Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Criação Automática de Datasets e Treinamento de Classificadores Sob Demanda a Partir de Web Crawling e Deep Learning

Hugo Luís Andrade Silva

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Marcus Vinicius Lamar

Brasília
2018

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Engenharia da Computação

Coordenador: Prof. Dr. José Edil Guimarães de Medeiros

Banca examinadora composta por:

Prof. Dr. Marcus Vinicius Lamar (Orientador) — CIC/UnB
Prof. Dr. Teófilo Emídio de Campos — CIC/UnB
Prof. Dr. Vinícius Ruela Pereira Borges — CIC/UnB

CIP — Catalogação Internacional na Publicação

Silva, Hugo Luís Andrade.

Criação Automática de Datasets e Treinamento de Classificadores Sob Demanda a Partir de Web Crawling e Deep Learning / Hugo Luís Andrade Silva. Brasília : UnB, 2018.
118 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2018.

1. Visão Computacional, 2. Aprendizado Profundo, 3. web crawling, 4. reconhecimento de padrões, 5. Aprendizado de Máquina

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Criação Automática de Datasets e Treinamento de Classificadores Sob Demanda a Partir de Web Crawling e Deep Learning

Hugo Luís Andrade Silva

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Marcus Vinicius Lamar (Orientador)
CIC/UnB

Prof. Dr. Teófilo Emídio de Campos
CIC/UnB

Prof. Dr. Vinícius Ruela Pereira Borges
CIC/UnB

Prof. Dr. José Edil Guimarães de Medeiros
Coordenador do Curso de Engenharia da Computação

Brasília, 08 de agosto de 2018

Dedicatória

Dedico este trabalho a meus pais, Salvane e Pierre, aos meus irmãos, Marina e André, e aos demais familiares e amigos que me apoiaram ao longo de minha jornada acadêmica.

Agradecimentos

Agradeço ao meu orientador, Professor Marcus Vinícius Lamar, pelo apoio ao longo do projeto, com constantes *feedbacks* sobre o texto e reuniões frequentes durante todo o desenvolvimento dele, ao meu amigo Pedro Henrique Pamplona Savarese, pelas discussões e referências a artigos sempre que necessário e ao professor Teófilo Emídio de Campos, pelas ideias sobre como melhorar determinadas partes, além de comentários e dicas acerca de como melhor articular o que foi desenvolvido no trabalho

Resumo

Aplicações de *Machine Learning* da maneira como são feitas hoje comumente requerem coleta de dados manual, que pode demorar para ser feita e, por isso, limita aplicações que requerem conjuntos de dados grandes. O trabalho trata do projeto de uma aplicação *web* onde o usuário terá as possibilidades de criar, armazenar e carregar detectores de objetos e *datasets* de maneira automática, removendo a necessidade de coletar manualmente as imagens. A criação é feita dinamicamente, de forma que os *datasets* utilizados no treinamento serão criados de maneira *on demand* fazendo uso de *web crawling*, tirando assim a restrição de se utilizar *datasets* prontos, que nem sempre são encontrados para aplicações específicas. Em seguida, são realizados testes relativos aos subsistemas de Inteligência Artificial envolvidos em cada etapa do projeto descrito e a viabilidade da criação de uma aplicação desse tipo é analisada, tendo em vista os resultados dos experimentos supracitados. O projeto consiste em dois passos principais: remoção de imagens ruidosas e treinamento de classificadores com as restantes. Os métodos propostos atingiram 94.4% no primeiro passo e 98.98% no segundo usando imagens obtidas usando classes do CIFAR-10 como *queries*.

Palavras-chave: Visão Computacional, Aprendizado Profundo, web crawling, reconhecimento de padrões, Aprendizado de Máquina

Abstract

Current Machine Learning applications usually require manual data collection, which take long to complete and thus may limit applications which require large datasets. This project designs a web application where the user will have the possibilities of creating, storing and loading object detectors and datasets automatically, removing the need to manually collect images. Creation is done dinamically, and the datasets used are created on demand using web crawling, removing the restriction of having to utilize already existing datasets, which may not exist for specific applications. Next, Artificial Intelligence subsystems present on each step of the described project are tested and the viability of creating such a web application is analyzed in light of the results of the above mentioned experiments. The project is consists of two main steps: removing noisy images and training classifiers with the rest. The proposed methods achieved 94.4% on the first step and 98.98% on the second one on images downloaded using CIFAR-10 classes as queries.

Keywords: Computer Vision, Deep Learning, web crawling, pattern recognition, Machine Learning

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

AML Applied Machine Learning. 52

API Application Programming Interface. 3, 52, 70, 71, 82, 91

CNN Convolutional Neural Networks. 24, 25

CPU Central Processing Unit. 63

FAIR Facebook Artificial Intelligence Research. 52

GPU Graphics Processing Unit. 7, 52, 63, 64, 98

HSV *Hue Saturation Value*. 84

HTML HyperText Markup Language. 70, 71

ILSVRC Imagenet Large Scale Visual Recognition Challenge. 1

JSON JavaScript Object Notation. 53

KNN K-Nearest Neighbors. 35

LBP Local Binary Patterns. 8, 9, 16, 39–41

LBPU Uniform Local Binary Patterns. 39

NLP Natural Language Processing. 8

REST Representational State Transfer. 53

ROC Receiver Operating Characteristic. 48, 49

SVM Support Vector Machines. 35, 41–43

Capítulo 1

Introdução

Atualmente, são vistos vários avanços na área de Inteligência Artificial, sobretudo com o desenvolvimento do conceito de Deep Learning, o estado na arte em relação a essa área. Reconhecimento facial de precisão próxima à classificação humana foi alcançado com DeepFace[20], classificadores usados em competições como o Imagenet Large Scale Visual Recognition Challenge (ILSVRC) conseguem atingir *top-5-error* (a classe certa está entre as 5 mais prováveis) de menos de 5% no Imagenet, dataset com aproximadamente 10 milhões de imagens. Apesar disso, esses avanços apresentam uma forte limitação no que tange à maneira como esses dados são obtidos, já que são colhidos a mão.

Tendo em mente o objetivo inicial de Inteligência Artificial, ou seja, o estudo ou projeto de agentes inteligentes, que percebem seu ambiente e tomam atitudes que maximizam suas chances de sucesso[21], esses sistemas certamente teriam que possuir classificadores de quase todos os objetos conhecidos, de forma a identificá-los. Para perceber isso, basta imaginar um robô olhando para determinada cena: ele teria que ser capaz de identificar cada objeto daquela cena para depois tomar ações com base nesse conhecimento. A coleta de imagens para o *training set* feita de uma maneira manual limita a possibilidade de obtenção de dados e, por isso, é um gargalo no contexto de Inteligência Artificial.

1.1 Problema

É notado que a coleta manual e criação de um *dataset* contendo todos os objetos existentes é algo de extrema dificuldade, porém existe uma estrutura *online* que pode ser usada como facilitadora desse processo.

Search engines, como Google ou Yahoo, operam de maneira a retornar um grande conjunto de imagens correspondendo à *keyword* digitada como chave de busca. Esse retorno, ilustrado na Figura 1.1 é implementado utilizando o conteúdo das *keywords*, de maneira que, nos códigos-fonte das páginas *web* disponíveis, são buscadas compatibilidades

com a *keyword* buscada, sendo retornadas as imagens relevantes para a busca. Nessa estrutura é comum haver algumas imagens retornadas que não correspondem ao esperado, como ilustrado na Figura 1.2, para utilização dessas *search engines*, isso certamente é um problema. Além disso, como podemos ver na Figura 1.1, as buscas ocorrem de maneira diferente dependendo do idioma em que se encontra a *keyword*, o que pode ser explorado para gerar mais resultados.

(a) Basketball (b) Bola de Basquete

(c) Elefante (d) Torta

Figura 1.1: Keywords e imagens retornadas

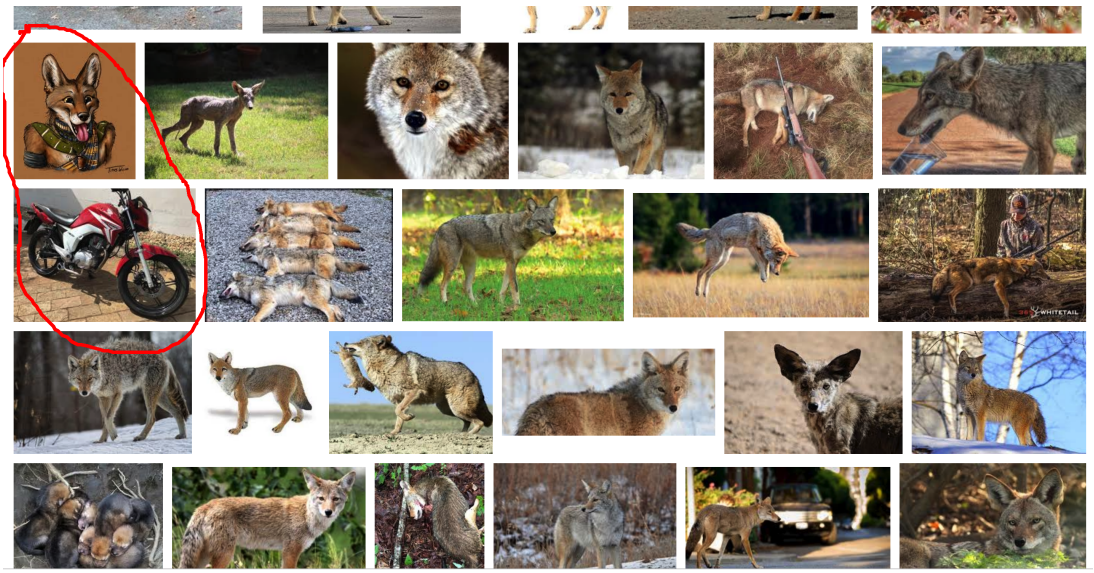


Figura 1.2: Imagem fora de contexto, enfatizada em vermelho

A ideia deste trabalho é utilizar o retorno dessas *search engines* para treinar classificadores, de forma a reduzir a necessidade de coleta manual de dados. Para isso, será

necessário retirar as imagens irrelevantes para a busca de maneira automatizada. É importante quantificar o número de imagens que uma busca como essa retorna, no decorrer do projeto, foi observado que, de maneira geral, o Google disponibiliza cerca de 800 resultados para cada busca. Se várias buscas forem feitas para o mesmo objeto, alterando, por exemplo, apenas o idioma em que o nome desse objeto é digitado, pode-se multiplicar esse número, além disso, existem outras *search engines* que também retornam resultados de boa qualidade.

1.2 Objetivo geral

O objetivo geral do trabalho é a proposição e estudo de um sistema de criação automática de classificadores e *datasets* baseados em *queries* definidas pelo usuário. Dessa forma, caso o usuário queira criar bancos de imagens ou aplicações baseadas em Machine Learning para algum domínio específico, esse sistema faria uso de *web crawling* para obter as imagens necessárias para realização desses objetivos. Nesse sistema, o usuário precisa apenas digitar as palavras referentes às classes desejadas e esperar o retorno do sistema com o classificador projetado.

Como uma implementação desse sistema requer o funcionamento consistente de várias sub-etapas, esse trabalho foca em realizar testes visando delimitá-las e mostrar como resolver cada uma delas, além de obter estatísticas que embasem o uso dos métodos propostos, a implementação real do sistema fica como trabalho futuro.

Na prática, um sistema como esse poderia ser lançado como API, em que o usuário da API cria esses classificadores para suas aplicações específicas. Ele poderia, por exemplo, criar uma aplicação que reconheça automaticamente diferentes tipos de espécies para auxiliar o trabalho de biólogos, ou criar uma aplicação que dissesse em qual parte do mundo determinada foto foi tirada. Tal sistema poderia, ainda, auxiliar a navegação de robôs permitindo que fossem reconhecidos diferentes tipos de objetos para guiar diferentes tipos de ação, como reconhecer que determinado objeto é um obstáculo e desviar dele ou reconhecer que outro objeto está fora do lugar e retorná-lo ao lugar devido.

1.3 Objetivos específicos

Com o propósito de organizar o trabalho, são enumerados os objetivos específicos:

1. Detalhar um sistema que atenda, em alto nível, aos requisitos propostos nos objetivos gerais usando diagramas de tela e de comunicação

2. Estudar o desempenho de arquiteturas profundas e comparar implementações em CPU e GPU
3. Obter imagens de *web crawlers* de um conjunto de classes pré-definido para realizar testes
4. Propor e avaliar um sistema de remoção automática de imagens não-relevantes que não possuem relação com a *query*
5. Treinar classificadores usando o *framework* Tensorflow para avaliar o quão adequadas as imagens relevantes são para essa tarefa
6. Avaliar dificuldades em se usar o sistema descrito para detecção de objetos
7. Descobrir e analisar possíveis dificuldades adicionais na implementação de um sistema desse tipo

A Figura 1.3 sumariza o fluxo de execução dos objetivos propostos, sendo que as dependências são representadas com setas.

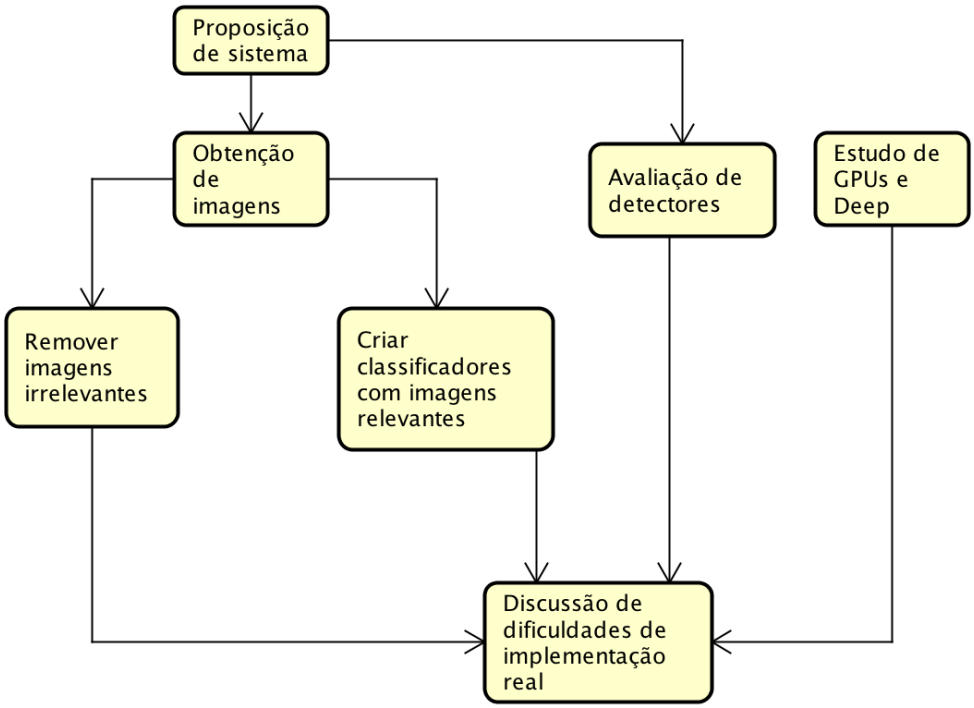


Figura 1.3: Fluxo de realização dos objetivos propostos

Vale ressaltar que, caso um sistema desses fosse de fato implementado, a etapa "remover imagens irrelevantes" teria necessariamente que vir antes de "treinar classificadores

com imagens relevantes", como será melhor explicado no decorrer do trabalho. No entanto, para facilitar a execução dos experimentos, esses estudos serão projetados sem que um necessite do retorno do outro, o que permite realização independente dos experimentos e evita que um deles fique impedido de ser explorado até que se complete a realização do outro.

1.4 Organização do trabalho

O trabalho foi organizado da seguinte forma: no Capítulo 2 é apresentado o referencial teórico necessário à compreensão do problema e das soluções propostas, sendo que ele trata tanto do material teórico de Deep Learning, que será altamente empregado para resolver os problemas desse trabalho, quanto do material teórico sobre detectores objetos, que será usado na discussão de uma parte não implementada do projeto. No Capítulo 3 é apresentado o sistema proposto, sendo que suas etapas são melhor discutidas e separadas e as soluções possíveis para cada uma são apresentadas. Os experimentos realizados e resultados obtidos são discutidos no Capítulo 4. Finalmente, o Capítulo 5 apresenta as conclusões deste trabalho e proposta de trabalhos futuros.

Capítulo 2

Referencial Teórico

Neste capítulo serão abordados os principais conceitos que serão explorados ao longo do trabalho. Inicialmente, será abordado o básico do Deep Learning e algumas de suas técnicas mais recentes: redes convolucionais, Residual Networks, Triplet Networks, NASNets e *transfer learning*. Essas técnicas são o estado da arte de visão computacional e vêm substituindo todas as técnicas clássicas que as precederam.

Por fim, o capítulo descreve o funcionamento básico de um detector de pedestres. Essa descrição é feita para embasar a discussão sobre as dificuldades da etapa de detecção de objetos proposta como um dos objetivos do trabalho.

2.1 Machine Learning

O Aprendizado de Máquina (Machine Learning), de maneira geral, consiste em métodos matemáticos que, quando implementados, tornam possível que computadores aprendam a prever comportamentos futuros baseados em padrões fornecidos. Em 1959, Arthur Samuel definiu Machine Learning como o "campo de estudo que dá aos computadores a habilidade de aprender sem serem explicitamente programados" [22]. Esse campo permite unir a velocidade de processamento de dados de computadores a comportamentos anteriormente exclusivo de humanos, permitindo automatização de várias tarefas. Alguns exemplos de aplicação são: tradução, reconhecimento de voz, previsão de valores da bolsa, reconhecimento facial e reconhecimento de ações.

2.1.1 Deep Learning

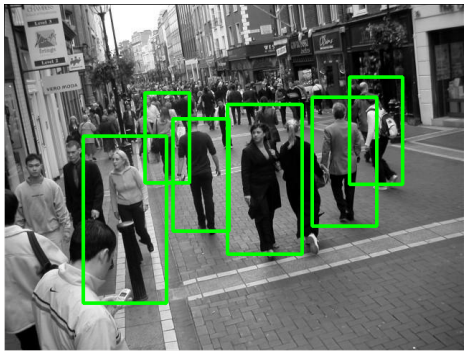
O Aprendizado Profundo [23] (Deep Learning) se refere a uma técnica popularizada nos anos 2000 em Inteligência Artificial e dominante em publicações de Machine Learning dos anos 2010 e que permite maior aglomeração de profissionais de áreas como processamento

de linguagem natural, visão computacional e reconhecimento de áudio para estudos de assuntos comuns. Pode ser vista como o próximo passo em direção ao objetivo inicial e utópico de fazer máquinas pensarem, visto que amplia enormemente a complexidade e possibilidades dos modelos já existentes na área de Machine Learning. O que permitiu essa evolução foi a modernização dos próprios computadores, com o desenvolvimento de GPUs cada vez mais poderosas, os cientistas responsáveis por pesquisa e desenvolvimento tiveram ideias de modelos que pudessem fazer uso dessas tecnologias. Anteriormente, o treinamento desses modelos seria inviável, ainda hoje, muitos deles ficam dias sendo executados até que o treinamento se estabilize.

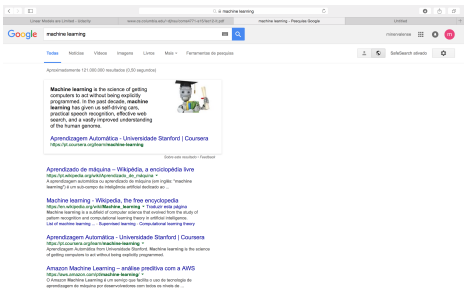
2.1.2 Aprendizado Supervisionada

Para entender a ideia por trás do *Deep Learning*, é necessário que se conheçam alguns conceitos básicos de Inteligência Artificial. O primeiro deles é o aprendizado supervisionado, que consiste na utilização de um conjunto (*set*) de dados de treino para que o sistema possa aprender algo. Nesse contexto, normalmente é buscado um classificador que consiga diferenciar entre várias classes, um exemplo comum é um sistema que, tendo como entrada fotos de 10 objetos diferentes, consiga diferenciar entre os 10, dizendo qual imagem se refere a qual objeto. Para isso, um conjunto de imagens pré-marcadas dos 10 objetos deve ser fornecido ao sistema, para que os padrões de características de cada objeto possam ser aprendidos.

De modo geral, haverá um *set* de treino, que poderá consistir em dados, por exemplo, de áudio, vídeo, ou texto. Características (*features*) são extraídas de cada exemplo desse *set*, com isso, é formado um vetor para cada exemplo. O sistema deverá aprender a utilizar esses vetores para prever algo, o exemplo citado acima tratava de classificação de objetos, mas existem também casos como regressão, em que os exemplos devem ser generalizados para que o sistema possa prever algum valor (um exemplo é a previsão da bolsa de valores usando Machine Learning). No aprendizado supervisionado, as previsões são feitas por humanos e dadas para o sistema, a partir delas, há uma tentativa de imitar o comportamento fornecido e realizar novas previsões para exemplos desconhecidos



(a) Detecção de pedestres



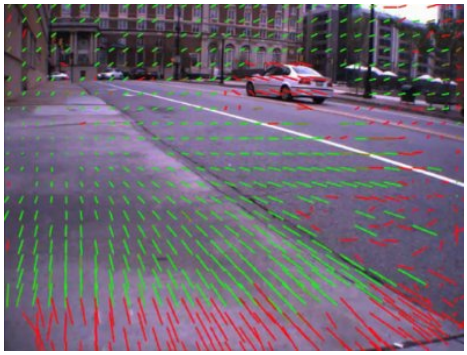
(b) Web searching

Figura 2.1: Alguns exemplos de aprendizado supervisionada

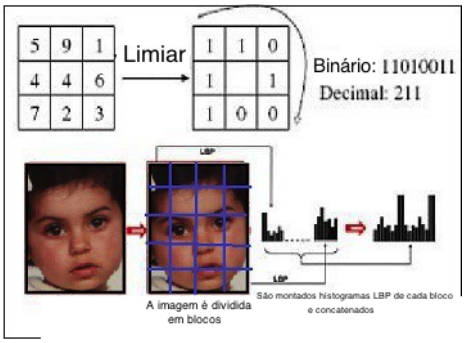
A Figura 2.1 apresenta alguns exemplos de aprendizado supervisionado aplicado ao rastreamento de pessoas, em que fotos de pedestres são fornecidas para que eles possam ser aprendidos pelo detector e pesquisa na Web, em que é usado Natural Language Processing (NLP) para que significados de palavras também possam ser levados em conta ao fazer a busca por páginas e com isso o resultado seja melhorado.

Regressão logística

Embora existam vários tipos de aprendizado em Inteligência Artificial, nessa seção focaremos na aprendizado supervisionado. Um dos modelos mais simples para classificação é a regressão logística[24][25] que, apesar de ser chamada regressão, trata-se de um problema de classificação. O primeiro passo para montar o classificador é definir o vetor de características(*features*) a ser utilizado e realizar possíveis preprocessamentos (normalização de medidas, por exemplo). Alguns vetores de *features* normalmente utilizados são representados na Figura 2.2. Desses, o *optical flow*[26] se refere a utilização de histogramas dos vetores de movimento dos quadros (*frames*) de determinado vídeo, sendo que os *bins* do histograma se referem às direções de cada vetor e as intensidades desses vetores também são consideradas na montagem do histograma. LBPs [27], por sua vez, são normalmente usados para classificadores que envolvem textura, o exemplo ilustrado atribui 1 para elementos com valor maior que o central e 0 para elementos com valor menor do que o central, em seguida, é definido um código para cada configuração centro-vizinhança e são montados histogramas com esses códigos.



(a) *Optical Flow*



(b) Local Binary Patterns (Fonte[15])

Figura 2.2: Alguns exemplos de geração de *features* para aprendizado

A partir do vetor de entrada, aqui referido como x, o sistema realizará uma operação do tipo $Ax + b$. Se tomamos p como o número de características e L o número de classes, x é $p \times 1$, A é $L \times p$ e b é $L \times 1$. Na regressão logística, o objetivo é ter Y com o número de entradas igual a L, o número de classes. Além disso, usualmente, cada entrada é uma probabilidade $\in [0, 1]$ de que o exemplo pertença àquela classe, tal que a soma das probabilidades seja 1. O vetor $Ax + b$ não possui essa característica, logo, deve ser utilizado algum método para converter as entradas reais desse vetor em probabilidades.

Softmax

A função usualmente utilizada para gerar tal formato é a chamada *softmax*[28], definida por

$$Y_{out\_i} = \frac{e^{Y_{in\_i}}}{\sum_{j=1}^{L} e^{Y_{in\_j}}} \tag{2.1}$$

onde Y_{in} são os valores de entrada e Y_{out} de saída.

Analisando a Equação 2.1, constata-se que a soma de todos os Y_{out_i} é igual a 1 e todos eles se encontram no intervalo $[0, 1]$. O próximo passo é determinar a matriz A e o vetor b que permitem com que o sistema melhor se adapte aos pares (x_i, y_i) fornecidos pelo classificador humano. Para isso, é necessária a definição de uma função de custo, que calcula o prejuízo dos erros de classificação. Posteriormente, essa função deverá ser otimizada tomando A e b como as variáveis de otimização.

Cross entropy

Uma escolha comum é definir a função de custo com base em entropia cruzada. A entropia cruzada entre vetores S e L é dada por

$$D(S, L) = -\sum_i L_i log(S_i) \tag{2.2}$$

.

Para o exemplo em questão, temos que S corresponde a $softmax(Ax+b)$, L é o vetor de classificação humana e é da seguinte forma, com a entrada 1 relativa à classe a qual o exemplo pertence:

$$L = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix} \tag{2.3}$$

Além disso, vale ressaltar que essa função não é comutativa, ou seja, $D(S, L) \neq D(L, S)$. A função de entropia para dois vetores é aplicada a todos os N exemplos do *set* de treino para formar a função de custo, que fica como na Equação 2.4.

$$F(A, b) = \frac{1}{N} \sum_{j=1}^{N} D(softmax(Ax_j + b), Y_j) \tag{2.4}$$

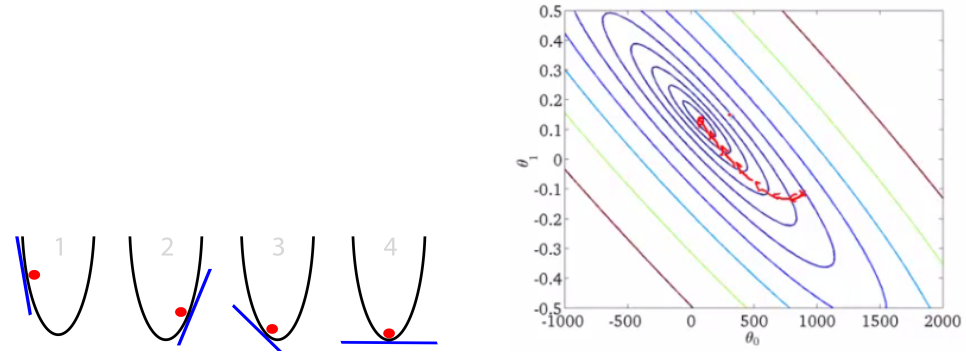
Após a definição do objetivo do classificador como uma função, o próximo passo é minimizá-la.

Gradiente Descendente

Para minimizar a função de custo, normalmente se usa métodos de otimização numérica. Nesse projeto, é usado o gradiente descendente[29], que consiste em utilizar, para um vetor de pesos genérico W o algoritmo descrito em 1, em que α é a chamada taxa de aprendizado (*learning rate*) e deve ser fornecida pelo programador. $\Delta F(W)$ se refere à variação do valor da função de custo entre duas iterações do algoritmo e $\nabla_W F$ se refere ao gradiente da função de custo. A Figura 2.3 ilustra exemplos de funções de custo se estabilizando para duas e três dimensões.

Algoritmo 1 Gradient descent

```
W ← W_inicial
i ← 0
while ΔF(W) > threshold ou i < max_iter do
    W^{i+1} ← W^i − α∇_W F
    i++
end while
```



(a) Função de custo para uma só variável

(b) Exemplo para duas variáveis (Fonte[3])

Figura 2.3: Alguns exemplos de gradiente descendente

A definição do limiar de variação da função de custo, número máximo de iterações e *learning rate* são feitas pelo programador, usando métodos como, por exemplo, *cross validation*, que será visto mais a frente no texto. Especificamente, a *learning rate* influencia na maneira como a função de custo irá se estabilizar, conforme ilustrado na Figura 2.4, caso ela seja muito grande, os pontos de mínimo serão muitas vezes perdidos devido a saltos de valores das variáveis muito grandes, caso ela seja muito pequena, a convergência será lenta e existe a possibilidade maior de a otimização ficar presa em pontos de mínimo locais, sem que sejam alcançados os mínimos globais (ou mínimos locais de menor valor).

Existem variações que utilizam derivada de segunda ordem (método de Newton) para maior precisão na definição das direções de variação, há também métodos que têm em vista a eficiência, como o gradiente descendente estocástico, que será visto mais a frente.

Preprocessamento

Em alguns casos, a escala das *features* pode influenciar em sua prioridade para o sistema. Uma *feature* variando 0 a 2000 nos exemplos tem mais peso na saída do que uma que varia entre 0 e 1. Além disso, devido à limitada capacidade de precisão dos computadores, somar números muito grandes a números muito pequenos pode, por vezes, introduzir erros.

A solução é fazer um preprocessamento visando igualar as escalas. Usualmente, para uma *feature*, são analisados todos os seus valores nos exemplos em questão e é feita uma normalização da forma

$$X_{proc} = \frac{X - \mu}{\sigma} \tag{2.5}$$

sendo μ a média e σ o desvio padrão, de forma que os dados preprocessados passam a ter média 0 e desvio padrão 1. Para imagens, basta fazer

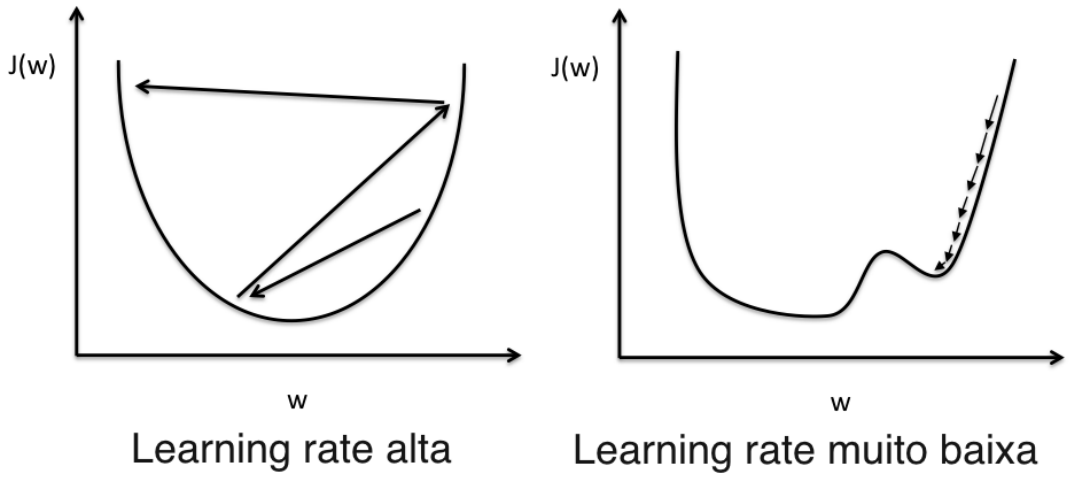


Figura 2.4: Efeito da learning rate (Fonte[1])

$$x_{proc} = \frac{x - 128}{128} \tag{2.6}$$

e o mesmo efeito é alcançado.

Inicialização de pesos

O *gradient descent* não garante a convergência ao valor mínimo da função, por isso, a inicialização dos pesos deve ser feita com cuidado, de forma a tentar aumentar a chance de parada em um mínimo local suficientemente bom. Por vezes, são feitas várias inicializações e a com melhor valor final de função de custo é mantida. Um conselho é inicializar os pesos aleatoriamente, pegando os valores de uma distribuição normal com média 0 e variância pequena.

Avaliando o desempenho

Para avaliar a performance, podemos fazer

$$acc = \frac{N(F(x_i) = y_i)}{N_{total}} \tag{2.7}$$

sendo $F(x_i)$ o resultado do classificador para o exemplo x_i e N significando o número de exemplos. Essa medida, se realizada no *training set*, é chamada de *training error*, porém, ela não necessariamente é precisa, pois a função de custo foi montada de forma a resolver o problema para aqueles exemplos específicos, não havendo garantia de generalização para

exemplos não vistos pelo classificador. O fenômeno em que o classificador fica "viciado" no *training set* e perde sua capacidade de generalização é chamado de *overfitting*.

Uma forma de avaliar se há *overfitting* é dividir os dados em *training set* e *test set*, sendo que o *test set* não deve ser fornecido em momento algum para o classificador: nem durante o treinamento, nem durante a escolha de parâmetros, ele apenas deve ser usado na avaliação final do classificador. Se o *training error* for baixo e o *test error* for alto, há *overfitting*. A Figura 2.5 ilustra na coluna à esquerda o fenômeno de *underfitting*, em que o sistema é simples demais para prever as entradas do modelo com precisão, ao centro é representado um sistema que, apesar de errar para algumas entradas, de maneira geral realiza as previsões de maneira correta e as generaliza bem para dados fora do *training set*, à direita, é representado o *overfitting*, em que há perda de generalização. A linha de cima representa regressão, enquanto a de baixo representa o problema da classificação.

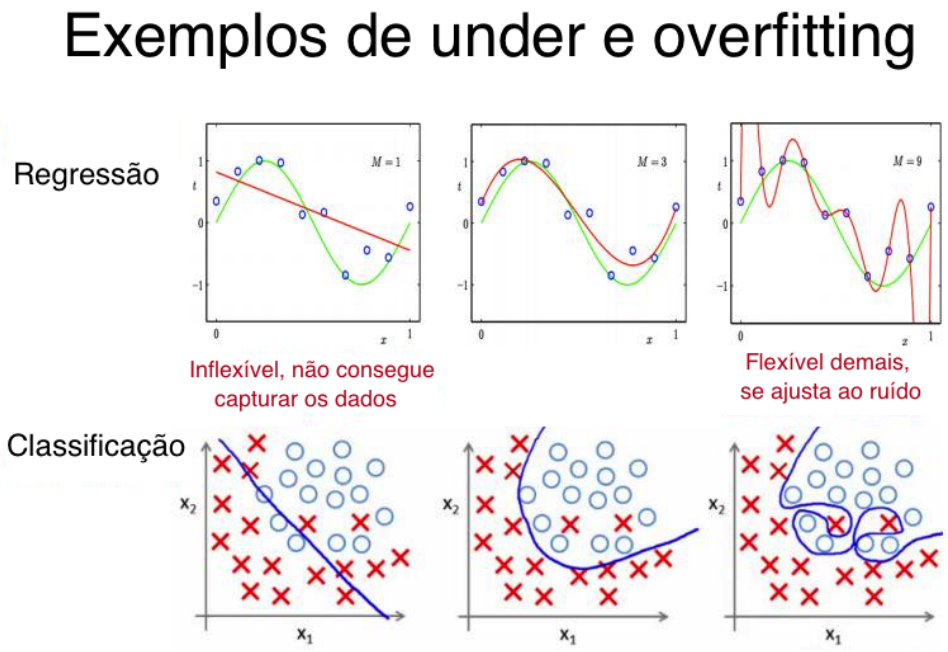


Figura 2.5: Efeitos do under e overfitting (Fonte[2])

Cross validation

Para definição de parâmetros, tais como a *learning rate* citada anteriormente, normalmente é usado *cross validation*[30], que consiste em separar novamente o *training set* inicial, dessa vez em um *training set* menor e um *cross validation set*. Para cada possibilidade de parâmetros w_i a ser utilizada, podemos fazer:

Algoritmo 2 Cross validation

```
min_err ← ∞
min_err_w ← nil
for w_i in W_possiveis do
    Executa o  classificador
    err ← cross_validation_error(w_i)
    if err < min_err then
        min_err ← err
        min_err_w ← w_i
    end if
end for
```

O uso do *cross validation set* para isso em vez do *test set* se faz necessário porque esse *set* também influencia diretamente o classificador. O *test set* deve ser usado apenas para avaliação final, nunca deve ser fornecido ao classificador. O motivo de o *training set* não ser usado diretamente para a melhora dos parâmetros tem a ver com o fato de que se deseja verificar a capacidade de generalização do sistema, a qual não pode ser inferida se é usado apenas o *training set*. Em alguns casos, após encontrar o espaço de melhores parâmetros no *validation set*, o modelo é retreinado, mas agora usando todo o conjunto *training+validation sets*.

Gradiente Descendente Estocástico

Gradiente descendente, embora seja um bom método de otimização, peca por sua baixa eficiência para computar os gradientes, já que, devido ao alto número de exemplos, a função de custo se torna complicada. Uma alternativa mais eficiente é o chamado gradiente descendente estocástico[31][32], que consiste em utilizar um *subset* do *training set* e montar a função de custo apenas com ele. A direção de variação do modelo simplificado é utilizada apenas uma vez, e, em seguida, é escolhido um outro *subset* e o procedimento é realizado novamente. O gradiente para o modelo simplificado pode ser calculado mais rapidamente, porém, esse cálculo deve ser repetido várias vezes. É interessante que os passos do gradiente descendente estocástico sejam mais curtos do que no modelo tradicional, ou seja, com uma *learning rate* menor, já que a direção de variação pode não ser a ideal. Apesar de a *learning rate* ser menor, o gradiente descendente estocástico é mais eficiente. A Figura 2.6 ilustra as curvas de nível para a função de custo e a otimização fazendo uso de gradiente descendente estocástico.

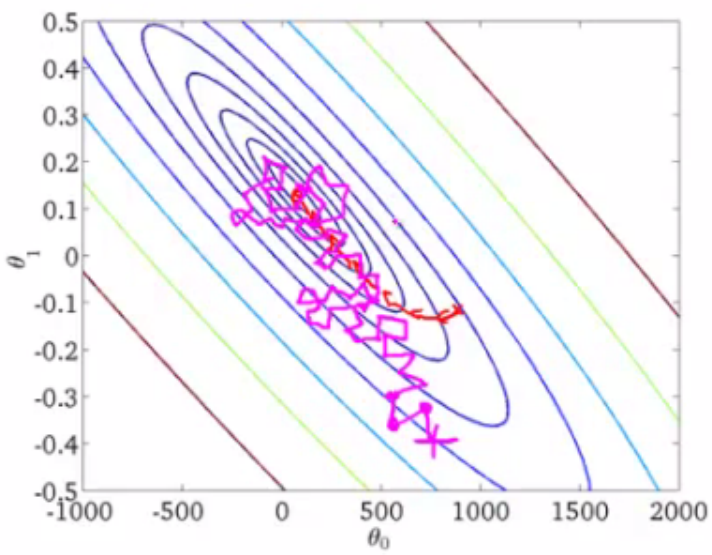


Figura 2.6: Gradiente descendente clássico (vermelho) versus estocástico (rosa) (Fonte[3])

Momentum e *learning rate decay*

Uma boa prática ao realizar gradiente descendente estocástico consiste em fazer uso de todas as direções (gradientes) computadas previamente, de forma que a direção atual passa a funcionar de maneira similar a uma média delas. Essa direção é chamada *momentum* e é calculada da seguinte forma, em que γ é usualmente um peso entre 0 e 1 e M passa a ser usado para fazer a atualização dos pesos no gradiente descendente estocástico.

$$M^{(i+1)} \leftarrow \gamma M^{(i)} + (1-\gamma)\nabla F \tag{2.8}$$

Além disso, como a precisão das direções é menor para o gradiente descendente estocástico, os passos tomados deverão ser menores. Ou seja, a *learning rate* deve ser diminuída. Usualmente isso é feito como uma função exponencial do número de iterações. Na ferramenta Tensorflow, que é utilizada no projeto, esse decaimento é dado de acordo com

$$decayed\_learning\_rate = learning\_rate \times decay\_rate^{(\frac{global\_step}{decay\_steps})} \tag{2.9}$$

Em que *decay_rate* e *decay_steps* são parâmetros e *global_step* se refere à iteração do treinamento.

Resumo

A Figura 2.7 resume o que foi falado anteriormente a respeito de regressão logística.

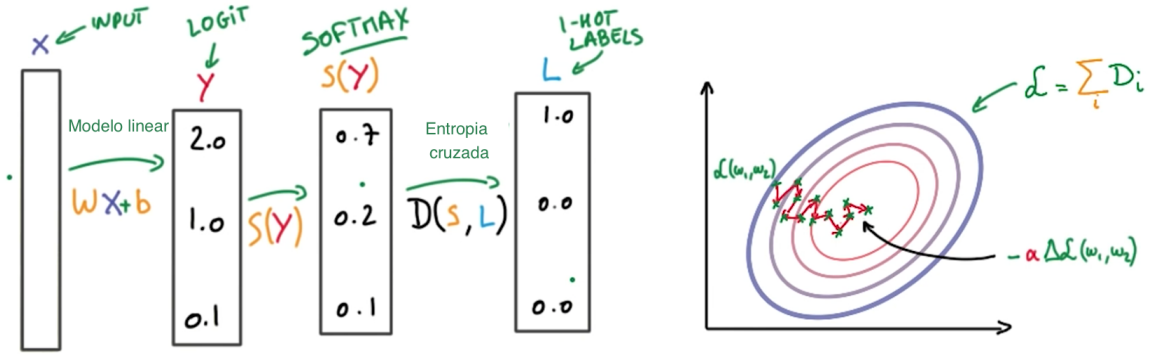


Figura 2.7: Resumo do modelo aprendido (Fonte[3])

Inicialmente são colhidos dados que, a partir de uma operação linear são mapeados para saídas (*logits*) que, através da Softmax, são remapeados para valores no intervalo $(0, 1)$ cuja soma é 1. Através da entropia cruzada entre essas saídas da Softmax e vetores como na Equação 2.3 é gerada uma função de custo, que é otimizada através do gradiente descendente.

2.1.3 *Deep Learning Neural Networks*

Embora esse tópico, da maneira como será descrito, se enquadre também em aprendizado supervisionado, foi criada uma seção à parte para maior organização. Os modelos lineares, embora relativamente simples, ficam limitados a um número $L \times (p + 1)$ de parâmetros, sendo L o número de classes e p o número de *features*. A aplicação pode requerer uma maior complexidade.

Uma solução possível seria utilizar combinações de grau maior das *features*, como por exemplo ao invés de $p_1x_1 + p_2x_2 + p_3$ usarmos funções de mais alta ordem tais como $p_1x_1 + p_2x_2 + p_3x_1x_2 + p_4x_1^2 + p_5x_2^2 + p_6$. Isso equivaleria, no sistema descrito, a partir do vetor x, adicionar coordenadas que envolvam produtos das já presentes, como x_1^2 ou x_1x_2. O problema dessa estratégia é que, para graus elevados, o número de parâmetros fica exageradamente alto. Existem maneiras de contornar esse problema, como o uso de *kernels*. Nesta seção, não será abordado esse tipo de modelo, a complexidade será aumentada através do uso de redes neurais.

Essas redes foram criadas com base no sistema nervoso humano e as interações entre seus neurônios. Anteriormente, vimos vetores de entrada que se baseiam em LBP e *optical flow*. Esse tipo de construção foi criada por humanos. O uso de redes neurais tenta automatizar esse tipo de construção. A entrada poderia ser, por exemplo, os valores

dos componentes RGB de todos os *pixels* de uma imagem e os valores nos neurônios refletiriam as construções automatizadas desejadas.

ReLU

A sigla ReLU significa *Rectified Linear Units*. Trata-se de uma função que zera valores negativos e mantém valores positivos, conforme

$$y = \begin{cases} 0, & \text{se } x \leq 0 \\ x, & \text{se } x > 0 \end{cases} \tag{2.10}$$

Seu gráfico é ilustrado na Figura 2.8.

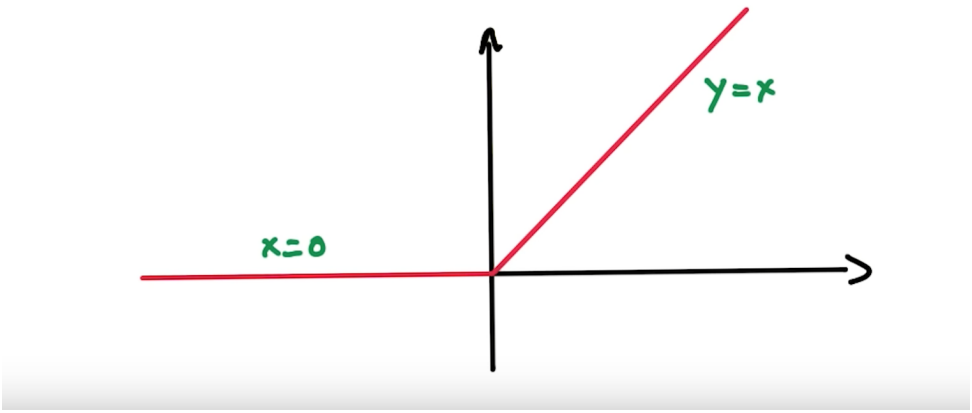


Figura 2.8: Gráfico ReLU (Fonte[3])

Em 2011[33], foi mostrado que o uso de ReLUs como não linearidades permite o treinamento de redes com várias camadas, como será visto a seguir, anteriormente, era necessário um pré-treinamento da rede usando aprendizado não-supervisionado. ReLUs permitem treinamento mais rápido e efetivo de redes profundas em *datasets* grandes e complexos em relação às funções que os antecederam, como a sigmoide.

Deep Learning Neural Networks de duas camadas

Para gerar uma *deep learning neural network* simples, de duas camadas, a entrada x é mapeada usando $Ax + b$, porém, o vetor de saída desse mapeamento não precisará ser de tamanho $L \times 1$. Em vez disso, ele é de tamanho $H \times 1$, sendo que H é o tamanho da segunda camada, também chamada *hidden layer*. Esse tamanho é definido pelo usuário. Após a execução dessa operação. As entradas do vetor recém-gerado passam por ReLUs. O vetor resultante dessa operação é usado como entrada para mais uma operação $Ax + b$, dessa vez de forma a mapeá-lo para a saída (o vetor resultante é $L \times 1$). Para estender

o modelo para várias camadas, basta repetir a *hidden layer* quantas vezes for desejado e finalizar mapeando para a saída.

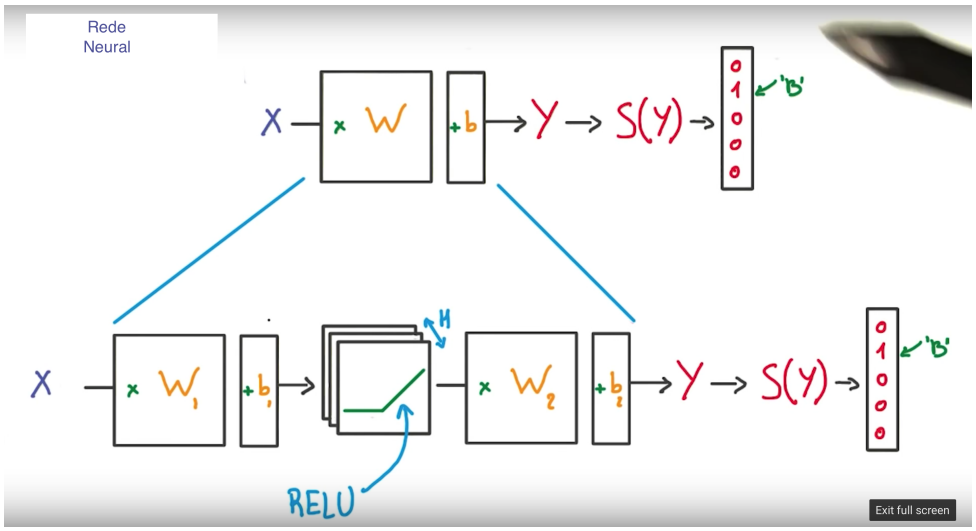


Figura 2.9: Estendendo o modelo linear para uma *deep neural network*(Fonte[3])

AFigura 2.9 ilustra o modelo explicado para o caso de duas camadas. Porém, o modelo pode ser construído com um número muito maior de camadas. Esse modelo é conhecido como *fully connected* e é a forma mais simples de rede profunda.

Backpropagation

Para calcular os gradientes, de forma a otimizar a função de custo ao realizar SGD (a função agora é mais complexa, já que a rede é maior e envole não-linearidades), é utilizado o algoritmo de *backpropagation*[34]. Esse algoritmo parte de uma derivação que toma como base a regra da cadeia para calcular valores intermediários que são combinados para cálcular os gradientes e usá-los na otimização.

Deep versus *wide*

Em problemas de Inteligência Artificial, muitas vezes desejamos tornar a rede neural utilizada mais complexa, o que permite maior precisão para problemas de difícil resolução. Uma estratégia para atingir esse objetivo consiste em aumentar o número de neurônios nas camadas intermediárias, ou seja, aumentar o número de linhas da matriz A e o número de ReLUs. Uma segunda abordagem, consiste em aumentar o número de camadas. Poderíamos, por exemplo, utilizar uma *Deep Learning Neural Network* com muitas camadas, em vez de apenas duas. Essas duas abordagens são ilustradas na Figura 2.10.

A segunda abordagem é superior, primeiro por ser mais eficiente no aprendizado dos parâmetros, segundo pelo fato de que os neurônios das camadas mais a frente passam a

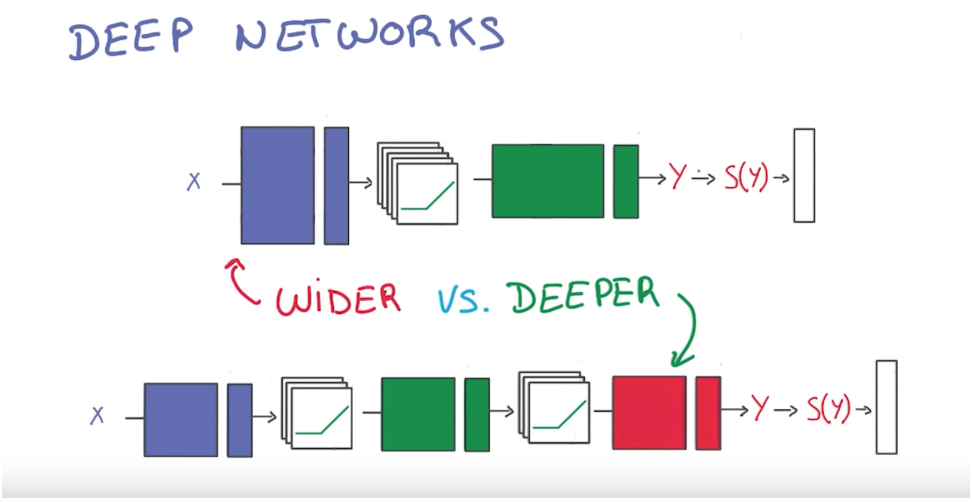


Figura 2.10: Deep vs wide (Fonte[3])

representar estruturas mais complexas, como por exemplo, os filtros que antes deveriam ser implementados manualmente. A Figura 2.11 mostra um exemplo ilustrativo dessas representações.

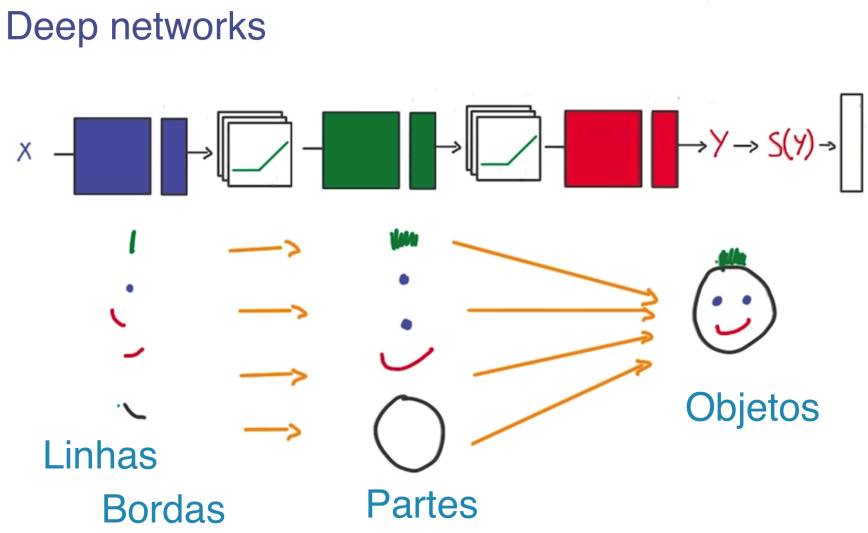


Figura 2.11: Estruturas aprendidas pelos neurônios (Fonte[3])

Nesse caso, podemos observar que as primeiras camadas diferenciam estruturas mais simples e, à medida que a complexidade do modelo vai aumentando, essas estruturas mais simples vão sendo combinadas e a rede vai aprendendo a diferenciar estruturas cada vez mais complexas, culminando com a diferenciação dos objetos desejados.

2.1.4 Controle de *Overfitting*

O *overfitting*, conforme explicado anteriormente, é o fenômeno em que o modelo fica viciado no *training set* e perde capacidade de previsão para imagens que estão fora dele. Isso geralmente ocorre quando a complexidade do modelo utilizado, excede a complexidade do *dataset* utilizado para treino. Nos problemas de aprendizado supervisionado, seria ideal se tivéssemos modelos com o número de parâmetros certo para o *dataset* em questão, já que muitos parâmetros podem causar *overfitting* e poucos parâmatros podem causar *underfitting*. Na prática, é difícil saber o número de parâmetros correto para cada *dataset*. Por isso, normalmente a rede é montada com mais parâmetros do que o necessário e são utilizados métodos de controle de *overfitting*.

Terminar o treinamento prematuramente

O programador deve atentar-se ao *training error* e ao *validation error* ao longo das iterações do SGD. Se em algum ponto tivermos o *training error* diminuindo, mas sem alteração do *validation error*, é possível que esteja havendo *overfitting*. Se isso ocorrer, pode ser necessário que o programador, ao perceber isso, termine o treinamento antes do número de iterações estipulado. Esse fenômeno pode ser visualizado na Figura 2.12.

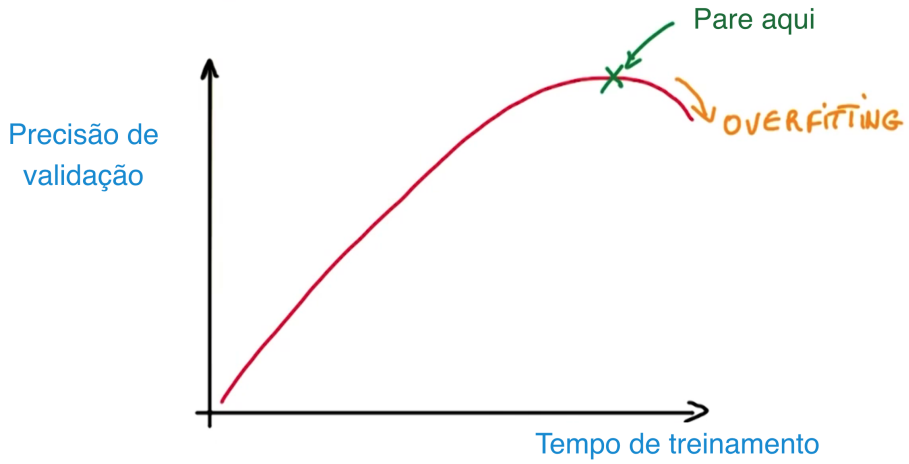


Figura 2.12: Momento correto de terminar o treinamento (Fonte[3])

Regularization

Uma forma de reduzir o número de parâmetros livres do modelo e, por consequência, sua complexidade é alterar diretamente a função de custo. Essa alteração é feita com a adição de um fator à função de custo, que corresponde a um parâmetro β multiplicado,

normalmente, pelo módulo do vetor de parâmetros do modelo, ω (o quadrado do módulo, por questões de eficiência). No caso de uma rede neural, todos os pesos, que correspondem às entradas das matrizes A citadas anteriormente, são colocados nesse vetor, os *biases* (fator b na fórmula $Ax + b$) não são incluídos no fator de regularização conforme

$$F(\omega) = \frac{1}{N}\sum_{j=1}^{N} D(f(x_j, \omega), Y_j) + \frac{\beta\|\omega\|_2^2}{2} \tag{2.11}$$

.

em que a primeira parte da equação é similar à Equação 2.4, β é o fator de regularização e ω é um vetor que contém os parâmetros do modelo, ressaltando que, apesar da notação simplificada, o bias não é incluído no módulo de ω.

O efeito da introdução desse fator é uma penalização de soluções cujos parâmetros tenham módulo muito elevado, caso os parâmetros sejam muito grandes, as operações na rede podem ultrapassar o valor máximo que pode ser armazenado no processador.

Deve-se, contudo, se atentar ao fato de que um valor de regularização muito grande pode transformar a otimização da função de custo numa minimização apenas dos módulos dos parâmetros. Nesse caso, as iterações do SGD mostram uma diminuição no valor da função de custo sem que haja melhora na precisão do sistema (*training error* e *validation error* permanecem praticamente constantes)

Dropout

Outra técnica recente que vem sendo muito usada para evitar *overfitting* é o chamado *dropout*[35]. Essa técnica consiste em, durante a fase de treinamento, zerar metade das ativações a cada iteração de SGD. No modelo de RNA com ReLU, por exemplo, isso corresponderia a, na hora de montar a função de custo para o *subset* daquela iteração, alterar a saída $f(x_j, \omega)$. Essa alteração é realizada tomando como entrada o vetor $ReLU(Ax_j + b)$ de determinada camada e zerando algumas entradas aleatoriamente (a probabilidade de isso ocorrer é um parâmetro a ser definido). Isso é realizado para várias camadas, sendo que deve ser feito apenas no treinamento.

Esse método força a rede a se tornar redundante, pois a otimização passa a ser feita sem depender da existência de nenhum peso específico. É como se a decisão da rede fosse agora um consenso de várias redes menores. A Figura 2.13 ilustra as operações citadas acima e a Figura 2.14 mostra a técnica como consenso de redes menores, em que as várias sub redes geradas a cada passo do SGD operam de maneira a otimizar o resultado com uma quantidade de entradas menor. Na etapa de testes, essa limitação é removida.

Um cuidado deve ser tomado ao se usar essa técnica. Tomando como exemplo uma *dropout_rate* de 0.5, a rede foi treinada para operar com apenas 50% das ativações

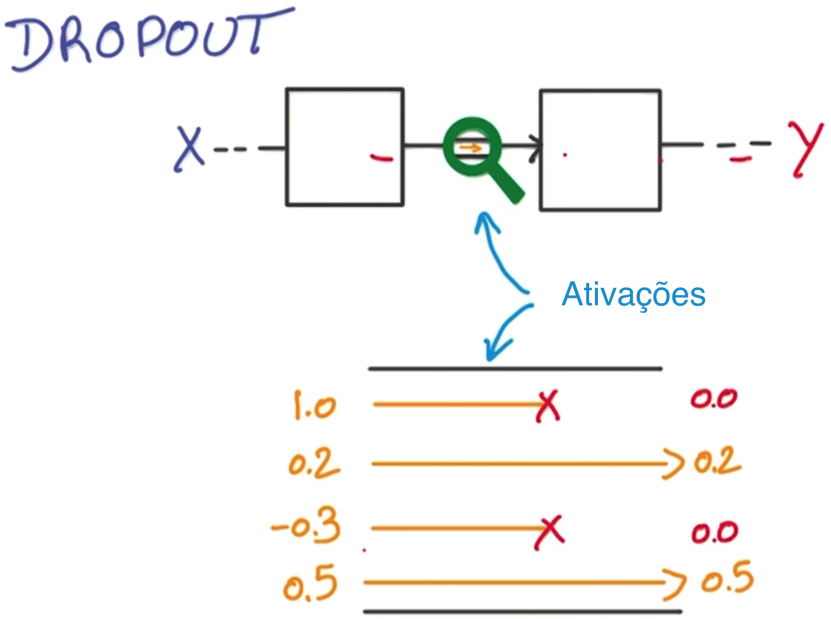


Figura 2.13: *Dropout* (Fonte[3])

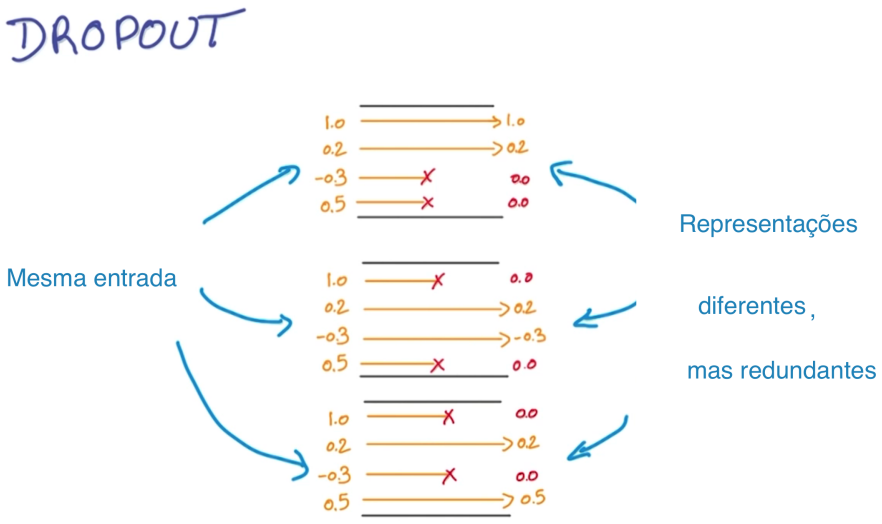


Figura 2.14: *Dropout* como consenso (Fonte[3])

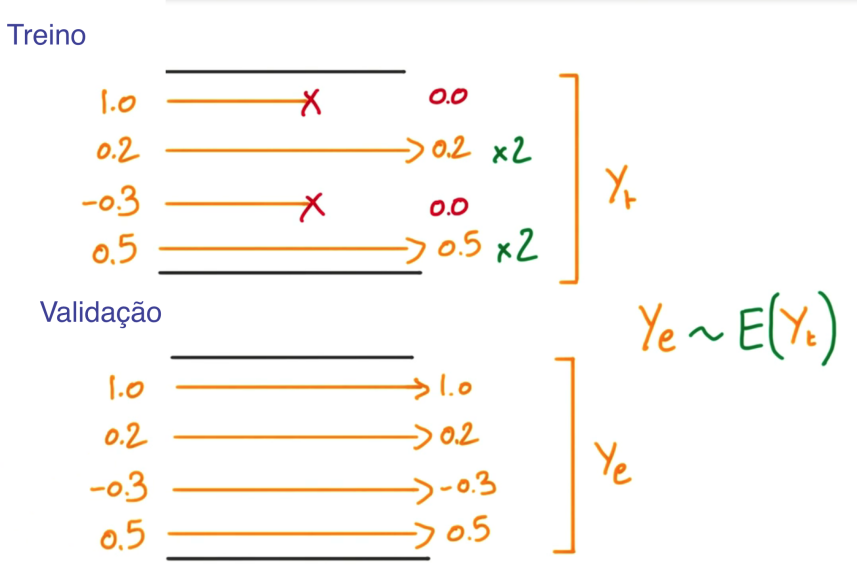


Figura 2.15: *Dropout* com ajuste de escala (Fonte[3])

(lembrando que a ativação corresponde à saída de determinada camada), ou seja, na fase de teste e validação, onde o *dropout* não será mais usado, as saídas que dependem dessas ativações serão dobradas, já que é como se duas redes criadas para gerar a saída correta com apenas metade da entrada estivessem sendo combinadas, resultando no dobro da saída.

Uma solução para esse problema poderia ser, por exemplo, dividir todas as ativações da camada em questão por 2 na validação e teste. Dessa forma, o efeito das saídas dobradas é negado. Essa técnica é ilustrada na Figura 2.15. Outra solução que tem o mesmo efeito que essa e é usada como padrão na função *dropout* da ferramenta TensorFlow, é multiplicar as entradas não zeradas por 2 durante o treinamento. Isso tira a necessidade de fazer qualquer alteração nas fases de validação e teste.

Dropout é uma técnica bastante poderosa, se ela não estiver produzindo resultados muito melhores, é um indicativo de que o programador deve usar uma rede maior.

2.2 *Convolutional Neural Networks*

O modelo de redes neurais visto aqui, conhecido como *fully connected layers*, é muito poderoso. Porém, a precisão dos classificadores pode ser melhorada ainda mais se utilizarmos algum tipo de indicativo da natureza dos dados de entrada. Em imagens, por exemplo, a proximidade dos *pixels* pode ser usada para adicionar informações relevantes à rede. Além disso, seria interessante se os dados pudessem ser invariantes a translação e escala, já que, para imagens, essas operações normalmente não alteram a classe em

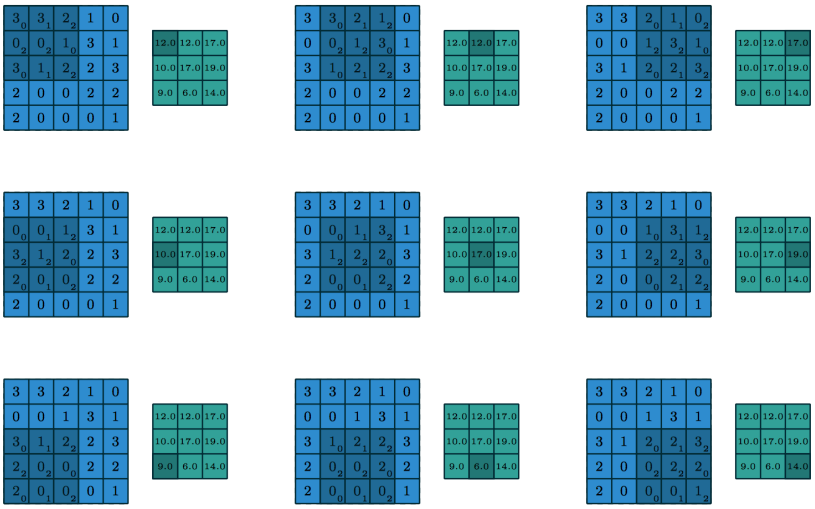


Figura 2.16: Convolução passo a passo (Fonte[4])

questão ou sequer trazem informação nova à rede. Dessa forma, poderíamos, por exemplo, tratar "objeto no canto superior esquerdo" da mesma forma que tratamos "objeto no canto inferior direito". Isso tudo é levado em conta por um outro modelo de redes neurais, as Convolutional Neural Networks (CNN)[36], também chamadas de Convnets, através dos artifícios de *weight sharing* e *pooling*.

As CNNs se baseiam em receber a entrada em seu formato original (uma matriz RGB ou *grayscale*, por exemplo) e mapeá-la para a saída por meio de convoluções (correlações na verdade, mas chamadas de convolução por definição). Podemos, por exemplo, considerar uma entrada *grayscale* 5×5 e um *kernel* 3×3, sendo que o reposicionamento do *kernel* não pula linhas ou colunas entre iterações consecutivas da convolução. Nesse caso, a convolução fica como ilustrado na Figura 2.16, em que a matriz da esquerda representa a entrada da convolução, com a submatriz referente ao passo atual destacada. A matriz da direita representa o resultado da convolução, com o resultado do passo atual destacado na matriz, os pesos usados na convolução aparecem como números pequenos na matriz da esquerda. Na prática, são usados *kernels* com profundidade maior do que 1, o que significa, para o exemplo anterior, que n *kernels* 3×3 são usados e as matrizes de saída são concatenadas (saída ficaria $3 \times 3 \times n$ no exemplo). A Figura 2.17 ilustra um exemplo de convolução com mais de um *kernel*

Após cada iteração da convolução, é somado um número à entrada correspondente, o *bias*. No exemplo ilustrado em 2.16, isso corresponderia a somar o número a cada entrada da matriz de saída. Se a imagem de entrada tivesse k canais, cada um dos n *kernels* deveria, no exemplo, ser $3 \times 3 \times k$ e as convoluções poderiam ser representadas por cubos,

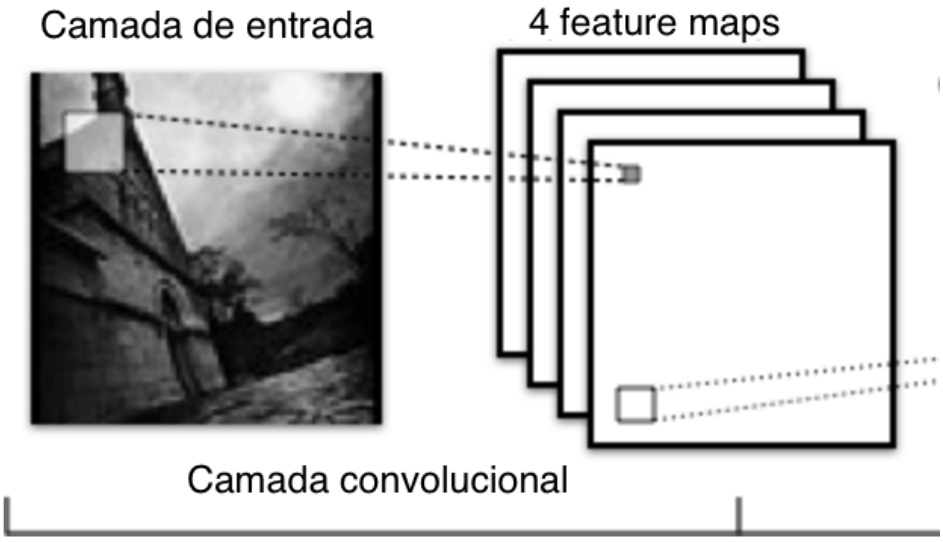


Figura 2.17: Convolução com *kernel* com profundidade maior que 1 (Fonte[5])

como em 2.18. Não há iterações que envolvam translações em eixos além da altura e largura, já que aqui estamos tratando de convoluções 2d.

Voltando para o exemplo de objetos posicionados em locais diferentes da imagem, em *fully-connected*, "canto superior esquerdo" e "canto inferior direito" eram tratados por parâmetros diferentes. Em CNNs, no entanto, o *kernel* de convolução e bias são compartilhados por todos os *pixels*. Isso faz com que esses parâmetros tentem otimizar todas as posições de uma só vez.

Por fim, como a saída da camada de convolução ainda tem formato similar a uma imagem, ela deve, em algum momento, ser convertida em vetor para ser mapeada para a saída através da função $softmax(Ax + b)$. A Figura 2.19 ilustra o esquemático resumido de uma CNN.

Quando se fala em convolução, normalmente são usados alguns termos referentes a partes específicas da operação, os principais são:

- *Stride*: quantidade de linhas ou colunas que devem ser puladas a cada reposicionamento do *kernel*
- *Padding*: adição de zeros às bordas da imagem para que o tamanho da saída possa ser alterado. A Figura 2.20 mostra um exemplo de adição de zeros para manipular o tamanho da saída. Nesse exemplo, a imagem de baixo é a entrada da convolução, a de cima é a saída e o *padding* é representado em linhas pontilhadas.
- *Patch*: o mesmo que *kernel*

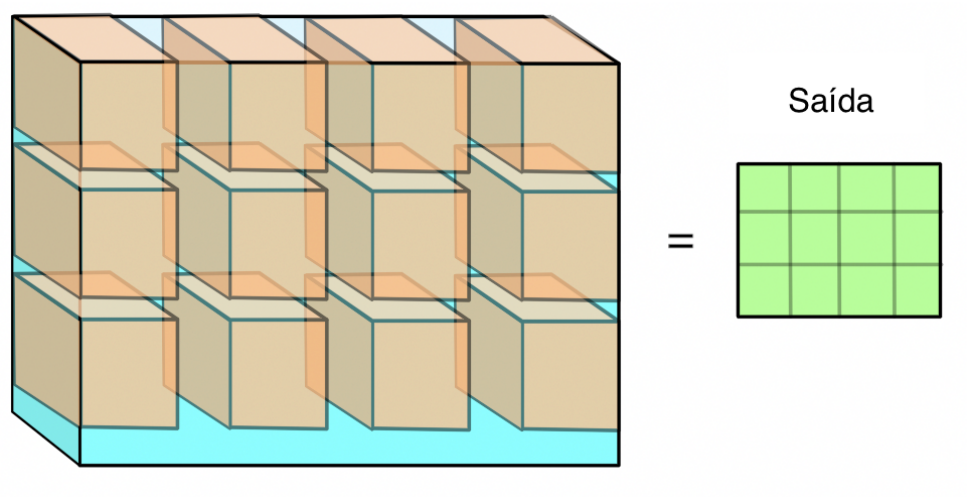


Figura 2.18: Convolução com imagem de entrada tendo múltiplos canais (Fonte[6])

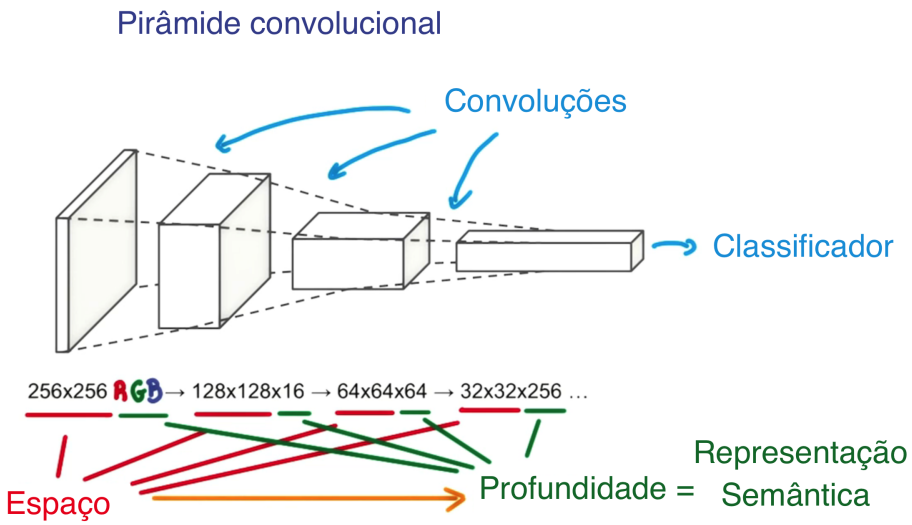


Figura 2.19: Esquemático completo de Convnet (Fonte[3])

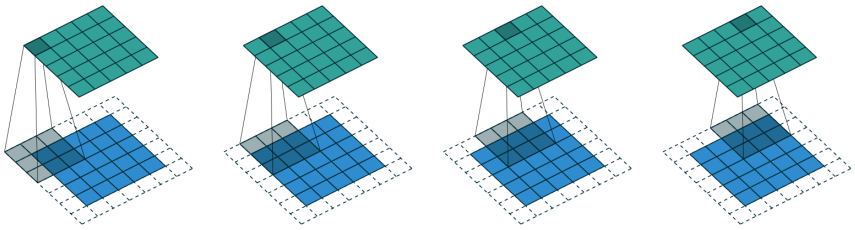


Figura 2.20: Convolução com *padding* (Fonte[4])

- *Feature map*: cada matriz da imagem de entrada (exemplo: a matriz R de uma imagem RGB) ou de alguma entrada de passo intermediário na rede
- *Zero padding*: não há *padding*
- *Half/same padding*: *padding* que faz a saída ter a mesma largura e a mesma altura da entrada

2.2.1 *Pooling*

Além do já citado mecanismo de *weight sharing*, para conseguir invariância à translação, devemos ter múltiplas camadas de convolução, cada uma delas diminuindo o tamanho da imagem e, de preferência, apresentando *kernels* com profundidade grande. Dessa forma, mais cedo ou mais tarde os parâmetros em locais diferentes acabam sendo agrupados e a invariância a escala é atingida. A maneira de reduzir a largura e altura que vimos até agora consiste em utilizar *stride* maior que 1 a cada convolução, porém, essa abordagem descarta boa parte da informação, já que pula colunas das matrizes.

Outra maneira de obter o mesmo efeito é o chamado *pooling*, que consiste em aplicar uma função a grupos $p \times p$ de valores para cada dimensão da entrada (cada uma das múltiplas matrizes geradas pelo *kernel*). Normalmente essa função consiste em tomar o máximo desses valores ou a média deles. Para que o *pooling* funcione, deve ser escolhido um *stride* para ele também, caso contrário, o tamanho da imagem não será diminuído. A Figura 2.21 ilustra um exemplo de convolução com *pooling* e a Figura 2.22 ilustra exemplos de redes de artigos científicos que são baseadas em *pooling*.

2.2.2 Convolução 1-por-1

Podemos visualizar uma iteração de uma convolução com *kernel* de tamanho $n \times n \times canais\_entrada \times canais\_saida$ como um mini-classificador linear. Basta imaginar o cubo de entrada como um vetor coluna x, cada unidade $n \times n \times canais\_entrada$ do

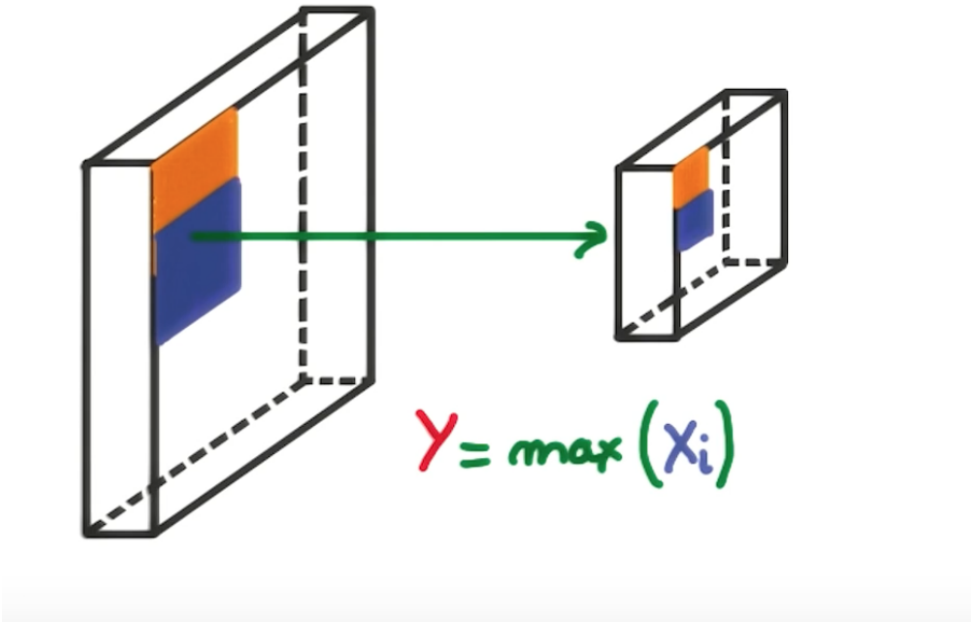


Figura 2.21: Convolução com *max pooling* (Fonte[3])

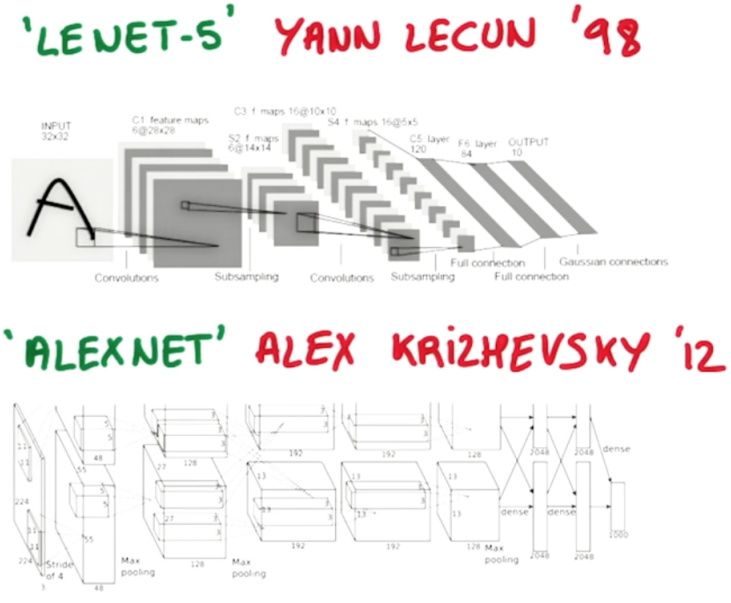


Figura 2.22: Redes que usam *pooling* (Fonte[3])

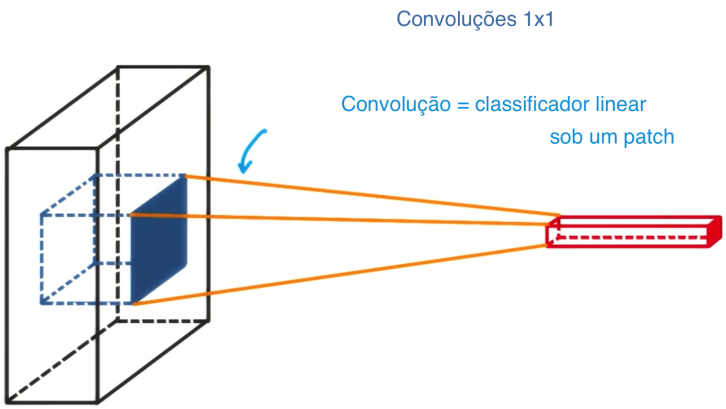


Figura 2.23: Convolução normal (Fonte[3])

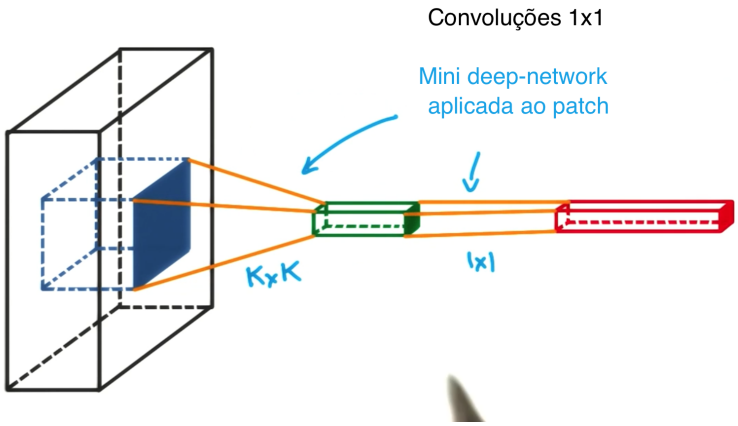


Figura 2.24: Convolução 1×1 (Fonte[3])

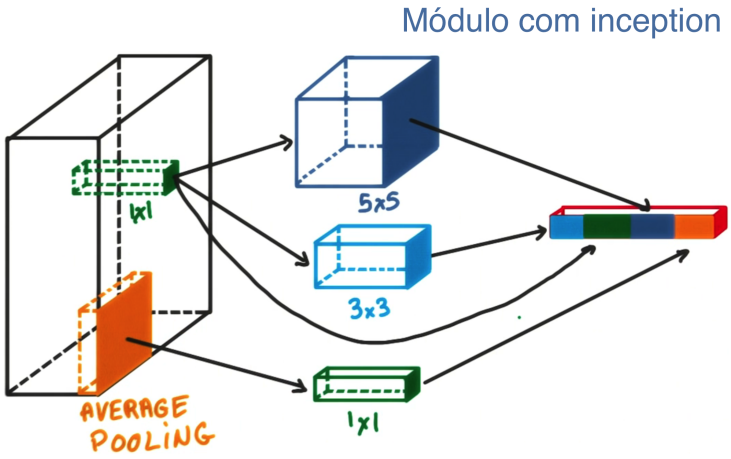
kernel como uma linha da matriz A e cada uma das *canais_saida* unidades de bias como o vetor coluna b, conforme ilustrado na Figura 2.23. Uma forma simples de transformar esse "mini-classificador" em uma "mini-rede neural" seria com a adição de uma convolução 1×1 após a convolução $n \times n$, conforme ilustrado na Figura 2.24. O efeito obtido é o de agrupar classificadores lineares no treinamento (a convolução $n \times n$ e a 1×1 estão sendo agrupadas) a custo baixo.

2.2.3 *Inception networks*

Muitas vezes, a escolha do tamanho do *kernel* pode não ser fácil para o programador. Como diversos tamanhos de *kernel* são benéficos ao sistema, assim como a combinação com 1×1 *convolutions*, uma maneira utilizada para conciliar essas soluções é concatenar todas elas [7]. Para isso, os *poolings* e *strides* devem ser regulados de forma que todas as saídas

de convoluções tenham a mesma altura e a mesma largura e possam ser concatenadas de acordo.

Com reaproveitamento de parâmetros, podemos tornar o treinamento menos custoso. A convolução 1×1 que for usada como sequência da $n \times n$ pode, por exemplo, também ser usada diretamente na entrada da camada. A Figura 2.25 ilustra um módulo que faz uso dessa técnica e a Figura 2.26 ilustra parte da rede proposta em [7], em que podemos ver que o padrão aqui explicado é aplicado a uma rede maior.



[H]

Figura 2.25: Módulo convolucional com *inception*

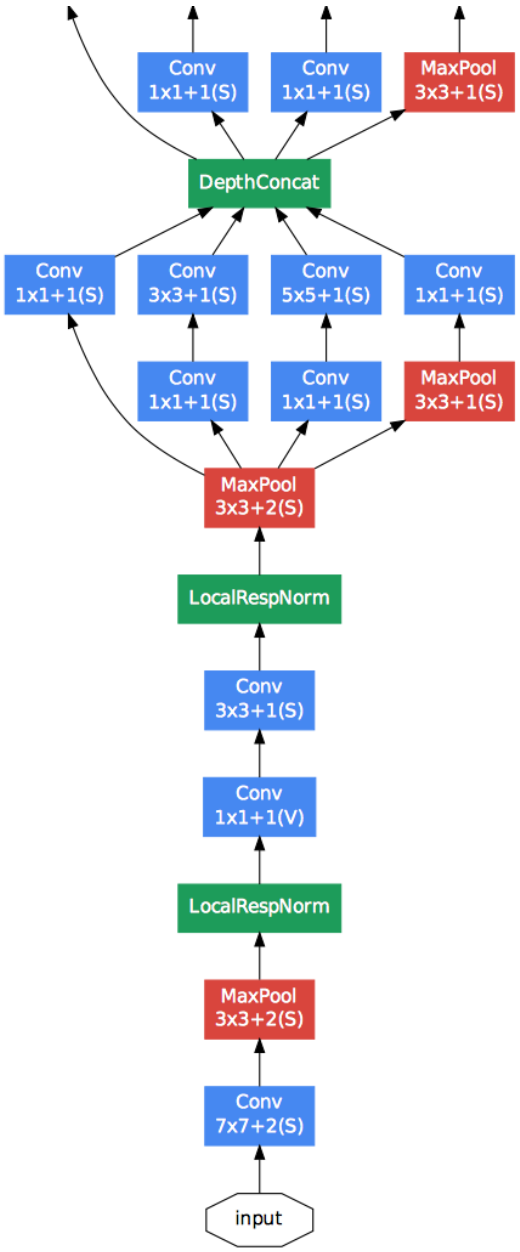


Figura 2.26: *Zoom in* da Googlenet (Fonte[7])

2.3 Técnicas mais recentes

2.3.1 Residual Networks

Atualmente, tem se popularizado o uso de redes convolucionais em Machine Learning, sendo que diversas novas arquiteturas são propostas e estudadas a cada ano. Uma das arquiteturas escolhidas para esse trabalho é a Resnet, proposta em [37] e melhorada em

[9]. Foram feitas mais melhorias nessa rede desde então, porém essas melhorias não foram exploradas neste trabalho. A ideia principal de uma Resnet é a separação da rede convolucional em blocos, sendo que, de maneira geral, os *feature maps* de entrada X_l do bloco passam por duas convoluções e a saída desse conjunto de operações é somada novamente à entrada, como ilustrado na Figura 2.27. O artigo verifica empiricamente que esse tipo de abordagem permite que a rede seja treinada com um número maior de camadas, sendo que esse era o estado da arte para o Imagenet no ano de 2016. A arquitetura geral da rede é ilustrada na Figura 2.28

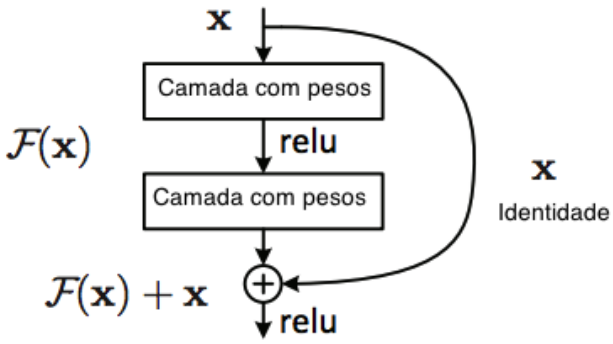
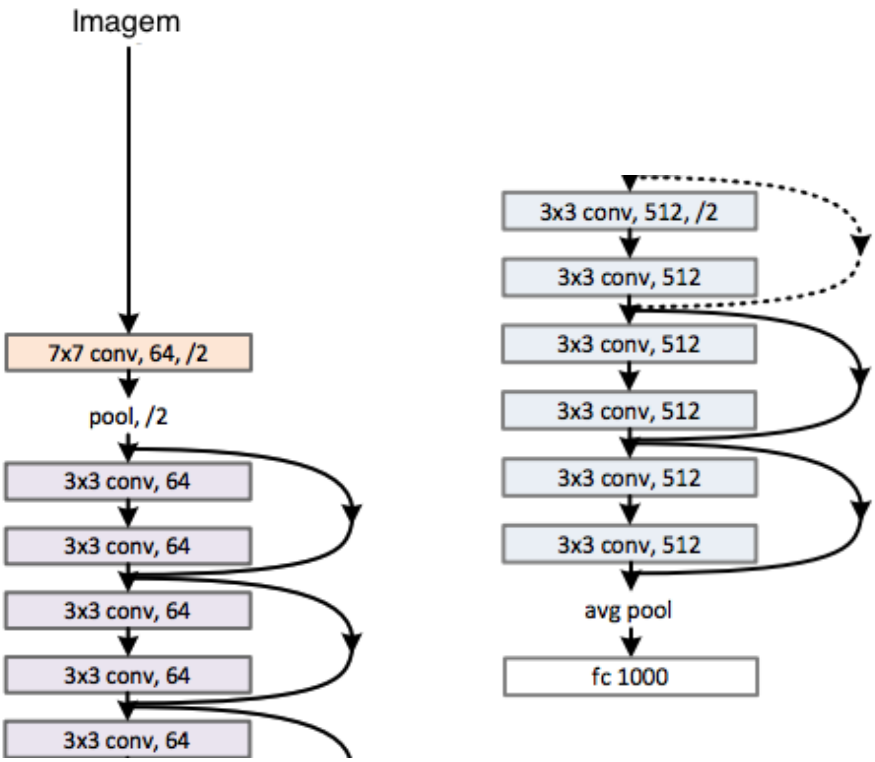


Figura 2.27: Bloco de operações na Resnet original (Fonte[8])



(a) Camadas iniciais (b) Camadas finais

Figura 2.28: Resumo da configuração de uma Resnet (Fonte[9])

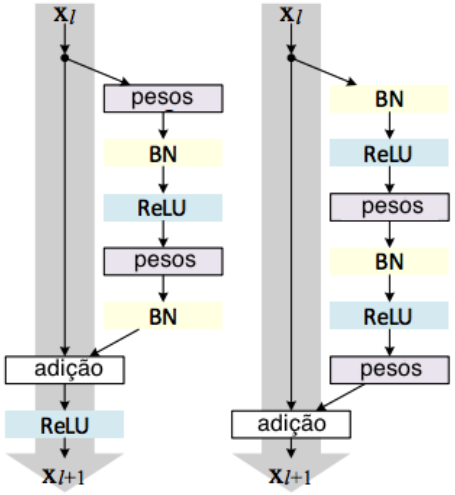


Figura 2.29: Original (direita) melhorado (esquerda) (Fonte[9])

Dessa forma, a Figura 2.28a ilustra que a imagem (dado) de entrada passa por uma camada convolucional inicial e, em seguida, blocos como os da Figura 2.27 são aplicados, sendo que o número de *feature maps* de cada bloco vai aumentando à medida que a largura e altura desses *feature maps* vão diminuindo através de *pooling* (note que as primeiras camadas começam com *feature maps* com 64 de profundidade e as últimas têm 512). Por fim, na Figura 2.28b a rede utiliza de *global average pooling* para reduzir a largura e altura dos últimos *feature maps* para 1, ficando com apenas um vetor cuja dimensão é a profundidade. Esse vetor passa por uma camada linear, sendo a saída igual a $softmax(Ax + b)$. Em [9] o autor propõe e testa uma melhoria na ordem em que as operações são executadas, como ilustrado em 2.29

2.3.2 Triplet Networks

Uma estratégia que é por vezes empregada no projeto de redes neurais consiste na utilização de fatores da funções de custo que não sejam totalmente focados na classificação, mas sim em objetivos diferentes. É o caso, por exemplo, da já mencionada L2 *regularization*, que utiliza a soma dos módulos dos parâmetros que estão sendo treinados na função de custo (multiplicados por uma constante). Isso é feito de modo a evitar que os parâmetros cresçam indefinidamente, já que seu módulo é penalizado e é uma das principais técnicas de controle de *overfitting*.

Outro exemplo é a função de custo das Triplet Networks [10]. Nesse caso, as saídas das redes são representações vetoriais dos elementos de entrada e a função de custo tem como objetivo gerar representações espaciais de forma que a norma da distância entre elementos de mesma classe seja pequena e a norma entre elementos de classes distintas

seja grande. A Figura 2.30 ilustra o funcionamento de uma rede desse tipo. A figura deve ser lida de baixo para cima e, nela, temos três exemplos, a âncora x, um exemplo da mesma classe da âncora, x_+, e um exemplo de classe diferente da âncora, x_-. Esses três exemplos são utilizados como entrada da rede e suas saídas, chamadas de *embeddings*, são comparadas para gerar a função de custo.

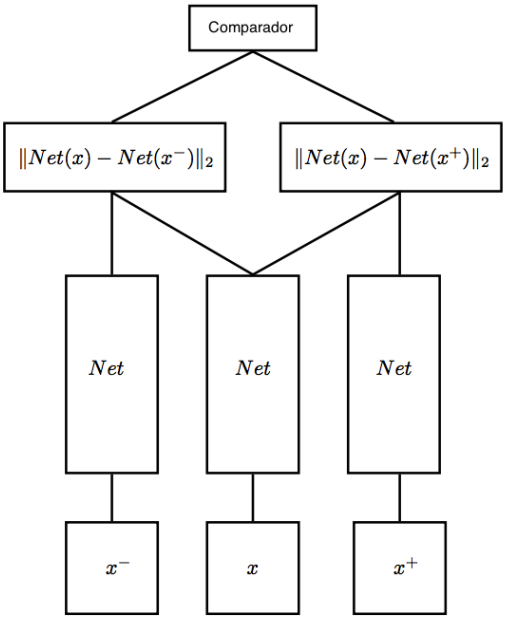


Figura 2.30: Esquemático de funcionamento da Triplet Network (Fonte[10])

A comparação de distâncias dos *embeddings* pode ser feita de diversas formas. Em [10], é aplicada a função *softmax* nas entradas, de forma a gerar

$$d_+ = \frac{e^{\|Net(x)-Net(x^+)\|_2}}{e^{\|Net(x)-Net(x^+)\|_2} + e^{\|Net(x)-Net(x^-)\|_2}} \tag{2.12}$$

e d_- de forma similar. Em seguida, a função de custo é montada a partir de fatores da forma $\|(d_+, d_- - 1)\|_2^2$.

Embora essas redes tenham sido propostas em [10], elas foram de fato popularizadas em [38], quando foram utilizadas para os objetivos de reconhecimento e clusterização de faces na rede Facenet da Google. Nesse artigo, temos o objetivo como sendo

$$\|Net(x) - Net(x^+)\|_2^2 + m < \|Net(x) - Net(x^-)\|_2^2 \tag{2.13}$$

e a margem m sendo um parâmetro a ser definido pelo usuário. A função de custo passa a ser então

$$F(x) = \sum[\|Net(x) - Net(x^+)\|_2^2 + m - \|Net(x) - Net(x^-)\|_2^2]_+ \tag{2.14}$$

Em seguida, após serem gerados os *embeddings* que separam bem os exemplos, aplica-se algum classificador simples, como um SVM, um classificador linear ou KNN.

2.3.3 NASNets

Esse modelo de rede foi obtido através de uma técnica chamada *Neural Architecture Search*, originalmente proposta em [39]. Essa técnica consiste em automaticamente determinar o modelo ótimo de rede, em vez de confiar em arquiteturas projetadas por humanos como as apresentadas anteriormente.

Em [11] essa técnica foi aplicada para determinar modelos ótimos para conjuntos de imagens. Por motivos de complexidade, foi obtida a estrutura ótima para o CIFAR-10, que é um *dataset* menor, e transferida para o Imagenet através de replicação das estruturas encontradas, obtendo os resultados que são estado da arte para esse último no momento de realização desse trabalho. A Figura 2.31 ilustra um bloco típico obtido usado nessa rede, sendo que os blocos indicados como "sep" se referem a convoluções separáveis, como descrito em [40]. A Figura 2.32 ilustra as porcentagens de erro obtidas no Imagenet para diferentes modelos de rede no TFLite, uma disponibilização móvel da ferramenta Tensorflow e a Figura 2.33 as ilustra para o TFSlim, um repositório de redes para pesquisa do Tensorflow. Em ambos os casos, podemos ver que essa rede supera todas as outras no momento de publicação desse trabalho.

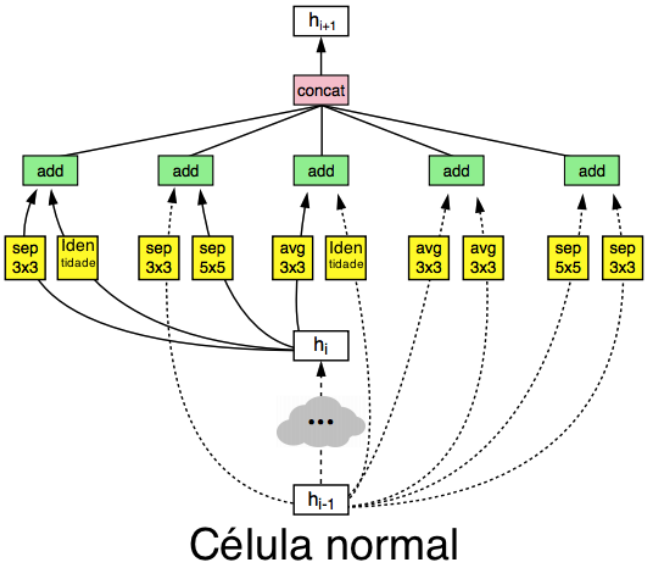


Figura 2.31: Esquemático de NASNet (Fonte[11])

Nome do modelo	Artigo e arquivo-modelo	Tamanho	Precisão Top-1	Precisão Top-5	Performance no TFLite
DenseNet	paper, tflite&pb	43.6 Mb	64.2%	85.6%	894 ms
SqueezeNet	paper, tflite&pb	5.0 Mb	49.0%	72.9%	224 ms
NASNet mobile	paper, tflite&pb	21.4 Mb	72.2%	90.6%	261 ms
NASNet large	paper, tflite&pb	355.3 Mb	82.1%	95.8%	6697 ms
ResNet_V2_50	paper, tflite&pb	102.3 Mb	68.1%	88.4%	942 ms
ResNet_V2_101	paper, tflite&pb	178.3 Mb	70.4%	89.6%	1880 ms
Inception_V3	paper, tflite&pb	95.3 Mb	76.9%	93.5%	1433 ms
Inception_V4	paper, tflite&pb	170.7 Mb	79.6%	94.6%	2986 ms
Inception_ResNet_V2	paper, tflite&pb	121.0 Mb	76.8%	93.5%	2731 ms

Figura 2.32: Tabela de implementações de redes atuais no TFLite (Fonte[12])

Nome	Código	**Checkpoint**	Precisão top-1	Precisão top-5
Inception V1	Code	inception_v1_2016_08_28.tar.gz	69.8	89.6
Inception V2	Code	inception_v2_2016_08_28.tar.gz	73.9	91.8
Inception V3	Code	inception_v3_2016_08_28.tar.gz	78.0	93.9
Inception V4	Code	inception_v4_2016_09_09.tar.gz	80.2	95.2
Inception-ResNet-v2	Code	inception_resnet_v2_2016_08_30.tar.gz	80.4	95.3
ResNet V1 50	Code	resnet_v1_50_2016_08_28.tar.gz	75.2	92.2
ResNet V1 101	Code	resnet_v1_101_2016_08_28.tar.gz	76.4	92.9
ResNet V1 152	Code	resnet_v1_152_2016_08_28.tar.gz	76.8	93.2
ResNet V2 50^	Code	resnet_v2_50_2017_04_14.tar.gz	75.6	92.8
ResNet V2 101^	Code	resnet_v2_101_2017_04_14.tar.gz	77.0	93.7
ResNet V2 152^	Code	resnet_v2_152_2017_04_14.tar.gz	77.8	94.1
ResNet V2 200	Code	TBA	79.9*	95.2*
VGG 16	Code	vgg_16_2016_08_28.tar.gz	71.5	89.8
VGG 19	Code	vgg_19_2016_08_28.tar.gz	71.1	89.8
NASNet-A_Large_331#	Code	nasnet-a_large_04_10_2017.tar.gz	82.7	96.2
PNASNet-5_Large_331	Code	pnasnet-5_large_2017_12_13.tar.gz	82.9	96.2

Figura 2.33: Tabela de implementações de redes atuais no TFSlim (Fonte[13])

2.3.4 *Transfer Learning*

O *transfer learning* [41] é uma técnica recente que trata da transferência de conhecimento de um domínio complexo para algum domínio diferente, normalmente mais simples. Essa transferência, muitas vezes, permite reaproveitamento do conhecimento do domínio mais

complexo e obtenção de bons resultados no novo domínio, mesmo que esse último possua poucos exemplos de treino.

Uma das formas de realizar essa transferência é com o pré-treinamento de redes. Nesse caso, normalmente a rede é inicialmente treinada em um *dataset* maior, como o Imagenet, e os pesos das camadas da rede são mantidos, com exceção da camada *softmax*. A ideia é que, com o pré-treinamento, a rede aprende estruturas intermediárias, como na Figura 2.11, tais que os *feature maps* da penúltima camada permitem fácil separação entre os exemplos por um classificador linear, que no caso é a última camada. Essas estruturas podem ser reaproveitadas para domínios diferentes e isso permite que os resultados sejam bons mesmo que o novo domínio seja escasso de exemplos, já que as estruturas que diferenciam objetos já foram aprendidas.

2.4 Detecção de Objetos

Uma parte importante do projeto consiste em entender como funciona a detecção de objetos e avaliar sua viabilidade para sistema proposto. Nesta seção, apresentamos um modelo simples de detector de pedestres baseado em *sliding windows*[14][42]. Para uma imagem dada, o sistema deve ser capaz de identificar as regiões onde se situam os pedestres, conforme ilustrado na Figura 2.34. Um sistema desse tipo deve operar em três passos:

1. Geração de janelas candidatas
2. Classificação de janelas candidatas
3. Refinação da decisão

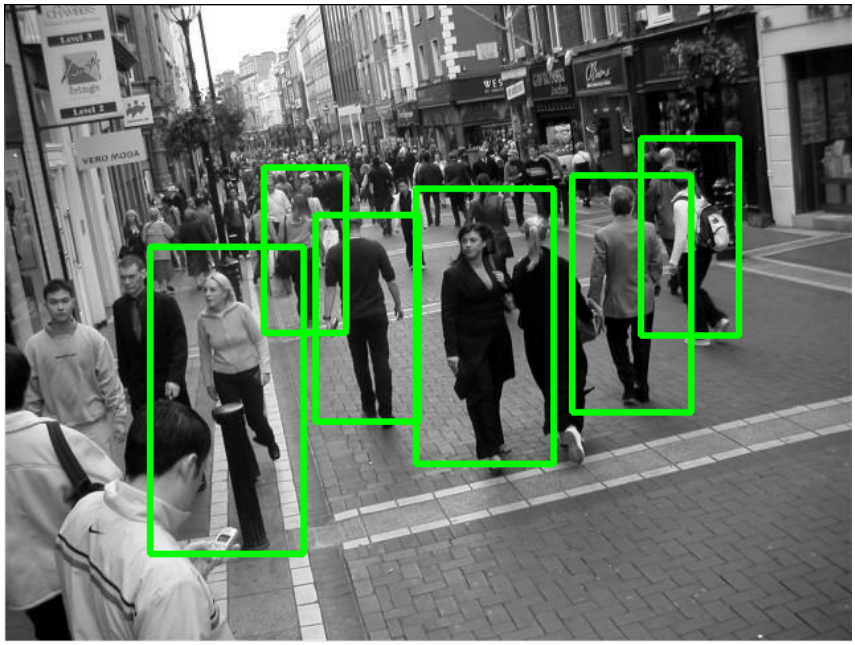


Figura 2.34: Exemplo de detecção de pedestres

2.4.1 Geração de candidatos

A parte de geração de candidatos poderia, de maneira ingênua, ser feita utilizando toda a imagem e fazendo retângulos do mesmo tamanho que variem de posição, começando, por exemplo, na parte superior esquerda da imagem e indo até a parte inferior direita seguindo um conjunto pré fixado de variações Δx e Δy. Essa abordagem, no entanto, desconsidera alguns fatores:

- Podem existir regiões da imagem que não interessam ao detector, por exemplo, os *pixels* acima da linha do horizonte de determinada imagem, a definição dessa região e sua incorporação ao algoritmo pode reduzir o tempo de processamento.
- Normalmente, para uma abordagem de Machine Learning clássico (sem Deep Learning), as janelas são processadas antes de serem fornecidas ao classificador, de forma que é gerado um vetor de *features* para cada janela e esse vetor é fornecido ao classificador no lugar dos valores RGB das imagens. Conforme ilustrado na Figura 2.35, muitas vezes há interseções entre as janelas e os descritores relativos a essas interseções são calculados múltiplas vezes (uma vez para cada janela).
- A utilização de retângulos do mesmo tamanho limita o classificador a detectar pedestres do tamanho do retângulo utilizado (chamado tamanho canônico). Na prática esses pedestres terão tamanhos diversos.

Figura 2.35: Exemplo de detecção de pedestres (Fonte[14])

Local Binary Patterns

Os Local Binary Patterns (LBP)[27], são uma maneira de gerar os descritores de janelas acima citadas antes que as janelas sejam passadas para a etapa de classificação. De maneira simplificada, o código LBP de um *pixel* é o resultado de sua comparação com os 8 *pixels* vizinhos para uma imagem em nível de cinza: para cada vizinho, é gerado um *bit* com valor igual a 1 caso o valor exceda o do *pixel* cujo valor LBP está sendo determinado e com valor igual a 0 caso contrário. Em seguida, esses *bits* são concatenados e a concatenação passa a corresponder ao valor LBP do pixel. Existem variações dessa técnica, como o LBPU, que agrupa vários códigos LBP em apenas alguns principais tendo em vista a redução do número de códigos possíveis.

Normalmente, após ser aplicado o descritor para cada um dos *pixels*, a janela é separada em sub-regiões e são montados histogramas LBP para cada uma dessas sub-regiões, conforme ilustrada na Figura 2.36. A concatenação desses histogramas é o descritor utilizado na etapa de classificação.

Para o exemplo da detecção de pedestres, Δx e Δy podem ser escolhidos de forma que os cálculos dos histogramas de sub-regiões comuns a mais de uma janela sejam reaproveitados para outras janelas. Isso evita cálculos redundantes para as interseções comuns entre janelas e torna o sistema de detector de pedestres mais eficiente.

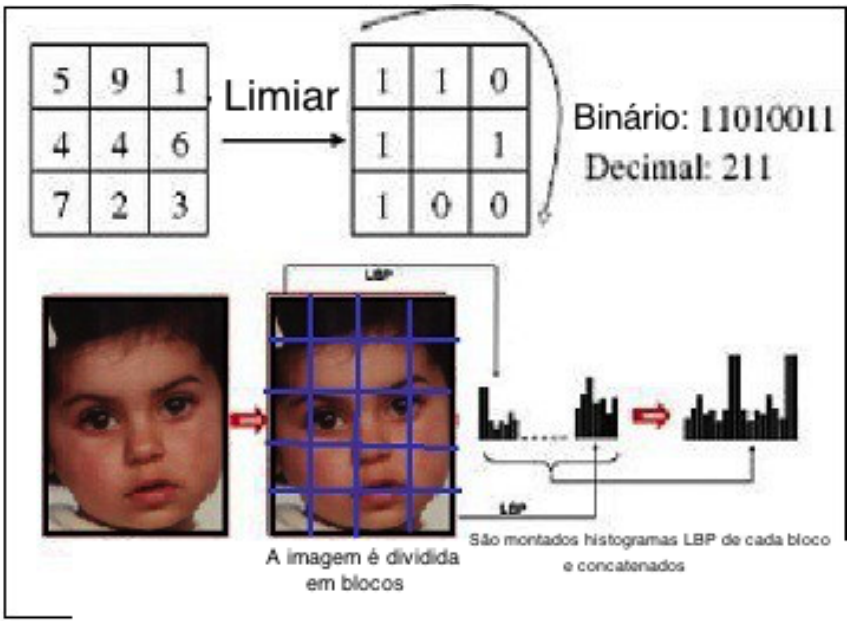


Figura 2.36: Resumo de LBP (Fonte[15])

Pyramidal Sliding Window

Para contornar o problema dos diferentes tamanhos de pedestres, que é consequência do fato de que pedestres estão a diferentes distâncias da câmera, é utilizada a detecção de janelas com pirâmide. Essa técnica consiste em manter o tamanho da janela canônica, mas variar o tamanho da imagem pela qual a janela desliza. Para imagem original com largura w_0 e altura h_0, é definido um parâmetro s e as imagens geradas passam a ter altura igual a $\frac{h_0}{s^i}$ e largura igual a $\frac{w_0}{s^i}$, sendo i o índice da imagem. O algoritmo de geração de candidatos é então aplicado na imagem original e nas imagens reduzidas.

É importante ressaltar que, antes de as imagens reduzidas serem geradas, se faz necessária a aplicação um filtro de suavização na imagem a ser reduzida, caso contrário a imagem resultante possuirá artefatos, ou seja, padrões gerados automaticamente e que podem comprometer o funcionamento do detector. É importante também que seja definido o mapeamento entre janelas nas imagens reduzidas e suas correspondentes na imagem original, pois esse mapeamento será utilizado em etapas posteriores. Com a geração de janelas dessa forma, é possível detecções de pedestres de diferentes tamanhos, apesar de que a razão entre largura e altura de cada janela ainda fica limitada a um só valor. A Figura 2.37 ilustra as várias imagens geradas e o número de janelas candidatas para cada imagem.

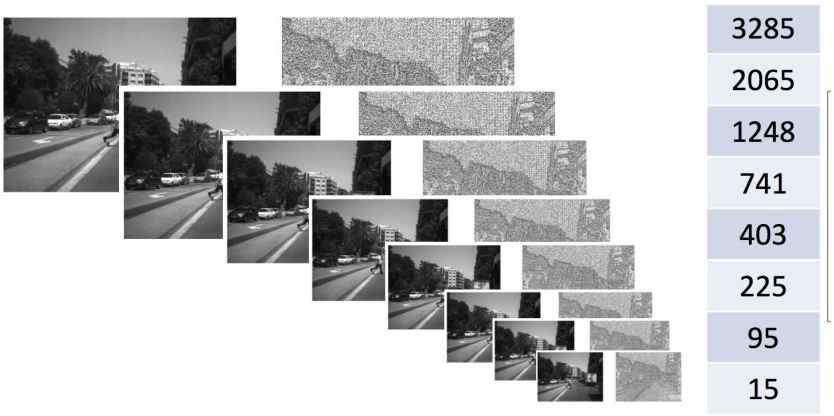


Figura 2.37: Exemplo de Pyramidal Sliding Window (Fonte[14])

2.4.2 Classificação de candidatos

A classificação de candidatos parte de um conjunto de janelas candidatas em que há uma marcação prévia, normalmente feita por humanos, de janelas com pedestres e janelas sem pedestres. Esses exemplos são fornecidos como *training set* para um classificador supervisionado. Para o caso de LBPs, normalmente o classificador usado é Support Vector Machines (SVM)[43]. De maneira simplificada, o SVM é um classificador que tenta separar as classes com a máxima margem possível, sendo que a margem é a distância em relação aos *support vectors*, que são os exemplos que delimitam a fronteira que separa determinada classe das outras.

A Figura 2.38 ilustra uma visualização simplificada, em que os eixos x e y correspondem aos valores de duas *features* para exemplos em $\mathbb{R}^2$ e os *support vectors* são destacados em amarelo. Vale ressaltar que, em exemplos reais, as classes normalmente não são linearmente separáveis como na figura, ou seja, não é possível traçar um hiperplano sem que exemplos caiam na classe errada. A tolerância a classificações erradas na geração desse hiperplano normalmente aparece como parâmetro do classificador, sendo que uma menor tolerância significa uma menor margem.

2.4.3 Refinação de Decisão

Conforme ilustrado na Figura 2.39, muitas vezes os mesmos pedestres acabam sendo detectados várias vezes em múltiplas janelas. Nesse caso, deve haver algum tipo de algoritmo que tenha como objetivo a eliminação desses resultados múltiplos. O algoritmo conhecido como *Non-Maximum Suppression* [14] é utilizado aqui como exemplo de solução desse problema e é apresentado a seguir.

1. Inicia uma lista P contendo as janelas classificadas como pedestres em ordem decrescente de confiança. Sendo a confiança o valor $\in [0, 1]$ retornado pelo algoritmo:

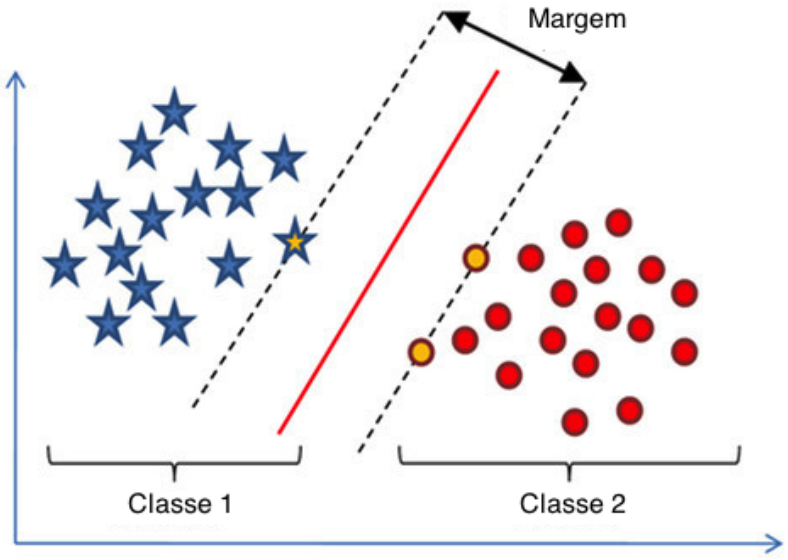


Figura 2.38: Ilustração simplificada de SVM (Fonte[16])

valores mais próximos de 1 indicam chance maior de que a janela contenha um pedestre.

2. Cria uma lista vazia G.

3. Verifica se o primeiro elemento de P já está presente em G. Se estiver, o elemento é descartado, se não estiver, ele é adicionado a G. Para definir se o elemento é o mesmo, são usadas as coordenadas e o tamanho da janela na imagem original e é avaliada a interseção entre o elemento de G e o elemento de P, se a área da interseção entre as janelas for maior ou igual a 50% da área do elemento de P, é considerado que o elemento já está presente em G. Esse último passo é repetido até que não haja mais elementos em P.

Ao final do procedimento, G contém todas as detecções não-redundantes. Existem outros algoritmos que resolvem esse problema, mas nenhum deles é tido como o melhor, já que algoritmos mais sofisticados tendem a ser mais lentos. Ou seja, a escolha depende da aplicação. Existem também algoritmos que não retornam diretamente uma das detecções, e sim uma média das várias detecções redundantes.

2.4.4 Conseguindo exemplos melhores para o classificador

A marcação de pedestres em imagens normalmente é um trabalho manual e feito por humanos e, por isso, está sujeita a erros. Existem ferramentas *online* que viabilizam esse tipo de marcação, como o Mechanical Turk, da Amazon, ou o LabelMe, do MIT,

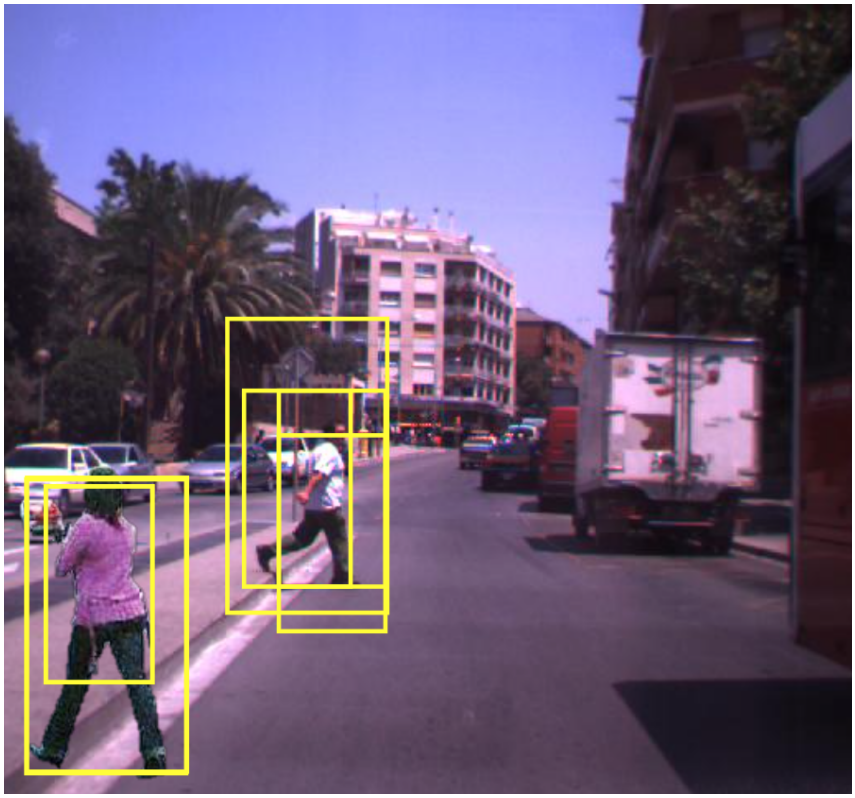


Figura 2.39: Várias detecções dos mesmos pedestres (Fonte[14])

em que pessoas se disponibilizam a marcar as imagens. Para detecção de pedestres, é empiricamente sabido que incluir um pouco do fundo ao redor do pedestre na janela em questão melhora a performance. Para as imagens relativas ao fundo, é possível a obtenção tanto a partir de imagens onde há pedestres quanto de imagens em que não há nenhum pedestre, conforme ilustrado na Figura 2.40, em que as janelas com pedestres são representadas em amarelo e as com fundo são representadas em vermelho.

Conforme explicado na parte de SVM, algumas imagens são mais importantes que outras na delimitação do limiar que separa as classes. A figura 2.41 ilustra isso. No canto esquerdo, são representados todos os exemplos, sendo as classes baseadas em *features* bidimensionais assim como no gráfico de SVM. Ao centro, é representada a separação

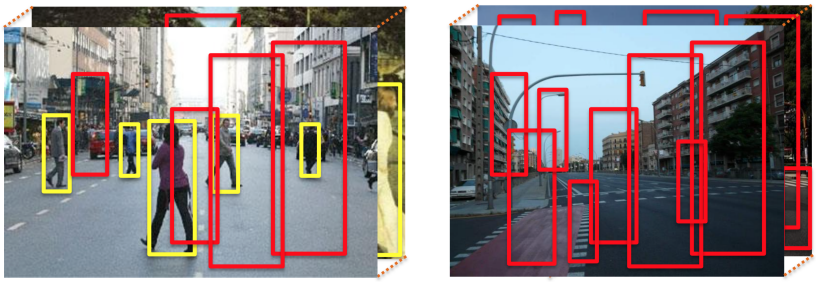


Figura 2.40: Exemplos (Fonte[14])

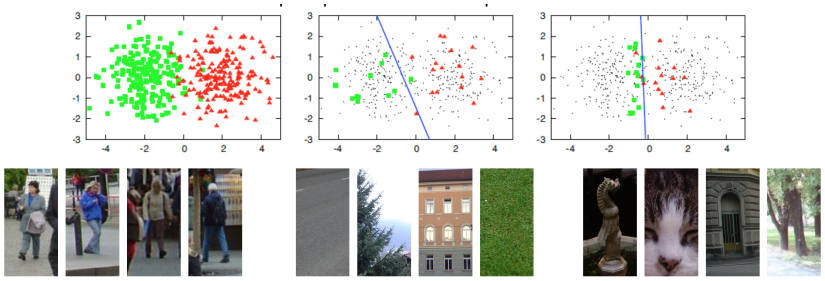


Figura 2.41: Influência de exemplos para a determinação do limiar do classificador (Fonte[14])

caso apenas alguns poucos exemplos aleatórios fossem tomados para geração do limiar. No canto direito, são tomados novamente poucos exemplos, mas agora esses exemplos são os *support vectors* e, por isso, o limiar de classificação gera um separador bem melhor do que caso exemplos aleatórios fossem usados. Abaixo à esquerda, há a ilustração de alguns pedestres, no meio, a de fundos considerados de fácil classificação, que teriam efeitos similares aos exemplos ilustrados na parte superior ao centro e, à esquerda, há a representação de fundos correspondentes aos *support vectors*, ou seja, que delimitam o limiar do classificador. As seções a seguir são dedicadas a detalhar métodos utilizados para coletar exemplos mais importantes ao classificador.

Bootstrapping

O *bootstrapping* [14] tem como objetivo a obtenção de imagens de fundo que ajam como *support vectors* para o modelo em questão. Ele funciona de maneira iterativa e opera conforme descrito a seguir e ilustrado na Figura 2.42.

- São fornecidas, de um lado, imagens com fundo e pedestres e o marcador humano tem a função de colocar janelas sobre os pedestres. De outro lado, são fornecidas imagens que contêm apenas fundo e as janelas sem pedestres são geradas automaticamente a partir dessas imagens
- Após treinado o classificador, são fornecidas imagens apenas com o fundo e o detector de pedestres é aplicado a essas imagens. Caso o algoritmo detecte algum pedestre nessas imagens, sabemos que a detecção é incorreta e a janela detectada é adicionada a um conjunto Q
- As imagens em Q são incorporadas ao conjunto de imagens de treinamento e o classificador é retreinado. Em seguida, o processo é repetido: mais imagens contendo apenas fundo são avaliadas e os erros são incorporados ao classificador, que será retreinado. Esses passos se repetem até que haja convergência ou até que um número pre-determinado de iterações seja atingido

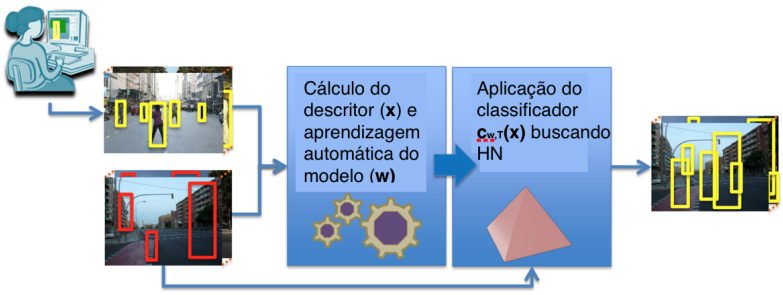


Figura 2.42: Esquemático de *bootstrapping* (Fonte[14])

Active Learning

O *active learning*, da maneira como foi definido em [14] é um mecanismo que opera de maneira similar ao *bootstrapping*, mas com o objetivo de encontrar janelas que contenham pedestres e ajam como *suport vectors*. Ele consiste em, de um lado, utilizar imagens que só contenham fundo e, de outro lado, utilizar uma marcação de pedestres feita por humanos, de maneira similar ao *bootstrapping*. A diferença é que agora o humano pode marcar um conjunto reduzido de pedestres, em vez de marcar todos. O classificador é treinado e aplicado em um conjunto de imagens de teste. Em seguida, o humano verifica essas imagens de teste e marca manualmente todos os pedestres nelas contidos que não foram detectados pelo sistema. Esses pedestres são tratados como *suport vectors* e passam a ser incorporados no conjunto de janelas usadas para treinamento. Finalmente, com esse novo conjunto, o classificador é retreinado e aplicado novamente em imagens. O processo se repete até que haja convergência ou um número de iterações pré-definido seja atingido.

Combinando *active learning* e *bootstrapping*

A Figura 2.43 mostra a combinação dos dois métodos. Uma maneira como essa combinação é feita á a seguinte:

- É utilizada, primeiramente, uma iteração do *bootstrapping* para gerar exemplos de fundo (alimenta parte inferior da figura)
- Em seguida, é utilizada uma iteração de *active learning* para gerar exemplos de pedestres (alimenta parte superior da figura)
- O processo é repetido até que haja convergência ou um número de iterações pré-definido seja atingido, permitindo que sejam gerados tanto exemplos relevantes positivos quanto negativos.

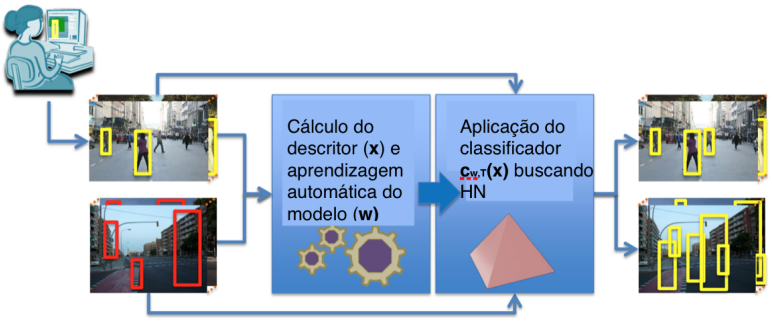


Figura 2.43: Esquemático que combina os dois métodos (Fonte[14])

2.4.5 Avaliando precisão da etapa de classificação

Considerando apenas a etapa de classificação, algumas medidas comumente usadas são baseadas na matriz de confusão. Normalmente, uma matriz de confusão plota como linhas as classes reais e como colunas as classes resultantes da classificação. Para esse tipo de detector, a matriz é 2×2, já que a classificação é binária e é representada na Figura 2.44.

		Classe prevista: *P*	Classe prevista: *N*
Classe real	*P*	True Positives (TP)	False Negatives (FN)
	N	False Positives (FP)	True Negatives (TN)

Figura 2.44: Matriz de confusão (Fonte[17])

Seus elementos são denominados como a seguir:

- ***True positives*:** elementos classificados como contendo a classe e que de fato a contêm
- ***True negatives*:** elementos classificados como não contendo a classe e que de fato não a contêm
- ***False positives*:** elementos classificados como contendo a classe, mas que não a contêm
- ***False negatives*:** elementos classificados como não contendo a classe, mas que a contêm

Algumas medidas de qualidade do classificador binário são definidas a partir desses elementos[44]:

$$Accuracy = \frac{true\_positives + true\_negatives}{total\_de\_classificações} \tag{2.15}$$

$$Precision = \frac{true\_positives}{true\_positives + false\_positives} \tag{2.16}$$

$$Recall = \frac{true\_positives}{true\_positives + false\_negatives} \tag{2.17}$$

$$Specificity = \frac{true\_negatives}{true\_negatives + false\_positives} \tag{2.18}$$

$$True\_Positive\_Rate = Recall \tag{2.19}$$

$$False\_Positive\_Rate = 1 - specificity = \frac{false\_positives}{true\_negatives + false\_positives} \tag{2.20}$$

Esses são parâmetros que normalmente aparecem em ferramentas de aprendizado e são combinados de algumas maneiras diferentes para avaliar a qualidade da classificação. A medida mais simples é a *accuracy*, mas nem sempre é um bom indicador. Há problemas em que uma das classes aparece com muito mais frequência nos exemplos do que a outra, como no caso de anomalias que se manifestem raramente em imagens de órgãos: classes que aparecem dessa forma são conhecidas como *skewed classes*. Num caso em que *positive* represente a presença da anomalia rara e *negative* represente a sua ausência, caso o classificador atribua sempre *negative* para os exemplos, a *accuracy* será alta, mas isso não significa que é um bom classificador, já que ele deixa passar todas as imagens em que há anomalia. Para esse mesmo classificador, a *precision* fica como indeterminada, já que não pode ser calculada, e o *recall* fica como 0. Ou seja, esses indicadores mostram que o classificador é ruim. Um outro indicador normalmente usado é o *F1-score*, que é calculado por

$$F1\_score = 2 \times \frac{precision \times recall}{precision + recall} \tag{2.21}$$

O indicador varia entre 0 e 1. Alguns medidores aqui apresentados podem ser usados para melhoria de parâmetros. Pode-se supor um exemplo simplificado de fronteira de classificação representada na Figura 2.45, em que os exemplos dentro do círculo são considerados pessoas e os exemplos fora dele são considerados objetos. A variação de um

parâmetro, nesse modelo didático, produz a variação da fronteira de classificação conforme ilustrado.

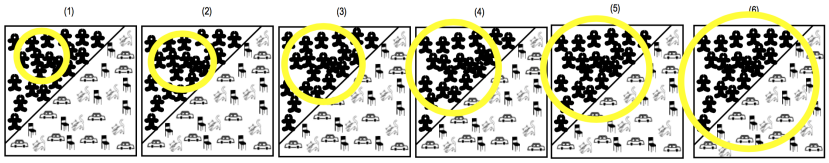


Figura 2.45: Efeito da variação de parâmetro na fronteira de classificação (Fonte[14])

Nesse contexto, uma forma de decidir o melhor parâmetro possível é plotando a chamada curva Receiver Operating Characteristic (ROC). Nessa curva, a *true positive rate* é plotada no eixo y e a *false positive rate* é plotada no eixo x. Para entender como funciona a curva, basta pensar que cada valor do parâmetro equivale a um ponto (x_0, y_0) nesse gráfico. O classificador ideal tem *true positive rate* igual a um, pois acerta todos os exemplos que classifica como positivos em relação aos que são de fato positivos e *false positive rate* igual a zero, já que não faz classificações incorretas. Isso significa que o classificador ideal corresponde ao ponto $(0, 1)$ do gráfico. Para cada valor do parâmetro a ser variado, são calculadas a *false positive rate* e a *true positive rate* e o ponto correspondente é identificado no gráfico. O valor que gera o ponto com a menor distância em relação a $(0, 1)$ é tomado como melhor valor do parâmetro, sendo que a distância pode ser o próprio valor de distância euclidiana entre os pontos. Um exemplo de curva ROC é ilustrado na Figura 2.46, em que o melhor parâmetro está entre os valores que geraram os pontos $(0.04; 0.85)$ e $(0.16; 0.9)$.

2.4.6 Avaliando precisão do detector como um todo

A seção anterior focou na avaliação apenas da etapa de classificação. Para avaliar o detector como um todo, primeiramente devem ser feitas as marcações corretas (*ground truth*), que posteriormente serão conferidas com os resultados do detector. Deve-se pensar que as janelas detectadas, mesmo que correspondam ao mesmo pedestre da *ground truth*, não necessariamente terão o tamanho e posição exatos do que foi marcado, até porque há limitações de posições possíveis no algoritmo de *sliding windows* dependendo de Δx e Δy definidos. A maneira de verificar se uma janela A e uma janela B são equivalentes é utilizando a equação a seguir:

$$S = \frac{\acute{a}rea(A \cap B)}{A \cup B} > Limiar \tag{2.22}$$

Sendo que o limiar é um parâmetro a ser definido pelo usuário. Algumas medidas importantes na avaliação do detector são[14]:

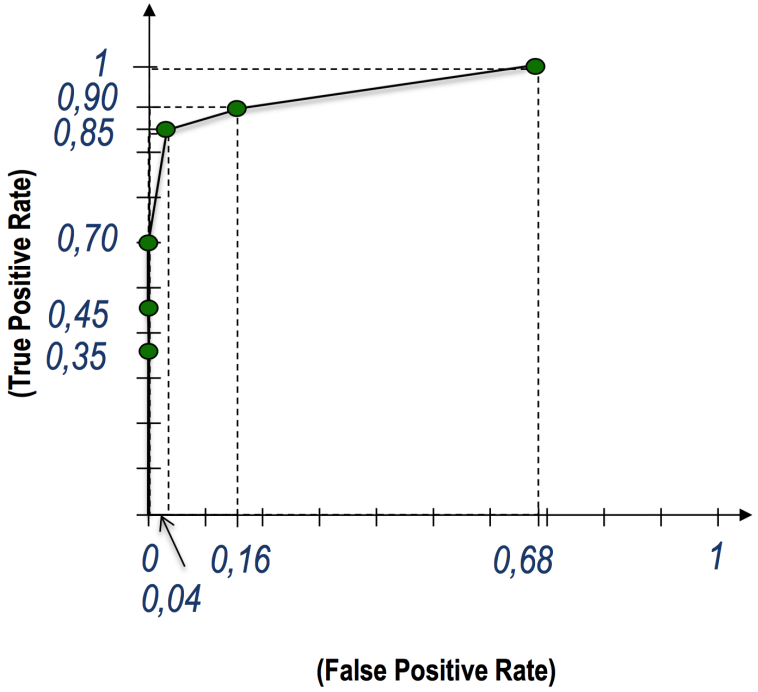


Figura 2.46: Exemplo de curva ROC (Fonte[14])

$$Taxa\_de\_detecção = \frac{reais\_positivos}{total\_de\_pedestres\_ground\_truth} \tag{2.23}$$

$$Taxa\_de\_erro = 1 - taxa\_de\_detecção \tag{2.24}$$

$$FPPI = \frac{falsos\_positivos}{total\_de\_imagens\_ground\_truth} \tag{2.25}$$

Uma maneira de avaliar a qualidade do classificador é plotando $taxa\_de\_erro \times FPPI$, em que cada ponto corresponde a um valor fixado de limiar da equação 2.22. Nesse contexto, caso o limiar seja muito pequeno, mais janelas vão ser consideradas como detecções positivas, o que resulta numa $taxa\_de\_erro$ baixa, já que vários pedestres serão corretamente detectados, mas uma $FPPI$ alta, já que, vários falsos positivos aparecem como consequência desse limiar baixo. Pelo raciocínio oposto, um limiar muito alto reduz $FPPI$, mas aumenta $taxa\_de\_erro$. Um exemplo de gráfico conforme descrito aqui pode ser visto na Figura 2.47. Para avaliar se o detector é bom, normalmente se usa a área abaixo da curva. O detector ideal tem $taxa\_de\_erro$ e $FPPI$ iguais a zero, logo, o quanto mais próximo dele melhor o detector, o que significa que uma área menor é indicativo de melhor qualidade do detector.

Essa seção conclui o que o leitor deve saber sobre Machine Learning e detecção de

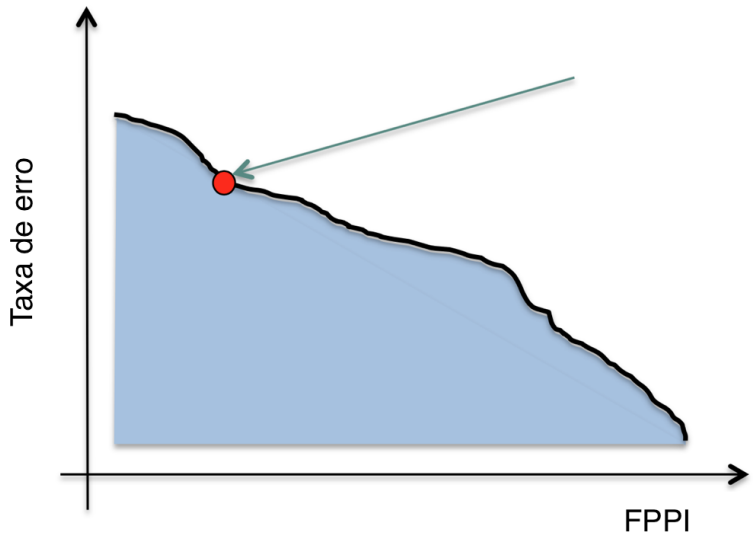


Figura 2.47: Exemplo de curva de avaliação de detector (Fonte[14])

objetos para entender o trabalho. A seção seguinte reúne um conjunto de estudos recentes que tentam resolver alguns dos problemas que aparecerão ao longo do trabalho.

2.5 Estado da arte

O primeiro problema cujos estudos recentes são citados é a remoção de não-relevantes, que é uma das principais etapas do trabalho a ser descrito. A segunda seção trata de um estudo geral sobre uso de imagens marcadas por usuários comuns da Internet para aplicação em Machine Learning.

2.5.1 Remoção de não-relevantes

O problema aqui consiste em, dado um grande conjunto de imagens de determinado objeto, mas com presença de imagens indesejadas de objetos diferentes dele, treinar algum sistema ou classificador capaz de remover essas imagens de objetos diferentes, aqui chamadas de imagens não-relevantes. Grande parte dos trabalhos tenta realizar essa separação usando a similaridade entre as imagens, já que a maioria representa o objeto procurado. Liu et al [45] resolve o problema através de *graph-cuts*, utilizando uma abordagem que considera apenas o conteúdo das imagens para encontrar um *cluster* de imagens com conteúdo similar. Já em [18], é feito um trabalho que utiliza informações que vão além do conteúdo da imagem para encontrar o que é relevante: são usadas as informações relativas a palavras contidas na página *web* de determinada imagem da *query* para verificar se essa imagem está entre os principais resultados de alguma *query* não sinônima à original. Caso

esteja, significa que a palavra não é o objeto que se procura. Esse método é ilustrado na Figura 2.48, em que o contexto original se refere à *query* "arc de triomphe", mas a imagem destacada é uma das principais para "Volubilis arc de triomphe" e, por isso, deve ser descartada.

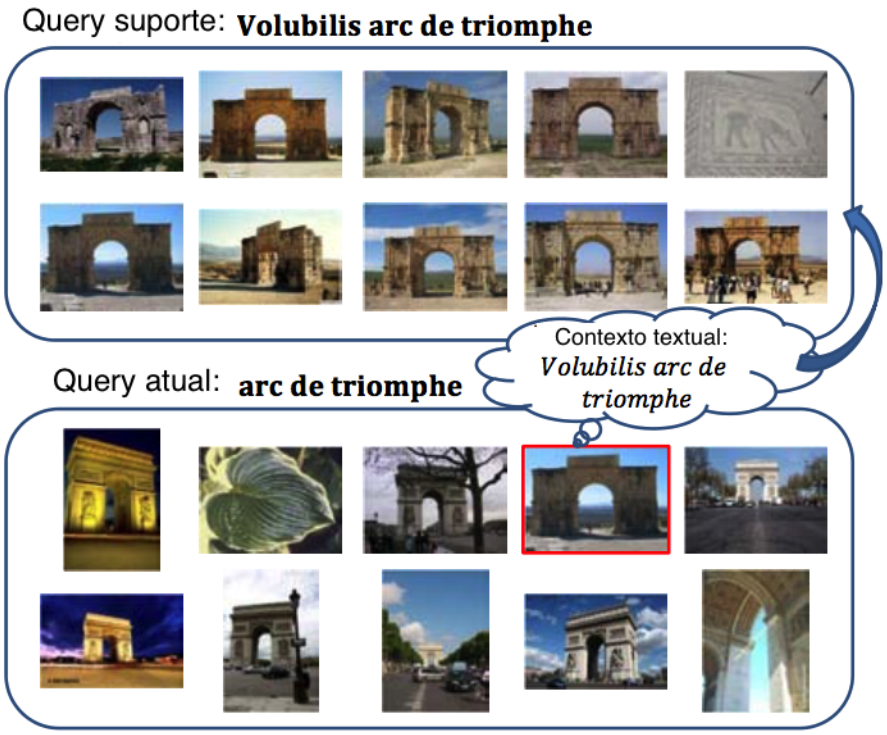


Figura 2.48: Eliminação de *outliers* usando *deep contexts* (Fonte[18])

O método descrito no artigo foi implementado baseado em um *dataset* pronto, o INRIA [46], que possui 353 *queries* e 71.478 imagens. São aproximadamente 200 imagens por *query*. Cada imagem tem um *score* inicial de *ranking* dado por uma *search engine*, um *ground truth* indicando sua relevância e o contexto textual relacionado à imagem. A precisão desse método fica em torno de 70% a 80%, sendo que o artigo original deve ser consultado para mais detalhes.

O método implementado em [18] tem complexidade $O(m^3)$, sendo m o número de imagens a ser considerado. Devido a essa limitação, o artigo trabalha apenas com $m = 40$. Como a implementação manual de *deep contexts* usando *crawlers* é muito complexa e inviável dado o prazo, optou-se por não implementar nenhuma abordagem similar a essa nesse trabalho.

2.5.2 Uso de imagens fornecidas por usuários

A ideia de se usar imagens com marcação reaproveitada das imagens normais da Internet foi analisada por Mahajan et al [47] em estudos recentes. Quando a ideia do trabalho aqui apresentado foi concebida e durante a maioria da relização dele, o autor não tinha conhecimento desses estudos, que foram publicados poucos meses antes da apresentação. No artigo, que foi uma parceria entre o grupo Applied Machine Learning (AML) e o grupo Facebook Artificial Intelligence Research (FAIR), ambos do Facebook, foram usadas imagens com *hashtags* como base. A premissa do estudo é que, atualmente, os avanços em Deep Learning ficam limitados devido ao fato de as imagens usadas serem marcadas manualmente sob demanda: no futuro, *datasets* com milhões de imagens serão insuficientes, as aplicações passarão a requerer bilhões.

Dentre os vários experimentos realizados, o maior envolveu 3,5 bilhões de imagens e 17.000 *hashtags*. Após treinar o sistema para classificar essas *hashtags*, foi realizado *transfer learning* e treinamento de redes no Imagenet. Um dos *sets* de treino usados, contendo 1 bilhão de imagens e um vocabulário de 1.500 *hashtags*, foi capaz de superar o estado da arte no Imagenet, com *top-1-error* de 85,4% (mais de 2% de melhora em relação ao melhor resultado até então) e *top-1-error* de 97.6%. Os *embeddings* desses modelos serão disponibilizados futuramente como ferramenta *open source* a fim de ajudar a comunidade científica.

Esses resultados mostram que, de fato, a ideia é boa. No caso do grupo, eles são proprietários de grande conjunto de imagens pré-marcadas e não houve necessidade de realizar *web crawling* da maneira como foi feito neste trabalho (embora as APIs que lhes trouxeram essas imagens provavelmente realizem *web crawling* para obtê-las). Além disso, a quantidade de recursos computacionais disponíveis, assim como de pessoal e dinheiro envolvidos, permitiram resultados bem superiores aos aqui apresentados. Segue um trecho de post sobre o estudo [48], que ilustra bem essa diferença:

> Como uma única máquina levaria mais de um ano para completar o treinamento do modelo, criamos um jeito de distribuir a tarefa entre 336 GPUs, diminuindo o tempo total de treinamento para apenas algumas semanas.

Após a apresentação desses conceitos, o leitor está pronto para entender as discussões e experimentos das seções seguintes, em que os modelos de rede serão utilizados de fato e a detecção de objetos será utilizada em discussão.

Capítulo 3

Sistema Proposto

A ideia de usar imagens já presentes na *web* para obter bancos de dados e classificadores é o ponto principal desse trabalho. Nesse capítulo é proposta uma ferramenta que, a partir de *queries* quaisquer informadas pelo usuário, permite o treinamento de classificadores e criação bancos de imagens genéricos tomando proveito desse conjunto de imagens com marcação de *labels* já feita pelos usuários normais da *Internet*. As aplicações em que esse tipo de sistema pode ser usado são diversas, sendo que a principal vantagem é o fato de que não será necessário investir grandes quantidades de tempo e dinheiro para obter esses bancos ou classificadores.

O capítulo começa introduzindo o sistema, com diagramas das telas a serem apresentadas. Em seguida, é mostrado o diagrama de comunicação para o caso de uso principal do sistema: o treinamento de classificadores, sendo que cada mensagem desse diagrama é detalhada. Por fim, são apresentadas as técnicas específicas que podem ser usadas para resolver cada parte apresentada no diagrama.

3.1 O sistema

O sistema de treinamento de classificadores e *datasets* é descrito aqui como uma aplicação *web*, mas vale ressaltar que, no mercado, sistemas desse tipo possuem tanto uma interface *web* quanto uma interface em código, em que o usuário, ao programar sua aplicação específica, faz chamadas usando REST à aplicação aqui apresentada e o resultado é normalmente retornado em formato JSON, facilitando assim a utilização da ferramenta diretamente em código. Nesse caso, a documentação de cada como cada chamada é feita, como o retorno é obtido e qual a funcionalidade que cada chamada executa é normalmente fornecida em um guia. Como o escopo do trabalho não é referente a uma ferramente real de mercado, foi feita essa simplificação.

3.1.1 Janela de treinamento de classificadores

A Figura 3.1 se refere à criação de novos classificadores.

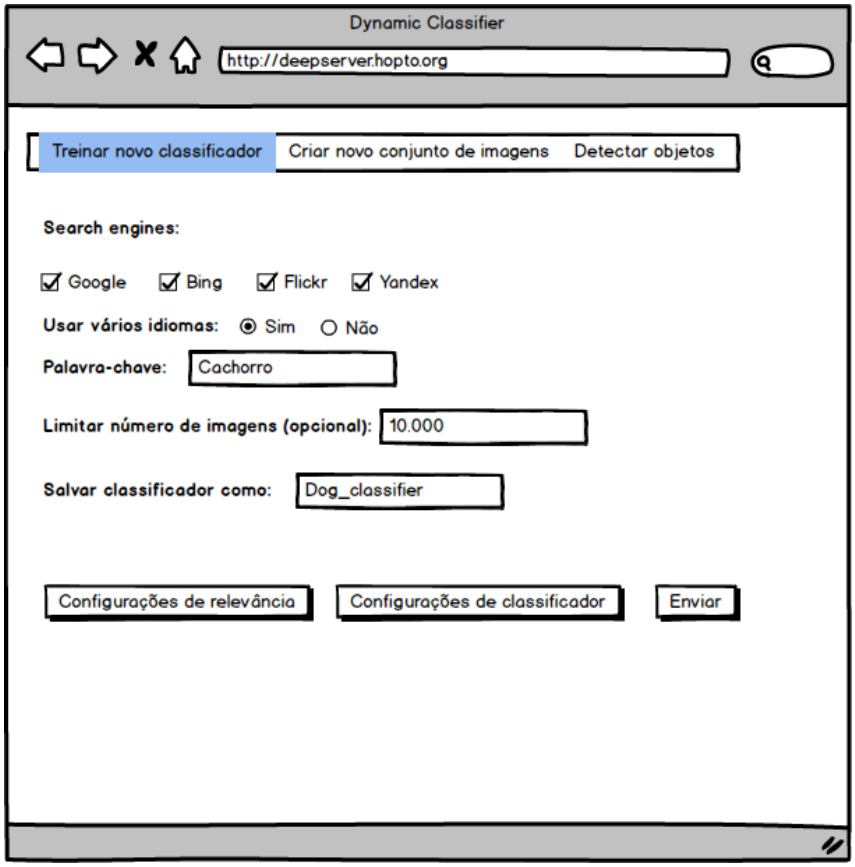


Figura 3.1: Janela de criação de novo classificador

Alguns parâmetros que podem ser definidos pelo usuário nessa criação são:

- ***Search engines:*** permite ao usuário escolher quais *search engines* serão utilizadas como fonte das imagens a serem buscadas
- **Usar vários idiomas:** através dessa opção o usuário pode definir se, nas *search engines* Google e Bing, a palavra digitada deverá ser procurada apenas em inglês ou se ela deve ser buscada tanto no idioma original quanto em outras línguas. As línguas presentes no sistema são: inglês (original), japonês, espanhol, holandês, russo, italiano, alemão, francês, coreano, árabe e chinês.
- **Palavra-chave:** a palavra desejada
- **Limitar número de imagens:** parâmetro opcional, caso o usuário decida limitar o número de imagens retornado (por motivos de tempo total menor, por exemplo)
- **Salvar classificador como:** o classificador fica salvo num banco de dados. Essa opção define o nome que será utilizado por esse classificador

- **Configurações de relevância:** esse modal se refere a parâmetros da fase de separação de imagens relevantes das não-relevantes, que será melhor explicada nas próximas seções. Esses parâmetros dependem do algoritmo a ser utilizado nessa fase e, por isso, fica difícil defini-los *a priori*
- **Configurações de classificador:** esse modal se refere a parâmetros da fase de treinamento de classificadores (a ser melhor explicada a seguir), como número de camadas, número de parâmetros por camadas, tipos das camadas, funções de ativação...

3.1.2 Janela de criação de bases de imagens

Essa janela permite ao usuário a criação de bases de imagens e é retratada na Figura 3.2. O usuário pode tanto criar uma base de imagens do zero, usando configurações que são iguais às apresentadas na tela anterior quanto reaproveitar a base de imagens que foi usada para criar algum classificador.

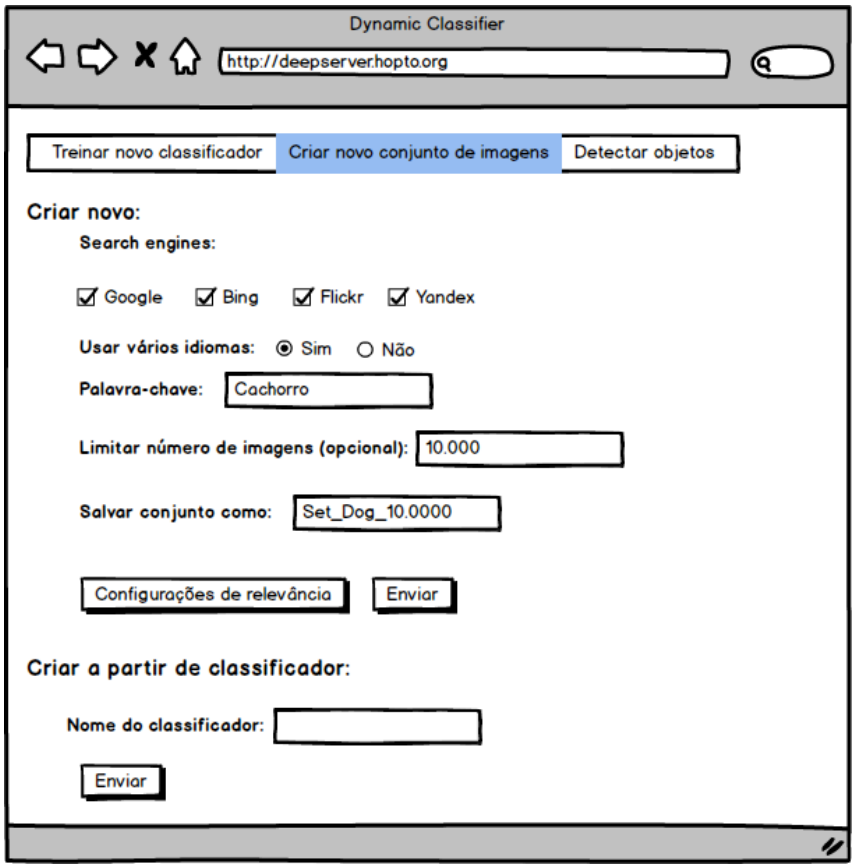


Figura 3.2: Janela de criação de novo conjunto de imagens

3.1.3 Janela de detecção de objetos

A Figura 3.3 é relativa à detecção de objetos.

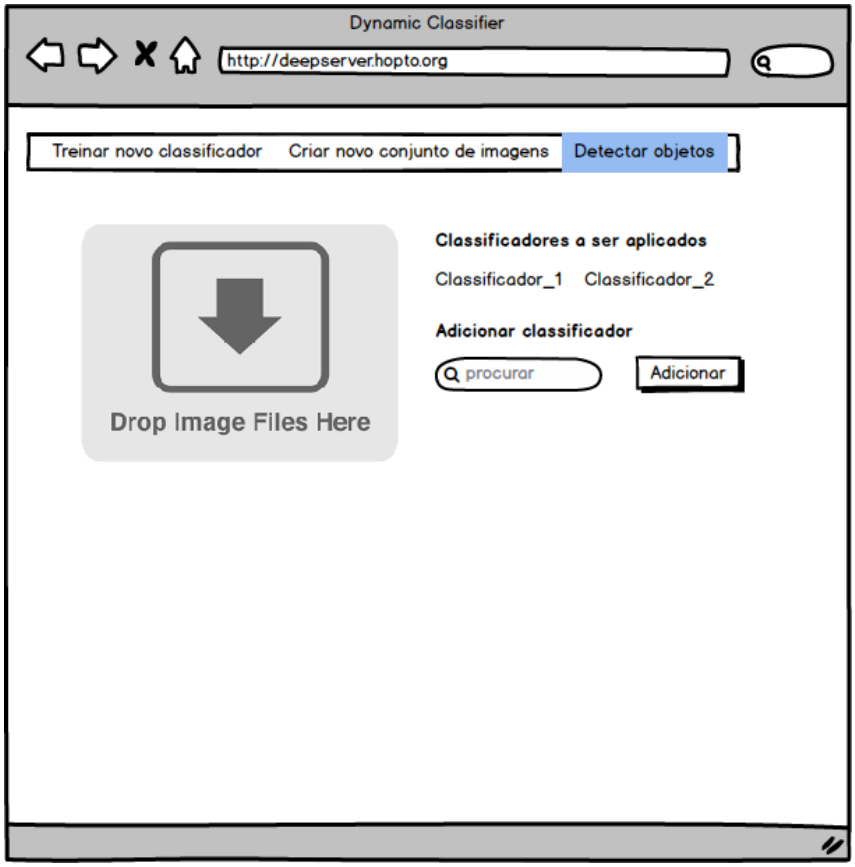


Figura 3.3: Janela de detecção de objetos

Nessa tela, o usuário faz *upload* alguma imagens de seu computador e, através do campo "adicionar classificador", tem a possibilidade de selecionar classificadores cujos parâmetros foram gravados no banco de dados. Esses classificadores são utilizados com uma técnica chamada *sliding windows*, que, de maneira resumida, consiste em verificar, para quadrados de tamanho pré-definido, se o objeto está presente no quadrado ou não, sendo que são feitas verificações para quadrados em todas as posições da tela (o leitor pode voltar à seção de *background* teórico para mais detalhes). A Figura 3.4 mostra um exemplo de detecção de múltiplos animais na tela utilizando dois classificadores diferentes: um para hipopótamo e um para leão.

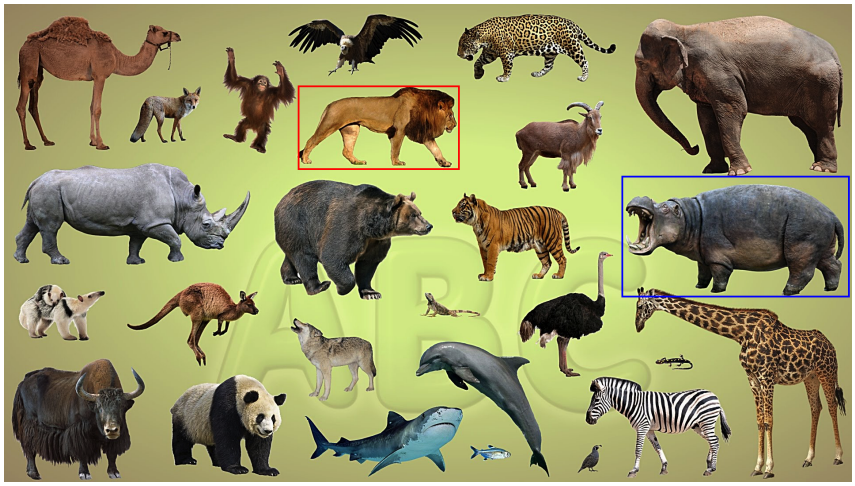


Figura 3.4: Detecção de animais

3.2 Funcionamento

A Figura 3.5 ilustra o diagrama do principal caso de uso do sistema: o treinamento de classificadores.

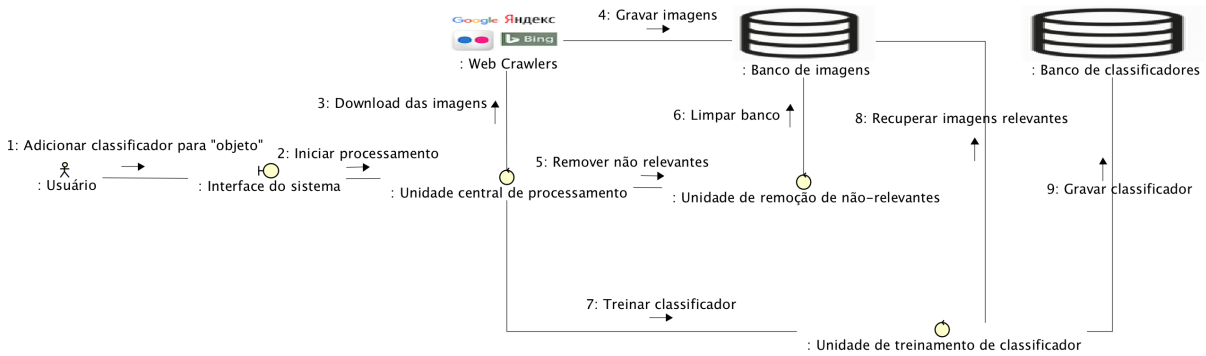


Figura 3.5: Diagrama de comunicação do caso de uso “treinar classificador”

Nesse diagrama, as mensagens são enumeradas de acordo com a sequência em que ocorrem e os diferentes componentes do sistema são ilustrados, de maneira a resumir o funcionamento. As mensagens são melhor explicadas a seguir:

1. O usuário envia uma mensagem à interface do sistema pedindo a criação de um classificador para determinado objeto
2. A interface repassa a mensagem para a unidade central de processamento, que dará início à criação
3. A unidade central faz uma requisição ao conjunto de *crawlers* para que baixem as imagens da *query* pedida usando as configurações especificadas

4. As imagens, da maneira como foram baixadas, são gravadas em um banco de imagens
5. A unidade central de processamento faz uma requisição à unidade de remoção de não-relevantes para que remova as imagens recém adquiridas que não correspondem à *query*
6. A unidade de remoção de não-relevantes, através de análise do conteúdo das imagens recém-obtidas, decide quais são os *outliers* (imagens não-relevantes) e os exclue do banco
7. A unidade central de processamento se comunica com a unidade de treinamento de classificador para que o treinamento do classificador seja iniciado
8. A unidade de treinamento de classificador recupera as imagens que serão usadas para essa criação do banco de imagens
9. Após criado o classificador, o modelo e parâmetros obtidos são salvos no banco de classificadores

3.3 Principais desafios

3.3.1 Obtenção de imagens com *web crawling*

O *web crawling* corresponde ao uso de ferramentas automatizadas de *software* que têm a função de navegar por páginas *web* seguindo comandos pré-definidos pelo programador. Na ilustração da Figura 3.5, ele corresponde ao processamento realizado entre as mensagens 3 e 4. Para obter o grande conjunto de imagens necessário nesse projeto, foram estudadas algumas ferramentas de busca, sendo que as principais são:

- **Google:** fundada por Larry Page e Sergey Brin em 1998[49], a Google é uma das empresas mais conhecidas atualmente, sobretudo por seus sistemas de busca de páginas *web* e imagens.
- **Bing:** outra ferramenta de busca muito conhecida, que implementa inclusive a busca de imagens do Yahoo.
- **Flickr:** é uma rede social especializada no compartilhamento de fotos entre usuários.
- **Yandex:** é uma companhia de Internet russa que opera o maior motor de busca da Rússia e ocupa o quarto lugar na lista mundial dos maiores provedores, baseado em informações do *site* Comscore, com mais de 150 milhões de buscas por dia, como observado em Abril de 2012, e mais de 25,5 milhões de visitantes diariamente (em todos os serviços), como observado em Maio de 2012.

Após decididas as ferramentas de busca que serão usadas, deve ser programado um sistema que navegue por cada uma delas e faça *download* de todas as imagens retornadas, levando em conta as especificidades de cada página. No Google, por exemplo, basta deslizar até o final da página para que sejam mostrados mais resultados, enquanto no Yandex é necessário apertar um botão ao fim da página para que mais resultados sejam mostrados. Cada ferramenta dessas tem um código HTML, que deve ser analisado para que os campos corretos correspondentes a botões e imagens mostrados na tela sejam acessados.

3.3.2 Remoção de imagens não-relevantes

Após a obtenção de várias imagens, é importante ressaltar que muitas delas não possuem relação com a *query* procurada e atrapalham a criação de *datasets* e classificadores que tomam como base essas imagens. A remoção dessas imagens indesejadas se dá antes e durante a mensagem 6 da Figura 3.5. É argumentável que, muitas vezes, ferramentas que realizam *retrieval* devem retornar certa variedade de resultados, de forma a, caso o usuário digite uma *query* ambígua, todas as possibilidades sejam contempladas. Por exemplo, caso o usuário digite a *query* "arco", não se sabe se ele procura um arco do esporte arco e flecha, uma construção, como em "Arco do Triunfo" ou algum gráfico de arco nos eixos XY. No entanto, o propósito de remover as imagens não relevantes não tem a ver necessariamente com esse tipo de problema, na Figura 3.6, por exemplo, são mostrados alguns resultados incorretos para a *query* "*airplane*", em que alguns objetos nada têm a ver com o que foi procurado, na Figura 3.7 é mostrado esse efeito para a *query* "*deer*" e na Figura 3.8 é mostrado esse efeito para a *query* "*coyote*". Esse tipo de resultado é normalmente encontrado nas últimas imagens retornadas pelas ferramentas de busca (abaixo de todas as outras nos resultados), sendo que, mesmo que haja muitas imagens incorretas, ainda há imagens relevantes em meio a elas e é benéfico ao classificador fazer uso dessas imagens relevantes.

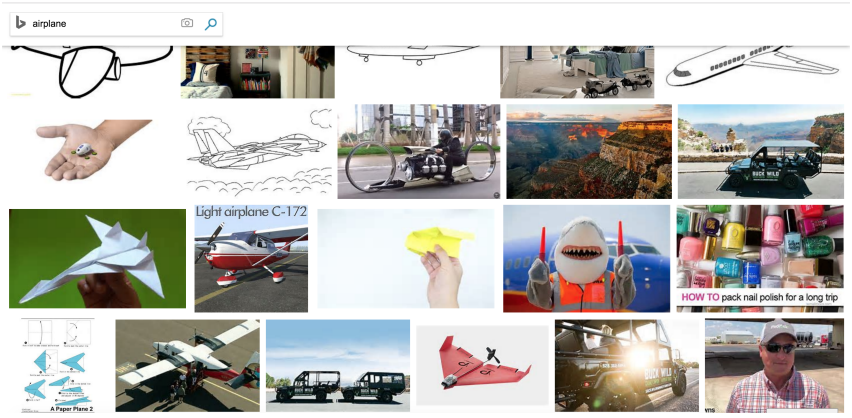


Figura 3.6: Resultados incorretos do Bing para a *query* "*airplane*"

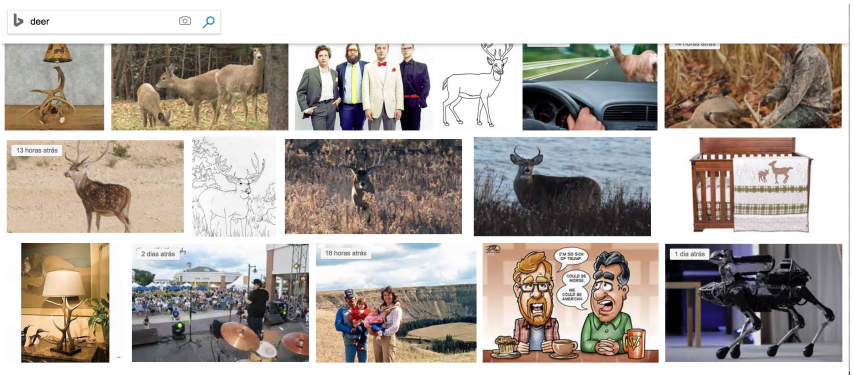


Figura 3.7: Resultados incorretos do Bing para a *query* "*deer*"

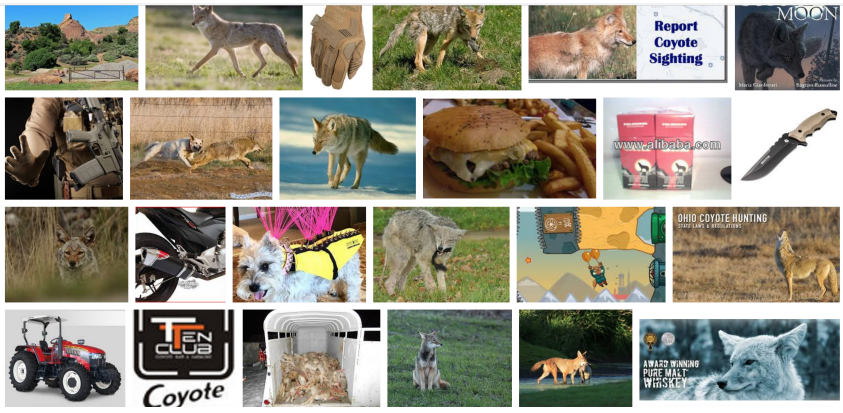


Figura 3.8: Resultados incorretos do Google para a *query* "*coyote*"

É importante, no entanto, notar que o problema da ambiguidade de *queries* citado acima ainda existe. Uma maneira de resolver esse problema é com o uso de *queries* específicas, sendo escolhidas sempre palavras com significado mais restrito: em vez de "arco", podem ser usadas as *queries* "arco e flecha", "arco monumento" e "arco gráfico" para o exemplo acima. Porém, mesmo essa solução pode trazer efeitos indesejados, como a

perda de resultados, por exemplo. Outra solução é a intervenção direta do usuário: após o *download* e gravação iniciais das imagens (mensagens 3 e 4 na Figura 3.5), algumas imagens podem ser mostradas ao usuário, sendo que ele escolhe manualmente quais delas são relevantes ou não. Essa marcação manual é algo que pode ser feito em poucos minutos, visto que o usuário, a princípio, não precisa marcar muitas imagens (o sistema é projetado para funcionar com poucas marcações), e o propósito de obtenção fácil de classificadores e *datasets* não é violado, pois a intervenção é mínima.

Vale ressaltar ainda que esse tipo de ambiguidade é muitas vezes dependente da aplicação, o que significa que a intervenção humana nessa parte pode ser inevitável. Após essa marcação por parte do usuário, é possível utilizar a informação fornecida por ele para definir se o restante das imagens é ou não relevante. Para isso, alguns métodos possíveis são:

- **Redes convolucionais:** nesse caso, o número de exemplos necessários para obter um bom resultado, em teoria, deve ser grande. Os limites desse número devem ser verificados empiricamente.
- ***Triplet networks*:** embora essas redes tenham sido projetadas para funcionar em contextos de autenticação de face [38], a possibilidade de clusterização apresentada no artigo indica que pode ser um método adequado para generalizar a informação obtida através de um conjunto pequeno de exemplos. As Siamese Networks são um modelo um pouco mais antigo do que as Triplet e que já foram usadas para classificação com poucos exemplos de treino por classe, também chamada de *few shot learning* [50], embora o domínio também tenha sido bem diferente (reconhecimento de dígitos de alfabetos diferentes), é possível que esse método funcione bem nessa parte
- ***Transfer learning*:** a possibilidade de realizar o treinamento em um domínio maior e reaproveitar o que foi aprendido no caso em que poucos exemplos estão disponíveis se encaixa muito bem nessa fase, diminuindo o número de imagens que o usuário precisaria marcar manualmente.

Em Machine Learning, o objetivo de aprender a diferenciar classes a partir de poucos exemplos é conhecido como *few-shot learning*, sendo que, particularmente, o *one-shot learning* se refere ao aprendizado usando um exemplo por classe. Trabalhos como os implementados em [51] e [52] abordam esse problema diretamente e conseguiram bastante popularidade. Nesse trabalho, optou-se por explorar as Triplet Networks apresentadas em [10] e [38], pois são bem mais simples do que esses outros métodos, além de existirem tutoriais sobre elas disponíveis *online* e funções do próprio Tensorflow que implementam a função de custo das Triplet Networks.

Após a seleção das imagens que são de fato relevantes e descarte das que não são, é possível proceder para a etapa seguinte: treinar classificadores a partir das imagens relevantes.

3.3.3 Treinamento de classificadores

Essa etapa corresponde ao uso das imagens relevantes para treinar classificadores para cada objeto desejado. Na Figura 3.5 é executada entre as mensagens 8 e 9. Como o sistema foi proposto com detecção de objetos, esses classificadores devem ser *one-vs-all*, permitindo assim que o algoritmo de *sliding windows* faça uso deles.

Para treinar classificadores desse tipo para uma *query* X, deve-se tomar o problema como de classificação binária: os exemplos positivos correspondem a imagens relevantes de X e os exemplos negativos correspondem a imagens de várias outras classes. Para obter imagens de várias outras classes, a princípio, pode-se utilizar métodos de detecção de sinonimidade entre *queries*, como em [53] e [54], para decidir quais *queries* do banco de imagens são semanticamente diferentes de X (evitando que objetos sinônimos sejam usados como classes diferentes). Em seguida, basta usar as imagens relevantes dessas *queries* como exemplos negativos. Podem ainda, ser usadas imagens de *datasets* prontos, como o Imagenet, em vez de imagens obtidas pelos *crawlers*, desde que a sinonimidade seja verificada antes do uso. Em seguida, de posse das imagens e marcações, basta escolher qualquer método de classificação desejado e treinar classificadores com base nele. Como os métodos que fazem uso de *deep learning* são normalmente superiores e vêm substituindo os clássicos, eles são os principais candidatos para essa etapa, até porque não é uma aplicação em tempo real.

3.3.4 Detecção usando classificadores

A última etapa é o uso dos classificadores para realizar a detecção, conforme ilustrado na Figura 3.3. A detecção pressupõe bom funcionamento das outras partes, e a implementação básica é como na seção de teoria. As dificuldades adicionais em realizar detecção no contexto desse projeto específico ficam melhor evidenciadas após análise dos experimentos e, por isso, essa discussão foi colocada na seção de conclusão.

Capítulo 4

Resultados Obtidos

O sistema não foi implementado de fato, pois a implementação necessitaria que cada uma das etapas funcionasse de forma robusta em um conjunto de testes amplo e representativo de várias situações. Na prática, esses testes necessitariam de muito mais tempo do que foi possível alocar para esse trabalho. Essa seção detalha os experimentos feitos, assim como seus resultados. Em cada subseção, primeiro é descrito o que foi feito, para que depois sejam mostrados os resultados do procedimento em questão e que possa ser realizada uma breve discussão acerca deles.

Primeiramente, são feitos experimentos genéricos relativos ao uso de GPU e sua comparação com CPU. Esses experimentos não tiveram relação direta com a aplicação, mas serviram para familiarização com as ferramentas.

Em seguida, inicialmente foi baixado um conjunto de imagens da Internet usando *web crawling*. No entanto, para que fosse possível saber o quão bem os algoritmos de separação de relevantes e não-relevantes funcionam, foi necessário marcar manualmente a relevância das imagens obtidas, permitindo assim saber se o que o sistema julgou como relevante é de fato relevante. No sistema real, essas marcações por relevância de todas as imagens não seriam feitas, sendo que o único momento em que é necessário que o usuário marque algo é assim que as imagens são obtidas. Nesse momento o usuário marca um pequeno número de imagens para que fique definido o que é ou não relevante, conforme discutido no Capítulo 3. Vale ressaltar que esse tipo de marcação do usuário no sistema é bem rápido: será mostrado que o sistema funciona razoavelmente bem com apenas 40 marcações e, se as marcações forem feitas usando algum esquema de marcação como o aqui mostrado, 40 marcações podem ser feitas em questão de segundos.

Em seguida, foi testada a possibilidade de se treinar classificadores com as imagens relevantes. A fim de simplificar o trabalho, essa etapa não foi feita com classificadores *one-vs-all*, como descrito no sistema, mas sim como problema de classificação normal (separar entre um certo número de classes). Isso fez com que fossem evitadas preocupações de

como obter a classe “all” de cada classificador. Outra simplificação que foi feita foi o fato de que não foi usado o retorno real da etapa de remoção de não-relevantes, mas sim o conjunto de marcações manuais colocadas como “relevantes”. Isso significa assumir que a etapa de remoção de não-relevantes funcionou perfeitamente, e foi feito apenas por possibilitar realização independente dos experimentos.

Por fim, outra simplificação que foi feita foi a limitação do escopo das classes. Como será visto mais adiante, o número de imagens obtido por classe chega a cerca de 22 mil, caso fossem usadas muitas classes, seria impossível realizar a marcação manual a fim de avaliar o funcionamento. Por isso, foi decidido que seriam usadas apenas 10 classes para os testes. Por motivos de se obter compatibilidade com um *dataset* padrão, foram escolhidas as 10 classes do CIFAR-10. Essa decisão permitiu, ainda, que fossem realizados experimentos que usassem as imagens dos *crawlers* como *set* de treino e as do CIFAR-10 como *set* de teste, permitindo tirar conclusões sobre a qualidade das imagens obtidas.

4.1 Etapas de experimentação projeto

A Figura 4.1 representa de maneira resumida as etapas dos experimentos no projeto em questão, que são descritas de forma resumida a seguir e detalhadas nas próximas seções.

- **Familiarização com as ferramentas:** aprendizagem básica sobre *Deep Learning* e familiarização com as ferramentas utilizadas, sobretudo as com suporte para GPU
- **Obtenção de imagens:** *download* de grande quantidade de imagens da *internet* e marcação manual dessas imagens para que possam ser conferidos os resultados de testes do desempenho de algoritmos de separação
- **Treinamento de classificadores:** utilização do *dataset* baixado da *internet* e das marcações da etapa anterior para testar performance de algoritmos supervisionados assumindo que a fase anterior funcionou perfeitamente (para isso foram usadas as marcações)
- **Detecção usando *sliding windows*:** essa parte não chegou a ser implementada, mas é feita uma discussão de como seria a implementação dos detectores baseados em *sliding windows* criados a partir dos classificadores obtidos nas etapas anteriores
- **Desenvolvimento da interface *web*:** outra parte que não chegou a ser implementada, mas se refere à união de todas as etapas das fases anteriores e possibilidade de uso por meio de um servidor *web* com as telas conforme descrito anteriormente

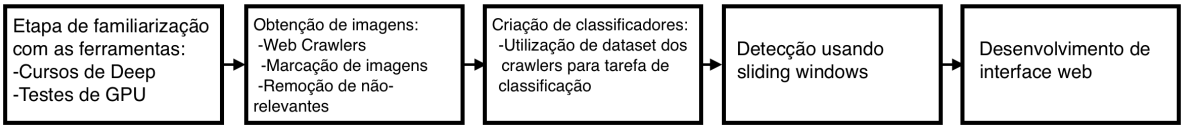


Figura 4.1: Esquemático resumido das etapas do projeto

4.2 Testes e familiarização com as ferramentas

4.2.1 Material *online* de Deep Learning

Antes de se iniciar o projeto, foi consultado material *online* sobre Deep Learning. Particularmente, o curso Deep Learning, da plataforma Udacity, criado por Vincent Vanhoucke e Arpan Chakraborty [3] foi utilizado. Boa parte do material desse curso foi resumida na seção de introdução teórica sobre Deep Learning.

4.2.2 Testes de GPU

Como o projeto faz uso de *datasets* grandes, sua realização em CPU não é viável. Por isso, foi necessária uma familiarização inicial com GPU. O curso citado acima fazia uso da plataforma Tensorflow, desenvolvida para a linguagem de programação Python, essa plataforma possui suporte para GPU, embora isso não tenha sido abordado no curso. Para testar o suporte a GPU do Tensorflow, foram realizados testes utilizando uma versão modificada do *dataset* MNIST[55], que, por sua vez, é um *dataset* popular usado para reconhecimento de dígitos manuscritos. No curso, esse *dataset* é referenciado como NotMNIST. Trata-se de um conjunto de imagens inicialmente grande, mas de onde foram tomados *subsets* de forma a gerar $20,000$ imagens de treino, $10,000$ de validação e $10,000$ de teste, sendo que todas elas têm tamanho 28×28 . O propósito desse teste é analisar o desempenho de uma Rede Neural Convolucional com Inception Module em CPU (duas CPUs diferentes foram usadas), GPU e duas GPUs. A Figura 4.2 ilustra alguns exemplos colhidos desse *dataset*.

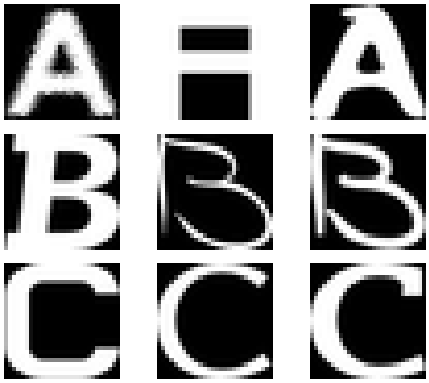


Figura 4.2: Exemplos do dataset NotMNIST

Arquitetura da Rede

A rede utilizada é uma rede convolucional com módulo Inception. Sua arquitetura é ilustrada na Figura 4.3, sendo que os retângulos de mesma cor indicam que os mesmos *kernels* foram usados nessas partes, o modelo tem um total de $124,018$ parâmetros. A configuração utilizada nos módulos Inception segue apenas o princípio de combinar convoluções diferentes para obter resultado melhor do que na rede convolucional normal. Nesse código, as convoluções foram escolhidas de forma arbitrária. A precisão da classificação no *test set* após todo o treinamento varia entre 96.3% e 96.7%, dependendo da inicialização dos parâmetros, que é aleatória. Além do que foi colocado, vale ressaltar também que foi utilizado *dropout* para evitar o efeito do *overfitting*, sendo que ele foi usado nas camadas *fully connected* com taxa de 0.8.

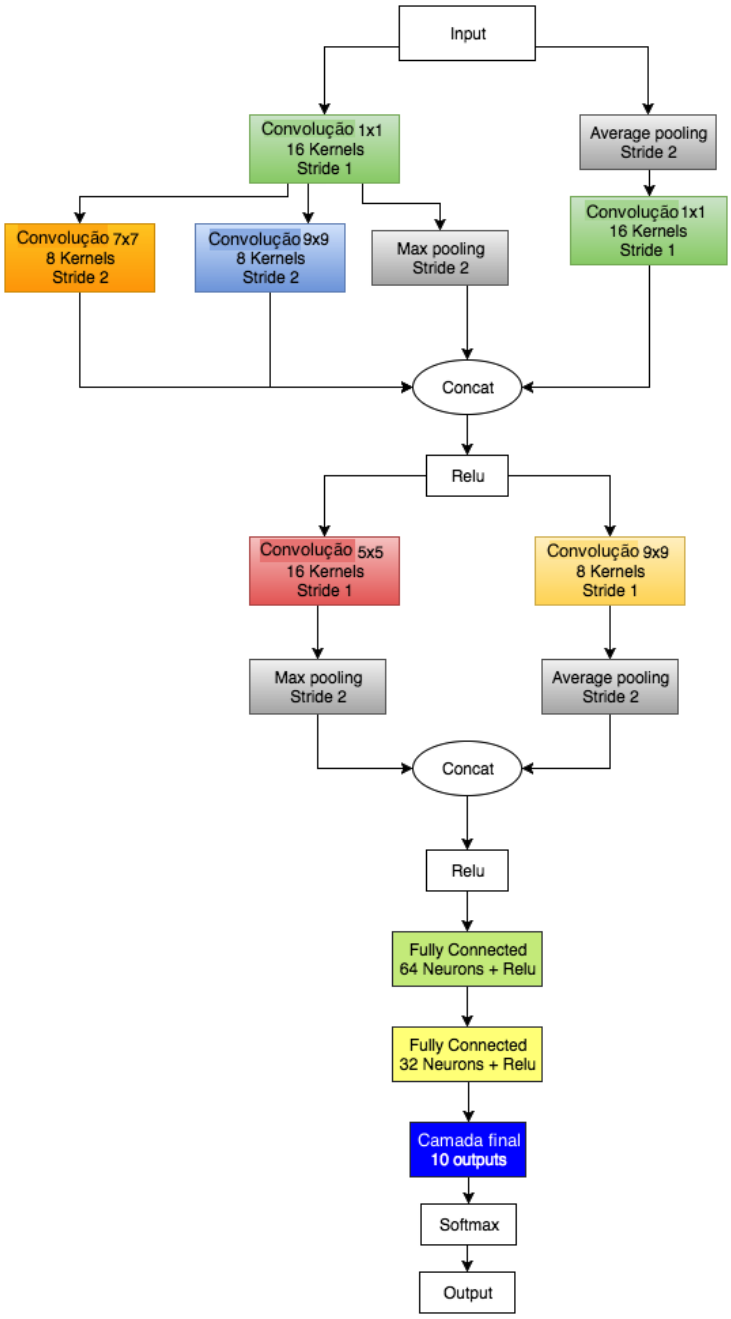


Figura 4.3: Rede neural utilizada

Modelo de treinamento Multi GPU

Nessa etapa, o código foi executado em diversos sistemas computacionais, sendo que um deles fazia uso de duas GPUs. Para rodar usando duas GPUs foi necessário fazer algumas modificações no código, o funcionamento básico é ilustrado na Figura 4.4: os valores das variáveis são passados ao mesmo tempo para as duas GPUs, cada uma roda um *batch* diferente do Stochastic Gradient Descent e computa os gradientes da função de custo

montada com os *batches* passados à GPU em questão. Em seguida, esses gradientes são repassados à CPU, que computa a média deles e atualiza as variáveis correspondentes.

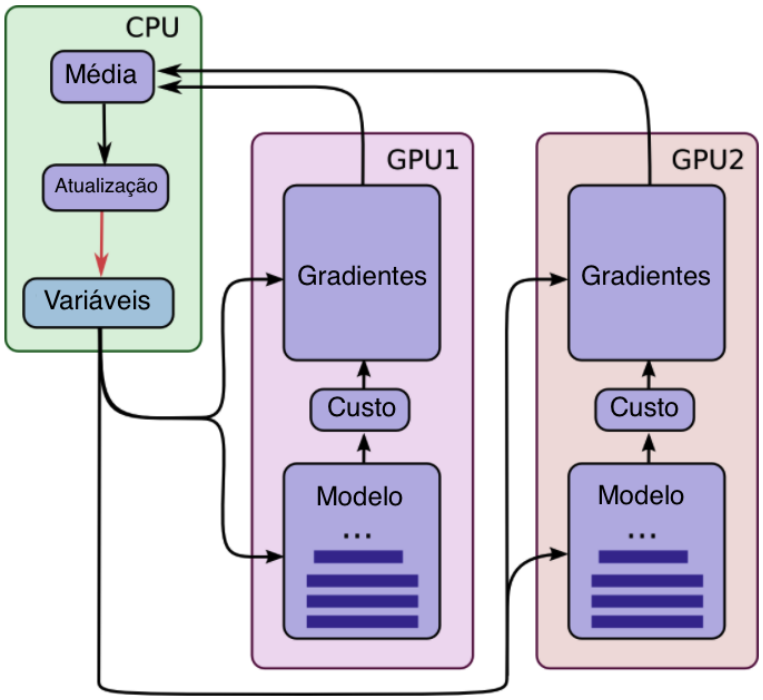


Figura 4.4: Funcionamento da execução em duas GPUs (Fonte[19])

Resultados

Em todos os ambientes de teste foi utilizada a ferramenta Tensorflow na linguagem de programação Python. O treinamento foi feito usando *Stochastic Gradient Descent*, com $20,000$ *batches* de 128 exemplos. A Tabela 4.1 resume os resultados.

Tabela 4.1: Tabela com resultados do requisito 1

Sistema	*Acc*	*Tempo*	$N_{batches}$
Comp Portátil	$96,7\%$	4h 24min 25s	20.000
CPU	$96,3\%$	2h 56min 55s	20.000
GPU	$96,4\%$	4min e 47s	20.000
MultiGPU #1	$96,2\%$	2min 33s	20.000
MultiGPU #2	$96,5\%$	3min 34s	28.000

Computador portátil

O primeiro ambiente de teste utilizado foi uma CPU de MacBook, mais especificamente um Intel Core I7 de quatro *cores*, 16GB de RAM DDR3L e rodando a 2.5GHz. O tempo

de execução do código foi de 15865 segundos, o que equivale a 4h 24min 25s e a precisão foi de 96.7%. Nesse computador o Tensorflow estava instalado sem suporte a GPU.

Desktop

O segundo ambiente de teste foi a CPU de um Desktop. Nesse caso, trata-se de um i7-6700 de quatro *cores*, memória RAM DDR4 de 16GB, rodando a 3.4GHz. O tempo de execução agora foi de 10615 segundos, o que equivale a 2h 56min 55s e a precisão foi de 96.3%. Nesse caso, o Desktop estava com o Tensorflow instalado com suporte a GPU, para forçar o programa a usar apenas a CPU, foi necessário declarar essa restrição especificamente no código.

GPU

O terceiro ambiente de teste foi o mesmo Desktop do ambiente anterior, porém agora foi utilizada uma GPU. O modelo da GPU é Nvidia GTX 1080, com RAM interna de 8GB GDDR5x. O tempo de execução foi de 287 segundos, o que equivale a 4min e 47s e a precisão foi de 96.4%. Nesse caso, o Tensorflow atribui as operações à GPU sempre que possível.

Multi GPU

O tempo de execução para rodar os $20,000$ batches foi de 153 segundos, ou 2min 33s, porém verificou-se que a precisão no *test set* foi de 96.2%, valor inferior ao esperado à priori. Nesse caso, não se trata de problema na inicialização, pois o modelo foi rodado algumas vezes, produzindo sempre resultados na faixa de 96 a 96.2%, essa leve inferioridade também foi verificada no set de validação. Vale ressaltar que essa preocupação na precisão não foi enfatizada no teste de CPU que obteve precisão de 96.3% pois naquele caso o código era o mesmo dos exemplos anteriores, ao contrário do caso atual, em que o código sofreu várias modificações.

Para entender porque o mesmo número de *batches* produziu resultados levemente inferiores nesse treinamento, voltamos ao modelo da Figura 4.4 e consideramos 2 *batches* diferentes, aqui chamados A e B. No treinamento convencional, o gradiente é calculado inicialmente usando os exemplos de A, as variáveis são atualizadas e, em seguida, o gradiente é recalculado usando os exemplos de B. Nesse caso, o cálculo do gradiente usando os exemplos de B teve acesso a valores de variáveis atualizados após a computação usando os exemplos de A. No caso do treinamento usando duas GPUs, os *batches* A e B são processados simultaneamente, ou seja, a computação dos gradientes nesses dois casos é feita

com base nas mesmas variáveis iniciais. Isso faz com que o treinamento seja levemente inferior ao sequencial.

Apesar disso, a adição de mais alguns passos de treinamento torna possível alcançar a precisão anterior. Nesse caso, o número de *batches* usado foi aumentado de $20,000$ para $28,000$, valor verificado empiricamente como adequado. A precisão obtida foi de 96.5% num tempo de 214 segundos, ou 3min 34s, esse tempo ainda é bastante inferior ao treinamento usando uma GPU (em termos percentuais).

4.3 Obtenção de imagens

O foco dessa etapa é o desenvolvimento de um sistema que recebe uma quantidade de imagens relativas à determinada *query* e descarta as imagens que não são relevantes à busca, permitindo assim que sejam treinados classificadores na etapa seguinte.

4.3.1 *Web Crawlers*

Como o foco do projeto é a automatização de geração de *datasets*, o primeiro passo para que testes possam ser realizados é o *download* de grande número de imagens da Internet. Para que esse *download* fosse possível, foi utilizada a ferramenta Selenium com a linguagem de programação Python. Essa ferramenta recebe como entrada o código HTML da página renderizada e permite realizar ações como deslizar até o final da página, apertar o botão cujo nome do identificador HTML (id ou classe) é dado pelo programador, esperar até que o código carregue por completo (para casos em que algumas imagens ainda estão sendo carregadas). Através dessa navegação automática, é possível fazer *download* das imagens retornadas por ferramentas de busca. A desvantagem desse tipo de abordagem é que qualquer atualização no formato do HTML da ferramenta de pesquisa faz com que sejam necessárias manutenções no código dos *crawlers*. Por exemplo, caso a Google decida que o *link* para a imagem, que antes ficava dentro de determinado * * passe a ser um atributo JSON do conteúdo de outro elemento HTML, essa alteração deverá ser percebida pelo programador do código e a atualização deve ser feita no *crawler* também. Em suma, um código que funciona perfeitamente em determinado mês, pode (e normalmente vai) parar de funcionar dentro de alguns meses.

No escopo desse artigo, foi baixado um *dataset* inicial contendo as 10 classes do CIFAR-10 como *search queries* e essas imagens foram utilizadas para realizar os testes detalhados nas seções seguintes. Caso a ferramenta *web* proposta anteriormente fosse implementada, a manutenção com base nas atualizações discutidas no parágrafo anterior precisaria ser feita. Inicialmente, a ideia era utilizar APIs de busca, em que o programador digita a palavra desejada, uma requisição HTML é feita a essas APIs e um conjunto de imagens

são retornadas. Essa abordagem é mais simples do que programar *web crawlers*, porém, a ideia do projeto era retornar o maior número de imagens possível. O CIFAR-10 possui 10 classes com 6000 imagens cada, totalizando 60.000 imagens. Para competir com esse número, uma grande quantidade de imagens deveria ser retornada. Porém, essas APIs são pagas e o preço necessário para retornar grande quantidade delas é alto. As APIs testadas foram as do Bing, Google e Pixabay.

As ferramentas de busca utilizadas para navegação dos *crawlers* foram as seguintes:

- **Google:** a busca de imagens utilizando essa ferramenta retornava cerca de 800 imagens por *query* na língua inglesa (às vezes um pouco menos). Para aumentar esse número, foi utilizada a tradução da palavra informada para outras línguas (japonês, espanhol, holandês, russo, italiano, alemão, francês, coreano, árabe e chinês) e busca nessas línguas também. O número de imagens obtidas passa a ser então $11 \times 800 = 8800$ (lembrando que muitas dessas imagens são imagens não-relevantes que são retornadas junto com as relevantes). A pressuposição é que imagens obtidas em determinado país e postadas com o nome na língua desse país muitas vezes são diferentes das imagens postadas em páginas de língua inglesa. Claro que essa premissa nem sempre é verdade e algumas imagens serão repetidas. Porém, esse efeito é ignorado, já que, a princípio, não prejudica de maneira significativa o desempenho dos algoritmos

- **Bing:** também retornava cerca de 800 imagens em inglês e, assim como no Google, foi utilizada a tradução para outras línguas para que mais imagens pudessem ser obtidas. Como a limitação em 800 resultados é algo imposto pela ferramenta e o algoritmo utilizado para obtê-los é diferente do Google, assume-se que muitas imagens serão diferentes entre essas ferramentas.

- **Flickr:** em geral, as imagens retornadas por essa ferramenta eram de maior qualidade e bem diferentes do retorno das ferramentas anteriores. Além disso, para as *queries* testadas, a maioria das imagens era relevante. Eram retornadas cerca de 2500 imagens para cada pesquisa e, por ser uma ferramenta com pequena quantidade de resultados para outras línguas, apenas o inglês foi utilizado.

- **Yandex:** as imagens dessa ferramenta eram de qualidade similar às do Flickr e no geral eram retornadas 1500 imagens por busca. Apenas o inglês foi utilizado.

Os códigos dos *crawlers* em geral, funcionavam da seguinte forma: baseado no HTML inicial, eram utilizados os *links* das imagens para que fossem feitos *downloads* delas, em seguida, o navegador deslizava até o fim da página, clicava no botão "visualizar mais imagens" (caso houvesse) e esperava que mais imagens fossem carregadas. O processo se

repetia até que o fim dos resultados fosse alcançado. Baseado nas informações fornecidas anteriormente, o total de imagens deveria ser $(8800 + 8800 + 2500 + 1500) \times 10 = 216.000$. Na prática, foram retornadas 222.073 imagens, totalizando $2,9GB$. Vale ressaltar que foram feitos *downloads* dos *thumbnails*, ou seja, das imagens reduzidas que são exibidas na tela de busca. As imagens em tamanho original têm tamanho muito grande e é inviável utilizá-las em redes neurais.

4.3.2 Marcação de imagens

Para que seja possível avaliar se os algoritmos de separação por relevância que serão testados funcionam de fato, as imagens retornadas pelas *queries* da etapa anterior devem ser marcadas de acordo com a relevância. Dessa forma, qualquer separação entre imagens relevantes ou não feita pelos algoritmos poderá ser validada tomando as marcações como *ground truth*. Existem serviços *online* em que é possível pagar uma quantia de dinheiro para contratar pessoas que fazem marcações desse tipo, como o Amazon Mechanical Turk, mas, como o preço é em dólar, a utilização de tais serviços torna-se cara, principalmente se for levado em consideração o número de imagens disponíveis.

Com essa limitação em mente, foi programada uma interface gráfica utilizando a ferramenta TKinter, disponível para Python. Nessa interface, o marcador pode escolher se a imagem é relevante, irrelevante ou, caso o marcador não esteja certo se a imagem é relevante ou não, como, por exemplo, para a classe "avião", pode ser que a imagem retornada seja um desenho de avião, ou a parte interna de um avião, ou uma foto de uma revista onde são representados alguns aviões. O caso de desenho e parte interna foram considerados como não-relevantes, a revista é considerada como "incerto", pois é sabido que alguns algoritmos baseados em redes neurais utilizam o contexto da imagem para escolher a classe à qual ela pertence. A Figura 4.5 mostra mais algumas imagens que foram marcadas como "incertas".

Figura 4.5: Exemplos de imagens cuja relevância é de difícil definição

De maneira geral, não é tão simples definir a relevância de uma imagem, já que ela depende da aplicação. Esse foi o motivo por se ter optado pela proposição do sistema com marcação manual de poucas imagens como base para que o sistema aprenda o que

é relevante, que não deve ser confundida com a marcação que está sendo descrita nesse capítulo: a primeira se refere ao *feedback* de quem está usando o sistema assim que as imagens são obtidas para definir o que é relevância e a segunda se refere a marcações experimentais feitas para registro de porcentagens.

De maneira geral, nessa etapa de marcação de relevância das 10 classes do CIFAR-10, para se definir a relevância de uma imagem foi usada uma filosofia básica: existe um padrão de como as imagens normalmente se apresentam no CIFAR-10, imagens que claramente seguem esse padrão são relevantes, imagens que representam objetos completamente diferentes do procurado são irrelevantes, imagens que não seguem exatamente esse padrão, mas que ainda assim representam o objeto em questão são marcadas como "incertas". Nos experimentos das etapas posteriores, as imagens marcadas como "incertas" foram descartadas para simplificação da análise. Essa decisão tem como motivação o isolamento de fatores de interferência nos resultados. Sendo realizados estudos sem essas imagens e, em seguida, com elas, é necessário apontar que a causa específica de determinado efeito foi a presença delas. Caso o experimento fosse feito usando as imagens incertas e se obtivessem resultados insatisfatórios por causa delas, não seria possível apontar que a causa específica desse resultado foi a presença dessas imagens.

Para que fosse possível marcar essa grande quantidade de imagens (mais de 200 mil), foi necessário o desenvolvimento de uma interface onde o usuário não precisasse clicar nas imagens uma de cada vez. O resultado ficou como ilustrado na Figura 4.6 e funciona da seguinte forma: as imagens começam todas com a margem roxa, simbolizando que ainda não foram visualizadas, e com alguma marcação pré-definida (todas começam como relevantes, mas a marcação padrão para as imagens não vistas pode ser alterada caso as imagens carregadas naquele momento estejam, de forma geral, pertencentes a outra classe), o usuário vai descendo e visualizando mais imagens, quando ele clicar em alguma, a classe da imagem muda, de forma que ele pode escolher a classe da imagem clicada e, no momento do clique, a margem de todas as imagens acima e à esquerda da clicada somem, indicando que agora elas foram visualizadas. A ideia é que, ao clicar, o usuário não está apenas definindo a classe da imagem que está sendo clicada, mas também confirmando a classe de todas as imagens que vieram antes. O motivo disso é que muitas imagens têm classe igual: as primeiras imagens carregadas costumam ser todas relevantes e as últimas costumam ser irrelevantes ou indefinidas. Dessa forma, o usuário pode apenas olhar rapidamente e confirmar a classe de cada uma delas clicando apenas em uma.

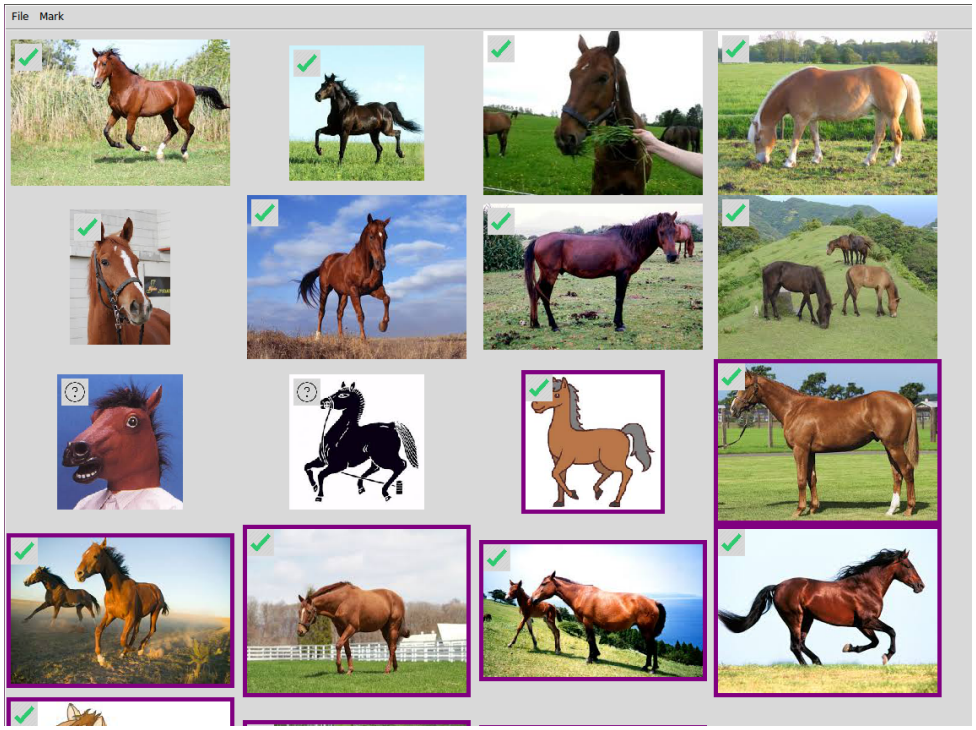


Figura 4.6: Interface para marcação de imagens

Com a interface feita dessa forma, a marcação de imagens se tornou possível. Mesmo assim não foram marcadas todas as imagens, pois essa tarefa tomaria muito tempo. Para cada classe foram escolhidas:

- Todas as imagens do Google em inglês
- Todas as imagens do Bing em inglês
- Todas as imagens do Yandex
- Todas as imagens do Flickr

O total de imagens marcadas foi de 56.994.

4.3.3 Qualidade das imagens de cada *crawler*

Além dos estudos propostos, foi feita uma análise quantitativa acerca das imagens fornecidas por cada *crawler*. Dessa forma, fica possível quantificar quais deles foram mais importantes para a geração dos dados utilizados.

Qualidade das imagens agrupadas por *crawler*

O gráfico da Figura 4.7 ilustra quantas imagens boas, ruins e incertas foram marcadas para cada um dos *crawlers*, sendo que as imagens boas são representadas em verde, as ruins

são representadas em vermelho e as incertas são representadas em amarelo. Conforme citado anteriormente, Flickr e Yandex forneceram as melhores imagens, sendo as do Flickr próximas de 25 mil e as do *Yandex* próximas de 15 mil. Vale ressaltar que, embora Google e Bing tenham fornecido menos imagens, essas ferramentas retornam uma quantidade de imagens para *queries* em outras línguas similar ao retornado para inglês, mas essas imagens não puderam ser exploradas devido à falta de disponibilidade de tempo para a marcação de todas essas imagens (foram marcadas apenas 56.994 das 222.073 fornecidas).

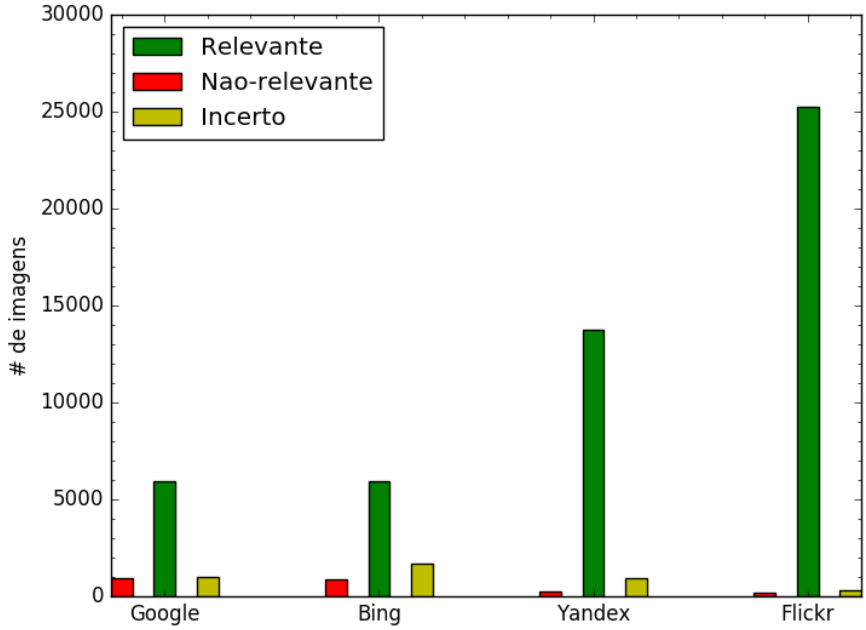


Figura 4.7: Imagens agrupadas por qualidade e separadas por *crawler*

Qualidade das imagens para cada *crawler* agrupadas por classe

A Figura 4.8 permite uma vizualização com agrupamento por classe para cada *crawler*.

Para Yandex e Flickr, a qualidade das imagens foi igual entre as classes, com exceção da classe sapo para o Yandex, em que uma grande quantidade de imagens foi incerta. Para Bing e Google, em geral a maioria das imagens foi de boa qualidade, sendo que as exceções parecem ser de natureza aleatória: pássaro, gato, veado, cachorro, cavalo e caminhão tiveram mais qualidade, Bing teve dificuldade com avião, sapo e navio, já Google, teve dificuldade com automóvel e avião. Em particular, a *query* navio para o Bing retornou um número muito menor de imagens em relação às outras *queries*: foram retornadas menos de 300 imagens. O motivo disso provavelmente foi alguma falha na execução do *crawler* que não foi detectada anteriormente, como por exemplo alguma

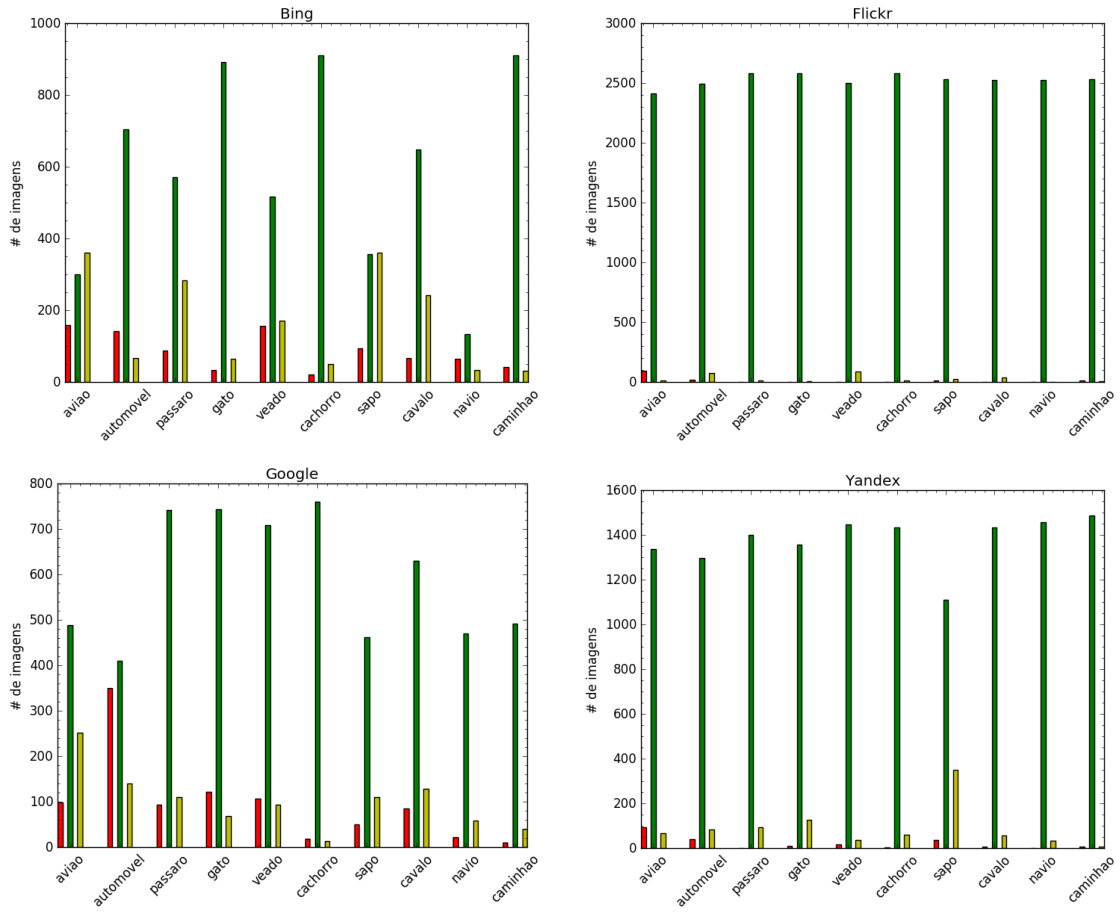


Figura 4.8: Imagens agrupadas por qualidade e separadas por classe para cada *crawler*

falha de conexão. O ideal executar novamente o *crawler* para essa *query* e idioma, mas o erro foi detectado muito tarde.

Qualidade das imagens para cada classe agrupadas por *crawler*

A Figura 4.9 ilustra, para cada classe, a quantidade de imagens agrupada por qualidade e separada por *crawler*. No geral, podemos ver que as classes avião e automóvel parecem ser as que aparecem com maior quantidade de imagens ruins, sendo que, para automóvel, essa quantidade de imagens ruins se deve às imagens fornecidas pelo Google. Também é possível ver que o Bing e o Google, em geral, retornam o mesmo tanto de imagens boas (em algumas classes o Google retorna mais, em outras, é superado pelo Bing). Assim como nos outros gráficos, Flickr é responsável pela maior quantidade de imagens de boa qualidade, seguido por Yandex.

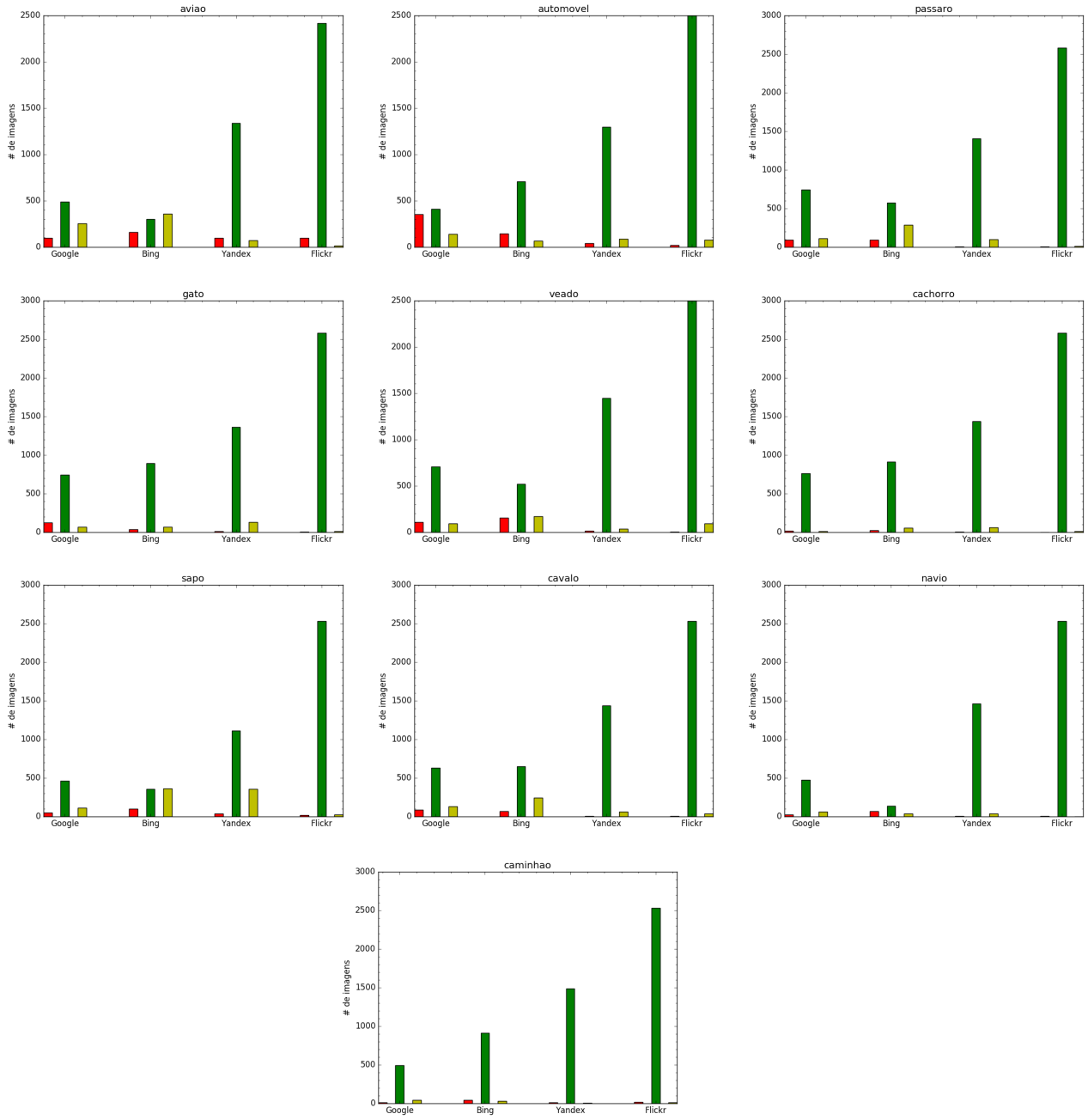


Figura 4.9: Imagens agrupadas por qualidade e separadas por *crawler* para cada classe

4.3.4 Separação entre relevantes e não-relevantes

A próxima etapa do trabalho consiste em testar métodos que separem as imagens consideradas relevantes das não-relevantes e é aqui descrita resumidamente e detalhada em seguida. Para esses experimentos, apenas as imagens da classe "avião" foram usadas. São fornecidos ao classificador alguns exemplos pré-marcados como *training set*, sendo que a marcação é 0 para irrelevante e 1 para relevante (as incertas foram descartadas). A

partir desses poucos exemplos, o sistema deve conseguir generalizar a informação para dizer quais imagens são relevantes ou não no conjunto inteiro.

Foram testadas as Triplet Networks, devido à hipótese de que redes de *one-shot learning* funcionam bem nesse caso, as redes convolucionais tradicionais, para comparação com as Triplet e também foram testadas as NASNets com *transfer learning* devido a adequação de *transfer learning* para esse contexto e ao fato de as NASNets terem os melhores resultados da atualidade.

Descrição do experimento

Para essa parte, foi preferido focar apenas em uma das classes do CIFAR-10, em vez de se utilizar as 10. Foi escolhida aleatoriamente a classe avião e foram usadas todas as suas imagens (todos os *crawlers* e todos os idiomas). Como as marcações de idiomas diferentes do inglês não haviam sido feitas, foram realizadas marcações extras exclusivamente para essa etapa. Ao final, as marcações totalizaram 18.684 imagens (5.825 não relevantes e 12.859 relevantes), sendo que as incertas foram descartadas para simplificação do experimento.

Para o problema em questão, as classes usadas são "não relevante" (0) ou "relevante" (1), e os tamanhos dos *sets* de treino e validação são variados. Especificamente, são testados os tamanhos treino-validação a seguir: 20-6, 100-30, 200-60, 400-120, em que os exemplos são distribuídos igualmente entre as classes, ou seja, 20-6, por exemplo, corresponde a 10 exemplos relevantes para treino, 10 não relevantes para treino, 3 relevantes para validação e 3 não relevantes para validação. Para cada divisão dessas, o resto das 18.684 imagens é usado para teste.

Foram usados 3 modelos de rede neural nessa etapa: Resnet[8], Triplet[10] e NASNet[11]. A Resnet utilizada tem uma camada inicial com 1 convolução, seguida por 9 Relu-blocks (com 2 convoluções cada), *global average pooling* e a camada Softmax que mapeia para a saída (20 camadas ao total), conforme ilustrado na Tabela 4.2. É usada entropia cruzada como *loss function*. Já a Triplet Network usa o mesmo modelo da Resnet, mas em vez da Softmax é feito um mapeamento para *embeddings* de 128 dimensões e normalização, sendo que esses *embeddings* são usados na função de custo. A classificação é feita com base em KNN. A NASNet utilizada é a padrão do Tensorflow-HUB, disponibilizada pela Google e pré-treinada no Imagenet, apenas a última camada é retreinada no *dataset* dessa etapa. Para a Resnet e Triplet, as imagens foram redimensionadas para 100×100 e para a NASNet, foi usado o tamanho padrão de 331×331

Tabela 4.2: Resnet utilizada para separação de relevantes

Layer	Conv type	Feature maps	Pooling
Initial	5×5	64	Sim (após conv)
Relu block 1	3×3	64	Não
Relu block 2	3×3	128	Sim (pré conv)
Relu block 3	3×3	128	Não
Relu block 4	3×3	256	Sim
Relu block 5	3×3	256	Não
Relu block 6	3×3	256	Não
Relu block 7	3×3	256	Não
Relu block 8	3×3	512	Sim
Relu block 9	3×3	512	Não
Global avg pooling	-	512	-
Softmax layer	-	-	-

Resultados

Os resultados são sumarizados na Tabela 4.3 e representados graficamente em 4.10. Nessas ilustrações fica claro que o uso da Triplet não contribuiu para melhora dos resultados, ficando bem similar ao da Resnet comum. Além disso, fica evidente a limitação dos métodos de Deep Learning sem *transfer learning* para casos em que poucos dados estão disponíveis.

Tabela 4.3: Acurácia obtida na remoção de não-relevantes por tamanho do training set

Mod\Tamanho	20	100	200	400
Triplet	60,2%	73,8%	76,5%	81,3%
Resnet	67,1	67,8%	74,2%	80,3%
NASNet	91,5	92,1%	**94,4%**	93.8%

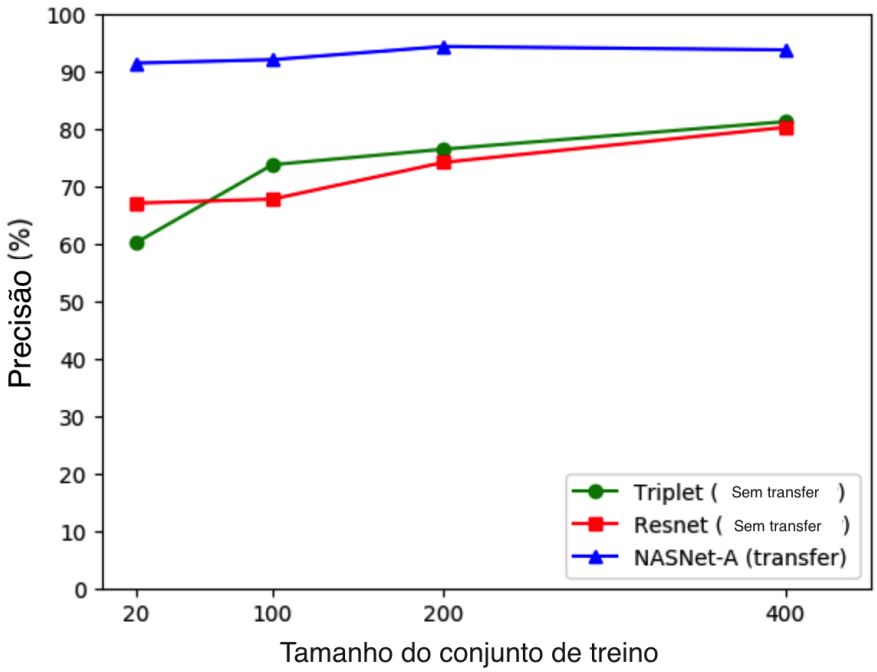


Figura 4.10: Acurácia obtida na remoção de não-relevantes

Para a NASNet treinada com 200 exemplos, em que o sistema obteve os melhores resultados, a tabela 4.4 mostra a matriz de confusão e a Figura 4.11 ilustra alguns exemplos classificados erroneamente pelo modelo.

Tabela 4.4: Matriz de confusão para NASNet com training set de 200 exemplos

Original\Pred	Não-relevante	Relevante
Não-relevante	5.388	307
Relevante	720	12.009

(a) Relevantes classificadas incorretamente

(b) Não-relevantes classificadas incorretamente

Figura 4.11: Erros do classificador

Pela matriz de confusão, podemos imaginar a seguinte situação no sistema descrito:

1. O usuário digita a *query* "avião"
2. Os *crawlers* fazem *download* de 18.684 imagens: 5.825 não-relevantes e 12.859 relevantes (vale relembrar que as incertas foram removidas para simplificar o experimento)
3. O sistema diz que 12.316 imagens são relevantes
4. Nessa marcação, o sistema julga corretamente que 12.009 são relevantes, mas deixa de incluir 720 que deveriam ter sido colocadas como relevantes e inclui 307 que não deveriam ter sido colocadas como relevantes

O modelo deixaria de incluir 720 imagens de avião no *dataset* de saída e teria incluído 307 imagens falsas, sendo esse segundo número mais danoso à aplicação. Percentualmente, essa quantidade de imagens não relevantes que foram aceitas é de 2,4% de um *dataset* que teria 12.316 exemplos, o classificador teria que ser robusto a pelo menos essa porcentagem de erro. Além disso, podemos observar na figura que as imagens relevantes classificadas incorretamente correspondem a ângulos ou situações incomuns, como o avião dentro da água ou um aeroporto em que os aviões ficam representados como miniatura, sendo discutível a marcação desse último como relevante em vez de incerto pelo critério apresentado.

4.4 Treinamento de classificadores

A fase de treinamento de classificadores tem como objetivo verificar se, dado que a fase anterior foi concluída com sucesso, as imagens disponíveis são boas o bastante para treinar classificadores da maneira como foram retornadas, ou seja, se a qualidade das imagens obtidas sem qualquer tipo de controle se adequa à tarefa em questão. Essa verificação é feita pois, muitas vezes, *datasets* já prontos possuem algum tipo de tratamento ideal: as imagens normalmente são do mesmo tamanho e os objetos já vêm marcados com *bounding boxes* definidas pela pessoa que os marcou, de forma que apenas o objeto de interesse aparece na imagem em questão. Em imagens cuja obtenção foi automatizada não há esse tipo de controle, já que não há marcador humano garantindo isso.

Para simplificar o experimento, não foram feitos classificadores *one-vs-all* como no sistema proposto, em vez disso, foi feito um classificador para diferenciar entre as 10 classes do CIFAR-10, sendo que esse classificador usa as imagens dos *crawlers* como *training set*. Foram realizados realizados experimentos usando dois *test sets* de naturezas distintas, sendo um deles o próprio *test set* do CIFAR-10 e o outro formado com imagens dos *crawlers*. O motivo disso é que há uma grande diferença de escala entre o conjunto de imagens dos *crawlers* e as imagens do CIFAR-10, que têm tamanho reduzido. Essa diferença de escala pode prejudicar os resultados, por isso foi feito esse segundo experimento que faz uso das próprias imagens dos *crawlers* e remove assim essa dificuldade.

Quanto aos classificadores usados, foram uma Resnet pequena sem pré-treinamento, devido à popularidade do modelo, uma Resnet grande sem pré-treinamento, permitindo a avaliação do efeito das camadas extras e uma API de classificação *black box*: Clarifai. O motivo de se ter feito uso desse sistema é o fato de ser uma aplicação profissional, garantindo assim que os resultados sejam os melhores possíveis e dispensando a necessidade de *parameter tuning* e *debug* de redes neurais. As duas Resnets foram utilizadas tomando o CIFAR-10 como *test set*, enquanto o Clarifai foi testado nas próprias imagens dos *crawlers*.

4.4.1 Preprocessamento das imagens

Para que as imagens pudessem ser usadas como entrada numa rede neural convolucional, é necessário que elas tenham tamanho único. A princípio havia se pensado em resolver isso a partir de *cropping* e *padding*, sendo que, caso a imagem tivesse uma das dimensões muito maior que a outra, seria feito *padding* para que as dimensões ficassem iguais. Caso uma dimensão fosse levemente maior do que a outra, seria executado *cropping* de maneira a deixar as dimensões iguais. Essa abordagem tem a vantagem de não distorcer as imagens de entrada, mas, por outro lado, tem um alto risco de retirar grande parte do objeto,

fazendo com que deixe de pertencer à classe original e passe a atrapalhar o classificador. Esse efeito pode ser verificado na Figura 4.12, em que grande parte do avião é perdida caso seja realizado *cropping*.

Figura 4.12: Imagem que prejudicaria o classificador caso fosse utilizado *cropping*

É realizada uma alteração da razão $\frac{altura}{largura}$ de forma que as dimensões fiquem iguais. Foi feito um banco de imagens em MongoDB onde elas ficaram armazenadas com um tamanho padrão de 150×150.

4.4.2 Transformações aplicadas nas imagens

Assim como na explicação teórica, a abordagem de classificação usando redes neurais aqui empregada parte de uma rede relativamente grande, ou seja, bastante propensa a *overfitting*, sendo usados métodos para controlar esse efeito. Um desses métodos é a aplicação de transformações aleatoriamente às imagens de entrada, fazendo com que as representações internas das imagens nas camadas da rede sejam robustas a essas transformações. A maioria das transformações são disponibilizadas no próprio Tensorflow, sendo que são aplicadas no momento em que os exemplos do *batch* são capturados para que o modelo os processe, de forma que o mesmo exemplo passa por transformações diferentes dependendo da época em que é capturado (época é o nome dado ao conjunto de iterações necessário para que todo o *set* seja processado uma vez).

As transformações e probabilidades de ocorrência são descritas a seguir:

- **Espelhamento vertical:** com uma probabilidade pré-definida, os lados direito e esquerdo da imagem são invertidos
- **Rotação:** com uma probabilidade pré-definida, a imagem é rotacionada. O ângulo de rotação da imagem é definido aleatoriamente e varia entre $-45°$ e $45°$
- **Brilho:** com uma probabilidade pré-definida, é aplicado brilho com intensidade aleatória à imagem de entrada
- **Contraste:** com uma probabilidade pré-definida, é aplicado contraste com intensidade aleatória à imagem de entrada

- ***Hue*:** com uma probabilidade pré-definida, é alterado o componente H da representação HSV da imagem. Isso tem o efeito de alterar as cores da imagem em questão
- **Saturação:** com uma probabilidade pré-definida, é alterada a saturação da imagem, sendo que a intensidade da alteração é aleatória
- ***Zoom*:** com uma probabilidade pré-definida, é aplicado *zoom* na imagem de entrada. Esse efeito é obtido em duas etapas: primeiro, há aumento ou diminuição da imagem em escala aleatória, em seguida, o tamanho original é recuperado com aplicação de *cropping*, para *zoom in*, ou *padding*, para *zoom out*. A escala de *zoom* varia num intervalo pequeno, sendo que os valores foram escolhidos de forma a minimizar possibilidade de perda da informação da imagem, por isso não os efeitos de aproximação/afastamento não são muito intensos.
- ***Skew em X*:**, com uma probabilidade pré-definida, é aplicada uma pequena distorção horizontal nas imagens
- ***Skew em Y*:**, com uma probabilidade pré-definida, é aplicada uma pequena distorção vertical nas imagens
- ***Translação*:**, com uma probabilidade pré-definida, é aplicada uma translação vertical e horizontal com componentes pequenos e aleatórios nas imagens

Após essas transformações a imagem é normalizada antes de ser fornecida à rede. É questionável se algumas dessas transformações, como o brilho e o contraste fariam alguma diferença, já que são seguidas pela normalização. Porém, a verificação lado-a-lado de reconstrução das imagens com e sem aplicação desses efeitos (em que a normalização é remapeada para o intervalo $[0, 255]$) indica que há diferença entre as imagens.

4.4.3 *Deep convolutional network*

Essa parte do projeto consistiu na aplicação de *Deep Neural Networks* convolucionais no *dataset* obtido através dos *crawlers*, sendo que foram usadas apenas as imagens classificadas como relevantes na etapa de marcação manual. Ou seja, assume-se que a etapa de remoção de imagens não-relevantes funcionou perfeitamente. Essa pressuposição foi feita para evitar que esses experimentos fiquem impedidos de serem executados até que os anteriores fossem concluídos. O conjunto de imagens disponíveis consiste em um total de 50.887 imagens e elas foram divididas em 40.887 imagens de treino de 10.000 de validação.

Para que a qualidade dessas imagens pudesse ser avaliada de fato, decidiu-se por tomar o *test set* como um subconjunto de 10.000 imagens obtidas aleatoriamente do CIFAR-10.

Como no CIFAR-10 as imagens são de tamanho 32×32, foi utilizada a interpolação de Lanczos[56] para reduzir as imagens dos *crawlers* ao tamanho adequado mantendo o máximo de informação possível. Em seguida, essas imagens foram utilizadas como entrada de uma Resnet[37][9] com formato descrito na Tabela 4.5, que será referida como SmallerResnet. Vale ressaltar que essa tabela apenas resume os parâmetros, sendo que a sequência exata de operações realizadas foi feita como em [9].

Tabela 4.5: SmallerResnet

Layer	Conv type	Feature maps	Pooling
Initial	5×5	64	Não
Relu block 1	3×3	64	Não
Relu block 2	3×3	64	Não
Relu block 3	3×3	64	Não
Relu block 4	3×3	64	Não
Relu block 5	3×3	64	Não
Relu block 6	3×3	128	Sim
Relu block 7	3×3	128	Não
Relu block 8	3×3	128	Não
Relu block 9	3×3	128	Não
Relu block 10	3×3	128	Não
Relu block 11	3×3	256	Sim
Relu block 12	3×3	256	Não
Relu block 13	3×3	512	Sim
Relu block 14	3×3	512	Não
Global avg pooling	-	512	-
Softmax linear layer	-	-	Não

Foi também utilizado um segundo modelo, com mais camadas que o primeiro. A Tabela 4.6 ilustra esse segundo modelo, que será chamado BiggerResnet nesse trabalho.

Tabela 4.6: BiggerResnet

Layer	Conv type	Feature maps	Pooling
Initial	5×5	64	Não
Relu block 1	3×3	64	Não
Relu block 2	3×3	64	Não
Relu block 3	3×3	64	Não
Relu block 4	3×3	64	Não
Relu block 5	3×3	64	Não
Relu block 6	3×3	64	Não
Relu block 7	3×3	64	Não
Relu block 8	3×3	64	Não
Relu block 9	3×3	64	Não
Relu block 10	3×3	64	Não
Relu block 11	3×3	128	Sim
Relu block 12	3×3	128	Não
Relu block 13	3×3	128	Não
Relu block 14	3×3	128	Não
Relu block 15	3×3	128	Não
Relu block 16	3×3	128	Não
Relu block 17	3×3	128	Não
Relu block 18	3×3	128	Não
Relu block 19	3×3	128	Não
Relu block 20	3×3	128	Não
Relu block 21	3×3	256	Sim
Relu block 22	3×3	256	Não
Relu block 23	3×3	256	Não
Relu block 24	3×3	256	Não
Relu block 25	3×3	512	Sim
Relu block 26	3×3	512	Não
Relu block 27	3×3	512	Não
Relu block 28	3×3	512	Não
Global avg pooling	-	512	-
Softmax linear layer	-	-	Não

Conforme explicado na seção de introdução teórica, cada bloco referido aqui como "Relu block" tem duas convoluções, o que significa que a SmallerResnet tem $1+14\times2+1 =$

30 camadas e a BiggerResnet tem $1 + 28 \times 2 + 1 = 58$ camadas. Esse modelo foi escolhido por ser relativamente recente e altamente popular, sendo que, nesse caso, não foi utilizado *transfer learning*, essa técnica é utilizada no próximo modelo a ser descrito após esses resultados.

4.4.4 Resultados no CIFAR-10

Os resultados para o *training set* e o *validation set*, composto por imagens obtidas pelos *crawlers* e para o *test set*, composto pelas imagens do CIFAR-10, são retratados na Tabela 4.7 com otimizadores de parâmetros e inicializadores de variáveis indicados. Foi usado, além do já explicitado, as *moving averages* das variáveis registradas durante o treinamento: é realizada uma operação que computa essas médias a cada iteração de treino, mas não as usa. Na hora da execução da validação e do teste, a utilização delas em vez das variáveis finais é uma prática comum em Machine Learning.

Tabela 4.7: Tabela com resultados do treinamento de classificadores

Modelo	Inicializador	Otimizador	LR Schedule	Acc_{val}	Acc_{test}
SmallerResnet	He Kaiming [57]	Nadam [58]	Exponential	92.95%	83.44%
BiggerResnet	He Kaiming [57]	Nadam [58]	Exponential	93.21%	84.06%

A Figura 4.13 e a Figura 4.15 ilustram as matrizes de confusão no conjunto de teste para SmallerResnet e BiggerResnet respectivamente, sendo que as linhas representam as classes reais e as colunas representam o retorno do classificador. A Figura 4.14 e a Figura 4.16 ilustram alguns erros cometidos por esses classificadores, também no conjunto de testes. Fica evidente que os classificadores cometem erros óbvios e seriam necessárias mais imagens para que o aprendizado desses objetos fosse melhorado. Em particular, a classe “avião” foi muitas vezes confundida com “pássaro”, embora “pássaro” não tenha sido muito confundida com “avião”.

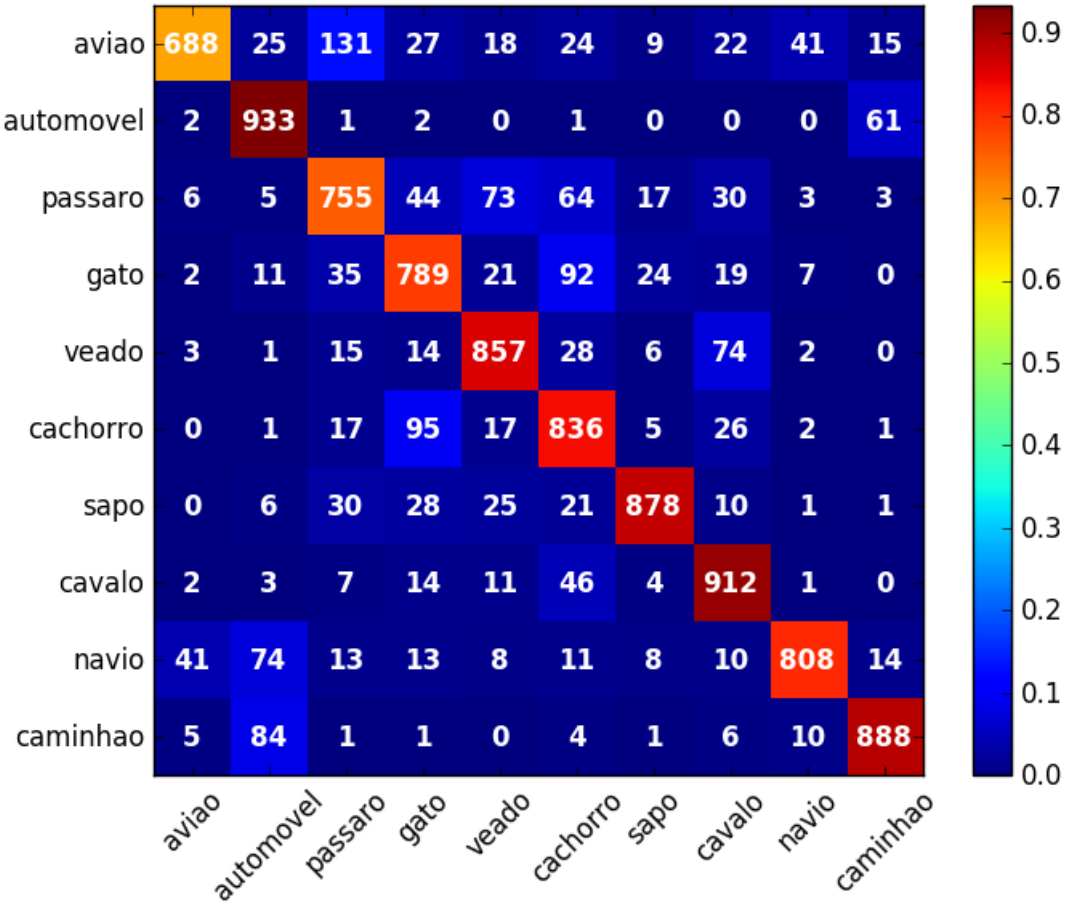


Figura 4.13: Matriz de confusão dos resultados da SmallerResnet

(a) PAS:CAC (b) PAS:CAM (c) GAT:CAV (d) GAT:CAV (e) GAT:CAV

(f) VEA:AVI (g) VEA:AVI (h) SAP:CAC (i) SAP:CAC (j) CAM:SAP

Figura 4.14: Alguns erros cometidos pela SmallResnet. Lêem-se as legendas como <original>:<previsto> e as abreviações são AVI: avião, AUT: automóvel, PAS: pássaro, GAT: gato, VEA: veado, CAC: cachorro, SAP: sapo, CAV: cavalo, NAV: navio e CAM: caminhão

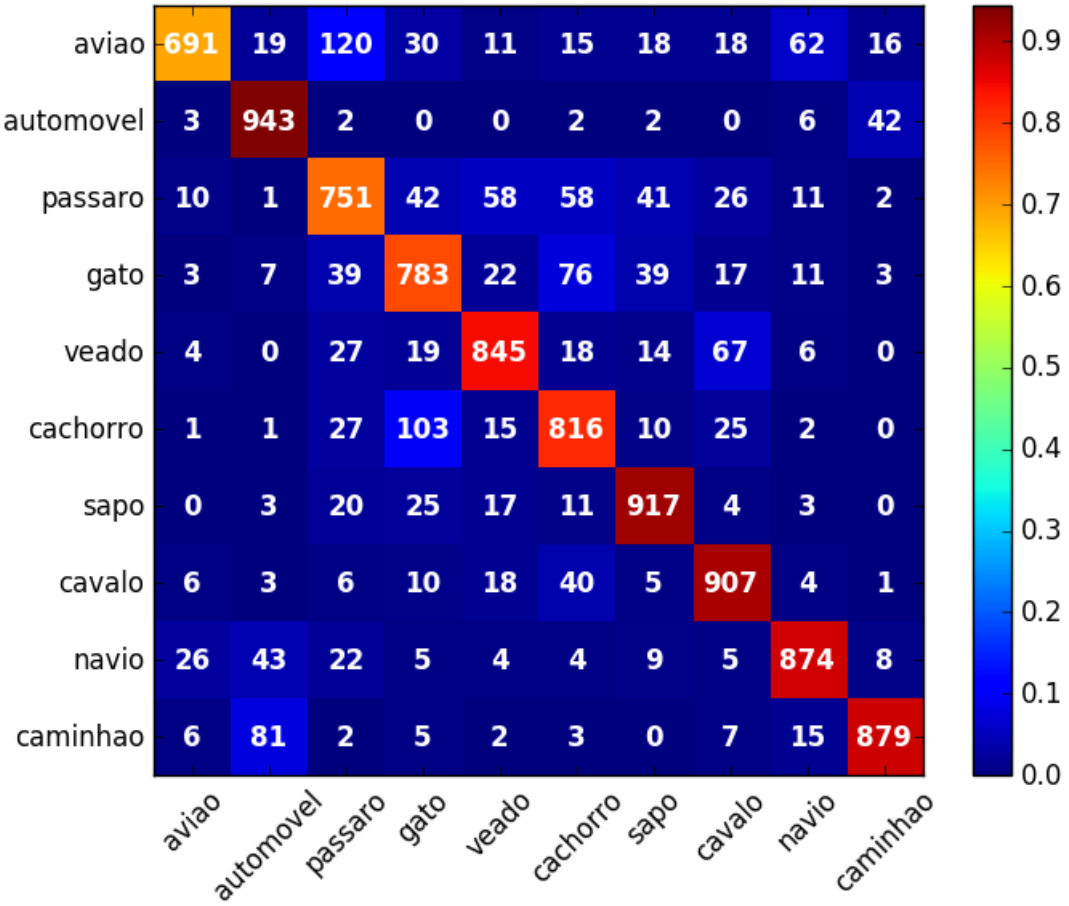


Figura 4.15: Matriz de confusão dos resultados da BiggerResnet

(a) AVI:PAS (b) CAR:PAS (c) CAR:PAS (d) GAT:CAR (e) GAT:CAC

(f) VEA:PAS (g) VEA:PAS (h) CAC:AVI (i) CAM:PAS (j) CAM:PAS

Figura 4.16: Alguns erros cometidos pela BiggerResnet. Lêem-se as legendas como <original>:<previsto> e as abreviações são AVI: avião, AUT: automóvel, PAS: pássaro, GAT: gato, VEA: veado, CAC: cachorro, SAP: sapo, CAV: cavalo, NAV: navio e CAM: caminhão

A partir desses dados, pode-se constatar que, embora o classificador tenha precisão razoável com essas imagens de treino, o resultado é claramente inferior a estudos do estado-da-arte em que é usado o *set* de treino do próprio CIFAR-10. Alguns desses resultados são ilustrados na Tabela 4.8. Vale ressaltar que muitas das arquiteturas dessa tabela são bem mais recentes que a utilizada, mas mesmo na Resnet original [37] a diferença é visível. Essa inferioridade possivelmente se deve ao fato de o CIFAR-10 ser um *dataset* controlado, com *bounding boxes* já pré-definidas e qualidade das imagens garantida pelos marcadores. Embora as imagens obtidas pelos *crawlers* nessa fase tenham sido marcadas manualmente também, a maneira como foram obtidas pressupõe menos controle, conforme citado anteriormente. Uma possibilidade da obtenção de imagens da forma como foi feita nesse trabalho é a de compensar essa diferença através do fornecimento de um número de exemplos bem maior do que o do CIFAR-10, mas, como não foi possível marcar todas as imagens obtidas dentro do prazo, o *set* de treino acabou tendo 40.887 imagens de treino e 10.000 de validação, enquanto o CIFAR-10 normalmente é separado em 40.000 de treino e 10.000 de validação. Isso significa que a vantagem desse método não pôde ser explorada em sua totalidade. Além disso, a diferença de escala também é fator importante, imagens como as da Figura 4.17 provavelmente se comportam como de classe diferente caso a escala seja muito reduzida, já que se torna difícil identificar o objeto em questão.

Tabela 4.8: Resultados recentes para o CIFAR-10

Modelo	Camadas	*Acc*
Resnet[37]	20	91.25%
Resnet[37]	44	92.83%
Wide Resnet[59]	16	95.19%
Densenet[60]	40	94.76%
Resnext[61]	-	96.35%

Figura 4.17: Algumas imagens que se comportam como incorretas caso haja grande redução de escala

4.4.5 Clarifai API

Tendo em vista a limitação da qualidade do estudo anterior devido à redução da escala para se adequar ao CIFAR-10, foi utilizada uma segunda maneira de se verificar a adequação das imagens obtidas pelos *crawlers* como classificadores: separar as próprias imagens dos *crawlers* em conjuntos de treino e teste. Para essa fase, foi escolhida uma API de Machine Learning chamada Clarifai. O motivo dessa escolha foi que essa API é de uso profissional, usada na indústria e com equipe de desenvolvimento, teste e suporte que, além de forte *background* teórico, dedica grande quantidade de tempo para garantir o bom funcionamento de seu sistema. Espera-se que esse sistema funcione bem melhor do que qualquer outra rede usada aqui, e é o equivalente a grande quantidade de tempo com testes e *parameter tuning* dos modelos aqui usados. O motivo dessa ferramenta não ter sido usada em todo o estudo é o fato de ser uma ferramenta paga. No sistema real, não seria desejável fazer uso dessa ferramenta devido ao custo financeiro, sendo o ideal gastar um tempo com *parameter tuning* de algum dos outros modelos, como as NASNets com *transfer learning*.

Essa ferramenta opera em cima de imagens enviadas ao sistema pelo usuário. O usuário tem a possibilidade de definir *concepts* presentes em imagens e, para cada imagem enviada, escolher quais desses *concepts* estão ou não presentes. Após *upload* de grande número de imagens, o usuário seleciona a opção de treinar o sistema, que é tratado como uma *blackbox* pela ferramenta. A Figura 4.18 ilustra a ferramenta descrita. No caso desse projeto, os 10 conceitos gerados foram referentes às 10 classes presentes no CIFAR-10. Além disso, foram marcadas as opções de "conceitos mutuamente exclusivos", o que significa que a presença de um conceito impede que haja outros conceitos na imagem e "ambiente fechado", que

significa que nenhuma das imagens fornecidas ao sistema terá muita variação em relação aos conceitos fornecidos, ou seja, todas as imagens serão de algum dos 10 objetos.

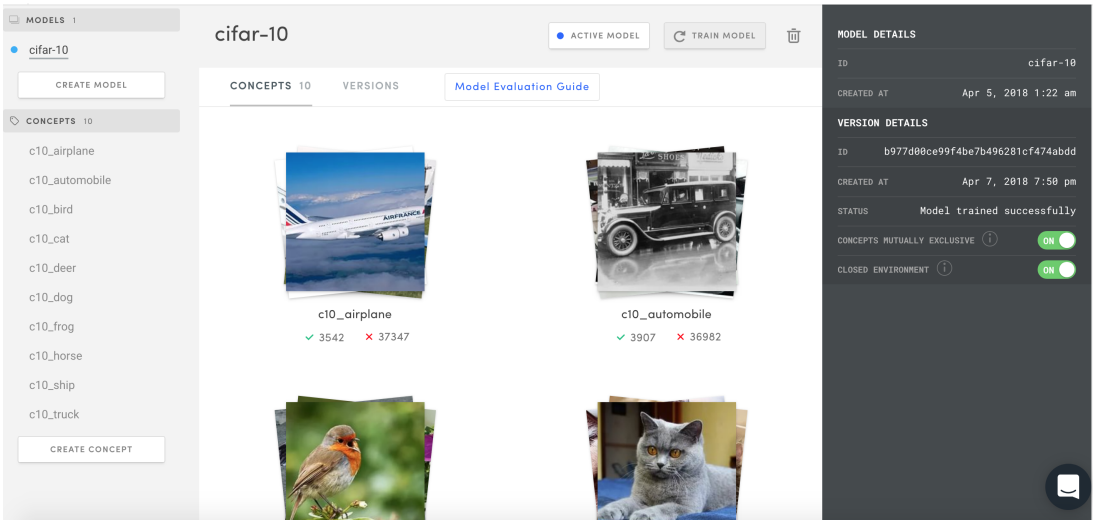


Figura 4.18: Ilustração da API Clarifai

As imagens obtidas através dos *crawlers* foram divididas em imagens de treino e de teste, sendo que a conjunto de treino ficou com 40.889 imagens e o conjunto de teste ficou com 10.000 imagens. A obtenção dessa divisão foi feita com embaralhamento das imagens de cada uma das classes separadamente e as 1000 primeiras imagens do conjunto embaralhado de cada classe foram reservadas para o *test set*, sendo o resto pertencente ao *training set*.

4.4.6 Resultados na Clarifai API

A precisão do sistema no *test set* foi de 98.98% (vale ressaltar que o esse conjunto vem das imagens dos *crawlers*, mas que não foram fornecidos ao classificador durante treino e validação) e a Figura 4.19 ilustra a matriz de confusão dos resultados dessa etapa do projeto, sendo que as linhas representam as classes reais e as colunas representam o retorno do classificador. Através da figura, é possível perceber que duas imagens foram perdidas durante o *upload*: uma de *bird* e uma de *truck* (note que essas linhas somam 999 em vez de 1000). Além disso, é possível também notar que a maior dificuldade do classificador esteve em *automobile* e *truck*, que são objetos cuja diferenciação é de fato mais difícil, visto que são parecidos. A Figura 4.20 ilustra alguns dos erros desse classificador para esses dois objetos e a Figura 4.21 ilustra erros gerais cometidos pelo classificador.

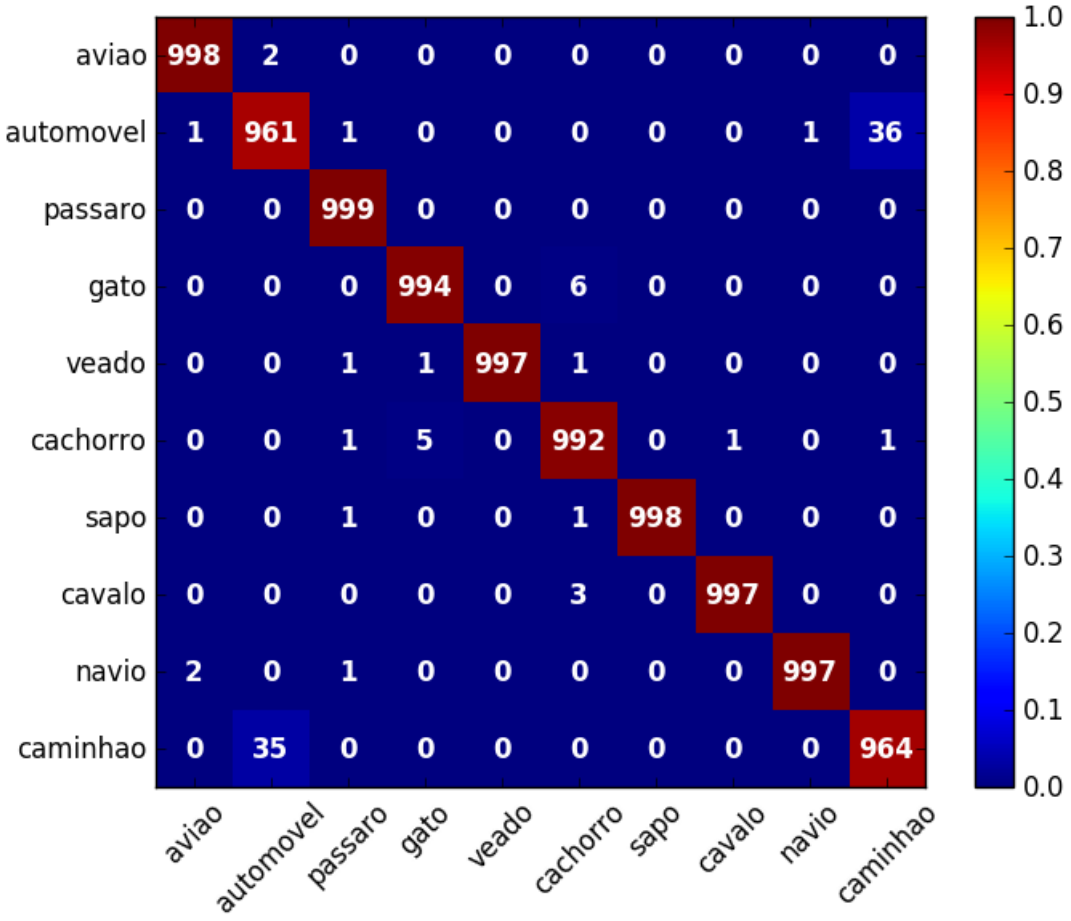


Figura 4.19: Matriz de confusão dos resultados do Clarifai

(a) Automóveis que foram confundidos com caminhões

(b) Caminhões que foram confundidos com automóveis

Figura 4.20: Erros cometidos pelo Clarifai

(a) Gato como cachorro

(b) Veado como pássaro

(c) Cachorro como cavalo

(d) Sapo como cachorro

(e) Navio como pássaro

Figura 4.21: Figuras com classificação incorreta

Vale ressaltar que, na Figura 4.20, a maioria dos caminhões que foram confundidos com automóveis tratam-se de caminhonetes, que podem ser consideradas como um tipo de automóvel, apesar de que, nos exemplos de treino, caminhonete entra na classe *truck*, já que caminhonete é *pickup truck* em inglês e a expressão é comumente abreviada como *truck*. Além disso, nessa mesma figura, é possível ver que houve erros de marcação: um dos automóveis dessa figura na verdade é uma caminhonete e deveria ter sido marcado como *truck*. Na Figura 4.21, acontece de dois conceitos estarem presentes na mesma foto, o que dificulta a classificação. O sapo com pele diferente do comum, que é possivelmente uma imagem artificial, também parece ter causado problemas para o classificador, assim como o veado com rabo similar a um pássaro e o desenho do navio (que deveria ter sido marcado como incerto).

A maioria dos erros encontrados aparentemente foram resultado de confusão de conceitos (como em *truck*) ou de erro da marcação manual. Ou seja, a princípio, o classificador conseguiu aprender bem com os exemplos fornecidos. Porém, como o sistema foi apresentado como uma *blackbox*, havia a dúvida se bases de dados internas ao Clarifai foram de alguma forma usadas para melhorar os resultados com exemplos além dos fornecidos com *transfer learning*. A equipe de suporte do Clarifai foi contatada acerca disso, e a resposta obtida foi:

> Quando "*general*" é selecionado como a base do *workflow* de um modelo customizado, essa opção está de fato utilizado do poder de um modelo geral para melhorar a precisão do customizado. São utilizados *embeddings* de cada um dos 10.000 itens treinados à *priori* (como céu, escova de dentes, bicicleta, etc) para possibilitar isso.

Além disso, a equipe acrescentou:

> Os nomes dos conceitos por você criados não fazem diferença, já que os únicos "conceitos" do modelo são os seus. *Embeddings* são uma entidade totalmente separada.

Isso significa que houve *transfer learning* no modelo. Outro ponto que não foi considerado quando o experimento foi elaborado é o fato de que, durante o fluxo de experimentos, não foi checado se havia repetição entre as imagens. Isso significa que, para esse experimento específico, é possível que algumas das imagens de teste sejam iguais a algumas das imagens de treino, facilitando a classificação de maneira injusta.

Capítulo 5

Conclusões

Nesse trabalho, primeiro foi notado que os estudos em Machine Learning da atualidade fazem marcação manual de *datasets* para cada aplicação nova e foi questionado se isso é realmente necessário, dado que imagens com informações de classes associadas são diariamente disponibilizadas na Internet. O reuso dessas imagens poderia trazer enormes benefícios para a área. Com base nessa ideia, foi proposto um sistema, sendo que foram apresentados diagramas de telas e de comunicação para melhor entendimento. A partir desses diagramas, foram propostas soluções para cada etapa.

A primeira delas é a obtenção de imagens, que pode ser realizada com *web crawling*. O problema de tal abordagem é que manutenções devem ser constantemente realizadas para que os *crawlers* continuem funcionando, visto que as estruturas sobre as quais esses *crawlers* navegam estão em constante mudança. Também seria ideal que a máquina onde os *crawlers* são executados possua acesso de qualidade à Internet, pois o tempo necessário para *download* de todas as imagens geralmente é longo, principalmente se forem usados todos os *crawlers* aqui estudados. Tomando como base a média de imagens obtidas para cada classe, uma *query* retorna cerca de 22.000 imagens, incluindo as não-relevantes. Para propósito de realizar os experimentos referentes a cada etapa, foram escolhidas as 10 classes do CIFAR-10 e suas imagens foram usadas como base.

Como é necessário ter um *ground truth* para que se saiba o quão bem funcionou cada etapa, foi feita uma ferramenta específica com a finalidade de obter essas marcações. No contexto do CIFAR-10, as imagens obtidas pelos *crawlers* relevantes eram as que representavam claramente o objeto pretendido, sendo que essa representação deveria se apresentar em moldes próximos ao que aparece nas imagens do CIFAR-10. As imagens não-relevantes eram as que representavam objetos claramente diferentes dos procurados e as incertas eram representações do objeto pretendido, mas que, por algum motivo, estavam distantes do encontrado em imagens do CIFAR-10. Para simplificar os experimentos, as imagens incertas foram retiradas do experimento. Com a análise dos resultados, é possível

constatar que esses critérios acabam sendo muito subjetivos, sendo que alguns erros dos classificadores dos experimentos foram na verdade erros de marcação.

Em seguida, é necessário fazer uma remoção das imagens não-relevantes. Essa etapa é requerida pois há muitas imagens incorretas em meio às imagens obtidas, que nada têm a ver com a *query* pretendida. No caso de criação de *datasets*, isso significa que há erros em meio as imagens, o que prejudica a aplicação pretendida. Da mesma forma, essas imagens com classes incorretas prejudicam os classificadores que serão criados com base nessas imagens. Nessa etapa, foram testadas as Triplet Networks, uma Resnet convolucional com mesma arquitetura base da Triplet e uma NASNet com *transfer learning*. Inicialmente assumiu-se que a Triplet Network melhoraria os resultados, já que tanto ela quanto sua concorrente, a Siamese Network, são usadas para *few shot learning*. Porém, as aplicações de *few shot learning* sempre contam com um grande número de classes, embora haja poucos exemplos por classe. É o caso de [38], em que os experimentos com faces chegam à ordem de milhões de imagens e também de [50], em que é usado o dataset Omniglot de alfabetos, com 1623 caracteres de 50 alfabetos e, no estudo, foram usados *training set* com tamanho variando de 30.000 a 150.000 exemplos. Ou seja, caso hajam poucos exemplos por classe e poucas classes, esse modelos não são bons: tanto a Resnet quanto a Triplet causam *overfitting*.

Nesse contexto, o *transfer learning* seria de fato a abordagem adequada para resolver o problema. Vale ressaltar que, nesse trabalho, as classes escolhidas foram relativamente comuns, pois na época de escolha delas não havia sido decidido que algumas partes seriam feitas com *transfer learning*. Como o Imagenet contém classes similares a essas 10 do CIFAR-10, isso facilita a classificação com esse método. Porém, vale também ressaltar que, para a fase de remoção de não-relevantes, a classe "não-relevante" inclui qualquer tipo de objeto, de forma que não são objetos imediatamente previstos por Imagenet *pretraining*.

A próxima fase tratou de treinamento de classificadores usando as imagens obtidas automaticamente. A ferramenta foi proposta como de detecção de objetos, o que implicaria que esses classificadores deveriam ser *one-vs-all*. Alguns cuidados extras devem ser tomados quando são treinados classificadores desse tipo. Um deles é a geração de uma classe negativa "all" representativa dos outros objetos possíveis, sendo que essa classe não pode conter o objeto em questão. Outro problema é o fato de que detectores são majoritariamente baseados em *sliding windows*, sendo que, nesses algoritmos, são usadas janelas de tamanho canônico e as variações desse tamanho mantêm o *aspect ratio* inicial, mas as imagens dos *crawlers* podem ser dos mais variados *aspect ratios*. Soluções para esse tipo de problema serão propostas em discussão a seguir, mas, para simplificar a implementação desse trabalho, esses problemas foram ignorados: em vez de serem feitos classificadores *one-vs-all* foi feito um classificador que diferencia entre as 10 classes do CIFAR-10, sendo

que as marcações manuais foram usadas como *ground truth*.

Inicialmente, foi usado o CIFAR-10 como conjunto de teste, mas os resultados, que ficaram em torno de 84%, indicam que a diferença de escala entre as imagens de treino e o CIFAR-10 prejudicou a classificação, além do fato de não ter sido usado todo o conjunto de imagens obtidas devido a falta de tempo para realizar as marcações. Nesse experimento, a precisão de validação ficou próxima a 94%, sendo que a diferença entre ela e a precisão de teste serve como indicador adicional de que essa diferença entre escalas prejudicou os resultados no CIFAR-10. Por isso, foi decidido realizar um experimento que não tivesse uma diferença de escala entre treino e teste tão grande. O modo mais simples de se fazer isso é aproveitando as próprias imagens obtidas pelos *crawlers* e, nesse caso, foi utilizado o Clarifai (por motivos de se obter a melhor performance possível) para realizar a classificação. O resultado foi de aproximadamente 99%. No entanto, vale ressaltar que, como as classes a serem diferenciadas eram comuns, o *transfer learning* realizado internamente pelo Clarifai tem mais facilidade em classificá-las corretamente.

5.1 Dificuldades em se criar um detector

Tendo em vista o embasamento teórico acerca de treinamento de classificadores descrito na seção teórica, o primeiro aspecto a ser considerado é o tempo para se detectar os objetos. Como os modelos de rede aqui usados em geral foram grandes, isso é um indicativo de que possíveis modificações do sistema apresentado para uso em tempo real podem ser inviáveis, já que não se tratam de modelos projetados com foco em eficiência, como os de *Machine Learning* clássico citados no capítulo de *background* teórico na parte de detecção. Por outro lado, é sabido que, com o uso de GPUs, a inferência se torna mais rápida, sendo que a viabilidade de se implementar isso em tempo real pode ser possível com uso delas. Dados mais específicos podem ser obtidos apenas com a realização de testes.

Além disso, esses detectores são normalmente feitos com base em *sliding windows*. Como mencionado na seção anterior, esses algoritmos dependem de janelas canônica com tamanho pré-definido. No decorrer dos experimentos, as imagens usadas sempre foram quadradas, de forma que, caso determinada imagem tivesse *aspect ratio* muito diferente de um quadrado, ela sofreria deformações horizontais ou verticais. Essas deformações foram ignoradas no trabalho. Porém, no caso de imagens enviadas ao sistema na tela de detecção, a limitação de haver apenas objetos em janelas quadradas não existe, o que significa que certos objetos com formato muito vertical ou horizontal ficariam de fora. Uma solução para isso seria não usar apenas uma janela canônica quadrada, mas sim de alguns tipos diferentes. Por exemplo, uma janela canônica majoritariamente vertical, uma majoritariamente horizontal e uma quadrada seriam usadas e, na fase de treinamento de

classificadores, a imagem seria deformada a fim de se enquadrar na janela canônica mais próxima. Um problema ao se fazer isso é as redes pré-treinadas disponibilizadas *online*, que foram as que obtiveram os melhores resultados nesse trabalho, esperam imagens em tamanhos específicos, o que significa que elas não poderiam ser usadas e o pré-treinamento teria que ser realizado do zero.

Outro problema é o fato de que não há controle de *bounding boxes* nas imagens obtidas. No detector de pedestres descrito na seção teórica, a marcação dos pedestres era manual e, por isso, o marcador tinha como garantir que o objeto estaria bem delimitado pelas *bounding boxes*. O fato de as imagens obtidas automaticamente não terem essa garantia torna ainda mais difícil o uso dessas imagens para detecção.

Quanto a técnicas para conseguir melhores exemplos para o classificador, como *bootstrapping* e *active learning*, o momento de usá-las seria em conjunto com a criação dos classificadores, logo após a remoção de não-relevantes, sendo esse uso realizado de maneira iterativa: primeiro o classificador é criado, depois é testado em imagens e os exemplos considerados bons são reincorporados ao classificador e o processo se repete. O *active learning*, no entanto, não pode ser usado nesse sistema, já que envolve participação do usuário.

Por fim, como mencionado anteriormente, existe o problema de se garantir que o objeto sendo detectado não esteja presente na classe "all" no momento de criação do classificador *one-vs-all*. Caso se tenha acesso a *queries* que geraram as imagens candidatas à classe "all", basta usar redes que identificam *queries* sinônimas para evitar esse problema [53] [54].

5.2 Trabalhos futuros

Os experimentos aqui descritos sofreram várias simplificações para que fosse viável sua realização no tempo disponível e com a quantidade de pessoas disponível. Porém, mesmo sendo simplificados, os resultados não devem ser descartados, pois servem para motivar ou guiar experimentos mais complexos. Um trabalho futuro possível seria a realização desses experimentos com todas as imagens sendo marcadas. Há de se mencionar que essa marcação pode possivelmente fazer uso de redes pré-treinadas ou do próprio Clarifai, tomando como base alguns exemplos marcados manualmente, isso pouparia o marcador várias semanas de trabalho manual. O problema em fazer uso desse tipo de abordagem é que há várias situações inesperadas nessas imagens, em que, caso o marcador seja uma pessoa, é possível tomar atitudes apropriadas, como a de criar uma classe de relevância extra chamada "incerta" no experimento que foi feito. Caso seja decidido confiar nas redes para realizar essa tarefa, há de se considerar que há essa perda de controle. Um

outro experimento possível seria a remoção dessa classe extra, sendo as imagens julgadas estritamente como relevantes ou não-relevantes.

Além disso, outro trabalho futuro possível é o de se considerar o retorno real da fase de remoção de não-relevantes, em vez do *ground truth* manual. Isso permitiria tirar conclusões acerca da robustez desses métodos frente aos falsos positivos encontrados (cerca de 2.4% no experimento realizado).

Outro estudo que também deve ser considerado é a replicação do que foi feito para o CIFAR-10, mas com uso de algum *dataset* diferente. Isso traria conclusões importantes, primeiramente porque a escala reduzida do CIFAR-10 prejudicou a classificação, mas também pelo fato de o CIFAR-10 conter um número pequeno de classes, o que traz dúvidas quanto a possibilidade de se generalizar os resultados obtidos, embora deva-se atentar para o fato de que cada classe retornou cerca de 22 mil imagens, sendo necessário preparo para lidar com o número de imagens obtido com uma quantidade de classes maior. Houve, ainda, o problema de as classes do CIFAR-10 não serem muito diferentes do que é visto em *datasets* usados para pré-treinamento, como o Imagenet. Uma seleção de um *dataset* diferente eliminaria essa facilidade artificial que alguns classificadores tiveram, mas deve-se tomar cuidado para que o *dataset* usado seja padrão em publicações, caso contrário é possível argumentar contra a qualidade dos resultados, já que teria sido usado um *dataset* cuja qualidade das marcações não é conhecida pela comunidade.

Um outro experimento possível é o uso desse sistema como *black box* para alguma aplicação. Um exemplo simples seria escolher um conjunto de classes, obter um *test set* com imagens dessas classes e usar as *queries* delas como entrada do sistema. Nessa proposta, o classificador poderia ser geral, como foi feito no estudo apresentado, em vez de ser um *one-vs-all*. Após a criação desse classificador geral, ele poderia ser avaliado nesse conjunto de teste. A vantagem desse experimento seria que não seriam necessárias marcações das imagens dos *crawlers*. Por fim, vale ressaltar que não houve em momento algum verificação de imagens repetidas. Na realização de qualquer um desses experimentos, deve-se conferir se não há a possibilidade de algumas imagens de treino serem iguais a algumas de teste, o que faz com que a acurácia fique maior do que deveria.

Referências

[1] Anubhav Anushi Tushar. 3 types of gradient descent algorithms for small & large data sets. `https://www.hackerearth.com/blog/machine-learning/3-types-gradient-descent-algorithms-small-large-data-sets/`. Accessed: 2018-07-31. x, 12

[2] Victor Lavrenko Charles Sutton. Introductory Applied Machine Learning. Technical report, University of Edinburgh, 2010. Course avaliable on Youtube. x, 13

[3] Vincent Vanhoucke. Deep Learning. Technical report, 2016. Course avaliable on Udacity. x, 11, 15, 16, 17, 18, 19, 20, 22, 23, 26, 28, 29, 65

[4] Vincent Dumoulin e Francesco Visin. A guide to convolution arithmetic for Deep Learning. *arXiv preprint arXiv:1603.07285*, 2016. x, 24, 27

[5] Caglar Gulcehre. Convolutional neural networks (lenet). `http://deeplearning.net/tutorial/lenet.html`. Accessed: 2018-07-31. x, 25

[6] Pete Warden. Why GEMM is at the heart of Deep Learning. `https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/comment-page-1/`. Accessed: 2018-07-31. x, 26

[7] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, e Andrew Rabinovich. Going deeper with convolutions. Em *Computer Vision and Pattern Recognition (CVPR)*, 2015. x, 29, 30, 31

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, e Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015. xi, 32, 78

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, e Jian Sun. Identity mappings in deep residual networks. *arXiv preprint arXiv:1603.05027*, 2016. xi, 32, 33, 85

[10] Elad Hoffer e Nir Ailon. Deep metric learning using triplet network. Em *International Workshop on Similarity-Based Pattern Recognition*, páginas 84–92. Springer, 2015. xi, 33, 34, 61, 78

[11] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, e Quoc V Le. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2(6), 2017. xi, 35, 78

[12] Nathan Silberman e Sergio Guadarrama. Tensorflow-slim image classification model library. `https://github.com/tensorflow/models/tree/master/research/slim`. Accessed: 2018-07-18. xi, 36

[13] Tensorflow development team. List of tflite hosted models. `https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/lite/g3doc/models.md`. Accessed: 2018-07-18. xi, 36

[14] Maria Vanrell Antonio López Peña, Ernest Valveny. Detección de objetos. Technical report, Universidade Autônoma de Barcelona, 2015. Course avaliable on Coursera. xi, 37, 39, 41, 43, 44, 45, 46, 48, 49, 50

[15] Andrea Corriga. Computer vision project. `https://github.com/AsoStrife/Computer-Vision-Project`. Accessed: 2018-07-31. xi, 9, 40

[16] Ankit Sharma. Support vector machine without tears. `http://diggdata.in/post/94066544971/support-vector-machine-without-tears`. Accessed: 2018-07-18. xi, 42

[17] Sebastian Raschka. Confusion matrix. `https://rasbt.github.io/mlxtend/user_guide/evaluate/confusion_matrix/`. Accessed: 2018-07-31. xi, 46

[18] Junyang Lu, Jiazhen Zhou, Jingdong Wang, Tao Mei, Xian-Sheng Hua, e Shipeng Li. Image search results refinement via outlier detection using deep contexts. Em *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, páginas 3029–3036. IEEE, 2012. xi, 50, 51

[19] Tensorflow development team. Tensorflow. `https://www.tensorflow.org/`. Accessed: 2018-08-15. xi, 68

[20] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, e Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. Em *Proceedings of the IEEE conference on computer vision and pattern recognition*, páginas 1701–1708, 2014. 1

[21] David Lynton Poole, Alan K Mackworth, e Randy Goebel. *Computational intelligence: a logical approach*, volume 1. Oxford University Press New York, 1998. 1

[22] Phil Simon. *Too big to ignore: the business case for big data*, volume 72. John Wiley & Sons, 2013. 6

[23] Ian Goodfellow, Yoshua Bengio, e Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`. 6

[24] Strother H. Walker e David B. Duncan. Estimation of the probability of an event as a function of several independent variables. *Biometrika*, 54(1/2):167–179, 1967. 8

[25] David R Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society. Series B (Methodological)*, páginas 215–242, 1958. 8

[26] Steven S. Beauchemin e John L. Barron. The computation of optical flow. *ACM computing surveys (CSUR)*, 27(3):433–466, 1995. 8

[27] Dong-Chen He e Li Wang. Texture unit, texture spectrum, and texture analysis. *IEEE transactions on Geoscience and Remote Sensing*, 28(4):509–512, 1990. 8, 39

[28] Nasser M Nasrabadi. Pattern recognition and machine learning. *Journal of electronic imaging*, 16(4):049901, 2007. 9

[29] Krzysztof C Kiwiel. Convergence and efficiency of subgradient methods for quasiconvex minimization. *Mathematical programming*, 90(1):1–25, 2001. 10

[30] Seymour Geisser. *Predictive inference*. Routledge, 2017. 13

[31] Ilya Sutskever, James Martens, George Dahl, e Geoffrey Hinton. On the importance of initialization and momentum in Deep Learning. Em *International conference on machine learning*, páginas 1139–1147, 2013. 14

[32] Moritz Hardt, Benjamin Recht, e Yoram Singer. Train faster, generalize better: stability of stochastic gradient descent. *arXiv preprint arXiv:1509.01240*, 2015. 14

[33] Xavier Glorot, Antoine Bordes, e Yoshua Bengio. Deep sparse rectifier neural networks. Em *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, páginas 315–323, 2011. 17

[34] David E Rumelhart, Geoffrey E Hinton, e Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986. 18

[35] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, e Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014. 21

[36] Yann LeCun, Léon Bottou, Yoshua Bengio, e Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. 24

[37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, e Jian Sun. Deep residual learning for image recognition. Em *Proceedings of the IEEE conference on computer vision and pattern recognition*, páginas 770–778, 2016. 31, 85, 90

[38] Florian Schroff, Dmitry Kalenichenko, e James Philbin. Facenet: A unified embedding for face recognition and clustering. Em *Proceedings of the IEEE conference on computer vision and pattern recognition*, páginas 815–823, 2015. 34, 61, 97

[39] Barret Zoph e Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016. 35

[40] François Chollet. Xception: Deep Learning with depthwise separable convolutions. *arXiv preprint*, páginas 1610–02357, 2017. 35

[41] Jeremy West, Dan Ventura, e Sean Warnick. Spring research presentation: A theoretical foundation for inductive transfer. *Brigham Young University, College of Physical and Mathematical Sciences*, 1, 2007. 36

[42] Sanja Fidler. Object detection, sliding windows. Technical report, Universidade de Toronto, 2017. CSC420: Intro to Image Understanding course. 37

[43] Corinna Cortes e Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995. 41

[44] David Martin Powers. Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. 2011. 47

[45] Wei Liu, Gang Hua, e John R Smith. Unsupervised one-class learning for automatic outlier removal. Em *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, páginas 3826–3833, 2014. 50

[46] Josip Krapac, Moray Allan, Jakob Verbeek, e Frédéric Jurie. Improving web image search results using query-relative classifiers. Em *Computer Vision and Pattern Recognition (CVPR)*, páginas 1094–1101. IEEE, 2010. 51

[47] Dhruv Mahajan, Ross Girshick, Vignesh Ramanathan, Kaiming He, Manohar Paluri, Yixuan Li, Ashwin Bharambe, e Laurens van der Maaten. Exploring the limits of weakly supervised pretraining. *arXiv preprint arXiv:1805.00932*, 2018. 52

[48] Vignesh Ramanathan Manohar Paluri Laurens van der Maaten Dhruv Mahajan, Ross Girshick. Advancing state-of-the-art image recognition with Deep Learning on hashtags. `https://code.fb.com/ml-applications/advancing-state-of-the-art-image-recognition-with-deep-learning-on-hashtags/`. Accessed: 2018-07-23. 52

[49] Adi Ignatius. Meet the google guys. *Time*, página 40, 2006. 58

[50] Gregory Koch, Richard Zemel, e Ruslan Salakhutdinov. Siamese neural networks for one-shot image recognition. Em *ICML Deep Learning Workshop*, volume 2, 2015. 61, 97

[51] Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, e Timothy Lillicrap. One-shot learning with memory-augmented neural networks. *arXiv preprint arXiv:1605.06065*, 2016. 61

[52] Oriol Vinyals, Charles Blundell, Tim Lillicrap, e Daan Wierstra. Matching networks for one shot learning. Em *Advances in Neural Information Processing Systems*, páginas 3630–3638, 2016. 61

[53] Douglas B Lenat. Cyc: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11):33–38, 1995. 62, 99

[54] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995. 62, 99

[55] Yann LeCun, Corinna Cortes, e Christopher Burges. The MNIST database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`. Accessed: 2018-08-16. 65

[56] Ken Turkowski. Filters for common resampling tasks. Em *Graphics gems*, páginas 147–165. Academic Press Professional, Inc., 1990. 85

[57] Kaiming He, Xiangyu Zhang, Shaoqing Ren, e Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. Em *Proceedings of the IEEE international conference on computer vision*, páginas 1026–1034, 2015. 87

[58] Timothy Dozat. Incorporating Nesterov Momentum into Adam. Em *Proceedings of 4th International Conference on Learning Representations, Workshop Track*, páginas 2013–2016, 2016. 87

[59] Sergey Zagoruyko e Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016. 90

[60] Gao Huang, Zhuang Liu, e Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016. 90

[61] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, e Kaiming He. Aggregated residual transformations for deep neural networks. Em *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, páginas 5987–5995. IEEE, 2017. 90