede neuronal. Quando o conjunto de treino é muito grande, em geral, calcular o gradiente para todo o conjunto é impraticável em termos de recursos computacionais exigidos. A pensar nisto usa-se o gradiente descendente estocástico, calculado com alguns exemplos iterativamente (em vez de toda a base de treino). Outra vantagem de usar o SGD é a capacidade de reduzir a ocorrência de mínimos locais. São utilizados mini-lotes porque reduzem a variância na aprendizagem e, portanto, tem uma convergência mais estável, desde que a distribuição dos exemplos seja aleatória. Com o elevado poder computacional dos GPU[2], os mini-lotes podem também ser processados rapidamente, uma vez que a operação é facilmente paralelizada [do Nascimento, 2016]. A equação seguinte mostra o passo de atualização do SGD,

$$\theta = \theta - \alpha * \sum_{k=i}^{i+m} \nabla_\theta J(\theta; x^{(k)}, y^{(k)}) \tag{3.12}$$

onde θ é o parâmetro a atualizar, α é a taxa de aprendizagem e m é o tamanho do mini-lote.

3.6.2 Nesterov Accelerated Gradient (NAG)

O gradiente acelerado de Nesterov (NAG) é uma maneira de atribuir ao momento alguma presciência. *Momentum* é um método que ajuda a acelerar o SGD na direção relevante e amortece as oscilações. Isto é conseguido adicionando uma fração γ do vetor de atualização do último passo de tempo ao vetor de atualização atual. Assim usa-se o termo do momento $\gamma\vartheta_{t-1}$ para atualizar os parâmetros θ.

Calculando $\theta - \gamma\vartheta_{t-1}$ dá-nos uma aproximação da próxima posição dos parâmetros e é possível calcular uma posição futura aproximada dos parâmetros:

$$\vartheta_t = \gamma\vartheta_{t-1} + \eta\nabla_\theta J(\theta - \gamma\vartheta_{t-1}) \tag{3.13}$$

$$\theta = \theta - \vartheta_t \tag{3.14}$$

Esta atualização antecipada possibilita ao algoritmo acelerar resultando numa maior capacidade de resposta, o que aumenta significativamente o desempenho, por exemplo, das RNN em várias tarefas. Assim é possível adaptar as atualizações à inclinação da função de erro e acelerar o SGD [Ruder, 2016].

[2] *Graphics Processing Unit*

3.6.3 Adam

Adam[3] é um método para otimização estocástica eficiente que requer apenas gradientes de primeira ordem com pouca necessidade de memória. O método calcula as taxas individuais de aprendizagem adaptativa para diferentes parâmetros a partir das estimativas dos primeiro e segundo momentos dos gradientes. O nome Adam é derivado da estimativa do momento adaptativo. Este método é projetado para combinar as vantagens dos métodos AdaGrad e RMSProp [Kingma and Ba, 2014].

Em muitos problemas, possui um desempenho superior ao de outros métodos como AdaGrad, RMSProp, SGD e NAG. Em geral, o Adam é uma boa escolha porque é rápido e não precisa de um cronograma de aprendizagem muito rígido [do Nascimento, 2016].

3.6.4 RMSprop

O *RMSProp* (Root Mean Square Propagation) é um dos algoritmos de gradiente adaptativo mais usados para o treino de redes neuronais profundas. Tem sido utilizado frequentemente em visão por computador, por exemplo, para treinar a mais recente rede *InceptionV4*. O *RMSProp* mostrou em alguns estudos resultados superiores a outros métodos adaptativos, como *AdaGrad*, *Adadelta* e *SGD* (com momento) num grande número de testes. Apesar do seu enorme sucesso empírico, precisa ainda de uma análise teórica rigorosa [Mukkamala and Hein, 2017]. O seu algoritmo é dado por:

$$\begin{aligned} E[g^2]_t &= 0.9E[g^2]_{t-1} + 0.1g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \end{aligned} \tag{3.15}$$

onde g é o gradiente num dado instante t, η é o *learning rate* e θ representa os parâmetros a atualizar (W e b).

3.7 *Overfitting*

As redes de *Deep Learning* contêm múltiplas camadas ocultas não-lineares que podem aprender relações muito complexas entre as suas entradas e saídas. Com dados de treino limitados, no entanto, muitas dessas relações serão o resultado do ruído de amostragem que existe no conjunto de treino, mas não em dados de teste reais, mesmo que extraídos da mesma distribuição. Esta situação tende a originar *overfitting* [Srivastava et al., 2014].

As redes neuronais e outros modelos de *machine learning* são propensos ao *overfitting*, que se traduz numa sobre-adaptação de um determinado modelo a um conjunto de dados, perdendo este modelo a capacidade de generalização, isto é, perdendo a sua aplicabilidade a conjuntos de dados diferentes. A Figura 3.8 ilustra o conceito usando aproximações polinomais. Neste exemplo é apresentado um conjunto de treino com 21 pontos, de acordo com a equação $y = \sin(x/3) + v$, onde v é uma variável aleatória uniformemente distribuída entre -0.25 e 0.25. A equação foi avaliada em 0, 1, 2, ..., 20. Este conjunto de dados foi usado para adaptar modelos polinomiais com variações de ordem entre 2 e 20. Para o polinómio de ordem 2, a aproximação é pobre, sendo bastante melhor para o polinómio de ordem 10. No

[3]Nome derivado de *adaptive moment estimation*

entanto, à medida que a ordem aumenta, existe uma tendência para o *overfitting*. No exemplo de ordem 20 a função coincide com os dados de treino, no entanto, a interpolação entre pontos é muito pobre. O *overfitting* pode também ser um problema a ter em conta quando se trabalha com redes neuronais, tendo já sido desenvolvidas várias técnicas como paragem precoce e decaimento de pesos [Lawrence et al., 1997].

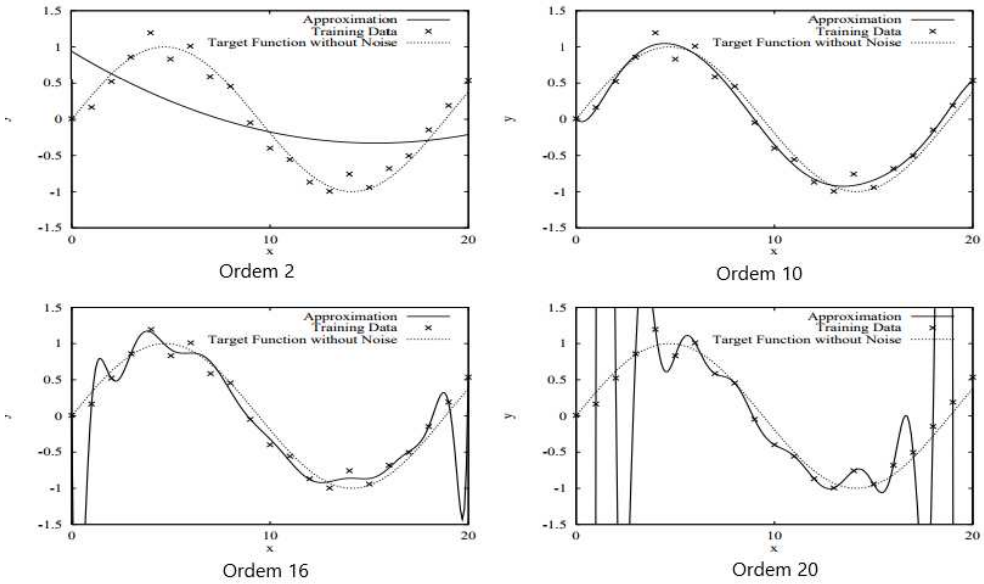


Figura 3.8: Adaptação de modelos polinomiais de ordem 2, 10, 16 e 20 à equação $y = \sin(x/3) + v$, com *overfitting* nos modelos de ordem 16 e 20.
Fonte:[Lawrence et al., 1997].

A seleção do tamanho do modelo que maximiza a generalização é um tópico importante. As redes com mais pesos do que o esperado podem resultar num treino menos eficiente e em erros de generalização em certos casos. O comportamento de *overfitting* é significativamente diferente em MLP e modelos polinomiais.

3.8 *Dropout*

Dropout é uma técnica utilizada para evitar o *overfitting*. Esta fornece uma maneira de combinar aproximadamente exponencialmente várias arquiteturas de redes neuronais diferentes de forma eficiente. O termo *dropout* refere-se à exclusão de unidades (escondidas e visíveis) numa rede neuronal. Ao descartar uma unidade removem-se, ao longo do treino, todas as suas conexões de entrada e saída, como mostra a Figura 3.9. A escolha das unidades a serem descartadas é aleatória. No caso mais simples, cada unidade é retida com uma probabilidade *p* independente de outras unidades, onde *p* pode ser escolhido usando um conjunto de validação ou pode simplesmente ser ajustado em 0.5, o que parece ser próximo ao ideal para uma ampla gama de redes e tarefas. No entanto, para as unidades de *input* a probabilidade ótima de retenção é geralmente mais próxima de 1 do que de 0.5 [Srivastava et al., 2014].

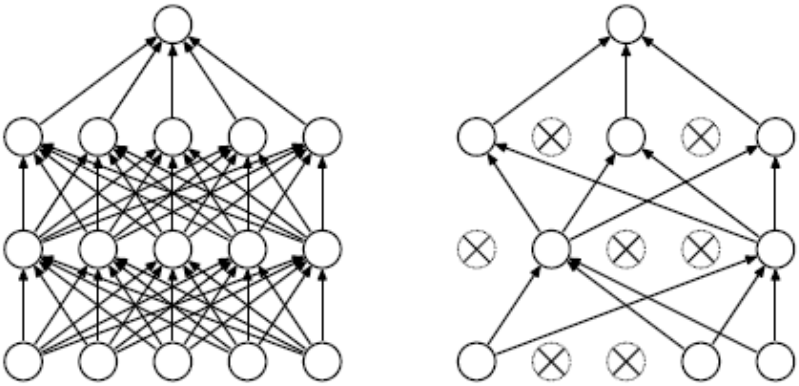


Figura 3.9: Modelo de rede neuronal. Esquerda: NN com duas camadas escondidas. Direita: Aplicação de *dropout* à NN anterior com exclusão das unidades assinaladas com *X*.
Fonte:[Srivastava et al., 2014].

3.9 *Batch normalization*

A *batch normalization* (normalização em lote) permite usar taxas de aprendizagem muito altas e ter menos cuidado com a inicialização dos pesos da rede, atuando como um regularizador, e eliminando em alguns casos a necessidade de *dropout*. Aplicada a um modelo de classificação de imagem de última geração, a *batch normalization* alcança a mesma precisão com 14 vezes menos *epochs* de treino, e bate o modelo original por uma margem significativa.

A utilização de *mini-batches* (mini-lotes), ao contrário de um exemplo de cada vez, é útil de várias formas. Primeiro, o gradiente da perda sobre um *mini-batch* é uma estimativa do gradiente sobre o conjunto de treino, cuja qualidade melhora à medida que o tamanho do *batch* (lote) aumenta. Em segundo lugar, o cálculo sobre um *batch* pode ser muito mais eficiente do que m cálculos sobre exemplos individuais, devido ao paralelismo proporcionado pelas plataformas de computação modernas.

Ainda que o gradiente estocástico muitas vezes utilizado seja simples e eficaz, requer uma cuidadosa afinação dos hiperparâmetros do modelo, especificamente da taxa de aprendizagem utilizada na otimização, bem como os valores de inicialização dos parâmetros do modelo. O treino é complicado devido ao facto de os *inputs* de cada camada serem afetados pelos parâmetros das camadas precedentes - de modo que pequenas alterações nos parâmetros da rede amplificam-se à medida que a rede se torna mais profunda.

A mudança na distribuição dos inputs das camadas representa um problema uma vez que as camadas precisam de se adaptar continuamente à nova distribuição. Quando a distribuição dos inputs do sistema muda, diz-se que apresenta uma mudança na covariância. Isto é normalmente tratado por meio de adaptação de domínio. Contudo a noção de mudança de covariável pode ser estendida para além do sistema de aprendizagem como um todo, para aplicar às suas partes, com uma sub-rede ou uma camada.

Considerando uma rede que efetua o cálculo:

$$l = F_2(F_1(u, \theta_1), \theta_2) \tag{3.16}$$

em que F_1 e F_2 são transformações arbitrárias, e θ_1, θ_2 são os parâmetros a aprender de forma a minimizar a perda l. A aprendizagem de θ_2 pode ser vista como se os inputs $x = F_1(x, \theta_2)$

são passados para a sub-rede.

$$l = F_2(x, \theta_2). \tag{3.17}$$

Por exemplo, um passo de descida do gradiente

$$\theta_2 \leftarrow \theta_2 - \frac{\alpha}{m} \sum_{i=1}^{m} \frac{\partial F_2(x_i, \theta_2)}{\partial \theta_2} \tag{3.18}$$

para um *batch* de tamanho m e taxa de aprendizagem α é exatamente equivalente a isto para uma rede isolada F_2 com *input* x. Assim, as propriedades de distribuição de input que fazem o treino mais eficiente - como o facto de ter a mesma distribuição entre os dados de treino e teste, aplicam-se ao treino da sub-rede também. É então vantajoso que a distribuição de x se mantenha fixa com o tempo. Então, θ_2 não tem que se reajustar para compensar a mudança na distribuição de x.

A distribuição fixa de inputs a uma sub-rede tem consequências igualmente positivas para as camadas fora da sub-rede. Considerando uma camada com uma função de ativação sigmoide $z = g(Wu + b)$ em que u é a camada de input, a matriz de pesos W e o vector bias *b* são parâmetros a ser aprendidos, e $g(x) = \frac{1}{1+exp(-x)}$. Como $|x|$ aumenta, $g'(x)$ tende para zero. Isto significa que para todas as dimensões de $x = Wu + b$, exceto para as que possuem valores absolutos pequenos, o gradiente que flui para u desaparece e o modelo treina lentamente. Contudo, uma vez que x é afetado por W, b e os parâmetros de todas as camadas abaixo, a mudança destes parâmetros durante o treino irão alterar muitas dimensões de x para um regime saturado da não-linearidade e abrandar a convergência. Este efeito é amplificado à medida que a profundidade da rede aumenta. Na prática, o problema da saturação e o resultante desaparecimento do gradiente são normalmente resolvidos pelo uso de Unidades de Retificação Linear $ReLU(x) = max(x, 0)$, inicialização programada e baixas taxas de aprendizagem. Se, contudo, se assegurar que a distribuição de inputs de não-linearidades se mantém mais estável à medida que a rede treina, então o otimizador tem menos probabilidade de ficar preso em regime de saturação, e o treino sofrerá uma aceleração.

Batch normalization, dá um passo em direção à redução da covariância interna e, ao fazê-lo, acelera drasticamente o treino de redes neuronais profundas. Isto é feito através de uma etapa de normalização que fixa as médias e as variações dos *inputs* da camada. A *batch normalization* também tem um efeito benéfico no fluxo do gradiente através da rede, reduzindo a dependência de gradientes na escala dos parâmetros ou dos seus valores iniciais. Isto permite usar taxas de aprendizagem maiores sem o risco de divergência [Ioffe and Szegedy, 2015].

3.10 Data Augmentation

É um dado adquirido o facto de que quanto maior a quantidade de dados a que um algoritmo de *Machine Learning* tenha acesso, mais eficaz este, tendencialmente, será.

O *data augmentation* apresenta-se como outra forma de reduzir o *overfitting* nos modelos de ML, em que se aumenta a diversidade de dados usando informação existente apenas no conjunto de dados disponível.

Uma prática comum para aumentar um conjunto de imagens é a realização de transformações ao nível da sua cor e geometria, como a reflexão, recorte, mudança na palete

de cores e tradução da imagem. As técnicas de transformação tradicionais (Figura 3.10) consistem no uso de combinações de transformações afins para manipular os dados de treino. Para cada imagem é gerada uma imagem "duplicada"que é deslocada, ampliada/reduzida, rodada, invertida, distorcida ou sombreada com uma tonalidade. A imagem e a sua duplicada são então fornecidas à rede neural [Perez and Wang, 2017].

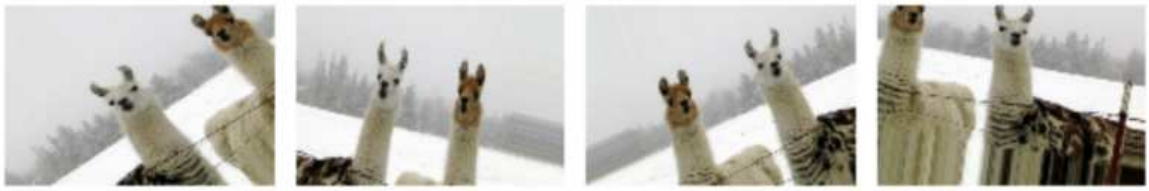


Figura 3.10: Exemplos de transformações tradicionais, realizadas numa imagem.
Fonte:[Perez and Wang, 2017].

3.11 *Fine-Tunning*

O fine-tunning é um procedimento baseado no conceito de transferência de aprendizagem (*transfer learning*)

Começa-se por treinar uma CNN para aprender características para um domínio amplo com uma função de classificação voltada para minimizar o erro nesse domínio. Em seguida, substitui-se a função de classificação e otimiza-se a rede novamente para minimizar o erro noutro domínio mais específico. Sob esta configuração, é feita a transferência de características e parâmetros da rede do domínio amplo para outro mais específico.

Assuma-se uma rede original em que a função de classificação é a *softmax*, que calcula a probabilidade em 1000 classes da base do *ImageNet*. Para começar o procedimento de *fine-tunning*, é feita a remoção deste classificador e inicializa-se outro com valores aleatórios. O novo classificador *softmax* é então treinado de raiz usando o algoritmo de *backpropagation* com informação relativa, por exemplo, à classificação de leucócitos. De forma a iniciar o *backpropagation* para o *fine-tunning* é necessário definir as taxas de aprendizagem de cada camada apropriadamente. A camada de classificação (*softmax*) necessita de uma elevada taxa de aprendizagem, devido à sua recente inicialização. O resto das camadas necessitam de taxas de aprendizagem relativamente pequenas (ou nulas) uma vez que se pretende preservar os parâmetros da rede anterior para transferir o conhecimento para a nova rede [Reyes et al., 2015].

Uma das grandes dificuldades na aplicação das CNNs a imagens de células reside na escassez de dados rotulados. Nesse sentido, foi já demonstrada a importância da reutilização de modelos treinados em tarefas diferentes. Contudo, a capacidade de transferência de recursos depende da distância entre a tarefa base e a tarefa de destino. No entanto, os recursos de tarefas distantes tendem a ter um desempenho melhor do que os recursos aleatórios. A inicialização de uma rede com recursos pré-treinados melhora então a sua capacidade de generalização [Kensert et al., 2018].

Capítulo 4

Redes Neuronais Convolucionais

As redes neuronais convolucionais (Convolutional Neural Networks, CNN) são úteis num grande número de aplicações, especialmente aquelas relacionadas com imagens, como classificação, segmentação e deteção de objetos [Wu, 2017].

Digamos que temos uma imagem de um leucócito em formato PNG[1] com tamanho 480×480. A matriz representativa será $480 \times 480 \times 3$. Cada um desses números recebe um valor de 0 a 255 que descreve a intensidade do pixel nesse ponto. A ideia é fornecer ao computador essa matriz de números. Com o output obtém-se a probabilidade de a imagem pertencer a uma categoria específica (por exemplo, 0.80 para neutrófilo, 0.15 para monócito, 0.05 para basófilo...). De forma semelhante aos humanos, o computador é capaz de realizar a classificação da imagem com base em características de baixo nível, como bordas e curvas, construindo depois conceitos mais abstratos através de uma série de camadas convolutivas. Esta é uma visão geral do que faz uma CNN [Deshpande, 2016].

4.1 Tensores

É vantajoso representar imagens (ou outros tipos de dados brutos) como tensores. Um escalar é um tensor de ordem 0, um vetor é um tensor de ordem 1, e uma matriz é um tensor de de segunda ordem. Uma imagem colorida é de facto um tensor de ordem 3. Se esta imagem é armazenada no formato RGB possui 3 canais (R, G e B, respetivamente) e cada canal é uma Matriz $H \times W$ (tensor de segunda ordem) que contém os valores R, G ou B de todos os pixeis. Nos estágios iniciais dos processos de visão por computador e reconhecimento de padrões, uma imagem a cores é muitas vezes convertida na versão em tons de cinzento (que é uma matriz) uma vez que é mais simples trabalhar com matrizes do que com tensores (de ordem 3). A informação de cores é perdida durante esta conversão. Mas a cor é muito importante em vários problemas de aprendizagem e reconhecimento baseados em imagem (ou vídeo). Neste sentido, os tensores são essenciais nas CNN. O *input*, a representação intermédia e os parâmetros de uma CNN são todos representados com tensores [Wu, 2017].

4.2 Primeira camada: convolução

A primeira camada numa CNN é sempre uma camada convolucional.

[1] *Portable Network Graphics*

Neste tipo de redes utiliza-se um filtro, que é na prática uma matriz de números (pesos ou parâmetros) com a mesma profundidade da imagem de input, por exemplo $5 \times 5 \times 3$ para uma imagem $32 \times 32 \times 3$ (RGB). O filtro vai deslizando, ou efetuando a convolução, em torno da imagem de entrada, multiplicando os valores no filtro pelos valores dos pixeis da imagem original. Obtém-se assim um único número representativo de cada posição do filtro ao longo do processo de convolução, o que origina uma nova matriz de 28×28, tendo, portanto, tamanho inferior ao da matriz original.

Se forem utilizados dois filtros $5 \times 5 \times 3$ em vez de um, o volume de saída será $28 \times 28 \times 2$. Ao usar mais filtros é possível preservar melhor as dimensões espaciais.

Cada um desses filtros pode ser considerado como identificador de características como bordas, retas, cores simples e curvas.

Imaginemos um filtro $7 \times 7 \times 3$, detetor de um tipo específico de linha curva. Consideremos apenas a fatia de profundidade superior do filtro e da imagem, para simplificar.

A Figura 4.1 exemplifica (com um quadrado amarelo) o local de aplicação do filtro na imagem.

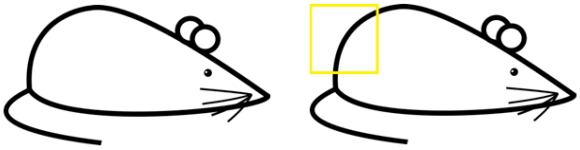
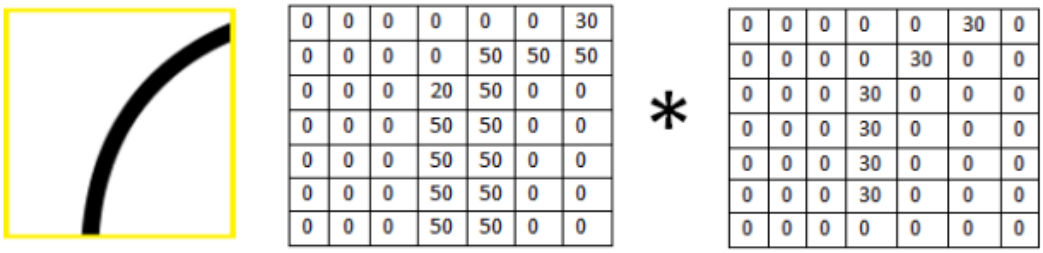


Figura 4.1: Local de aplicação de um filtro.
Fonte:[Deshpande, 2016].

É efetuada a convolução, isto é, a multiplicação dos valores no filtro pelos valores dos pixeis da imagem original, conforme apresentado na Figura 4.2.



0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

$*$

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Figura 4.2: Visualização do campo recetor; Representação matricial do campo recetor; Representação matricial do filtro.
Fonte:[Deshpande, 2016].

Basicamente, se houver na imagem de entrada uma forma que genericamente se assemelhe à curva que este filtro representa, todas as multiplicações somadas resultarão num valor elevado. Efetuando os cálculos temos $(50*30)+(50*30)+(50*30)+(20*30)+(50*30) = 6600$. Quando movemos o filtro para outro local da imagem, conforme exemplificado na Figura 4.3, o valor obtido é muito mais baixo ($(0*30)+(0*30)+(0*30)+(0*30)+(0*30) = 0$). Isto acontece porque não há nada na secção da imagem original que corresponda ao filtro. A saída da camada de convolução é um mapa de ativação. Assim, no exemplo apresentado o mapa de ativação mostrará a probabilidade de existência de curvaturas em determinados campos recetores da imagem. Quanto maior for o valor obtido, maior será a probabilidade.

Quantos mais filtros forem utilizados maior será a profundidade do mapa de ativação e consequentemente mais informações sobre o volume de entrada existirão.

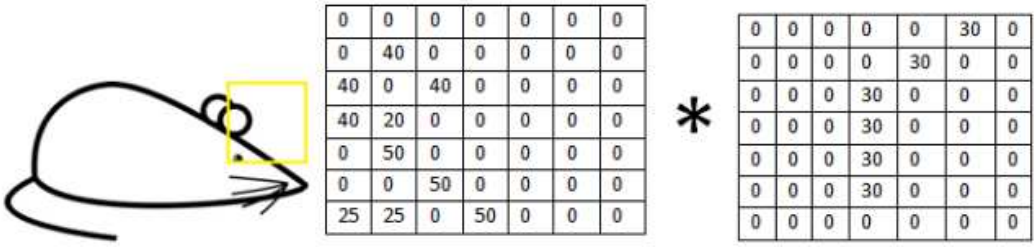


Figura 4.3: Exemplo da visualização de um campo recetor, da representação em pixeis deste campo e representação matricial de um filtro.
Fonte:[Deshpande, 2016].

Na Figura 4.4, apresentam-se exemplos de visualizações reais dos filtros da primeira camada de convolução de uma rede treinada.

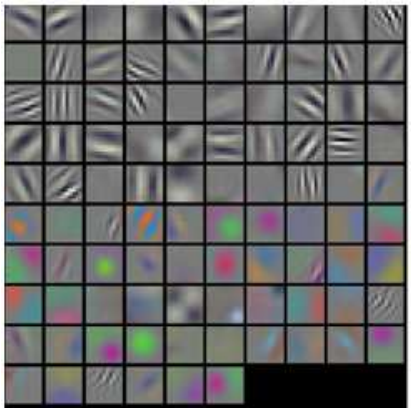


Figura 4.4: Visualização de filtros de um caso prático.

Para tensores de ordem 3, a operação de convolução é feita de forma semelhante à utilizada para tensores de ordem 2. Supondo que a entrada na l-ésima camada seja um tensor de ordem 3 com tamanho $H^l \times W^l \times D^l$, o *kernel* de convolução também será um tensor de ordem 3 com tamanho $H \times W \times D^l$. Quando se coloca o *kernel* em cima do tensor de *input* na localização espacial (0, 0, 0), calculam-se os produtos dos elementos correspondentes em todos os canais D^l e somam-se os produtos HWD^l para obter o resultado da convolução nesta localização espacial [Wu, 2017]. Embora existam outras formas de tratar tensores é esta a mais comum.

4.3 Agrupamento (*Pooling*)

O *pooling* é um processo de redução dos dados. Em geral, é calculado o máximo ou a média de uma pequena região dos dados de *input*. A vantagem principal deste *pooling* é o aumento de velocidade na rede, devido ao *downsampling* entre as camadas, reduzindo a quantidade de dados a serem processados [do Nascimento, 2016].

O objetivo das camadas de *pooling* é então alcançar a invariância espacial ao reduzir a resolução dos mapas de características. Cada mapa corresponde a um mapa de características da camada anterior. As suas unidades combinam a entrada de pequenos fragmentos $n \times n$, conforme indicado na Figura 4.5. Estas janelas de *pooling* podem ser de tamanho arbitrário

e pode haver sobreposição de janelas. A função *max pooling* é dada por

$$a_j = max_{N\times N}(a_i^{n\times n} u(n,n)) \tag{4.1}$$

que aplica uma função de janela $u(x,y)$ ao *patch* de entrada e calcula o máximo na vizinhança. O resultado é um mapa de características de menor resolução [Scherer et al., 2010].

4.4 Camadas mais profundas das CNN

AS CNN são representantes da arquitetura Hubel-Wiesel de vários estágios, que extraem características locais em alta resolução e as combinam, sucessivamente, em características mais complexas em resoluções mais baixas. A perda de informação espacial é compensada por um número crescente de mapas de recursos nas camadas mais altas. A arquitetura geral é mostrada na Figura 4.5 [Scherer et al., 2010].

Cada camada da entrada descreve locais na imagem original para onde determinadas características de baixo nível aparecem. Quando se aplica um conjunto de filtros a esta informação, a saída passa a representar recursos de nível superior. Os tipos dessas características podem ser semicírculos, ou quadrados. Ao passar por mais camadas de convolução, obtém-se mapas de ativação que representam recursos cada vez mais complexos [Deshpande, 2016].

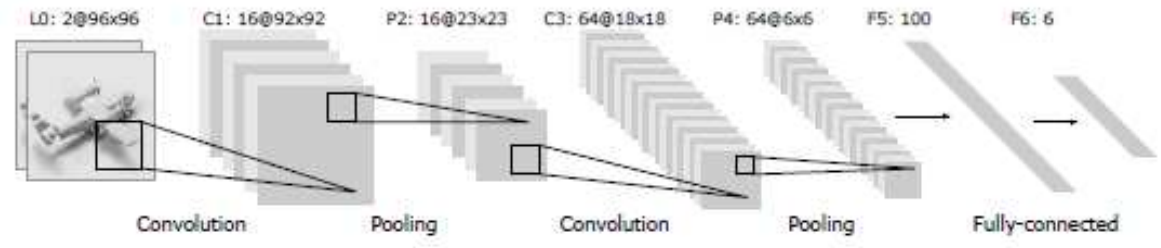


Figura 4.5: Arquitetura da CNN com alternância de camadas convolutivas e de agrupamento. As camadas de *pooling* podem implementar operações de subamostragem ou agrupamento máximo.
Fonte:[Scherer et al., 2010].

4.5 Camada Totalmente Conectada

Uma camada totalmente conectada é útil no final de um modelo CNN com várias camadas escondidas. Por exemplo, se, após várias camadas de convolução, *ReLU* e *pooling* a saída da camada atual contém representações distribuídas para a imagem de entrada, queremos usar todas essas características na camada atual para criar características com capacidades mais fortes na próxima. Uma camada totalmente conectada é útil para este propósito. Supondo que a entrada de uma camada x^l tem tamanho $H^l \times W^l \times D^l$. Se usarmos *kernels* de convolução com tamanho $H^l \times W^l \times D^l$, então os D núcleos formam um tensor de ordem 4 em $H^l \times W^l \times D^l \times D$. A saída é $y \in \Re^D$. É óbvio que para calcular qualquer elemento em y, precisamos de usar todos os elementos na entrada x^l. Portanto, esta camada é uma camada totalmente conectada, mas pode ser implementada como uma camada de convolução, não sendo necessário derivar regras de aprendizagem para uma camada totalmente conectada separadamente [Wu, 2017].

4.6 Arquiteturas de CNN

O projeto ImageNet é um banco de imagens de grandes dimensões destinado ao uso na investigação com software de reconhecimento de objetos. A ImageNet realizou (entre 2010 e 2017) uma competição anual de software, o ILSVRC onde programas competiam para detetar e classificar objetos e cenários. São apresentados em seguida alguns dos competidores de topo da ILSVRC até ao ano de 2015, bem como a LeNet-5, que serve de base a algumas das redes utilizadas atualmente.

4.6.1 LeNet-5 (1998)

A LeNet-5, uma rede convolucional pioneira de 7 camadas utilizada inicialmente para classificar dígitos, foi aplicada por vários bancos para reconhecer números escritos à mão em cheques digitalizados, em imagens em tons de cinzento de 32×32 pixels. O processamento de imagens de resolução mais alta requeriria mais camadas e convoluções, sendo necessários mais recursos de computação [Das, 2017].

A LeNet-5 é composta por 7 camadas (excluindo a camada de *input*) que contém parâmetros treináveis (pesos). A entrada aceita imagens com 32×32 pixeis, dimensão significativamente superior ao caratere maior da base de dados MNIST[2], na qual foi implementada a NN. O excedente dimensional permite que características potencialmente distintivas dos caracteres possam aparecer no centro do campo receptivo dos detetores de ordem mais elevada. Na LeNet-5 o conjunto de centros de campos recetivos da última camada de convolução (C3) formam uma área de 20×20 no centro do input de 32×32. Os valores dos pixeis de input são normalizados de forma a que o *background* corresponda ao valor -0.1 e o *foreground* corresponda a 1.175. Isto faz com que a média seja aproximadamente 0 e a variância cerca de 1, o que acelera a aprendizagem. Na Figura 4.6 as camadas convolucionais aparecem como C_x, as camadas de *pooling* aparecem como S_x e as camadas *fully-connected* aparecem como F_x, em que x é o índice da camada [Lecun et al., 1998].

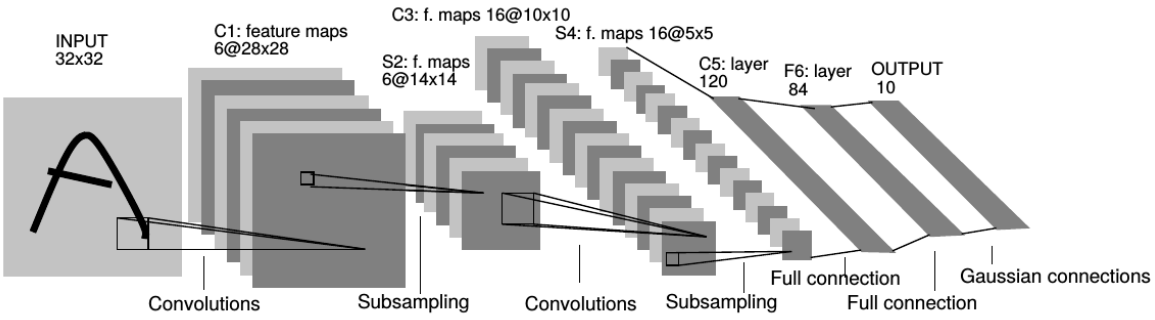


Figura 4.6: Rede neuronal LeNet.
Fonte:[Lecun et al., 1998]

4.6.2 AlexNet (2012)

Em 2012, a AlexNet superou significativamente todos os concorrentes anteriores e venceu o desafio de 2012 reduzindo o erro top-5 de 26% para 15,3%.

[2] *Modified National Institute of Standards and Technology database*

A rede tinha uma arquitetura muito similar à LeNet mas era mais profunda, com mais filtros por camada e com camadas convolucionais empilhadas. Consistia em convoluções de 11×11, 5×5, 3×3, *max pooling*, *dropout*, ativações de ReLU e SGD com momentum. Nesta rede existem ativações de ReLU após cada camada convolucional e totalmente conectada. A AlexNet foi treinada por 6 dias simultaneamente em dois GPUs Nvidia Geforce GTX 580, motivo pelo qual a rede é dividida em duas *pipelines*. A AlexNet foi projetada pelo grupo SuperVision, composto por Alex Krizhevsky, Geoffrey Hinton e Ilya Sutskever [Das, 2017].

Como mostra a Figura 4.7, a rede contém 8 camadas com pesos: as primeiras 5 camadas são convolucionais e as restantes 3 são *fully-connected*. O *output* da última camada *fully-connected* é passado à função *softmax* que produz a distribuição em 1000 classes rotuladas. Os *kernels* da segunda, quarta e quinta camadas são conectados apenas aos mapas de *kernel* da camada anterior que residem no mesmo GPU. Os *kernels* da terceira camada convolucional são conectados a todos os mapas *kernel* da segunda camada. Os neurónios nas camadas *fully-connected* são conectados a todos os neurónios da camada anterior. Camadas de normalização de resposta seguem-se às primeira e segunda camadas convolucionais. Camadas de *max-pooling* seguem camadas de normalização de resposta bem como a quinta camada de convolução.

As primeiras camadas de convolução filtram a imagem de *input* de $224 \times 224 \times 3$ com *kernels* de $11 \times 11 \times 3$ com passo (*stride*) de 4 pixeis. A segunda camada de convolução recebe o *output* da primeira camada e filtra-a com 256 *kernels* de $5 \times 5 \times 48$. A terceira camada de convolução tem 385 *kernels* de $3 \times 3 \times 192$, e a quinta camada de convolução tem 256 *kernels* de $3 \times 3 \times 192$. As camadas *fully-connected* têm 4096 neurónios cada [Krizhevsky et al., 2012].

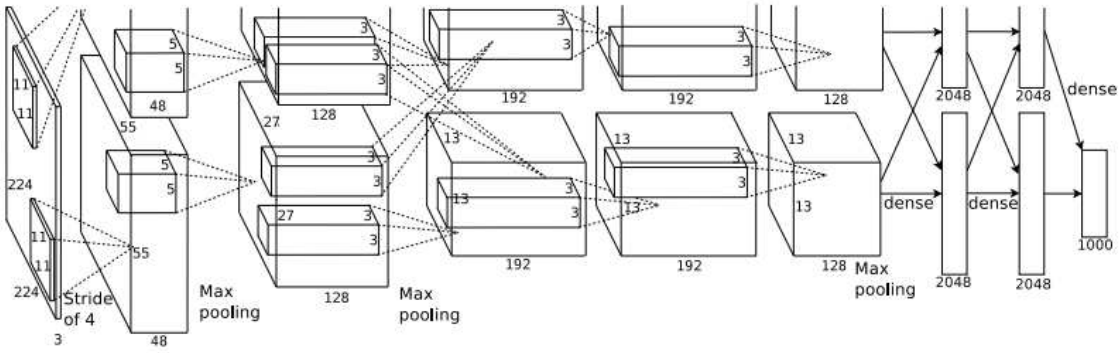


Figura 4.7: Esquema da arquitetura da rede neuronal AlexNet (2012).
Fonte:[Krizhevsky et al., 2012]

4.6.3 ZFNet(2013)

O vencedor do ILSVRC 2013 também foi uma CNN, que ficou conhecida como ZFNet. Atingiu uma taxa de erro de top-5 de 14,8%. Foi sobretudo resultado do aprimoramento dos hiper-parâmetros da AlexNet, mantendo a mesma estrutura mas com elementos adicionais do Deep Learning [Das, 2017].

A arquitetura da ZFNet (Figura 4.8) possui 8 camadas. O input é feito através de um corte de 224×224 de uma imagem (*RGB*). Esta passa por 96 diferentes filtros na primeira camada (a vermelho na figura), cada um com 7×7, usando um passo de 2 em x e y. O

mapa de *features* resultante é então (i) passado por uma função linear de retificação, (ii) alvo de *max-poling* com regiões de 3×3, com passo de 2 e (iii) normalizado com contraste para se obterem 96 mapas diferentes de 55×55. São repetidas operações similares para as camadas 2, 3, 4 e 5. As duas últimas camadas são fully-connected, aglomerando as *features* das camadas convolucionais do topo sob a forma de vetor (6 * 6 * 256 = 9216 dimensões). A camada final é uma função *softmax*, com número de saídas igual ao número de classes. Todos os filtros e mapas de *features* possuem forma quadrangular [Zeiler and Fergus, 2014].

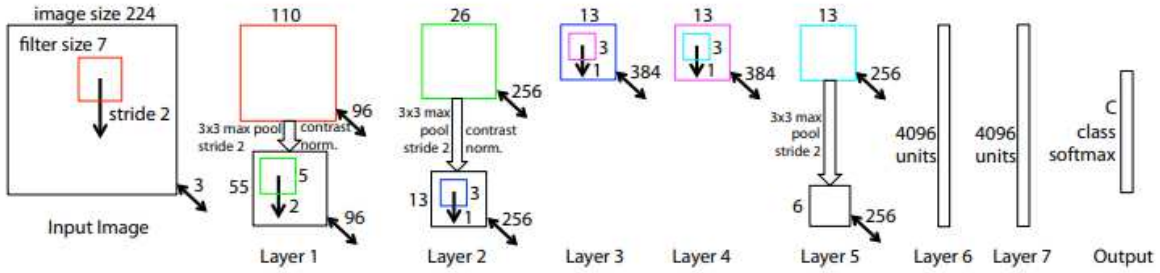


Figura 4.8: Esquema da arquitetura da rede neuronal ZFNet (2013).
Fonte:[Zeiler and Fergus, 2014]

4.6.4 GoogLeNet/Inception(2014)

O vencedor do concurso ILSVRC 2014 foi a GoogLeNet (Inception V1) da Google. Alcançou uma taxa de erro de top-5 de 6,67%, muito próximo do nível de desempenho humano, também avaliado pelos organizadores. Depois de alguns dias de treino, o especialista humano (Andrej Karpathy) conseguiu atingir uma taxa de erro de top-5 , de 5,1% (em modelo único) e 3,6% (conjunto). A rede usou uma CNN inspirada na LeNet, mas implementou um novo elemento denominado de módulo *Inception* (representado na Figura 4.9). Usou *batch normalization*, distorções de imagem e otimizador *RMSprop*. A sua arquitetura consistia numa CNN profunda de 22 camadas, mas reduziu o número de parâmetros de 60 milhões (da AlexNet) para 4 milhões [Das, 2017].

A *GoogLeNet* é uma rede com 22 camadas (não contando as camadas com parâmetros) ou 27 se contarmos o *pooling*. O número total de camadas usado na construção da rede é de cerca de 100. Contudo este número depende do sistema utilizado. O uso de *average pooling* permite a adaptação e *fine-tuning* para outros rótulos facilmente. Os autores chegaram à conclusão de que mudar de camadas *fully-connected* para camadas de *average pooling* melhorava os resultados top-1 em cerca de 0.61%, sendo o uso de *dropout* essencial. Dada a grande dimensão da rede, a capacidade para propagar o gradiente através de todas as camadas foi uma preocupação. Adicionando classificadores ligados a camadas intermédias possibilitaram a discriminação em estágios mais baixos do classificador, aumentando o sinal de gradiente propagado de volta, providenciando regularização adicional. Estes classificadores apresentam a forma de pequenas redes convolucionais colocadas no topo dos módulos de *Inception* (caminhos de *pooling* adicionais). Durante o treino, a sua perda é adicionada à perda total da rede com um desconto de peso. Aquando da inferência estas redes auxiliares são descartadas[Szegedy et al., 2015].

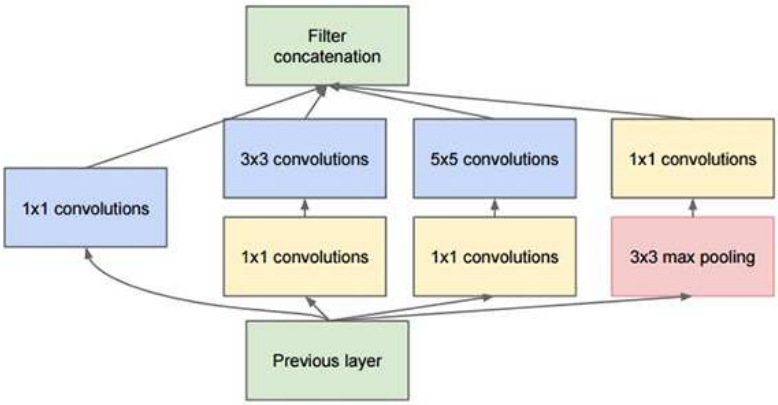


Figura 4.9: Módulo de *Inception* original usado na *GoogLeNet*
Fonte:[Szegedy et al., 2015]

4.6.5 VGGNet (2014)

O segundo classificado na competição ILSVRC 2014 foi a *VGGNet*, desenvolvida por Simonyan e Zisserman. A VGGNet consiste em 16 camadas convolucionais e apresenta uma arquitetura muito uniforme. Semelhante à *AlexNet*, com apenas convoluções 3×3, mas muitos filtros. Foi treinada em 4 GPUs durante 2 a 3 semanas. Mantém-se atualmente como uma das escolhas preferenciais para extrair recursos das imagens. A configuração de peso do *VGGNet* está disponível publicamente e tem sido usada em muitas aplicações e desafios como extrator de recursos de base [Das, 2017].

O input para as *ConvNets* VGG[3] (Figura 4.10) são imagens RGB com as dimensões 224×224. O único pré-processamento realizado aquando da sua implementação no ILSVRC foi a subtração da média do valor RGB, calculado no *dataset* de treino, em cada pixel. A imagem é então passada através de um conjunto de camadas convolucionais, onde são usados filtros com um campo recetor muito pequeno: 3×3 (que é o mais pequeno que permite captar as noções esquerda/direita, cima/baixo, e centro). Numa das configurações foram também utilizados filtros de 1×1, que podem ser vistos como transformações lineares dos canais de entrada (seguidos de não-linearidades). O passo de convolução é 1 pixel; o preenchimento espacial da camada de convolução é tal que permite que a resolução seja preservada após a convolução, por exemplo, o preenchimento é 1 pixel para camadas de convolução com 3×3. O agrupamento espacial é conseguido com 5 camadas de *max-pooling*, que seguem algumas das camadas de convolução. O *max-pooling* é feito com janelas de 2×2, e passo de 2.

O conjunto de camadas de convolução (que têm diferentes profundidades dependendo da arquitetura) é seguido por três camadas *fully-connected*: as duas primeiras contêm 4096 canais cada, a terceira faz a classificação das 1000 classes (com um canal para cada classe). A camada final é uma camada *softmax*. A configuração das camadas *fully-connected* foi a mesma em todas as redes testadas.

Todas as camadas escondidas estavam equipadas com retificação não-linear (ReLU) [Simonyan and Zisserman, 2014].

[3] *Visual Geometry Group*

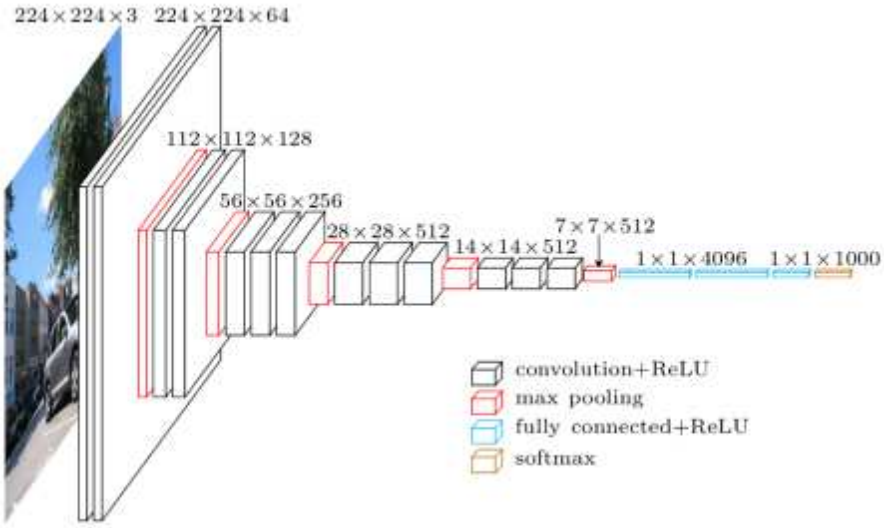


Figura 4.10: Visualização da arquitetura da VGG
Fonte:[Rosenbrock, 2017]

4.6.6 ResNet (2015)

No ILSVRC 2015, a Rede Neural Residual (*ResNet*), de Kaiming He et al., apresentou uma nova arquitetura com ligações não consecutivas (*skipped connections*) e o forte uso de *batch normalization*. As conexões ignoradas são também conhecidas como unidades bloqueadas ou unidades recorrentes bloqueadas (*gated recurrent units*) e têm uma forte semelhança com elementos de sucesso recentemente aplicados em RNNs. Graças a esta técnica, é possível treinar uma NN com 152 camadas, ainda que com complexidade inferior a uma *VGGNet*. A *ResNet* atinge uma taxa de erro de Top-5 de 3,57%, superando o nível de desempenho humano no *ImageNet* [He et al., 2016].

Os autores da *ResNet* inspiraram-se nas redes *VGG*. As camadas de convolução usadas possuem sobretudo filtros de 3×3 e seguem duas regras simples: (i) para o mesmo tamanho de mapa de *features*, as camadas têm o mesmo número de filtros e (ii) se o mapa de filtros vir o seu tamanho reduzido a metade, o número de filtros dobra de forma a preservar a complexidade temporal por camada. É efetuado o *downsampling* diretamente pelas camadas convolucionais com passo de 2 pixeis e a rede termina com um *pooling* global de média ligado a uma camada *fully-connected* com *softmax*. Este modelo possui menos filtros e complexidade inferior às redes VGG. Uma rede com 34 camadas equivale a 3.6 mil milhões de *FLOPs*[4], o que representa 18% da *VGG-19* (19.6 mil milhões de *FLOPs*).

Com base na rede de base anterior, os autores inseriram ligações de atalho transformando esta rede na sua versão contraparte residual. Os atalhos identidade (visíveis na Figura 4.11) podem ser usados diretamente quando o *input* e *output* são das mesmas dimensões. Quando as dimensões aumentam são consideradas duas opções: (A) A ligação de atalho realiza o mapeamento identidade, com entradas de preenchimento zero para aumentar as dimensões. Esta opção não induz qualquer parâmetro extra; (B) A ligação de atalho de projeção é usada para fazer corresponder as dimensões (com convoluções 1×1). Para ambas as opções, quando as ligações de atalho atravessam mapas de *features* de dois tamanhos, são realizadas com passo de 2 [He et al., 2016].

[4] *FLoating-point Operations Per Second*

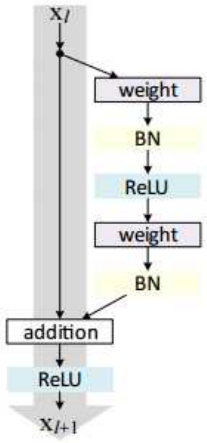


Figura 4.11: Módulo residual da ResNet conforme originalmente proposto pelos autores da rede.
Fonte:[Rosenbrock, 2017]

Na competição ILSVRC do ano de 2016 o erro de classificação situou-se 16.7% abaixo do erro de 2015. Esta classificação viria a melhorar em 23.3% no ano seguinte.

Na Figura 4.12 é possível observar a evolução dos resultados obtidos pelos vencedores do ILSVRC entre os anos de 2010 e 2015, com a passagem de uma percentagem de erro de 28,2% no primeiro ano para 3.57% em 2015 com a ResNet.

Em julho de 2017 foi anunciada a passagem da competição para o Kaggle, que é a maior comunidade mundial de *data science*. O Kaggle começou por oferecer competições de *machine learning* e fornece neste momento uma plataforma pública de dados e recursos educativos relativos à inteligência artificial. O *Kaggle* foi adquirido pela *Google* em março de 2017.

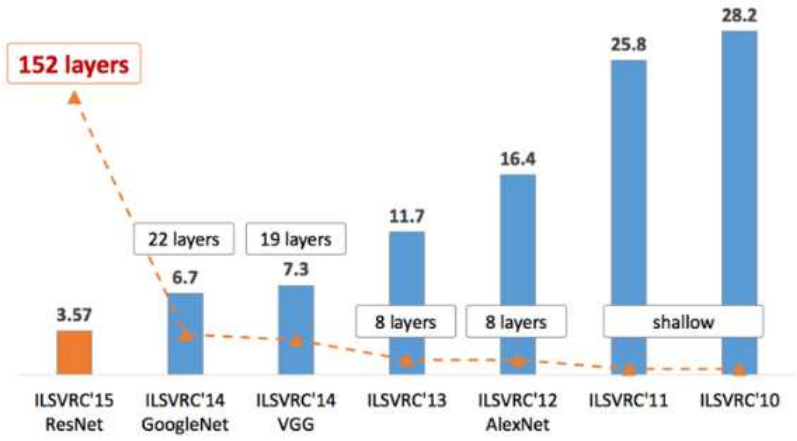


Figura 4.12: Representação gráfica da evolução dos erros de classificação obtidos pelos vencedores (e segundo classificado de 2014) do ILSVRC entre 2010 e 2015.
Fonte: [Das, 2017]

Capítulo 5

Implementação

5.1 Frameworks, linguagens informáticas e equipamento utilizado

Para a construção e implementação do projeto de classificação de leucócitos foram utilizados o *Jupyter Notebook*, com a linguagem *Python* e a API[1] *Keras* (utilizando o *TensorFlow* como *backend*).

O Jupyter Notebook é uma aplicação da *Web* de código aberto que permite criar e compartilhar documentos que contém código ativo, equações, visualizações e texto. Esta aplicação permite a execução de limpeza e transformação de dados, simulação numérica, modelação estatística, visualização de dados, aprendizagem de máquina, e muitas outras tarefas [Jupyter, 2018].

Python é uma linguagem de programação interpretada, orientada a objetos e de alto nível, com semântica dinâmica. As suas estruturas de dados incorporadas de alto nível, combinadas com digitação dinâmica e vinculação dinâmica tornam-a muito atraente para o desenvolvimento rápido de aplicações, bem como para uso como linguagem de *script* ou colagem para ligar componentes. A sintaxe simples e fácil de aprender do *Python* valoriza a legibilidade e, consequentemente, reduz o custo de manutenção do programa. O *Python* suporta módulos e pacotes, o que incentiva a modularidade dos programas e a reutilização de código. O interpretador *Python* e a extensa biblioteca padrão estão disponíveis em formato fonte ou binário, sem custo, para todas as principais plataformas, e podem ser distribuídos gratuitamente [Python, 2018].

O *TensorFlow* é uma biblioteca de *software* de código aberto para computação numérica que usa grafos de fluxo de dados. Os nodos no grafo representam operações matemáticas, e as arestas representam as matrizes ou tensores de dados multidimensionais que se comunicam com os nodos. A arquitetura flexível permite que a integração de aplicações de computação a um ou mais CPUs[2] ou GPUs num computador, servidor ou dispositivo móvel usando uma única API. O *TensorFlow* foi desenvolvido por investigadores e engenheiros da *Google Brain Team* no departamento de pesquisas de inteligência artificial da *Google* com a finalidade de realizar investigação nas áreas do *deep learning* e *machine learning*. No entanto, devido à característica abrangente do sistema, pode também ser aplicado a vários outros domínios [TensorFlow, 2018].

[1] *Application Programming Interface*
[2] *Central Processing Unit*

O *Keras* é uma API de alto nível de redes neuronais, escrita em *Python* passível de ser executada em cima do *TensorFlow*, *CNTK*[3] ou *Theano*. O *Keras* foi desenvolvido com o objetivo de permitir a experimentação rápida, isto é, a passagem de uma ideia ao resultado com o menor atraso possível. O Keras permite a criação de protótipos de forma fácil e rápida (devido à facilidade de uso, modularidade e extensibilidade), Suporta redes convolucionais e redes recorrentes, bem como combinações das duas, e funciona em CPU e GPU [Keras, 2018].

A máquina utilizada para codificar e implementar as NN possui as seguintes características: processador Intel(R) Core(TM) i3-2310M CPU @ 2.10GHz com 8.00 GB RAM. A partir desta máquina era feito o acesso remoto (utilizando o software *TeamViewer*, versão 9.x e numa segunda fase o *X2go*) a outra máquina com as características: Intel Core i7 5930K @ 3.7GHz com 32 GB RAM e 2× GTX 1080 Ti 11 GB.

5.2 Pré-processamento de dados

O conjunto de dados utilizado inicialmente no desenvolvimento do programa contém 598 imagens de leucócitos (corados). A maioria das imagens (aproximadamente 352) é proveniente de um repositório no *GitHub*[1]. As restantes imagens foram obtidas através de pesquisa realizada no *Google* de acordo com terminologia específica, nomeadamente '*monocyte*', '*leucocyte*', '*limphocyte*' e '*neutrophile*'. Os basófilos foram excluídos dada a escassa representatividade no sangue, que se refletiu na dificuldade em encontrar imagens destes em quantidades similares às conseguidas para os restantes leucócitos. Por motivos idênticos também os leucócitos de banda foram excluídos. As dimensões das imagens do *dataset* variam de acordo com a sua proveniência.

Uma grande parte do foco na realização deste trabalho incidiu no pré-processamento do conjunto de dados. Foi necessário rotular cada uma das imagens recolhidas de acordo com as 4 categorias a classificar. Posteriormente, procedeu-se à segmentação manual de cada imagem no sentido de fornecer à NN a utilizar, um registo, o mais específico possível, do enquadramento da célula cuja classificação era pretendida. Para a segmentação das imagens foi utilizado o software *GIMP2* [4]. As Figuras 5.1 e 5.2 exemplificam o processo manual de segmentação realizado. Partindo deste conjunto inicial de imagens, foi efetuado o *Data Augmentation*, tendo-se obtido um total de 10466 imagens distribuídas da seguinte forma:

- Eosinófilos: 2610 imagens de treino, 215 de validação e 215 de teste;
- Linfócitos: 2628 imagens de treino, 213 de validação e 213 de teste;
- Monócitos: 2604 imagens de treino, 211 de validação e 211 de teste;
- Neutrófilos: 2624 imagens de treino, 218 de validação e 218 de teste;

[3] *Computer Network ToolKit*

[1] disponível em `https://github.com/dhruvp/wbc-classification/tree/master/Original_Images`

[4] GIMP (acrónimo de GNU Image Manipulation Program) é um popular editor de imagens, com diversas ferramentas que facilitam tarefas de edição gráfica. Disponível em: https://www.gimp.org/

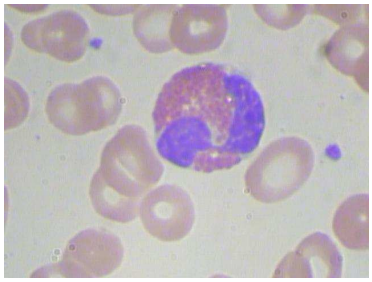


Figura 5.1: Imagem original de um eosinófilo (640 × 480).

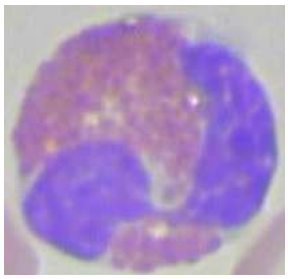


Figura 5.2: Imagem do eosinófilo segmentada (209 × 201).

O *Data Augmentation* consistiu, essencialmente, na realização de transformações morfológicas como:

- Rotações aleatórias até um limite de 20°;
- Preenchimento de pontos fora dos limites do input, de acordo com o modo selecionado (*constant*, *nearest*, *reflect* e *wrap*). A seleção do modo *nearest* no Keras traduz-se na extensão dos limites da morfologia (cor e forma) da imagem original até aos limites da nova imagem, isto é, a manutenção da cor e forma dos objetos limítrofes;
- Variações na área de input captada nos eixos longitudinal e transversal em até 10%, o que significa que poderá ter sido realizada uma movimentação da imagem de acordo com estes eixos;
- Inversões da imagem nos eixos horizontal e vertical;
- Alterações na cor.

A aplicação destas transformações permitiu a passagem das 598 imagens iniciais para as 10466 utilizadas no treino, validação e teste das redes neuronais.

Figura 5.3: Imagem de neutrófilo segmentada.

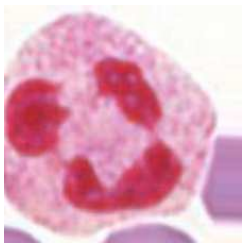


Figura 5.4: Imagem de neutrófilo originada por *Data Augmentation*

Em virtude dos resultados obtidos inicialmente após o treino das diversas redes (apresentadas na secção 5.3) não terem sido satisfatórios, foi ser necessário refazer a classificação e segmentação dos leucócitos. Neste sentido, as imagens foram separadas e categorizadas novamente. Todas as imagens que suscitavam dúvidas foram eliminadas tendo sido adicionadas outras, cuja classificação permitia um mais elevado nível de confiança. Foi feita uma nova segmentação utilizando o software *Paint 3D* [5] que possibilitou a separação das células (*foreground*) do resto da imagem (*background*) como é visível nas Figuras 5.5 e 5.6.

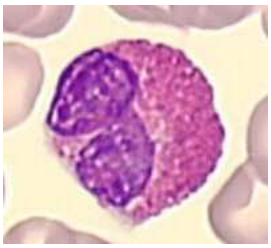


Figura 5.5: Imagem de eosinófilo segmentada de forma a obter a área pertinente da imagem original.

Figura 5.6: Imagem de eosinófilo segmentada com o Paint3D, com eliminação do *background*.

Deste segundo procedimento de seleção e segmentação obtiveram-se 100 imagens de treino, 10 de validação e 10 de teste, para cada uma das 4 categorias. Após novo processo de *Data Augmentation* obtiveram-se 4394 imagens de treino, 440 imagens de validação e 440 imagens de teste, distribuídas uniformemente pelas 4 categorias.

5.3 Modelos de NN utilizados

O *Keras* disponibiliza sob a topologia *Applications* alguns modelos de *Deep Learning*, bem como alguns pesos (pré-treinados na base de dados Imagenet) capazes de classificar até 1000 categorias. Na lista de modelos disponíveis encontram-se: Xception, VGG16, VGG19, ResNet50, InceptionV3, InceptionResNetV2, MobileNet, DenseNet, NASNet e MobileNetV2. Nas subsecções seguintes é feita uma breve descrição dos modelos utilizados no projeto descrito no presente documento, apresentados por ordem cronológica (do mais antigo para o mais recente) e estabelecendo a ligação aos modelos descritos na secção 4.6 do capítulo 4.

5.3.1 VGG16 e VGG19

A rede VGG introduzida por Simonyan e Zisserman no seu artigo de 2014, *Very Deep Convolutional Networks for Large Scale Image Recognition*, é caracterizada pela sua simplicidade, usando apenas convoluções 3×3 sobrepostas em profundidade crescente. A redução de volume de dados é conseguida através de *max pooling*. Duas camadas *fully-connected*, cada uma com 4096 nodos são frequentemente seguidas de um classificador *softmax* [Simonyan and Zisserman, 2014]. Os números *16* e *19* representam o número de camadas com pesos na rede. Algumas desvantagens da VGGNet são a morosidade do treino e a largura da arquitetura de pesos. Devido à sua profundidade e número de nodos *fully-connected* a VGG16 tem mais de 533MB e a VGG19 mais de 574MB, o que torna a implantação (*deploy*) destas redes complicada. Neste trabalho foram utilizadas as VGG16 e VGG19.

5.3.2 Inception V3

A micro-arquitetura *Inception* foi introduzida por Szegedy et al no artigo de 2014, e pretende atuar como um extrator de *features* de vários níveis calculando várias convoluções no mesmo

[5]Paint 3D a versão mais recente do popular editor de imagens *Paint* habitualmente disponível no sistema operativo *Windows*. Esta nova versão permite a criação de modelos tridimensionais

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	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 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figura 5.7: Tabela 1 de *Very Deep Convolutional Networks for Large Scale Image Recognition*, Simonyan and Zisserman (2014).

Fonte:[Rosenbrock, 2017]

módulo. A versão original desta rede era chamada de GoogLeNet. A arquitetura *Inception V3* incluída no *Keras* vem da publicação *Rethinking the Inception Architecture for Computer Vision* de 2015. A dimensão (ou espaço em memória requerido) da *Inception V3* é inferior à *VGG* e *ResNet*, ficando pelos 96MB [Rosenbrock, 2017].

5.3.3 Xception

A Xception foi proposta pelo criador do Keras, François Chollet e é uma extensão da arquitetura Inception que substitui os módulos *Inception* por convoluções separáveis por profundidade (Figura 5.8).

5.3.4 ResNet 50

A ResNet possui uma arquitetura que assenta em módulos de micro-arquitetura (*network-in-network architectures*). O termo micro-arquitetura refere-se ao blocos usados para construir a rede. Um conjunto de blocos de micro-arquitetura (juntamente com os blocos de convolução,

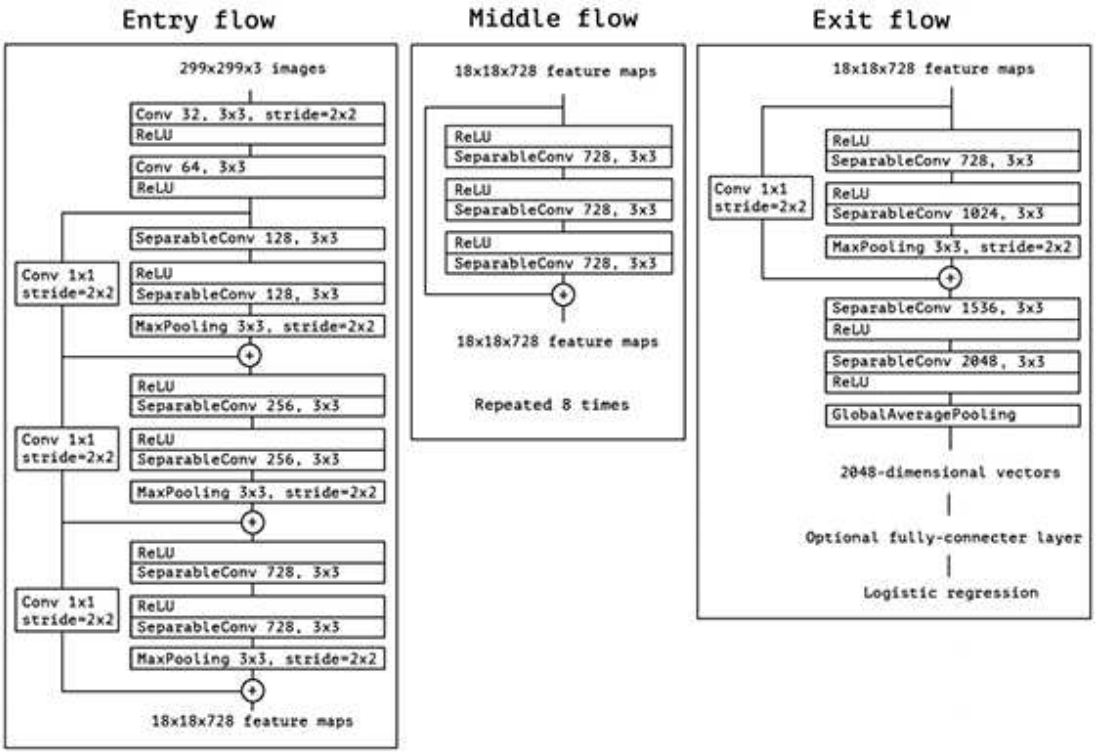


Figura 5.8: A arquitetura Xception: Os dados passam pelo fluxo de entrada, depois pelo fluxo médio que é repetido oito vezes e finalmente pelo fluxo de saída.
Fonte:[Chollet, 2017]

pooling, ...) leva a uma macro-arquitetura (a rede final em si). Esta rede demonstrou que é possível treinar redes extremamente profundas utilizando o SGD padrão através do uso de módulos residuais. A implementação da ResNet50 no Keras é baseada no artigo original de 2015. Ainda que a ResNet seja muito mais profunda do que a VGG o tamanho do modelo é substancialmente inferior (102MB) devido à maior abundância de *pooling* ao invés de camadas *fully-connected* [Rosenbrock, 2017].

5.3.5 Densenet 121, Densenet 169 e Densenet 201

Genericamente, a arquitetura da ResNet tem uma estrutura de blocos fundamental, e em cada um deles une-se uma camada anterior com uma camada futura. Em contraste, a arquitetura *DenseNet* propõe concatenar os outputs de camadas anteriores num bloco.

A arquitetura *DenseNet* diferencia explicitamente entre informações adicionadas à rede e informações preservadas. As camadas da *DenseNet* são muito estreitas (por exemplo, 12 *feature-maps* por camada), adicionando apenas um pequeno conjunto de mapas de recursos ao "conhecimento coletivo" da rede, mantendo inalterados os mapas de recursos restantes, e o classificador final toma uma decisão com base em todos os mapas de recursos da rede (Figura 5.9). Uma das grandes vantagens da *DenseNet* é o fluxo aprimorado de informações e gradientes em toda a rede, o que facilita o treino. Cada camada tem acesso direto aos gradientes da função de perda e do sinal de entrada original. Isto ajuda o treino de arquiteturas de rede mais profundas [Huang et al., 2017].

5.4 Resultados

Implementar um modelo num conjunto novo de dados é uma tarefa altamente complexa, cuja avaliação depende de inúmeros fatores inerentes ao tipo, proveniência de dados (bem como o nível de pré-processamento dos mesmos), da arquitetura de rede utilizada, de questões relacionadas com o *hardware* e *frameworks* utilizados, de entre outros... Algumas métricas não avaliadas neste trabalho são:

- Taxa de transferência: mede a eficiência do treino através do número de amostras de dados de entrada que são processadas por segundo.
- Utilização de computação de GPU: o GPU é a unidade responsável por executar as principais operações envolvidas no treino.
- Utilização de FP32: está relacionada com a capacidade do GPU realizar operações de vírgula flutuante. A utilização de FP32 fornece uma forma de calcular o limite superior teórico de melhoramentos de performance possíveis de atingir por uma melhor implementação.
- Utilização de CPU: Ainda que a maioria do treino seja normalmente realizada no GPU, o CPU está envolvido, por exemplo, na execução da *framework*, lançamento dos *kernels* da GPU e na transferência de dados entre CPU e GPU.
- Consumo de memória: além dos ciclos de computação, a quantidade de memória física disponível é um fator limitante no treino de CNNs de grandes dimensões. Para otimizar o uso de memória durante o treino é necessário entender quais as estruturas de dados que ocupam mais memória [Zhu et al., 2018].

Na classificação de leucócitos pretendida neste trabalho são utilizadas implementações de código aberto disponíveis na Internet e providenciadas pelos seus desenvolvedores e, neste caso, mediadas pela biblioteca *Keras*. A análise dos resultados do projeto centra-se sobretudo nas *accuracy* e *loss* de treino e validação de um conjunto de modelos (referidos na secção

Layers	Output Size	DenseNet-121($k = 32$)	DenseNet-169($k = 32$)	DenseNet-201($k = 32$)	DenseNet-161($k = 48$)
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	[1 × 1 conv, 3 × 3 conv] × 24	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48	[1 × 1 conv, 3 × 3 conv] × 36
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	[1 × 1 conv, 3 × 3 conv] × 16	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 24
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

Figura 5.9: Arquitetura da *DenseNet*.
Fonte:[Huang et al., 2017]

5.3) posteriormente testados na classificação de quatro categorias de leucócitos com base em imagens de microscopia. No *dataset* de teste, que possui imagens diferentes das imagens de treino e validação, afere-se a percentagem de acerto de cada um dos modelos utilizados (com e sem *transfer learning)* tendo em conta o tempo necessário para efetuar o treino.

Relativamente ao valor da *loss* (perda), quanto mais baixo for, melhor será o modelo em questão (a não ser que esteja a realizar *overfitting* ao *dataset* de treino). A *loss* é calculada durante os processos de treino e validação e a sua interpretação diz-nos o quão bom é o desempenho nestes *datasets*. Ao contrário da *accuracy* (precisão) a *loss* não é um valor percentual, sendo a soma dos erros para cada exemplo de treino (utilizando-se o termo *cost* para um conjunto de exemplos ou *batch*) nos *sets* de treino e validação. No caso das redes neuronais a *loss* é normalmente dada pelo logaritmo negativo da probabilidade de obtenção do número de categorias.

O objetivo principal da implementação do modelo é reduzir ou minimizar a função de *loss* através da alteração dos pesos da rede utilizando diferentes métodos de otimização. O valor da *loss* dá uma ideia do comportamento do modelo depois de cada iteração de otimização. Idealmente, dever-se-á esperar uma redução deste valor depois de uma ou várias iterações.

Apesar da *accuracy* poder ser verificada após cada iteração, no caso do treino este valor constitui apenas uma referência para a percentagem de acerto do modelo. O valor de precisão final, obtido após a aprendizagem de todos os parâmetros com a conclusão do treino, é aquele que deverá ser tido como correto. Após o treino são fornecidas as imagens de teste à rede e é feita a comparação das previsões da rede com os rótulos reais das imagens, calculando-se (habitualmente) a percentagem de erro.

Neste trabalho foi realizado o estudo comparativo entre alguns modelos efetuando o seu treino com recurso ao *transfer learning* e sem recurso a esta técnica, ou seja, treinando as redes de raiz na classificação de leucócitos.

5.4.1 Modelos com *Transfer Learning*

No caso do treino com *transfer learning*[6], foi excluído o topo das redes[7]. Assim foi possível utilizar os pesos do treino no ImageNet, ou seja, o conhecimento de algumas características de objetos aplicáveis à identificação de leucócitos. Foi então criada uma pequena rede com 3 camadas: uma *dense* com a ativação *ReLu*, uma camada de *Dropout* e uma camada de saída com a função *Softmax* para 4 categorias. O código necessário para programar esta rede no *Keras* é o seguinte:

```
model = models.Sequential()
model.add(layers.Dense(256, activation ='relu', input_dim =
7 * 7 * 512))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(4, activation ='softmax'))
```

A rede recebeu os pesos relativos às *features* do pré-treino e foi então treinada para classificar leucócitos em 4 categorias. O treino inicial teve a duração de 100 *epochs* tendo sido usado um *batch size* de 16 exemplos. O tempo médio de treino das 8 NN ('resnet50', 'vgg16', 'vgg19', 'inceptionV3', 'xception', 'densenet121', 'densenet169', 'densenet201')

[6]No anexo F é apresentado um exemplo do código em *Python* utilizado para treinar os modelos.

[7]função disponível no Keras, sendo possível especificar a inclusão ao não do topo da rede. O parâmetro `include_top = False` permite a utilização do modelo pré-treinado com o *dataset* do *ImageNet*.

para 100 epochs foi de 9 minutos e 59 segundos. A rede mais rápida, a VGG19, efetuou o treino em 5 minutos e 47 segundos, e a mais lenta, a Xception, demorou 16 minutos e 57 segundos. Os resultados obtidos, em termos de *loss* são apresentados no gráfico da Figura 5.10.

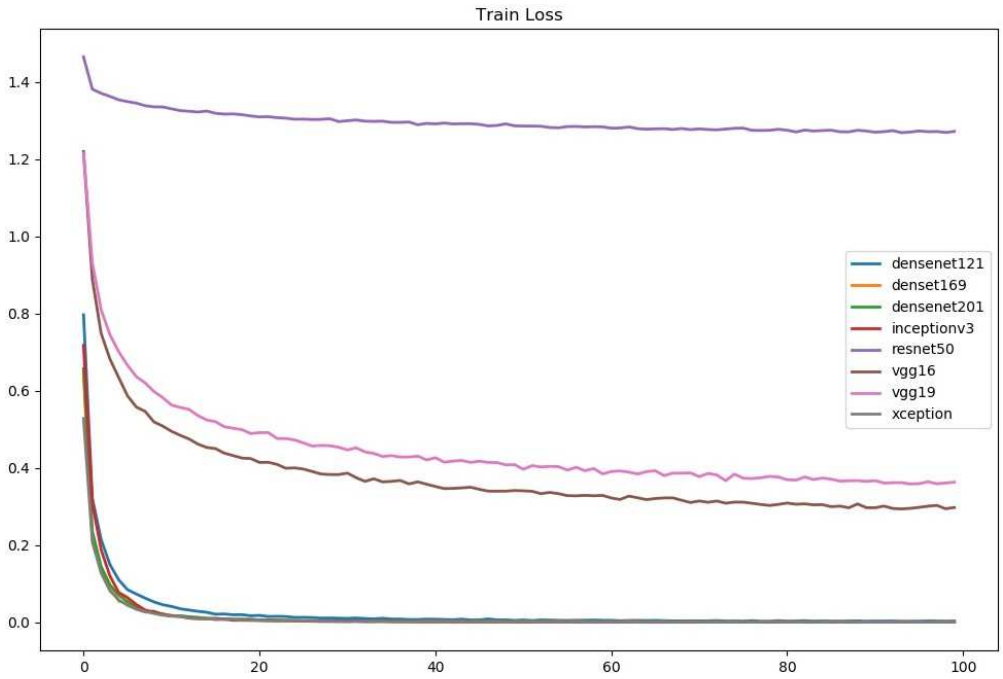


Figura 5.10: *Loss* registada no *dataset* de treino pelo conjunto de modelos treinados com *Transfer Learning* ao longo de 100 epochs.

As redes que conseguiram melhores resultados, com valores de *loss* inferiores, foram a Xception, a Densenet201, a Densenet169, a Densenet 121 e a InceptionV3. A rede que convergiu mais rapidamente para (aproximadamente) 0 foi a Xception. Os piores resultados foram obtidos pelas redes VGG e pela Resnet50.

Ainda que as redes tenham sido inicialmente apresentadas pelos seus autores para operarem com determinados otimizadores - todos utilizaram o *SGD*[8] -, para efeitos comparativos (consciente da possibilidade de penalização da performance de algumas das redes utilizadas, em função do otimizador utilizado), todos os treinos foram realizados com recurso ao optimizador *RMSprop* (com *learning rate* de $1e^{-5}$ e *decay* de 0.001).

O gráfico da Figura 5.11 mostra que a *accuracy* está intimamente relacionada com a *loss*, sendo que os modelos que obtiveram as melhores *accuracies* foram os mesmos que haviam obtido as *losses* com valores inferiores. Assim, a rede Xception foi a que convergiu mais rapidamente para 1 (com possibilidade de *overfitting*) e a ResNet50 foi a que obteve piores resultados, com precisão de treino de cerca de 40%. Os maus resultados da ResNet poderão estar relacionados com o otimizador.

Os resultados relativos ao processo de validação acompanham genericamente aqueles

[8]Definições dos otimizadores no *keras* com base nos artigos originais das redes:
DenseNet - `SGD(lr=0.01, momentum=0.9, decay=1e-4, nesterov=True)`
ResNet- `SGD(lr=0.01, momentum=0.9, decay=1e-4, nesterov=False)`
Inception V3 - `SGD(lr=0.01, momentum=0.9, decay=4e-5, nesterov=False)`
Xception - `SGD(lr=0.045, momentum=0.9, decay=1e-5, nesterov=False)`
VGG - `SGD(lr=0.01, momentum=0.9, decay=5e-4, nesterov=False)`

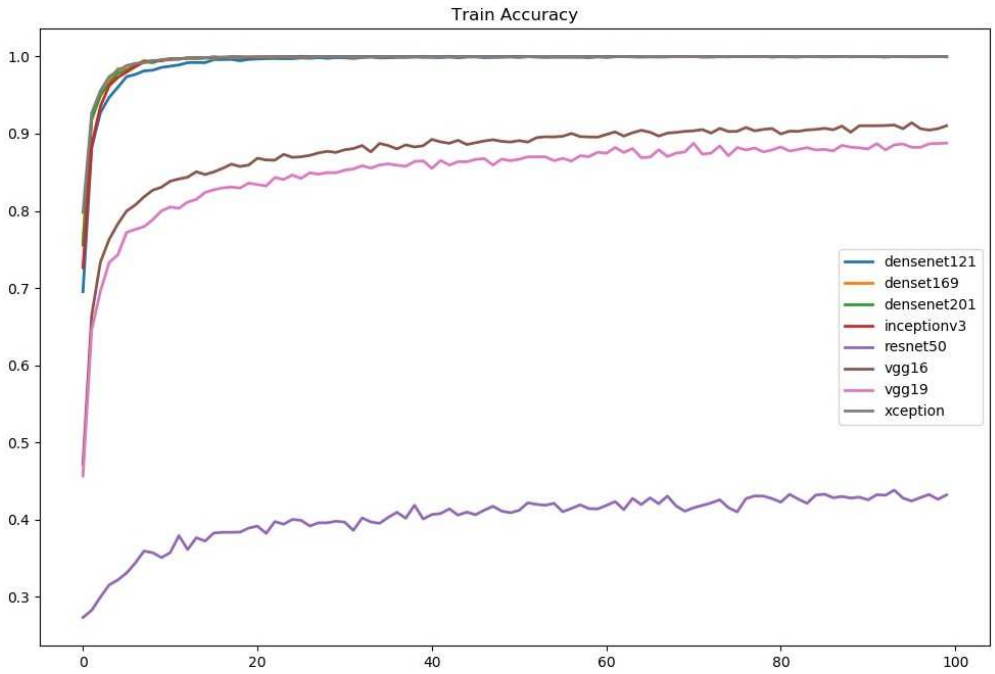


Figura 5.11: *Accuracies* registadas pelo conjunto de modelos no *dataset* de treino com *Transfer Learning* ao longo de 100 epochs.

obtidos durante o treino, sendo na globalidade as *accuracies* ligeiramente inferiores e as *losses* (apresentadas na Figura B.1 do anexo B) um pouco mais elevadas do que as de treino.

No caso do *transfer learning*, esperar-se-ia que a ResNet50 obtivesse melhores resultados, em virtude de ser uma das redes mais recentes, com erro de apenas 3.57% no ILSVRC'15. No entanto, nas condições de treino estabelecidas não se mostrou capaz de acompanhar as redes com melhores resultados [9].

A Figura C.1 do anexo C mostra como na generalidade a percentagem de acerto para cada uma das categorias tem valores similares, com exceção da ResNet50 onde é possível verificar alguma discrepância no processo de classificação.

5.4.2 Modelos treinados de raíz

Nos modelos treinados de raiz[10] é necessário treinar a totalidade da rede, sendo indispensável a especificação do número de `classes` ou categorias. O facto de ser necessário treinar a totalidade da rede faz com que o treino de raiz seja muito mais moroso do que o treino com *transfer learning*. Assim, o treino das oito redes demorou perto de 18 horas. Cada rede demorou em média 2 horas e 14 minutos para treinar durante 100 *epochs*. A rede mais rápida foi a VGG16 com 1 hora e 15 minutos e a mais lenta foi a Xception com 3 horas e 24 minutos. A Figura 5.12 mostra o comportamento dos modelos em termos de *losses*.

Os modelos que obtiveram valores de *loss* mais baixos foram a Xception e a Densenet201, ficando a ResNet50 com a *loss* mais elevada. As *accuracies* de treino apresentadas na Figura 5.13 refletem as losses previamente apresentadas, com resultados[11] similares às

[9] A secção A.1 do anexo A apresenta os gráficos referentes ao treino e validação para cada um dos modelos utilizados.

[10] No anexo G é apresentado um exemplo do código em *Python* utilizado para treinar os modelos.

[11] A secção A.2 do anexo A apresenta os gráficos referentes ao treino e validação para cada um dos modelos

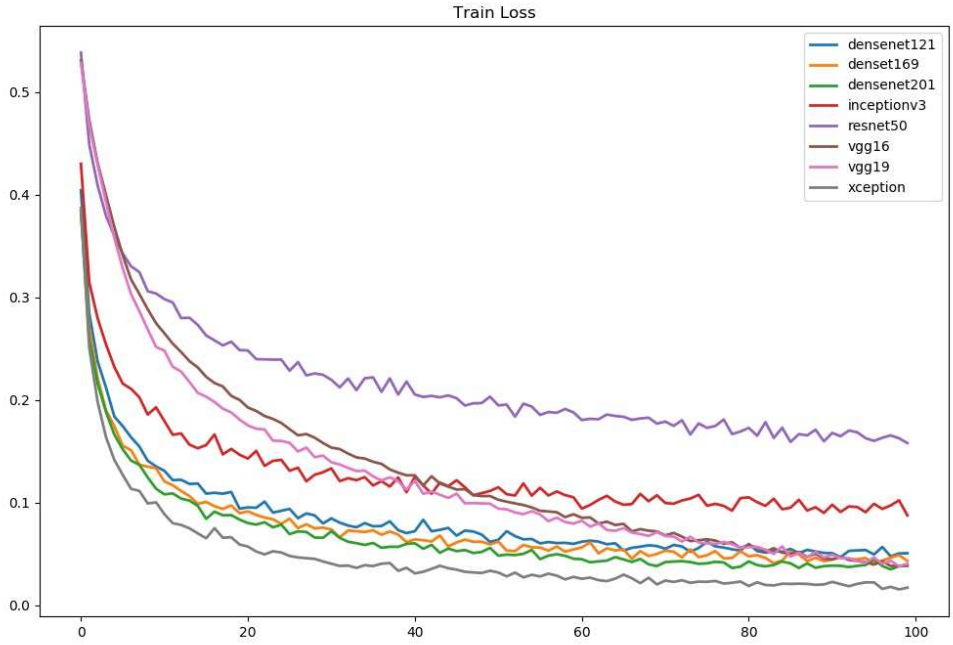


Figura 5.12: *Loss* registada pelo conjunto de modelos treinados de raiz no *dataset* de treino com 100 epochs.

losses e *accuracies* no *dataset* de validação.

A Figura C.2 do anexo C revela alguma variação nas percentagens de acerto em cada uma das categorias indicando que os modelos não estão devidamente treinados, isto atendendo ao facto de o número de exemplos de treino para cada categoria ser muito similar.

5.4.3 Modelos com *Transfer Learning vs.* Modelos treinados de raíz

Para se ter uma ideia geral do desempenho do conjunto de modelos treinados com *transfer learning* e dos modelos treinados de raiz, o gráfico da Figura 5.14 apresenta a média dos valores de *loss* obtidos pelo conjunto de modelos nos *datasets* de treino e validação. Este gráfico revela valores de *loss* mais baixos por parte dos modelos treinados de raiz relativamente aos modelos com *transfer learning*, implicando uma melhor performance, tanto no *dataset* de treino como no de validação. Os valores de *loss* para os modelos treinados de raiz situam-se, após as 100 *epochs* de treino, abaixo de 0.15, com tendência para convergir para 0 no caso do *dataset* de treino. Os modelos com *transfer learning* apresentam valores médios de *loss* de cerca de 0.4 para o *dataset* de validação e cerca de 0.3 para o *dataset* de treino.

Na Figura E.1 do anexo E são apresentadas também *accuracies* mais elevadas para os modelos treinados de raiz, com valores acima dos 95% em ambos os *datasets*. Os modelos treinados com *transfer learning* obtém resultados abaixo de 90% nos dois *datasets*.

A redução do valor da *loss* está associada a vários fatores. O facto de os resultados mostrarem valores muito baixos para esta métrica poderão ser indicativos da existência de *overfitting*, significando que o modelo memorizou os exemplos de treino, com a possibilidade de

utilizados.

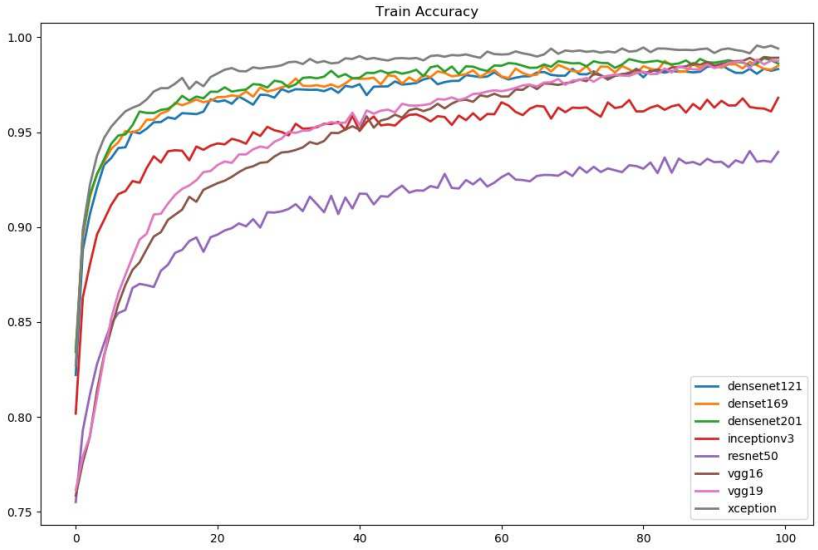


Figura 5.13: *Accuracy* registada pelo conjunto de modelos treinados de raiz no *dataset* de treino com 100 epochs.

se ter tornado ineficiente no *dataset* de teste. O *overfitting* acontece também quando não se aplica regularização, em modelos muito complexos (em que o número de parâmetros W, o conjunto de pesos, é grande) ou o número de dados N (número de exemplos) é muito baixo. No caso dos modelos treinados de raiz não foi utilizado qualquer método de regularização. Nos modelos com *transfer learning* foi utilizada uma camada de *dropout*[12]. A tabela 5.1 mostra os resultados obtidos pelos modelos treinados de raiz. Aquando do final do treino, estes modelos foram testados num *dataset* de imagens diferente das imagens utilizadas durante os processos de treino e validação. Da sua observação é possível verificar que a percentagem de acerto é muito baixa. Em média o conjunto dos modelos consegue uma precisão de 26,31% para as 4 categorias de leucócitos, o que significa que o processo tem resultados semelhantes a uma classificação aleatória.

Tabela 5.1: Tabela das percentagens de acerto na classificação de leucócitos obtidas por cada uma das redes (treinadas de raiz) no *dataset* de teste, após o treino de 100 *epochs*.

	Eosinófilos	Linfócitos	Monócitos	Neutrófilos
Densenet121	19,09	18,18	29,09	27,27
Densenet169	20,91	29,09	20,00	25,45
Densenet201	24,55	24,55	22,73	28,18
Inceptionv3	25,45	19,09	22,73	19,09
ResNet50	20,91	24,55	26,36	20,91
VGG16	26,36	23,64	21,82	27,27
VGG19	26,36	24,55	21,82	28,18
xception	25,45	26,00	30,00	30,91

Daqui se deduz a elevada probabilidade de ocorrência de *overfitting* durante o treino, uma

[12] No *Keras*: `dropout = 0.5`.

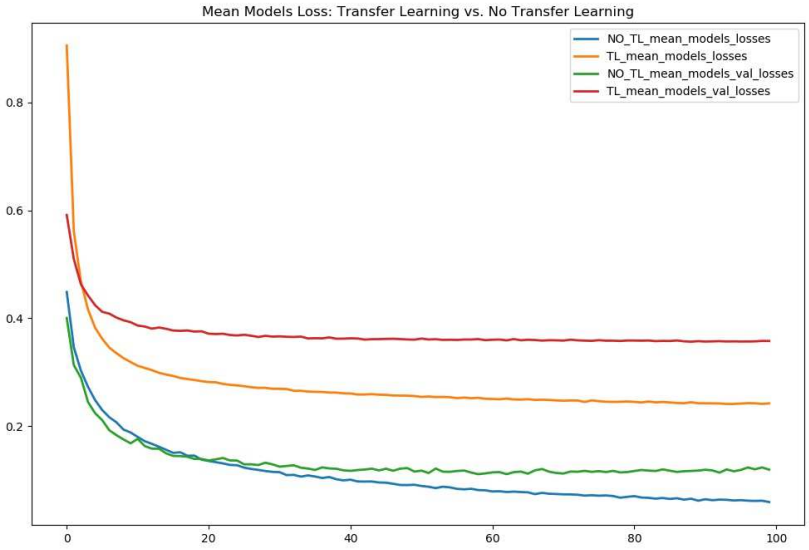


Figura 5.14: *Losses* de treino e validação médias do conjunto de modelos treinados de raíz ou com *transfer learning* nos *datasets* de treino e validação, durante 100 epochs.

vez que os resultados de treino e validação são muito bons, mas os modelos não conseguem efetuar classificações com elevado grau de precisão (ou *accuracy*) em imagens diferentes daquelas que foram utilizadas no treino.

Tabela 5.2: Tabela das percentagens de acerto na classificação de leucócitos obtidas por cada uma das redes (treinadas com *transfer learning*) no *dataset* de teste, após o treino de 100 *epochs*.

	Eosinófilos	Linfócitos	Monócitos	Neutrófilos
Densenet121	96,36	98,18	99,09	98,18
Densenet169	96,36	99,09	98,18	99,09
Densenet201	97,27	99,09	98,18	98,18
Inceptionv3	91,82	97,27	93,64	97,27
ResNet50	38,18	82,73	40,91	23,64
VGG16	78,18	86,36	86,36	90
VGG19	81,82	90	80	88,18
Xception	95,45	97,27	96,36	97,27

A tabela 5.2 apresenta os resultados obtidos pelos modelos com *transfer learning* no *dataset* de teste. Alguns modelos, como os *DenseNet* conseguiram percentagens médias de acerto (nas 4 categorias) acima de 97%. O modelo com piores resultados foi a ResNet50 com apenas 46.4% de acerto. Este valor dever-se-á à escolha do otimizador, à necessidade de regularização ou à necessidade de um número mais elevado de *epochs* de treino. As redes VGG16 e VGG19 obtiveram resultados de aproximadamente 85% para 100 *epochs* de treino. A InceptionV3 e a Xception conseguiram 95% e 96.6% respetivamente.

Da observação dos resultados obtidos é possível perceber que apesar da indicação inicial de que os modelos treinados de raiz conseguiriam melhor desempenho (com base nos melhores valores de *loss* e *accuracy* obtidos durante o treino) os modelos com *transfer learning*

conseguiram, para além de treinos muito menos demorados, efetuar classificações mais corretas no *dataset* de teste, com uma média global de acerto de 87,8%, contra os 26.3% dos modelos treinados de raiz.

5.4.4 Treino dos modelos com *Transfer Learning* durante 5000 *epochs*

Na tentativa de melhorar as performances dos modelos com *Transfer Learning* no *dataset* de teste, foi efetuado um novo treino dos mesmos modelos por 5000 epochs[13]. Atendendo a que a percentagem de acerto era já bastante elevada em alguns modelos, verificou-se que o treino veio aumentar pouco significativamente a percentagem de acerto destas redes. O treino de todos os modelos por 5000 epochs demorou 69 horas e 2 minutos, (enquanto que para as 100 *epochs* todos os modelos demoraram 1 hora e 20 minutos). A rede que demorou menos tempo a treinar foi a VGG16 com 4 horas e 56 minutos e a que demorou mais tempo a treinar foi a Xception com 14 horas e 32 minutos.

Em termos globais, o acréscimo médio da percentagem de acerto das redes entre o conjunto de modelos treinados por 100 e 5000 epochs foi de 1.68% passando de 87.81% em 100 *epochs* de treino para 89.49% em 5000 *epochs* de treino.

A Figura 5.15 mostra como as redes que já haviam obtido percentagens de acerto acima de 96% obtiveram melhorias pouco significativas, ainda que estas não devam ser negligenciadas. A DenseNet121 obteve resultados ligeiramente inferiores aos que havia obtido após o treino por 100 *epochs*.

As redes que mais beneficiaram do acréscimo do número de *epochs* de treino foram as VGG16, VGG19 e a ResNet50 com aumentos entre 3% e 5%, ainda que a ResNet50 tenha mantido percentagens de acerto muito baixas. No conjunto de figuras de A.3 (em anexo) observa-se uma tendência de aumento da *loss* de validação na generalidade dos modelos, o que sugere que os modelos estão a realizar *overfitting*, notável também pela elevada *accuracy* de treino (muito próxima de 1). Poder-se-ia utilizar um valor de *dropout* superior, aumentar o *decay* e diminuir o *learning rate* como forma de prevenir o *overfitting*. A título ilustrativo são apresentados no anexo D os resultados de precisão e abrangência bem como a *confusion matrix* para cada um dos modelos treinados por 100 e 5000 *epochs*.

A Figura C.3 do anexo C mostra, à semelhança do verificado no treino por 100 *epochs*, que na generalidade a percentagem de acerto para cada uma das categorias tem valores similares[14], com exceção da ResNet50 onde é possível verificar alguma discrepância no processo de classificação.

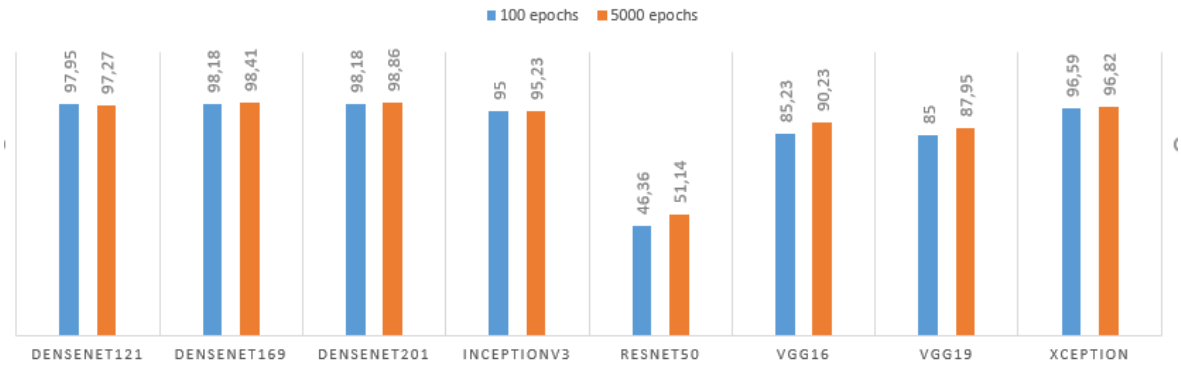
Na generalidade, tendo em conta a simplicidade da metodologia utilizada, os resultados são bastante encorajadores, mesmo quando comparados com trabalhos similares realizados recentemente. Em 2016 Sajjad et all criaram uma *framework* estrutura móvel na nuvem para segmentar e classificar leucócitos e conseguiram uma percentagem de acerto de 98,6% utilizando algoritmos *k-means* para segmentar as imagens e operações morfológicas para remover componentes indesejados. Para a classificação utilizaram uma EMC-SVM [Sajjad et al., 2017]. Em 2018 Habidzadeh et al utilizaram as redes ResNet V1 50, Resnet V1 152 e Resnet 101, treinadas por 3000 epochs, com 11200 exemplos de treino e 1244 exemplos de teste, tendo obtido resultados acima de 99% para 5 categorias de leucócitos

[13] A secção A.3 do anexo A apresenta os gráficos referentes ao treino e validação para cada um dos modelos utilizados.

[14] O anexo D mostra os valores de *precision*, *recall*, *score* e a *confusion matrix* dos modelos treinados com *transfer learning* em 100 e 5000 *epochs*

[Habibzadeh et al., 2018]. Também em 2018 Rawat et al classficaram quatro categorias de leucócitos, com uma precisão de 95% [Rawat et al., 2018].

Figura 5.15: Percentagens de acerto em imagens do *dataset* de teste obtidas com modelos de *transfer learning* treinados em 100 e 5000 epochs



Capítulo 6

Conclusões

O *machine learning* é um campo de destaque com impacto crescente nos últimos anos. Trata-se de um tipo de inteligência artificial que torna possível a determinadas aplicações a realização de previsões de várias índoles, mesmo não sendo expressamente programadas para tal.

No campo do *machine learning* aparece um subtipo, o *deep learning*, sob a forma de rede neuronal com uma ou várias camadas escondidas, capaz de realizar tarefas complexas de identificação, segmentação, classificação e outras, com precisão próxima (e em alguns casos superior) à dos humanos.

A aparente facilidade de programação deste tipo de tecnologia, vem com um custo, a necessidade de fornecimento de grandes quantidades de informação que permitem à rede aprender e reconhecer padrões que lhe facilitam a execução de determinadas tarefas.

Estamos ainda na era da *one purpose AI*, isto é, da inteligência artificial capaz de realizar tarefas específicas e relativamente isoladas. Mas a ideia de uma *general AI*, capaz de realizar tarefas complexas numa vasta gama de áreas, de forma semelhante aos humanos, já não é uma utopia.

A *one purpose AI* está já plenamente implementada e é utilizada em variados campos (recomendações online, anúncios em tempo real, resultados personalizados de pesquisa, otimização de pesquisas online, filtragem de *spam* no e-mail, reconhecimento de voz e semântica, reconhecimento de objetos e texto, entre outras...), sendo aproveitada por indivíduos, pequenas e grandes empresas.

As empresas que possuem a maior quantidade de informação (dados ou *data*) são aquelas que, nos dias que correm, têm uma vantagem inerente sobre a competição. Quantos mais dados se fornecer a uma rede, mais atualizações é possível fazer numa determinada aplicação, e mais afinada a rede está quando o software for para a produção. O *Facebook* (e o *Instagram*) podem usar todas as fotos dos milhares de milhões de usuários que atualmente possuem. A *Pinterest* pode usar informações dos 50 mil milhões de pinos que estão no seu site. A *Google* pode usar dados de pesquisa e a *Amazon* pode usar dados de milhões de produtos que são comprados todos os dias.

O objetivo primordial deste documento é a descrição do projeto de implementação de um conjunto de redes neuronais na classificação de glóbulos brancos ou leucócitos. Foram utilizadas implementações de código aberto com a *framework Keras*, sendo a maioria das implementações disponíveis referentes a redes vencedoras do ILSVRC. Os modelos do *Keras* utilizados foram o *ResNet50*, o *VGG16*, o *VGG19*, o *InceptionV3*, o *Xception*, o *Densenet121*, o *Densenet169* e o *Densenet201*.

Todos os modelos utilizados foram treinados de duas formas distintas: utilizando *transfer learning* e efetuando o treino de raiz. O *transfer learning* permite fazer a transferência dos valores dos pesos adquiridos num treino realizado previamente (no caso com 1000 categorias no *ImageNet*) sendo necessário treinar apenas o topo da rede para reconhecer características específicas do *dataset* em questão, e definir o número de categorias a classificar. O treino de raiz implica treinar a totalidade da rede, que é normalmente inicializada de forma aleatória, e é um processo significativamente mais moroso.

O conjunto de dados utilizado é normalmente pré-processado e, no caso deste trabalho, consistiu na seleção, classificação e segmentação de imagens contendo eosinófilos, linfócitos, monócitos e neutrófilos - os basófilos e leucócitos de banda foram excluídos devido à dificuldade em encontrar imagens em número equivalente às restantes categorias. Este processo de pré-processamento está também ele sujeito ao erro, humano, mas constitui o ponto de partida para a classificação automatizada realizada pelas redes neuronais. Foi feito o *data augmentation* para passar de 110 imagens em cada categoria para um total de 4394 imagens de treino, 440 de validação e 440 de teste.

O treino foi realizado por 100 *epochs* com os modelos treinados de raiz e com os modelos com *transfer learning*. Apesar dos elevados resultados de *accuracy* e baixa *loss* dos modelos treinados de raiz, verificou-se que estes estavam a realizar *overfitting* não conseguindo classificações de teste com precisão acima do aleatório para o número de categorias em questão. As redes com *transfer learning* conseguiram, no seu conjunto, precisões de classificação de 87.8% contra 26.3% das redes treinadas de raiz. As redes com melhores resultados foram as DenseNet169 e DenseNet201 ambas com 98.2% de acerto. Para além disto, o tempo médio de treino dos modelos com *transfer learning* foi de 9 minutos e 59 segundos, sendo necessárias, em média, 2 horas e 14 minutos para treinar um modelo de raiz. A solução para a melhoria dos resultados nos modelos treinados de raiz passaria pela utilização de regularização, que introduz ligeiras modificações no algoritmo de aprendizagem de forma a permitir uma melhor capacidade de generalização, o que se traduz numa melhoria do desempenho da rede. Algumas técnicas de regularização comuns são a regularização L2 e L1, o *dropout* ou *early stopping*. Atendendo a que se pretendia comparar os desempenhos com e sem *transfer learning*, não foi efetuada qualquer alteração das redes aquando do treino de raiz.

Na tentativa de melhorar o desempenho das redes foi realizado o treino dos modelos com *transfer learning* por 5000 epochs. Os resultados obtidos mostram um acréscimo de 1.68% na generalidade, na percentagem de acerto destas redes, quando testadas. Verificou-se também uma crescente tendência para o *overfitting* em virtude do escasso número de exemplos de treino utilizado para um grande acréscimo do número de *epochs* de treino. Também neste caso se aconselharia o uso de técnicas de regularização.

Os bons resultados obtidos com alguns modelos de *deep learning* estão diretamente relacionados com o trabalho de processamento realizado nas imagens. A ausência de *background* nas imagens veio permitir às redes uma mais fácil extração das *features* pertinentes para o processo de classificação. Aquando do primeiro processo de pré-processamento ou segmentação, onde apenas foi selecionada uma área aproximada contendo a célula a classificar, os resultados de classificação no *dataset* de teste foram significativamente inferiores àqueles obtidos após o "'isolamento"' dos leucócitos do *background*. Neste sentido, entende-se a importância do tratamento dos dados antes de os fornecer às redes neuronais para treino (bem como para a fase de teste).

A necessidade de uma grande capacidade de processamento informático poderá ser um

dos principais entraves ao desenvolvimento de projetos pessoais na área do *deep learning*, uma vez que num computador portátil convencional, o treino de uma rede neuronal poderá demorar dias ou semanas para um número não muito elevado de *epochs*. O acesso a GPUs poderosos acelera vertiginosamente o processo de treino, que mesmo assim poderá ser demorado para grandes quantidades de dados de treino e um elevado número de *epochs*.

Assim, como limitações para o presente trabalho assumem-se o tempo necessário para testar os modelos e, numa fase inicial, alguma indisponibilidade de recursos computacionais.

A análise dos resultados obtidos deve ser feita tendo em conta as limitações, sobretudo ao nível da quantidade de imagens utilizada, bem como da dificuldade inicial na classificação manual e individual das mesmas. A utilização de um conjunto de dados composto por mais imagens de células (com um aumento em duas ou três ordens de grandeza) seria de extrema importância como forma de corroborar com maior confiança os resultados obtidos e possibilitaria certamente o desenvolvimento com maior aplicabilidade prática (uma vez que foram utilizadas apenas 4 categorias).

Apesar de existir alguma variabilidade ao nível da morfologia das imagens, a segmentação realizada fez com que as diferentes condições de iluminação e microscopia não pesassem em demasia no resultado final. A existência de imagens provenientes de diferentes fontes constituiu uma forma de aumentar o poder de generalização das redes.

Contudo, a primeira fase de pré-processamento não funcionou e as redes mostraram-se incapazes de treinar devidamente, pelo que foi necessário rever todos os dados disponíveis e (re)segmentar cada uma das imagens. Fica em aberto a possibilidade de obter melhores desempenhos com redes mais recentes, ou com versões mais recentes das redes utilizadas.

De entre algumas propostas interessantes em termos de redes neuronais, surgidas no último par de anos, destacam-se as descritas nos seguintes artigos:

- *Identity mappings in deep Residual Networks [He et al.2016].* Propõe uma melhoria do design de blocos da ResNet. É criado um caminho mais direto para propagar informações em toda a rede (move a ativação para o caminho residual) conseguindo um melhor desempenho;

- *Aggregated Residual Transformations for Deep Neural Networks (RestNExt) [Xie et al. 2016].* Dos criadores da *ResNet*. Aumentam a largura dos blocos residuais através de múltiplas vias paralelas (”cardinalidade”) similares aos módulos de *Inception*.

- *Deep Networks with Stochastic Depth [Huang et al. 2016].* Tem o objetivo de reduzir os *vanishing gradients* e o tempo de treino através da utilização de redes curtas durante o treino. Não utilizam conjuntos de camadas, escolhidas aleatoriamente, durante uma passagem de treino. Fazem o *bypass* com a função identidade mas utilizam a rede completa na altura de testar.

- *FractalNet: Ultra-Deep Neural networks without Residuals [Larsson et al. 2017].* Defendem que a chave do bom desempenho é a capacidade de fazer a transição entre diferentes formatos de rede. Utilizam arquiteturas fractal que possuem caminhos de pouca e muita profundidades até ao output. São treinadas omitindo alguns caminhos. Utiliza-se a rede completa aquando do teste.

- *Efficient Networks... Sqeezenet (...)[Iandola et al. 2017]*: Possui módulos que consistem numa camada *'squeeze'* com filtros de 1×1 e 3×3. Os autores conseguem *accuracy* ao nível de uma *AlexNet* no *ImageNet* com $50\times$ menos parâmetros. A rede é comprimível até um tamanho $510\times$ mas pequeno que a *Alexnet* (0.5 MB).

Com este trabalho foi possível implementar um conjunto de redes na classificação de 4 categorias de leucócitos com percentagens de acerto acima de 98% no conjunto de dados disponível. É altamente provável que com um conjunto de imagens significativamente maior e com o acoplamento de um mecanismo computacional de identificação e segmentação de imagens (podendo também este ser uma rede neuronal) se conseguisse automatizar a identificação e contagem de todos os tipos de leucócitos existentes a partir de imagens de microscopia.

Ainda que a metodologia apresentada neste trabalho seja relativamente simples poder-se-á encarar como uma etapa inicial para o desenvolvimento de um dispositivo mais complexo e com maior autonomia na tarefa de classificação de leucócitos e de células sanguíneas na generalidade.

Bibliografia

[leu, 2018] (2018). Celulas sanguineas: globulos brancos. https://www.pinterest.pt/pin/369928556873360581/.

[Bayramoglu and Heikkilä, 2016] Bayramoglu, N. and Heikkilä, J. (2016). Transfer learning for cell nuclei classification in histopathology images. In *European Conference on Computer Vision*, pages 532–539. Springer.

[Bhagavathi and Thomas Niba, 2016] Bhagavathi, S. and Thomas Niba, S. (2016). An automatic system for detecting and counting rbc and wbc using fuzzy logic. *ARPN Journal of Engineering and Applied Sciences*, 11(11):6891–94.

[Braga, 2014] Braga, D. (2014). Contagem globular automatica: Parametros avaliados, significado clinico e causas de erro.

[Bush and Sejnowski, 1995] Bush, P. and Sejnowski, T. (1995). The cortical neuron.

[Cavaillon, 2011] Cavaillon, J.-M. (2011). The historical milestones in the understanding of leukocyte biology initiated by elie metchnikoff. *Journal of leukocyte biology*, 90(3):413–424.

[Chollet, 2017] Chollet, F. (2017). Xception: Deep learning with depthwise separable convolutions. *arXiv preprint*, pages 1610–02357.

[Das, 2017] Das, S. (2017). Cnn architectures: Lenet, alexnet, vgg, googlenet, resnet and more. https://medium.com/@sidereal/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5. Accessed: 2018-09-02.

[Deshpande, 2016] Deshpande, A. (2016). A beginner's guide to understanding convolutional neural networks. *Retrieved March*, 31:2017.

[do Nascimento, 2016] do Nascimento, P. P. M. (2016). *Applications of Deep Learning Techniques on NILM*. PhD thesis, Universidade Federal do Rio de Janeiro.

[Donnelly, 2017] Donnelly, P. (2017). Machine learning: the power and promise of computers that learn by example.

[Dutcher, 2016] Dutcher, T. F. (2016). Automated leukocyte differentials: A review and prospectus. *Laboratory Medicine*, 14(8):483–487.

[Falcón-Ruiz et al.,] Falcón-Ruiz, A., Taboada-Crispí, A., Orozco-Monteagudo, M., Aliosha-Pérez, M., and Sahli, H. Classification of white blood cells using morphometric features of nucleus.

[Fausett and Fausett, 1994] Fausett, L. and Fausett, L. (1994). *Fundamentals of neural networks: architectures, algorithms, and applications*. Number 006.3. Prentice-Hall,.

[Geijn et al., 2016] Geijn, G.-J. M., Gent, M., Pul-Bom, N., Beunis, M. H., Tilburg, A. J., and Njo, T. L. (2016). A new flow cytometric method for differential cell counting in ascitic fluid. *Cytometry Part B: Clinical Cytometry*, 90(6):506–511.

[Ghosh et al., 2011] Ghosh, P., Bhattacharjee, D., Nasipuri, M., and Basu, D. K. (2011). Automatic white blood cell measuring aid for medical diagnosis. In *Process Automation, Control and Computing (PACC), 2011 International Conference on*, pages 1–6. IEEE.

[Glorot et al., 2011] Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 315–323.

[Habibzadeh et al., 2018] Habibzadeh, M., Jannesari, M., Rezaei, Z., Baharvand, H., and Totonchi, M. (2018). Automatic white blood cell classification using pre-trained deep learning models: Resnet and inception. In *Tenth International Conference on Machine Vision (ICMV 2017)*, volume 10696, page 1069612. International Society for Optics and Photonics.

[He et al., 2016] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[Huang et al., 2017] Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. (2017). Densely connected convolutional networks. In *CVPR*, volume 1, page 3.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

[Jain et al., 1996] Jain, A. K., Mao, J., and Mohiuddin, K. M. (1996). Artificial neural networks: A tutorial. *Computer*, 29(3):31–44.

[Jupyter, 2018] Jupyter (2018). Jupyter, version 5.6.

[Kensert et al., 2018] Kensert, A., Harrison, P. J., and Spjuth, O. (2018). Transfer learning with deep convolutional neural network for classifying cellular morphological changes. *bioRxiv*, page 345728.

[Keras, 2018] Keras (2018). Keras documentation, keras version 2.2.2.

[Khan et al., 2012] Khan, S., Khan, A., Khattak, F. S., and Naseem, A. (2012). An accurate and cost effective approach to blood cell count. *International Journal of Computer Applications*, 50(1).

[Kingma and Ba, 2014] Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[Kloss, 2015] Kloss, A. (2015). *Object Detection Using Deep Learning-Learning where to search using visual attention*. PhD thesis, Universität Tübingen Tübingen, Germany.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

[Lawrence et al., 1997] Lawrence, S., Giles, C. L., and Tsoi, A. C. (1997). Lessons in neural network training: Overfitting may be harder than expected. In *AAAI/IAAI*, pages 540–545.

[Lecun et al., 1998] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324.

[Marr, 2017] Marr, B. (2017). A short history of machine learning every manager should read. https://www.forbes.com/sites/bernardmarr/2016/02/19/a-short-history-of-machine-learning-every-manager-should-read/#709519c215e7. Accessed: 2018-09-07.

[Moraes, 2017] Moraes, P. (2017). Leucocitos: Celulas que defendem o nosso organismo. https://mundoeducacao.bol.uol.com.br/biologia/leucocitos.htm. Accessed: 2018-09-09.

[Mukkamala and Hein, 2017] Mukkamala, M. C. and Hein, M. (2017). Variants of rmsprop and adagrad with logarithmic regret bounds. *arXiv preprint arXiv:1706.05507*.

[Nielsen, 2015] Nielsen, M. A. (2015). Neural networks and deep learning.

[Othman et al., 2017] Othman, M. Z., Mohammed, T. S., and Ali, A. B. (2017). Neural network classification of white blood cell using microscopic images. *INTERNATIONAL JOURNAL OF ADVANCED COMPUTER SCIENCE AND APPLICATIONS*, 8(5):99–104.

[Parthasarathy, 2017] Parthasarathy, D. (2017). Classifying white blood cells with deep learning. https://blog.athelas.com/classifying-white-blood-cells-with-convolutional-neural-networks-2ca6da239331. Accessed: 2017-05-15.

[Perez and Wang, 2017] Perez, L. and Wang, J. (2017). The effectiveness of data augmentation in image classification using deep learning. *CoRR*, abs/1712.04621.

[Python, 2018] Python (2018). Python software foundation. python language reference, version 3.7.0.

[Rawat et al., 2018] Rawat, J., Singh, A., Bhadauria, H., Virmani, J., and Devgun, J. S. (2018). Application of ensemble artificial neural network for the classification of white blood cells using microscopic blood images. *International Journal of Computational Systems Engineering*, 4(2-3):202–216.

[Reyes et al., 2015] Reyes, A. K., Caicedo, J. C., and Camargo, J. E. (2015). Fine-tuning deep convolutional networks for plant recognition. In *CLEF (Working Notes)*.

[Rosenbrock, 2017] Rosenbrock, A. (2017). Imagenet: Vggnet, resnet, inception, and xception with keras. https://www.pyimagesearch.com/2017/03/20/imagenet-vggnet-resnet-inception-xception-keras/. Accessed: 2018-10-11.

[Ruder, 2016] Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

[Russakovsky et al., 2015] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al. (2015). Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252.

[Saeki et al., 2014] Saeki, T., Hosokawa, M., Lim, T.-k., Harada, M., Matsunaga, T., and Tanaka, T. (2014). Digital cell counting device integrated with a single-cell array. *PloS one*, 9(2):e89011.

[Sajjad et al., 2017] Sajjad, M., Khan, S., Jan, Z., Muhammad, K., Moon, H., Kwak, J. T., Rho, S., Baik, S. W., and Mehmood, I. (2017). Leukocytes classification and segmentation in microscopic blood smear: a resource-aware healthcare service in smart cities. *IEEE Access*, 5:3475–3489.

[Scherer et al., 2010] Scherer, D., Müller, A., and Behnke, S. (2010). Evaluation of pooling operations in convolutional architectures for object recognition. *Artificial Neural Networks–ICANN 2010*, pages 92–101.

[Simonyan and Zisserman, 2014] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556.

[Srivastava et al., 2014] Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958.

[Szegedy et al., 2015] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[TensorFlow, 2018] TensorFlow (2018). Tensorflow, tensorflow version1.10.

[Verso, 1964] Verso, M. (1964). The evolution of blood-counting techniques. *Medical history*, 8(2):149.

[Wu, 2017] Wu, J. (2017). Introduction to convolutional neural networks. *National Key Lab for Novel Software Technology. Nanjing University. China*.

[Zeiler and Fergus, 2014] Zeiler, M. D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer.

[Zhang and Gupta, 2000] Zhang, Q.-j. and Gupta, K. C. (2000). *Neural networks for RF and microwave design (Book+ Neuromodeler Disk)*. Artech House, Inc.

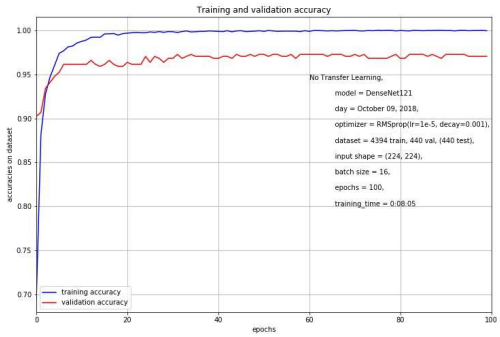
[Zhao et al., 2017] Zhao, J., Zhang, M., Zhou, Z., Chu, J., and Cao, F. (2017). Automatic detection and classification of leukocytes using convolutional neural networks. *Medical & biological engineering & computing*, 55(8):1287–1301.

[Zhu et al., 2018] Zhu, H., Akrout, M., Zheng, B., Pelegris, A., Phanishayee, A., Schroeder, B., and Pekhimenko, G. (2018). Tbd: Benchmarking and analyzing deep neural network training. *arXiv preprint arXiv:1803.06905*.

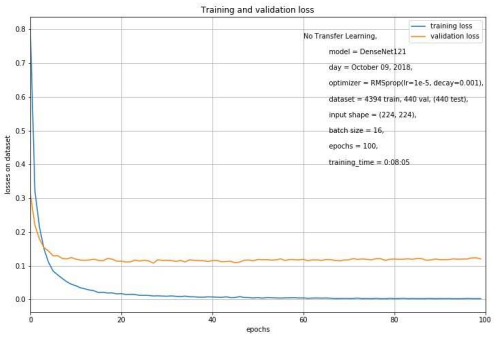
Apêndice A

Treino das redes por 100 *epochs*

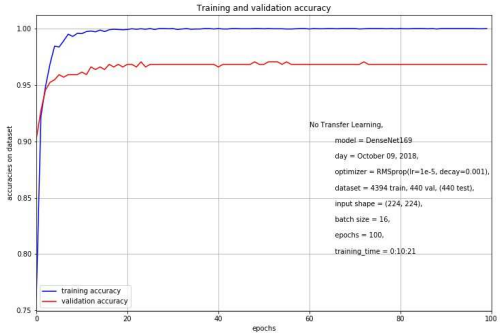
A.1 Transfer Learning (100 epochs): *accuracies*, *losses*



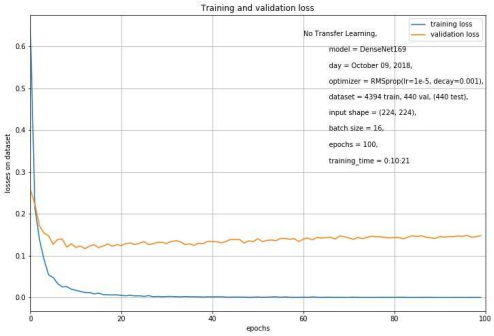
(a) DenseNet121 accuracy



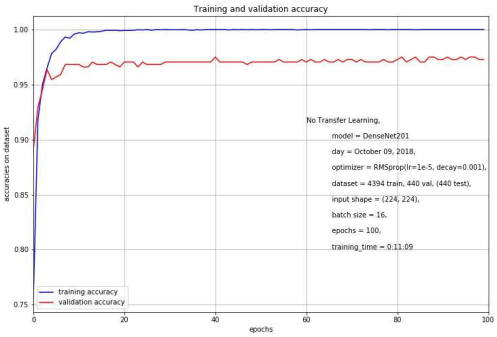
(b) DenseNet121 loss



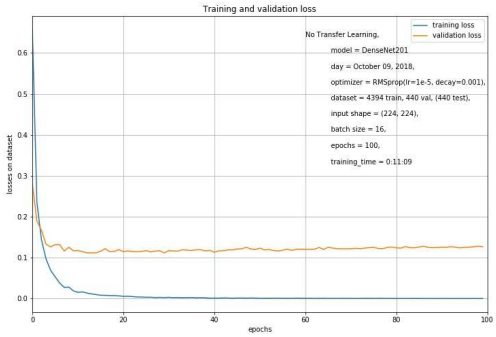
(c) DenseNet169 accuracy



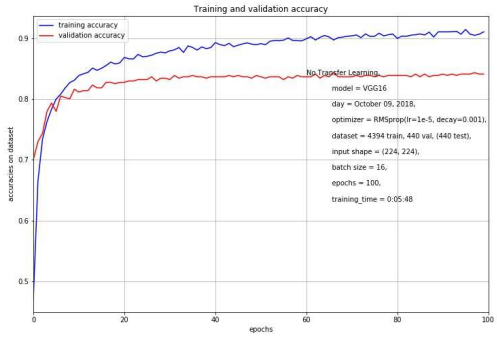
(d) DenseNet169 loss



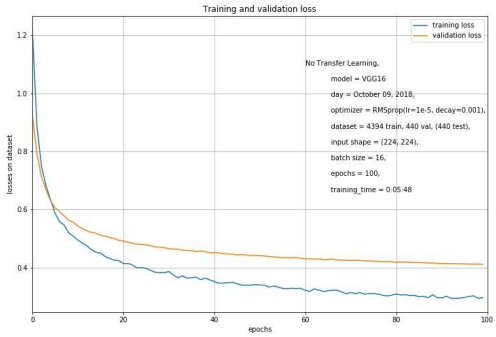
(e) DenseNet201 accuracy



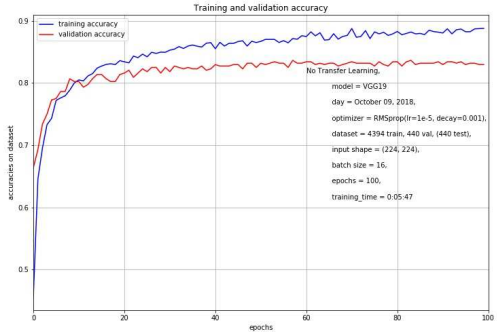
(f) DenseNet201 loss



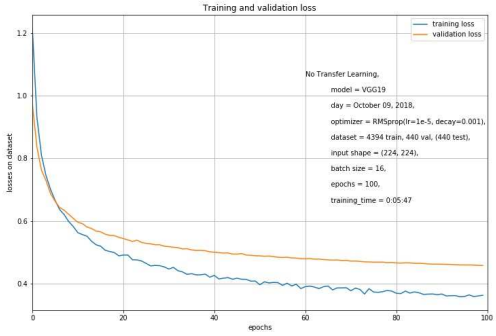
(g) VGG16 accuracy



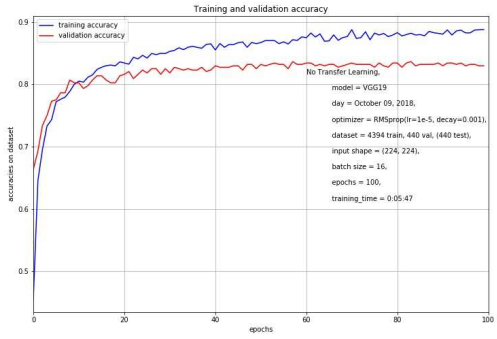
(h) VGG16 loss



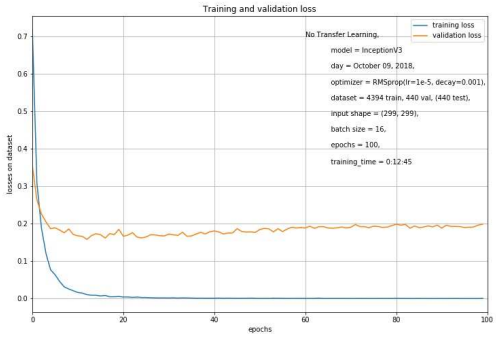
(i) VGG19 accuracy



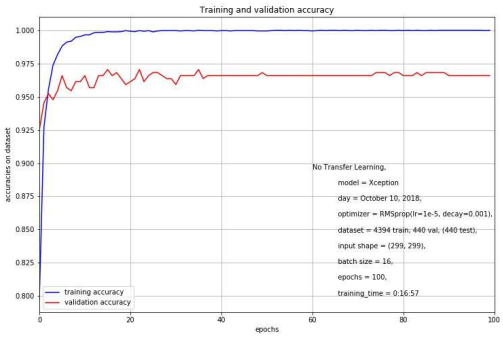
(j) VGG19 loss



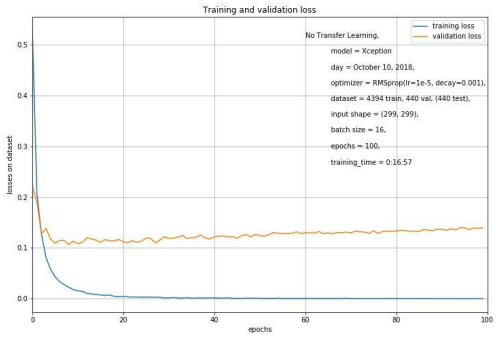
(k) InceptionV3 accuracy



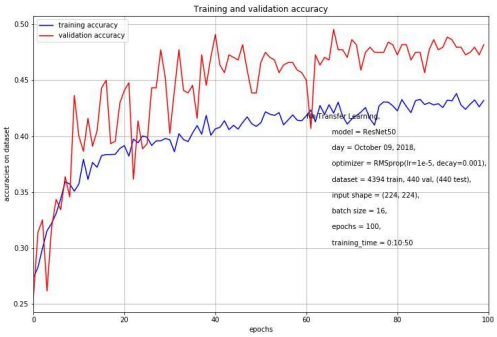
(l) InceptionV3 loss



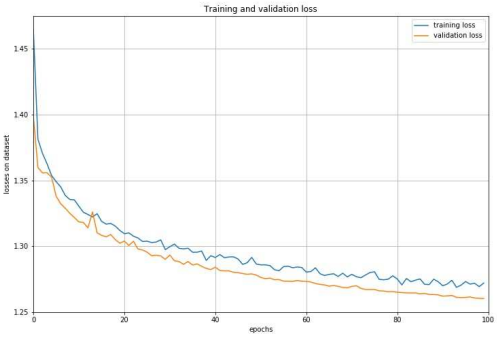
(m) Xception accuracy



(n) Xception loss



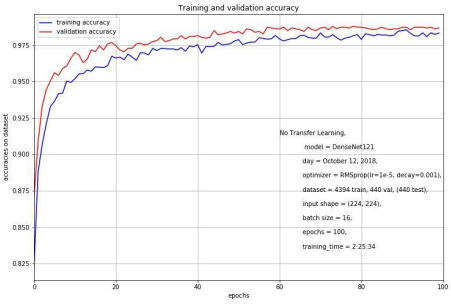
(o) ResNet50 accuracy



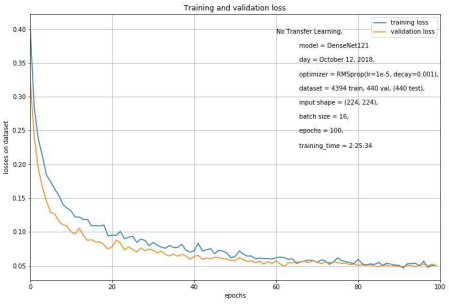
(p) ResNet50 loss

Figura A.1: *Train and validation*, *accuracies* and *losses* nos modelos treinados de raiz em 100 *epochs* .

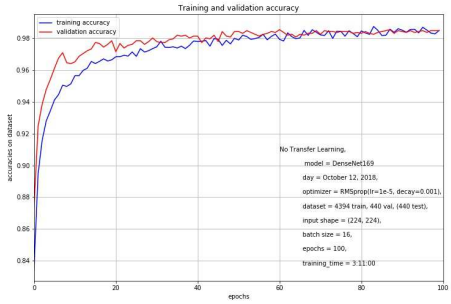
A.2 Treino de raiz (100 epochs): *accuracies*, *losses*



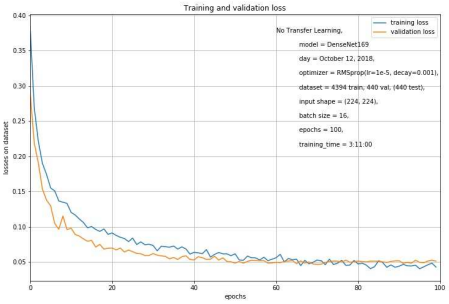
(a) DenseNet121 accuracy



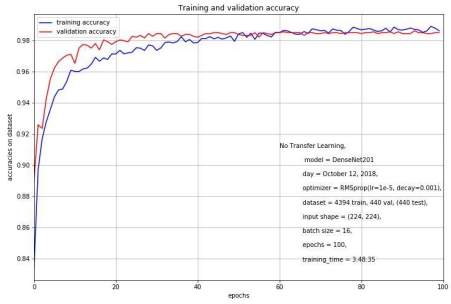
(b) DenseNet121 loss



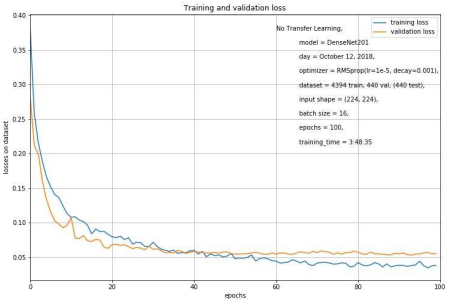
(c) DenseNet169 accuracy



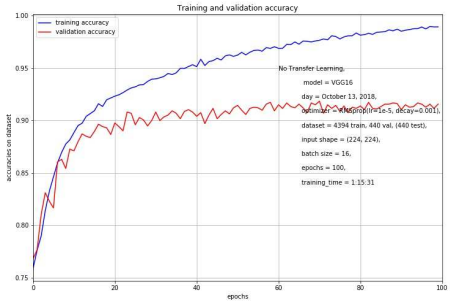
(d) DenseNet169 loss



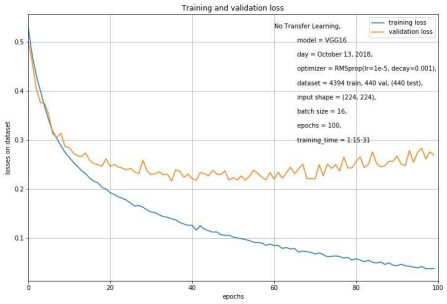
(e) DenseNet201 accuracy



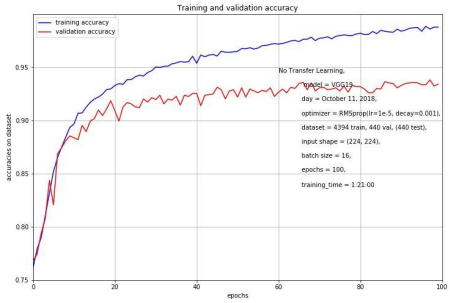
(f) DenseNet201 loss



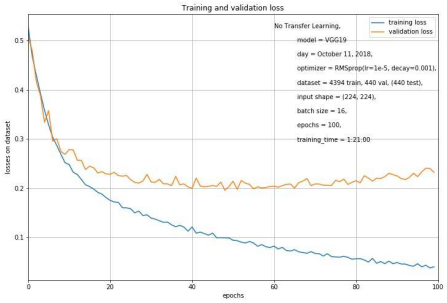
(g) VGG16 accuracy



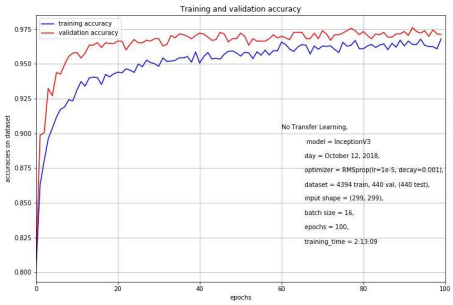
(h) VGG16 loss



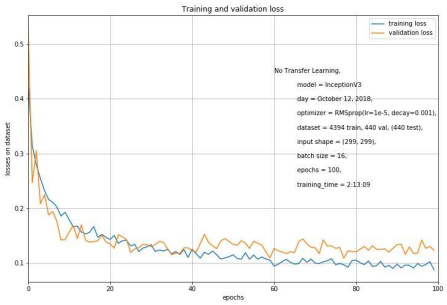
(i) VGG19 accuracy



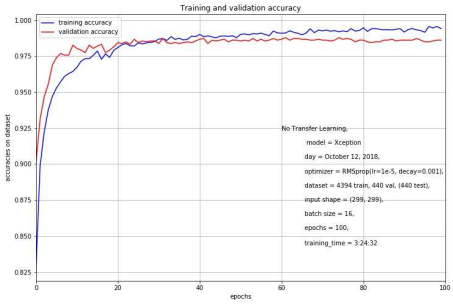
(j) VGG19 loss



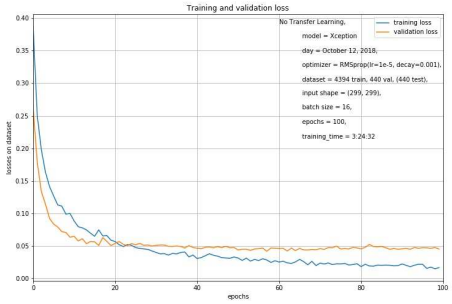
(k) InceptionV3 accuracy



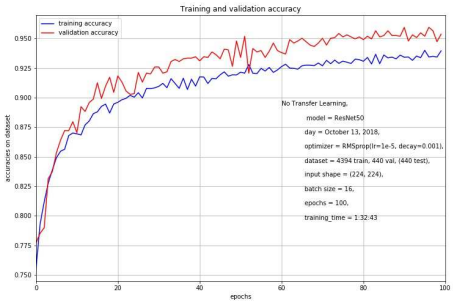
(l) InceptionV3 loss



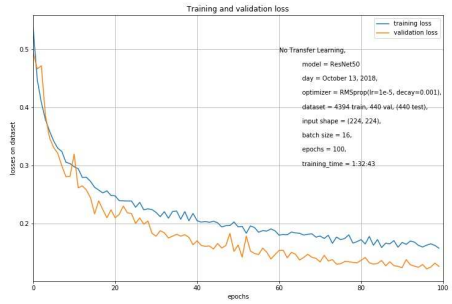
(m) Xception accuracy



(n) Xception loss



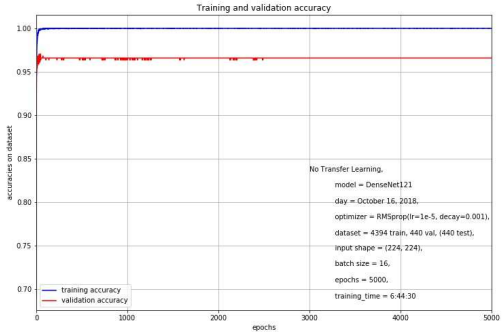
(o) ResNet50 accuracy



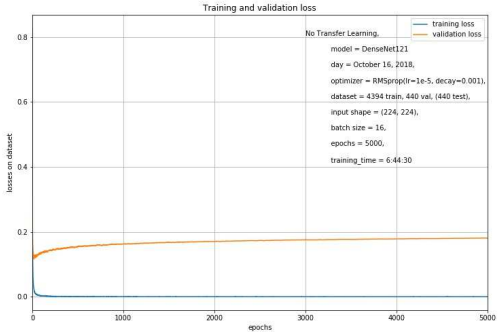
(p) ResNet50 loss

Figura A.2: *Train and validation*, *accuracies* and *losses* nos modelos treinados de raiz em 100 *epochs* .

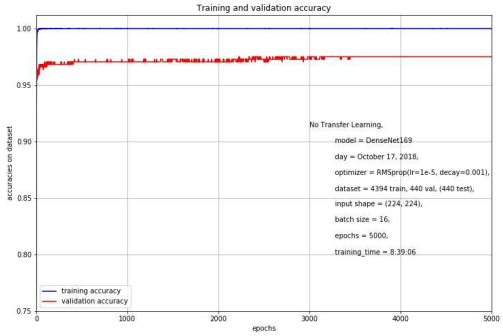
A.3 Transfer Learning (5000 epochs): *accuracies*, *losses*



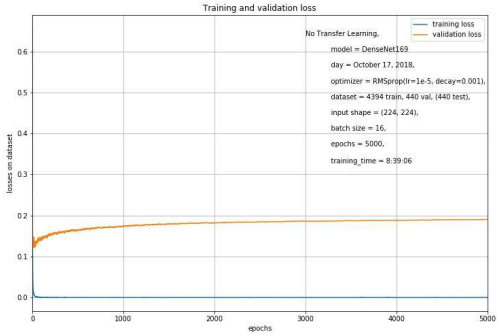
(a) DenseNet121 accuracy



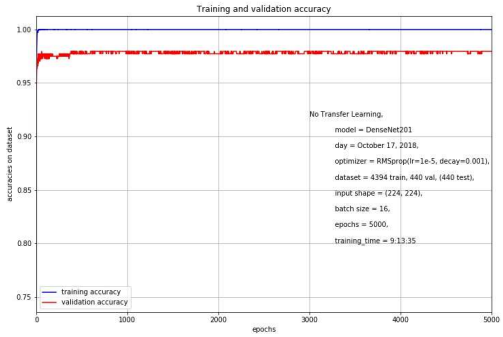
(b) DenseNet121 loss



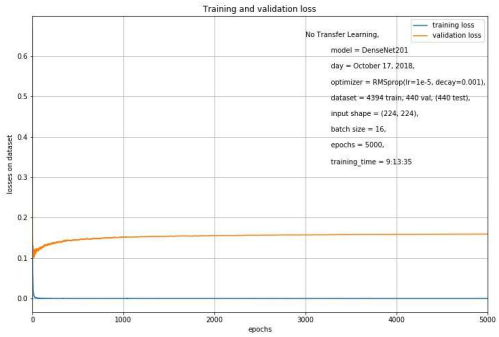
(c) DenseNet169 accuracy



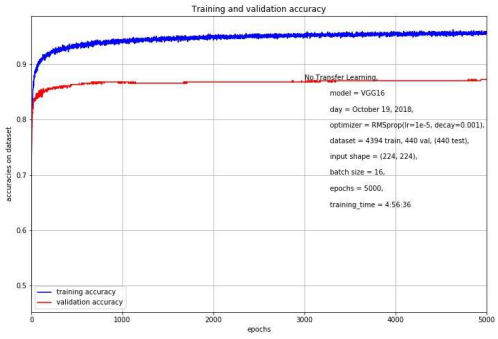
(d) DenseNet169 loss



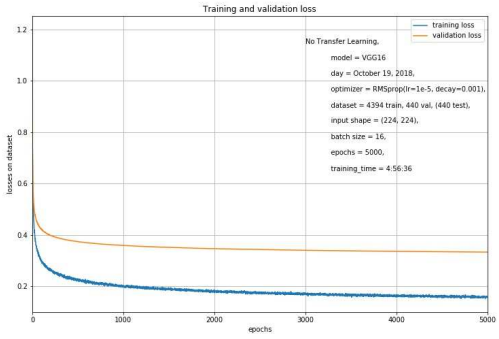
(e) DenseNet201 accuracy



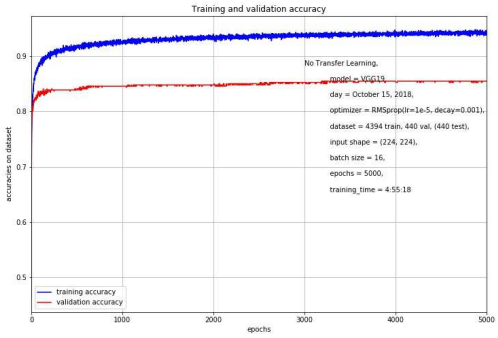
(f) DenseNet201 loss



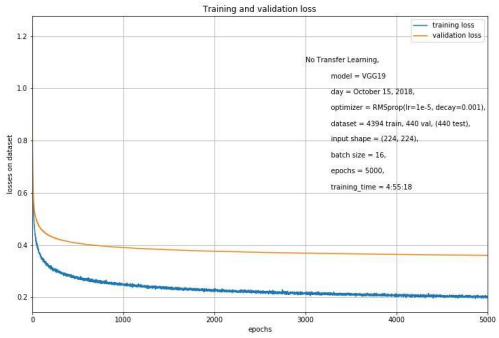
(g) VGG16 accuracy



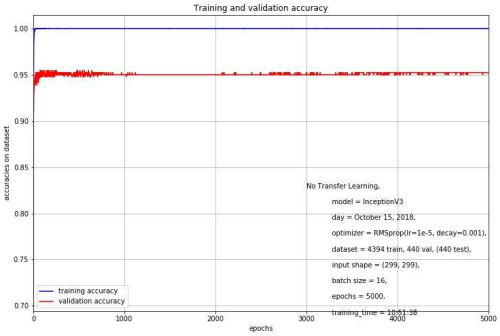
(h) VGG16 loss



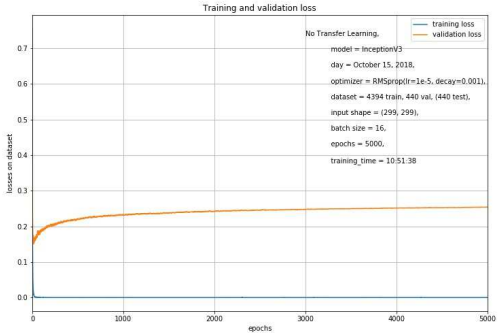
(i) VGG19 accuracy



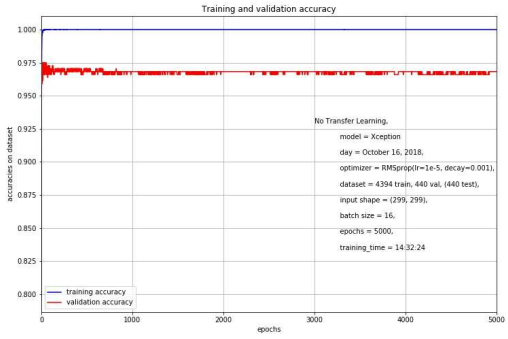
(j) VGG19 loss



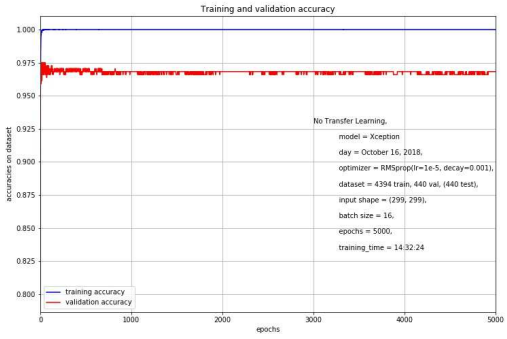
(k) InceptionV3 accuracy



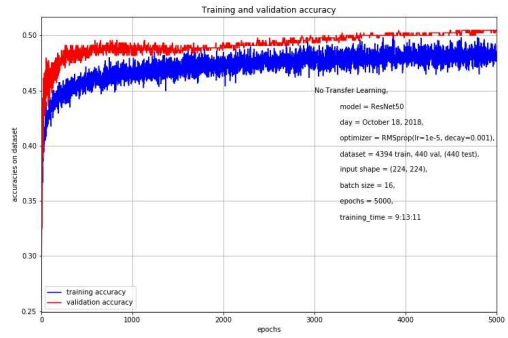
(l) InceptionV3 loss



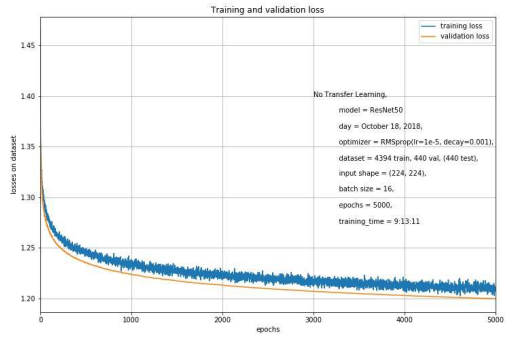
(m) Xception accuracy



(n) Xception loss



(o) ResNet50 accuracy



(p) ResNet50 loss

Figura A.3: *Train and validation*, *Accuracies* and *losses* nos modelos treinados de raiz em 5000 *epochs*.

Apêndice B

Losses de validação em modelos com *Transfer Learning*: 100 epochs

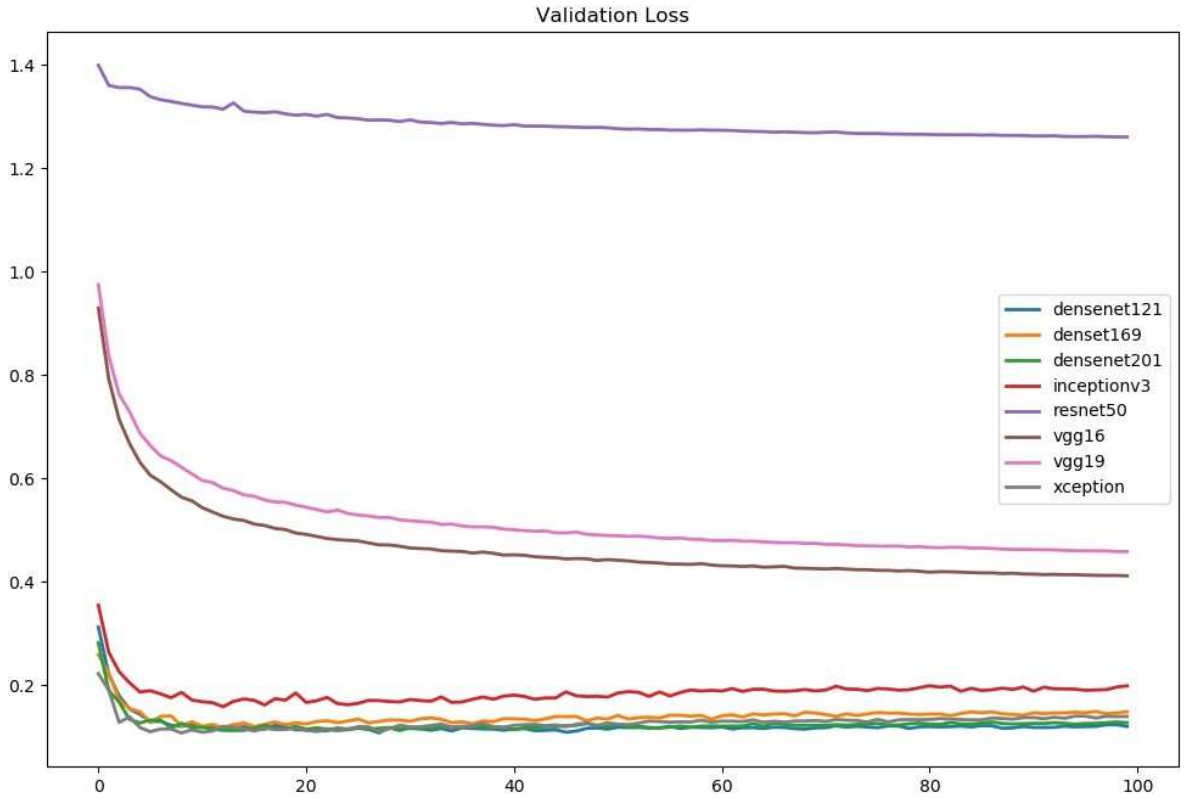


Figura B.1: *Losses* registadas pelo conjunto de modelos no *dataset* de validação com *Transfer Learning* ao longo de 100 epochs.

Apêndice C

Percentagens de acerto nas categorias

C.1 Percentagens de acerto em cada categoria.

Figura C.1: Percentagens de acerto em imagens do dataset de teste obtidas com modelos de *transfer learning* em 100 epochs

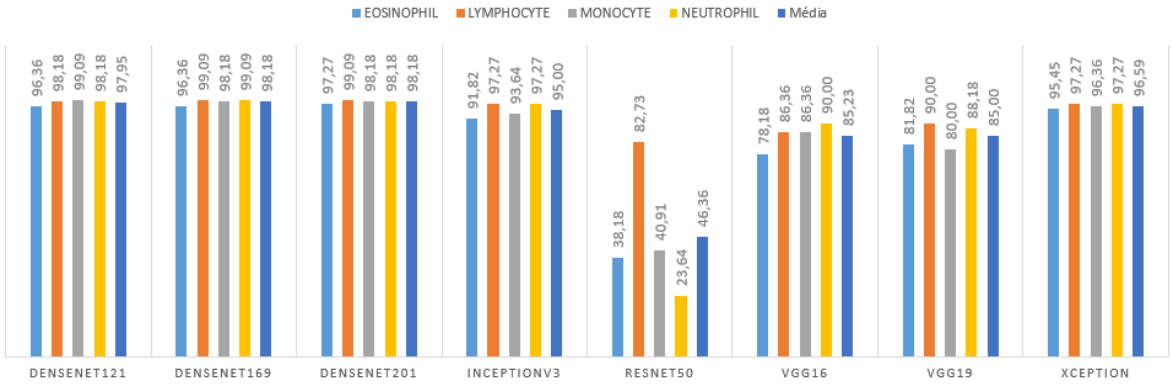


Figura C.2: Percentagens de acerto em imagens do dataset de teste obtidas com modelos treinados de raiz em 100 epochs

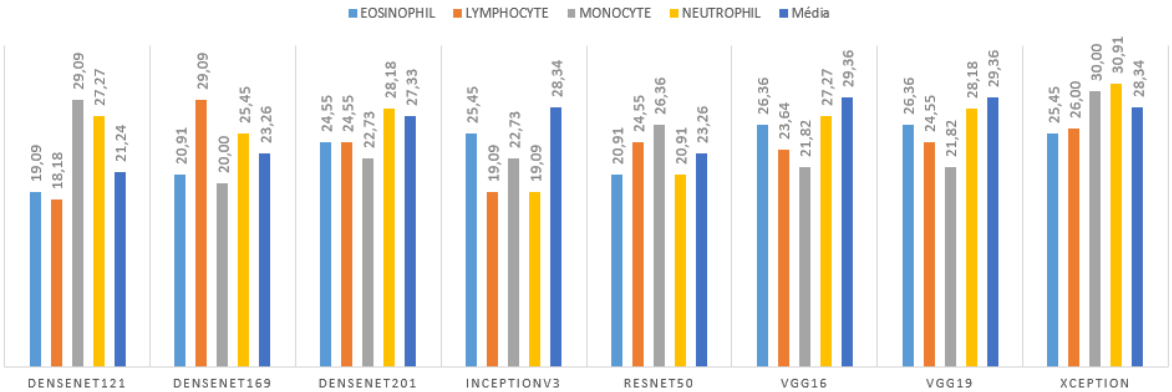
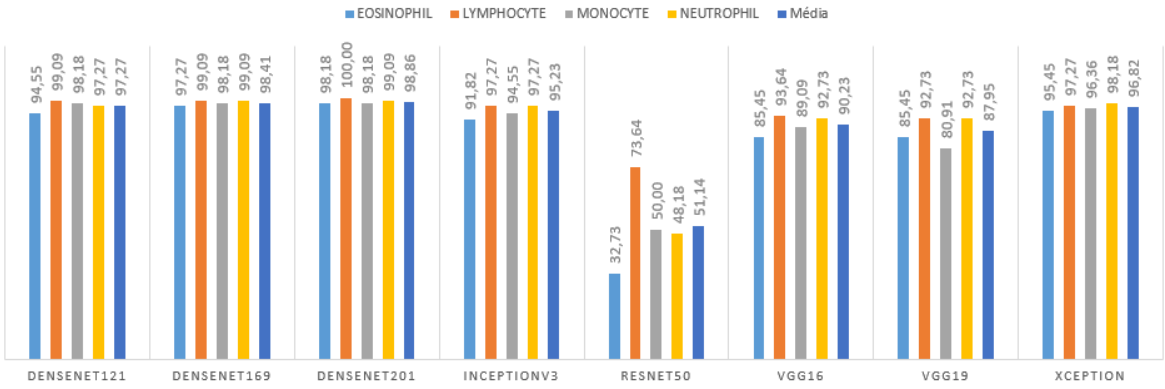


Figura C.3: Percentagens de acerto em imagens do dataset de teste obtidas com modelos de transfer learning em 5000 epochs



Apêndice D

Transfer Learning (100 *vs.* 5000 *epochs*)

PRECISION, RECALL, SCORE

	precision	recall	f1-score	support
class 0 (EOSINOPHIL)	0.98	0.96	0.97	110
class 1 (LYPHOCYTE)	1.00	0.98	0.99	110
class 2 (MONOCYTE)	0.97	0.99	0.98	110
class 3 (NEUTROPHIL)	0.96	0.98	0.97	110
avg / total	0.98	0.98	0.98	440

CONFUSION MATRIX

```
[[106   0   1   3]
 [  1 108   1   0]
 [  0   0 109   1]
 [  1   0   1 108]]
```

(a) DenseNet121 (100 epochs)

PRECISION, RECALL, SCORE

	precision	recall	f1-score	support
class 0 (EOSINOPHIL)	0.98	0.95	0.96	110
class 1 (LYPHOCYTE)	0.98	0.99	0.99	110
class 2 (MONOCYTE)	0.97	0.98	0.98	110
class 3 (NEUTROPHIL)	0.96	0.97	0.96	110
avg / total	0.97	0.97	0.97	440

CONFUSION MATRIX

```
[[104   1   1   4]
 [  1 109   0   0]
 [  0   1 108   1]
 [  1   0   2 107]]
```

(b) DenseNet121 (5000 epochs)

PRECISION, RECALL, SCORE

	precision	recall	f1-score	support
class 0 (EOSINOPHIL)	0.97	0.96	0.97	110
class 1 (LYPHOCYTE)	1.00	0.99	1.00	110
class 2 (MONOCYTE)	0.98	0.98	0.98	110
class 3 (NEUTROPHIL)	0.97	0.99	0.98	110
avg / total	0.98	0.98	0.98	440

CONFUSION MATRIX

```
[[106   0   1   3]
 [  0 109   1   0]
 [  2   0 108   0]
 [  1   0   0 109]]
```

(c) DenseNet169 (100 epochs)

PRECISION, RECALL, SCORE

	precision	recall	f1-score	support
class 0 (EOSINOPHIL)	0.97	0.97	0.97	110
class 1 (LYPHOCYTE)	1.00	0.99	1.00	110
class 2 (MONOCYTE)	0.98	0.98	0.98	110
class 3 (NEUTROPHIL)	0.98	0.99	0.99	110
avg / total	0.98	0.98	0.98	440

CONFUSION MATRIX

```
[[107   0   1   2]
 [  0 109   1   0]
 [  2   0 108   0]
 [  1   0   0 109]]
```

(d) DenseNet169 (5000 epochs)

PRECISION, RECALL, SCORE

	precision	recall	f1-score	support
class 0 (EOSINOPHIL)	0.98	0.97	0.98	110
class 1 (LYPHOCYTE)	0.99	0.99	0.99	110
class 2 (MONOCYTE)	0.97	0.98	0.98	110
class 3 (NEUTROPHIL)	0.98	0.98	0.98	110
avg / total	0.98	0.98	0.98	440

CONFUSION MATRIX

```
[[107   0   1   2]
 [  0 109   1   0]
 [  1   1 108   0]
 [  1   0   1 108]]
```

(e) DenseNet201 (100 epochs)

PRECISION, RECALL, SCORE

	precision	recall	f1-score	support
class 0 (EOSINOPHIL)	0.98	0.98	0.98	110
class 1 (LYPHOCYTE)	0.99	1.00	1.00	110
class 2 (MONOCYTE)	1.00	0.98	0.99	110
class 3 (NEUTROPHIL)	0.98	0.99	0.99	110
avg / total	0.99	0.99	0.99	440

CONFUSION MATRIX

```
[[108   0   0   2]
 [  0 110   0   0]
 [  1   1 108   0]
 [  1   0   0 109]]
```

(f) DenseNet201 (5000 epochs)

PRECISION, RECALL, SCORE

	precision	recall	f1-score	support
class 0 (EOSINOPHIL)	0.81	0.78	0.80	110
class 1 (LYPHOCYTE)	0.88	0.86	0.87	110
class 2 (MONOCYTE)	0.86	0.86	0.86	110
class 3 (NEUTROPHIL)	0.86	0.90	0.88	110
avg / total	0.85	0.85	0.85	440

CONFUSION MATRIX

```
[[86  8  5 11]
 [ 7 95  7  1]
 [ 7  4 95  4]
 [ 6  1  4 99]]
```

(g) VGG16 (100 epochs)

PRECISION, RECALL, SCORE

	precision	recall	f1-score	support
class 0 (EOSINOPHIL)	0.86	0.85	0.86	110
class 1 (LYPHOCYTE)	0.90	0.94	0.92	110
class 2 (MONOCYTE)	0.94	0.89	0.92	110
class 3 (NEUTROPHIL)	0.90	0.93	0.91	110
avg / total	0.90	0.90	0.90	440

CONFUSION MATRIX

```
[[ 94   6   2   8]
 [  6 103   1   0]
 [  4   5  98   3]
 [  5   0   3 102]]
```

(h) VGG16 (5000 epochs)

PRECISION, RECALL, SCORE

	precision	recall	f1-score	support
class 0 (EOSINOPHIL)	0.84	0.82	0.83	110
class 1 (LYPHOCYTE)	0.83	0.90	0.86	110
class 2 (MONOCYTE)	0.87	0.80	0.83	110
class 3 (NEUTROPHIL)	0.86	0.88	0.87	110
avg / total	0.85	0.85	0.85	440

CONFUSION MATRIX

```
[[90 11  2  7]
[ 5 99  5  1]
[ 7  7 88  8]
[ 5  2  6 97]]
```

(i) VGG19 (100 epochs)

PRECISION, RECALL, SCORE

	precision	recall	f1-score	support
class 0 (EOSINOPHIL)	0.87	0.85	0.86	110
class 1 (LYPHOCYTE)	0.88	0.93	0.90	110
class 2 (MONOCYTE)	0.92	0.81	0.86	110
class 3 (NEUTROPHIL)	0.86	0.93	0.89	110
avg / total	0.88	0.88	0.88	440

CONFUSION MATRIX

```
[[ 94   7   2   7]
[  3 102   3   2]
[  7   6  89   8]
[  4   1   3 102]]
```

(j) VGG19 (5000 epochs)

PRECISION, RECALL, SCORE

	precision	recall	f1-score	support
class 0 (EOSINOPHIL)	0.94	0.92	0.93	110
class 1 (LYPHOCYTE)	0.97	0.97	0.97	110
class 2 (MONOCYTE)	0.96	0.94	0.95	110
class 3 (NEUTROPHIL)	0.92	0.97	0.95	110
avg / total	0.95	0.95	0.95	440

CONFUSION MATRIX

```
[[101   2   0   7]
[  0 107   3   0]
[  4   1 103   2]
[  2   0   1 107]]
```

(k) InceptionV3 (100 epochs)

PRECISION, RECALL, SCORE

	precision	recall	f1-score	support
class 0 (EOSINOPHIL)	0.94	0.92	0.93	110
class 1 (LYPHOCYTE)	0.98	0.97	0.98	110
class 2 (MONOCYTE)	0.96	0.95	0.95	110
class 3 (NEUTROPHIL)	0.92	0.97	0.95	110
avg / total	0.95	0.95	0.95	440

CONFUSION MATRIX

```
[[101   2   0   7]
[  0 107   3   0]
[  4   0 104   2]
[  2   0   1 107]]
```

(l) InceptionV3 (5000 epochs)

PRECISION, RECALL, SCORE

	precision	recall	f1-score	support
class 0 (EOSINOPHIL)	0.95	0.95	0.95	110
class 1 (LYPHOCYTE)	0.99	0.97	0.98	110
class 2 (MONOCYTE)	0.96	0.96	0.96	110
class 3 (NEUTROPHIL)	0.96	0.97	0.97	110
avg / total	0.97	0.97	0.97	440

CONFUSION MATRIX

```
[[105   0   1   4]
[  1 107   2   0]
[  3   1 106   0]
[  2   0   1 107]]
```

(m) Xception (100 epochs)

PRECISION, RECALL, SCORE

	precision	recall	f1-score	support
class 0 (EOSINOPHIL)	0.95	0.95	0.95	110
class 1 (LYPHOCYTE)	0.99	0.97	0.98	110
class 2 (MONOCYTE)	0.96	0.96	0.96	110
class 3 (NEUTROPHIL)	0.96	0.98	0.97	110
avg / total	0.97	0.97	0.97	440

CONFUSION MATRIX

```
[[105   0   1   4]
[  1 107   2   0]
[  3   1 106   0]
[  1   0   1 108]]
```

(n) Xception (5000 epochs)

PRECISION, RECALL, SCORE

	precision	recall	f1-score	support
class 0 (EOSINOPHIL)	0.40	0.38	0.39	110
class 1 (LYPHOCYTE)	0.44	0.83	0.57	110
class 2 (MONOCYTE)	0.58	0.41	0.48	110
class 3 (NEUTROPHIL)	0.51	0.24	0.32	110
avg / total	0.48	0.46	0.44	440

CONFUSION MATRIX

```
[[42 43 17  8]
[10 91  4  5]
[24 29 45 12]
[28 45 11 26]]
```

(o) ResNet50 (100 epochs)

PRECISION, RECALL, SCORE

	precision	recall	f1-score	support
class 0 (EOSINOPHIL)	0.43	0.33	0.37	110
class 1 (LYPHOCYTE)	0.51	0.74	0.60	110
class 2 (MONOCYTE)	0.56	0.50	0.53	110
class 3 (NEUTROPHIL)	0.54	0.48	0.51	110
avg / total	0.51	0.51	0.50	440

CONFUSION MATRIX

```
[[36 34 27 13]
[ 9 81  7 13]
[14 21 55 20]
[24 24  9 53]]
```

(p) ResNet50 (5000 epochs)

Figura D.1: *Precision*, *recall*, *score* e *confusion matrix* dos modelos treinados com t*ransfer learning* em 100 e 5000 *epochs*

Apêndice E

Accuracy de treino e validação em modelos com e sem *T. Learning* (100 *epochs*)

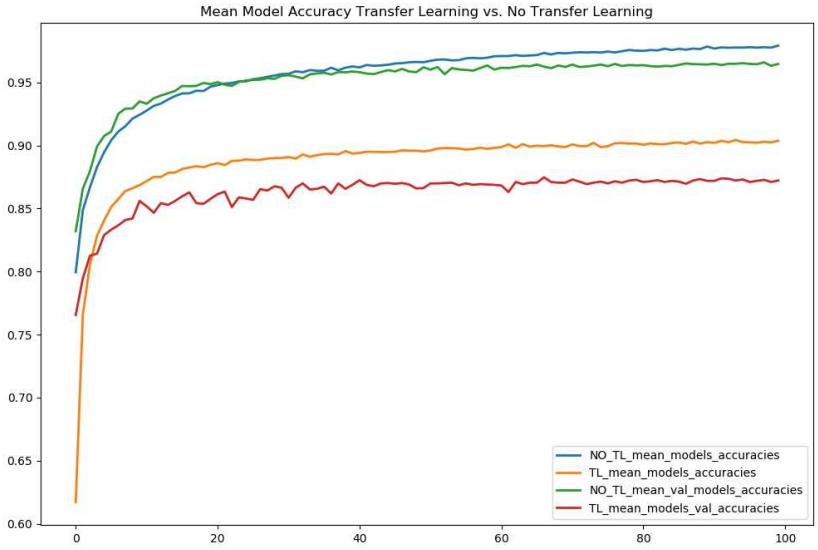


Figura E.1: *Accuracies* de treino e validação médias do conjunto de modelos treinados de raíz ou com *transfer learning* nos *datasets* de treino e validação, durante 100 epochs.

Apêndice F

Código Python para os modelos com *Transfer Learning*

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time
import datetime
import math

%matplotlib inline
from __future__ import print_function
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report, confusion_matrix

import keras
from keras import models
from keras import layers
from keras import optimizers
from keras.utils import to_categorical
from keras.preprocessing.image import ImageDataGenerator, load_img

from keras.applications.resnet50 import ResNet50
from keras.applications.resnet50 import preprocess_input,
    decode_predictions
from keras.applications.vgg16 import VGG16
from keras.applications.vgg16 import preprocess_input
from keras.applications.vgg19 import VGG19
from keras.applications.vgg19 import preprocess_input
from keras.applications.inception_v3 import InceptionV3
from keras.applications.densenet import DenseNet121
from keras.applications.densenet import DenseNet169
from keras.applications.densenet import DenseNet201
from keras.applications.nasnet import NASNetLarge
from keras.applications.nasnet import NASNetMobile
from keras.applications.inception_resnet_v2 import InceptionResNetV2
from keras.applications.xception import Xception
from keras.applications.mobilenet import MobileNet

import os
import cv2
import datetime
```

```
# MODEL CHOICE
model_choice = ResNet50(weights= 'imagenet',
                                                    include_top=False,
                                                    input_shape=(224, 224, 3)
                                                        )
model_name = 'ResNet50'
target_size = (224, 224)
sfvector = (7, 7, 2048) ##shape_feature_vector
reshape_feature_vector = (7* 7* 2048)

# TRAIN, VALIDATION AND TEST SET SIZE AND DIR
validation_dir = BASE_DIR +'images/TEST_SIMPLE2/'
train_dir = BASE_DIR + 'images/TRAIN/'
test_dir = BASE_DIR + 'images/TEST_SIMPLE/'
nTrain = 4394
nVal = 440
nTest = 440

# IMAGE DATA GENERATOR
def build(source=None):
        datagen = ImageDataGenerator(rescale=1. / 255)
            #featurewise_center=True,
            #featurewise_std_normalization=True)
        data_generator = datagen.flow_from_directory(
        source,  # this is the target directory
        target_size=(target_size[0], target_size[1]),
            batch_size=batch_size,
        class_mode='categorical',
        shuffle=False)
        class_dictionary = data_generator.class_indices
        return data_generator, class_dictionary


## TRAIN DATA, class dictionary, and train_features and train_labels
train_generator = []
train_features = np.zeros(shape=(nTrain, sfvector[0], sfvector[1],
    sfvector[2]))
train_labels = np.zeros(shape=(nTrain, 4))
train_generator = build(source=train_dir)
i = 0
for inputs_batch, labels_batch in train_generator[0]:
    features_batch = model_choice.predict(inputs_batch)
    train_features[i * batch_size : (i + 1) * batch_size] =
        features_batch
    train_labels[i * batch_size : (i + 1) * batch_size] = labels_batch
    i += 1
    if i * batch_size >= nTrain :
        break
train_features = np.reshape(train_features, (nTrain,
    reshape_feature_vector))


## VALIDATION DATA, class dictionary, and validation_features and
    validation_labels
validation_generator = []
validation_features = np.zeros(shape=(nVal, sfvector[0], sfvector[1],
```

```
    sfvector[2]))
validation_labels = np.zeros(shape=(nVal,4))
validation_generator = build(source=validation_dir)
i = 0
for inputs_batch, labels_batch in validation_generator[0]:
    features_batch = model_choice.predict(inputs_batch)
    validation_features[i * batch_size : (i + 1) * batch_size] =
        features_batch
    validation_labels[i * batch_size : (i + 1) * batch_size] =
        labels_batch
    i += 1
    if i * batch_size >= nVal:
        break
validation_features = np.reshape(validation_features,
                                 (nVal, sfvector[0] * sfvector[1] *
                                     sfvector[2]))


# MODEL TOP (OR MICRO-NET)
model = models.Sequential()
model.add(layers.Dense(256, activation='relu',
                       input_dim= sfvector[0] * sfvector[1] * sfvector
                           [2]))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(4, activation='softmax'))

# DEFINING THE OPTIMIZER AND COMPILING THE MODEL
optimizer = keras.optimizers.RMSprop(lr=1e-5, decay=0.001)
optim = 'RMSprop(lr=1e-5,␣decay=0.001)'
model.compile(optimizer=optimizer, #optimizers.RMSprop(lr=2e-4),
              loss='categorical_crossentropy',
              metrics=['acc'])

from keras.callbacks import CSVLogger
csv_logger = CSVLogger('TL_history_{}_{}.csv'.format(model_name,
                       datetime.date.today().strftime("%B␣%d,␣%Y")),
                       append=True, separator=';')

tic = time.process_time()

# SAVE THE HISTORY OS TRAINING AND VALIDATION
history = model.fit(train_features,
                    train_labels,
                    epochs=epochs,
                    batch_size=batch_size,
                    validation_data=(validation_features,
                                     validation_labels),
                    callbacks=[csv_logger])

toc = time.process_time()
print("----␣Total␣Computation␣time␣=␣" +
      str(datetime.timedelta(seconds=math.ceil(toc-tic))) + "␣seconds" )


## TEST DATA
datagen = ImageDataGenerator(rescale=1. / 255)
```

```
test_features = np.zeros(shape=(nTest,
                    sfvector[0], sfvector[1], sfvector[2]))
test_labels = np.zeros(shape=(nTest,4))

test_generator = datagen.flow_from_directory(
    test_dir,
    target_size=(target_size[0], target_size[1]),
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False)

i = 0
for inputs_batch, labels_batch in test_generator:
    features_batch = model_choice.predict(inputs_batch)
    test_features[i * batch_size : (i + 1) * batch_size] = features_batch
    test_labels[i * batch_size : (i + 1) * batch_size] = labels_batch
    i += 1
    if i * batch_size >= nTest:
        break

test_features = np.reshape(test_features,
                           (nTest,reshape_feature_vector))

predictions = model.predict_classes(validation_features)
prob = model.predict(validation_features)

## PPREDICT FOR TEST DATASET
Y_pred = model.predict(test_features)
print('\n PROBABILITIES')
print(Y_pred)

y_pred = np.argmax(Y_pred, axis=1)

print('\n RESULTS')
print(y_pred)

p=model.predict_proba(test_features) # to predict probability

target_names = ['class 0(EOSINOPHIL)', 'class 1(LYPHOCYTE)',
                'class 2(MONOCYTE)', 'class 3(NEUTROPHIL)']
print('\n PRECISION, RECALL, SCORE')
print(classification_report(np.argmax(test_labels,axis=1),
                            y_pred,target_names=target_names))
print('\n CONFUSION MATRIX')
print(confusion_matrix(np.argmax(test_labels,axis=1), y_pred))

keras.backend.clear_session()
```

Apêndice G

Código Python para os modelos treinados de raiz

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time
import datetime
import math

%matplotlib inline
from __future__ import print_function
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report, confusion_matrix

import keras
from keras import models
from keras import layers
from keras import optimizers
from keras.utils import to_categorical
from keras.preprocessing.image import ImageDataGenerator, load_img

from keras.applications.resnet50 import ResNet50
from keras.applications.resnet50 import preprocess_input,
    decode_predictions
from keras.applications.vgg16 import VGG16
from keras.applications.vgg16 import preprocess_input
from keras.applications.vgg19 import VGG19
from keras.applications.vgg19 import preprocess_input
from keras.applications.inception_v3 import InceptionV3
from keras.applications.densenet import DenseNet121
from keras.applications.densenet import DenseNet169
from keras.applications.densenet import DenseNet201
from keras.applications.nasnet import NASNetLarge
from keras.applications.nasnet import NASNetMobile
from keras.applications.inception_resnet_v2 import InceptionResNetV2
from keras.applications.xception import Xception
from keras.applications.mobilenet import MobileNet

import os
import cv2
import datetime
```

```
# MODEL CHOICE
model_choice = ResNet50(weights= 'imagenet',
                                                    include_top=False,
                                                    input_shape=(224, 224, 3)
                                                        )
model_name = 'ResNet50'
target_size = (224, 224)
sfvector = (7, 7, 2048) ##shape_feature_vector
reshape_feature_vector = (7* 7* 2048)

# TEST, VALIDATION AND TEST DIR AND SIZE
validation_dir = BASE_DIR +'images/TEST_SIMPLE2/'
train_dir = BASE_DIR + 'images/TRAIN/'
test_dir = BASE_DIR + 'images/TEST_SIMPLE/'
nTrain = 4394
nVal = 440
nTest = 440

#NUMBER OF EPOCHS, BATCH_SIZE
model_choice.summary()
epochs = 100
batch_size = 16
BASE_DIR = "/home/deep/Documents/Romeu_Beato/"

model = model_choice

# OPTIMIZER
optimizer = keras.optimizers.RMSprop(lr=1e-5, decay=0.001)
optim= 'RMSprop(lr=1e-5,_decay=0.001)'

# COMPILING THE MODEL
model.compile(loss='binary_crossentropy',
              optimizer=optimizer,
              metrics=['accuracy'])

train_datagen = ImageDataGenerator(rescale=1. / 255)
test_datagen = ImageDataGenerator(rescale=1. / 255)

#TRAIN DATA GENERATOR
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(target_size[0], target_size[1]),
    batch_size=batch_size,
    class_mode='categorical')

# TEST DATA GENERATOR
validation_generator = test_datagen.flow_from_directory(
    validation_dir,
    target_size=(target_size[0], target_size[1]),
    batch_size=batch_size,
    class_mode='categorical')


from keras.callbacks import CSVLogger
csv_logger = CSVLogger('NO_TL_history_{}_{}.csv'.format(model_name,
                    datetime.date.today().strftime("%B_%d,_%Y")),
```

```
                    append=True, separator=';')


tic = time.process_time()

# SAVE THE TRAINING AND VALIDATION HISTORY
history = model.fit_generator(train_generator,
                              steps_per_epoch=nTrain // batch_size,
                              epochs=epochs,
                              validation_data=validation_generator,
                              validation_steps=nVal // batch_size,
                              callbacks=[csv_logger])

toc = time.process_time()

print("---- Total Computation time = " +
    str(datetime.timedelta(seconds=math.ceil(toc-tic))) + " seconds")


# TEST DATA
test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(target_size[0], target_size[1]),
    batch_size=batch_size,
    class_mode='categorical')

Evaluate_history = model.evaluate_generator(test_generator,
                                            steps=None,
                                            max_queue_size=10,
                                            workers=1,
                                            use_multiprocessing=False,
                                            verbose=1)

test_generator.reset()
score = model.predict_generator(test_generator,
                                          steps=None,
                                          max_queue_size=10,
                                          workers=1,
                                          use_multiprocessing=False,
                                          verbose=1)



## PREDICT FOR TEST DATASET
Y_pred = score
print('\n PROBABILITIES')
print(Y_pred)

y_pred = np.argmax(Y_pred, axis=1)

print('\n RESULTS')
print(y_pred)

labels = (train_generator.class_indices)
labels = dict((v,k) for k,v in labels.items())
predictions = [labels[k] for k in y_pred]
```

```
filenames=test_generator.filenames

pprint.pprint(filenames)

results=pd.DataFrame({"Filename":filenames,
                      "Predictions":predictions})
print(results)

# SAVE RESULTS TO .csv
results.to_csv('''No_TL_{}_{}_{}_{}img.csv'''.format(model_name,
                                    datetime.date.today().strftime("%B_%d,_%
                                        Y"),
                                    optim,
                                    nTrain+nVal+nTest,
                                    index=False))

 keras.backend.clear_session()
```