Capítulo 4

Métodos baseados em Deep Learning para Análise de Vídeo

Gabriel N. P. dos Santos[1], Pedro V. A. de Freitas[1], Antonio José G. Busson[1], Álan L. Guedes[1], Sérgio Colcher[1] e Ruy L. Milidiú[1]

[1]Departamento de Informática, PUC-Rio

{gabrielpereira,pedropva,busson,alan}@telemidia.puc-rio.br

{colcher,milidiu}@inf.puc-rio.br

Abstract

Methods based on Deep Learning became state-of-the-art in several multimedia challenges. However, there is a gap of professionals to perform Deep Learning in the industry. This chapter focuses on presenting the fundamentals and technologies for developing such DL methods for video analyses. In particular, we seek to enable the reader to: (1) understand key DL-based models, more specifically Convolutional Neural Networks (CNN); (2) apply DL models to solve video tasks such as video classification, multi-label video classification, object detection, and pose estimation. The Python programming language is presented in conjunction with the TensorFlow library for implementing DL models.

Resumo

Os métodos baseados no Deep Learning tornaram-se state-of-the-art em vários desafios de multimídia. No entanto, existe uma lacuna de profissionais para realizar o Deep Learning na indústria. Este capítulo tem como foco apresentar os fundamentos e tecnologias para desenvolver tais métodos de DL para analise de vídeo. Em especial, buscamos capacitar o leitor a: (1) entender os principais modelos baseados em DL, mais especificamente Convolutional Neural Networks (CNN); (2) aplicar os modelos de DL para resolver tarefas de vídeo como: classificação de vídeo, classificação de multi-etiquetas de vídeo, detecção de objetos e estimação de pose. A linguagem de programação Python é apresentada em conjunto com a biblioteca TensorFlow para implementação dos modelos de DL.

4.1. Introdução

A popularização de equipamentos de captura de vídeo e serviços para seu armazenamento e transmissão, possibilitou a produção de um massivo volume de dados de vídeo. O Youtube, por exemplo, registrou em 2014[1] upload de 72 horas de vídeo por minuto. Enquanto que em 2018[2], esse número subiu para 400 horas de vídeo por minuto. Por exemplo no Brasil, serviços como o video@RNP[3] e ICD[4] constituem importantes redes de compartilhamento vídeo. Esse cenário apresenta desafio de controle do tipo de conteúdo que é carregado para esses serviços de armazenamento videos. A classificar esse tipo de conteúdo requer uma análise automática desse volume de forma eficiente e prática.

Métodos baseados em *Deep Learning* (DL) se tornaram o *estado-da-arte* em vários segmentos relacionados a análise automática de mídia. Em particular, as arquiteturas de DL baseadas em CNNs (*Convolutional neural network*), ou ConvNets, se tornaram o principal método usado para reconhecimento de padrões áudio-visuais. Tipicamente o treinamento de CNNs é feito de maneira supervisionada, e são treinadas em *datasets* que contém milhares/milhões de mídias e classes relacionadas. Durante o treinamento, as CNNs aprendem a hierarquia de *features* que são aplicadas a mídia de entrada para que seja possível realizar a classificação do seu conteúdo.

Este capítulo tem como foco avaliar e desenvolver tais métodos de DL para análise de vídeo. No decorrer dele, são implementados métodos de DL utilizando a linguagem Python. Para apresentar esse métodos, o restante deste trabalho está organizado como a segue.

- A Seção 4.2 apresenta o *framework TensorFlow*.
- A Seção 4.3 introduz os conceitos básicos de aprendizado de máquina.
- A Seção 4.4 apresenta os fundamentos de Redes Neurais
- A Seção 4.5 apresenta os fundamentos de Redes Neurais Convolucionais.
- A Seção 4.6 apresenta implementações para resolução de problemas de classificação de vídeo.
- A seção 4.7 descreve o modelo YOLO para detecção de objetos.
- A Seção 4.8 apresenta conceitos básicos de pose estimation, métricas e técnicas.
- A Seção 4.9 apresenta nossas considerações finais.

4.2. Framework TensorFlow

Nesta seção, apresentamos uma visão geral do *framework Tensorflow*. O *Tensorflow*[5] é um framework *open-source* para Python, Javascript, C/C++ e Go e tem o objetivo de

[1] https://www.domo.com/learn/data-never-sleeps-2
[2] https://www.domo.com/learn/data-never-sleeps-6
[3] http://video.rnp.br
[4] http://documentas.redclara.net/handle/10786/1104
[5] https://www.tensorflow.org/?hl=pt-br

auxiliar o processamento de dados em *machine learning* por meio de um modelo baseado em fluxo de dados. Ele foi criado em 2015 pelo time da Google responsável por pesquisas na área de Inteligência Artificial e *Deep Learning* (Google Brain). O *Tensorflow* começou como uma refatoração do antigo sistema DistBelief[6] (criado em 2011), com o intuito de melhorar seu desempenho.

Para apresentar o *Tensorflow*, o restante seção está organizado como segue. A Subseção 4.2.1 descreve o processo de instalação. Em seguida, a Subção 4.2.2 apresenta os fundamentos e estruturas básicas do *framework*. Por fim, a Subseção 4.2.3 descreve como utilizar o *framework*.

4.2.1. Instalação

Neste parte inicial, descrevermos como instalar o *Tensorflow* em sistemas Ubuntu e outras distribuições derivadas do Linux. O Python 2.7 e Pip (sistema de gerenciamento de pacotes do Python) já estão presentes no Ubuntu e na maioria das outras distribuições Linux. Porém, a versão python mais recente é o Python3. Para instalação do Python3, execute:

```
sudo apt-get install python3-pip python3-dev python-virtualenv
```

Tensorflow necessita que a versão do pip seja a 8.1 ou mais recente. Para verificar a versão atual do pip Python3 execute:

```
pip3 -V
```

Para atualizar o pip para a versão mais recente no Ubuntu:

```
sudo pip install -U pip
```

Para atualização do pip em distribuições diferentes da Ubuntu:

```
easy_install -U pip
```

A seguir, deve-se escolher a instalação com ou sem suporte para GPU. Para instalar o *Tensorflow* sem e com suporte para GPU execute respectivamente:

```
pip install -U tensorflow
```

ou

```
pip install -U tensorflow-gpu
```

Para testar a instalação inicie o terminal Python e execute:

```
import tensorflow as tf
tf.__version__
exit()
```

[6] https://ai.google/research/pubs/pub40565

4.2.2. Fundamentos e estruturas básicas do Tensorflow

O *Tensorflow* oferece facilidade para desenvolver modelos de redes neurais para uma multiplicidade de diferentes hardwares, bem como possibilita que o sistema seja executado sem ou com **GPU**s. Para utilizar o *framework*, primeiro é necessário entender os três conceitos que são descritos a seguir.

1. **Tensor**: consiste de um vetor de n dimensões. Tensores são as estruturas de dados básicas utilizadas no *TensorFlow*.

2. **Grafo de computação**: são malhas que consistem em nós conectados entre si por arestas. Cada nó possui seus *inputs* e outputs, assim como a operação que deve ser feita com os *inputs* para que o *output* seja criado. Tais arestas consistem nos valores que são passados de um nó para outro. Cada nó realiza a determinada operação assim que recebe todos os *inputs* necessários. Ao gerar seu resultado e passar para um próximo nó (ou vários) ligado a este.

3. **Sessões**: os nós do grafo de computação podem ser agrupados em sessões. Cada sessão pode ser executada separadamente em *threads* ou até mesmo em forma de computação distribuída.

4.2.3. Utilizando o Tensorflow

Após entender as estruturas básicas do *framework*, para usar o *Tensorflow*, como mostra a Listagem 4.1, basta importar o pacote para o projeto (linha 1). As linhas 3-5 mostram como criar o grafo de computação, para isso, são criados três tensores, onde o tensor **c**, consiste na multiplicação dos tensores **a** e **b**. Em seguida, as linhas 7 e 8 mostram como criar uma sessão e executar o grafo de computação. Por fim, a linha 10 mostra a saída do programa.

Listagem 4.1: Executando um grafo de computação no Tensorflow.

```
1  import tensorflow as tf
2
3  a = tf.constant([ [1.0, 2.0], [3.0, 4.0] ])
4  b = tf.constant([ [5.0, 6.0], [7.0, 8.0] ])
5  c = tf.matmul(a,b)
6
7  sess = tf.Session()
8  print(sess.run(c))
9  ---------------------------- OUTPUT -----------------------------
10 > [[19. 22.][43. 50.]]
```

Placeholders são tensores indefinidos, os quais receberão um valor posteriormente. Eles são úteis para receber as amostras de entrada e a saída que serão utilizadas no grafo de computação da rede neural. A Listagem 4.2 mostra como usar um *placeholder* no *Tensorflow*. Na linha 4 um *placeholder* do tipo *float* é criado com as dimensões 3x3. Em seguida a biblioteca Numpy é usada para gerar um tensor 3x3 com valores aleatórios. Nas linhas 9 e 10, a sessão é criada e o grafo de computação é executado. Note que desta vez o parâmetro **feed_dict** é explicitamente definido. Esse parâmetro recebe como valor

um *dict* que possui como chave o *placeholder* criado, e como valor o *array* de valores aleatório chamado *rand_array*). Por fim, na linha 13 é mostrada a saída do programa.

Listagem 4.2: Usando placeholders no Tensorflow.

```
1  import tensorflow as tf
2  import numpy as np
3
4  a = tf.placeholder(tf.float32, shape=(3,3))
5  b = tf.matmul(a,a)
6
7  rand_array = np.random.rand(3,3)
8
9  sess = tf.Session()
10 result = sess.run(b, feed_dict={a: rand_array})
11 print(result)
12 ---------------------------- OUTPUT -----------------------------
13 > [[1.9946331 1.5735985 2.126033 ][1.9040278 1.517215  2.0398355]
14 [1.7058356 1.3416395 1.8197424]]
```

4.3. Fundamentos de Aprendizado de Máquina

Aprendizagem de máquina, ou aprendizado automático é um subcampo da área de Inteligência Artificial que automatiza a construção de modelos analíticos a partir dos dados. Em 1959, o cientista da computação Artur Samuel definiu o conceito de aprendizado de máquina como "campo de estudo que dá aos computadores a habilidade de aprender sem serem explicitamente programados" [**?**]. Em outras palavras, tais algoritmos operam construindo de forma automática um modelo interno a partir das amostras de entrada e fazem previsões guiadas pelos dados ao invés de seguir instruções programadas.

O aprendizado de máquina é usado em um vasto domínio onde algoritmos tradicionais são impraticáveis. Este minicurso é focado especialmente em problemas da área de sistemas multimídia, mas existem aplicações de que vão desde problemas relacionados a robótica até problemas de sequenciamento de DNA. As tarefas de aprendizado de máquina geralmente são categorizados em três tipos: supervisionado, não-supervisionado e por reforço.

Aprendizado supervisionado: O humano fornece amostras pré-classificadas a máquina. O objetivo é aprender uma função que mapeia a entrada para um tipo de saída. Exemplos de tarefas supervisionadas são: classificação e regressão.

Aprendizado não-supervisionado: O humano fornece apenas os dados, sem classificação. O objetivo é encontrar alguma estrutura nos dados. Técnicas de agrupamento (*clustering*) são tipicamente tarefas não-supervisionadas, pois os grupos descobertos não são conhecidos previamente.

Aprendizado por reforço: A máquina recebe sinais (premiações ou punições) de um ambiente dinâmico em que se deve desempenhar um determinado objetivo.

Este minicurso é especializado em aprendizado do tipo supervisionado e aborda principalmente os algoritmos baseados em redes neurais. Por consequência, são utilizados *datasets* anotados, os quais são divididos em conjuntos distintos e são usados para avaliar as capacidades de generalização dos modelos. No método de validação cruzada

geralmente os *datasets* são divididos em três conjuntos: treino, validação e teste. O conjunto de treino é usado para realizar o treinamento do algoritmo, o conjunto de validação é usado para verificar a capacidade de generalização o algoritmo ainda em tempo de treinamento. Após o treinamento o conjunto de teste é usado avaliar o modelo. No entanto, devido ao tamanho pequeno da maioria dos *dataset* utilizados neste minicurso, eles são divididos em apenas dois conjuntos, treino e teste.

4.4. Redes Neurais

Métodos modernos de redes neurais são considerados os mais importantes modelos da área aprendizado de máquina. No entanto, até antes de 2006, não era possível treinar redes neurais para superar técnicas de aprendizado de máquina mais tradicionais (e.g. SVM, árvore de decisão), exceto em alguns domínios de problemas especializados. O que mudou em 2006 foi a descoberta de métodos que possibilitaram o advento dos modelos baseados em redes neurais profundas, também conhecido como aprendizado profundo (em inglês *Deep Learning*). A evolução das redes neurais profundas permitiu que esse tipo de arquitetura se tornasse o principal modelo para tarefas de classificação que estão relacionadas às áreas de visão computacional, reconhecimento de fala e processamento de linguagem natural.

Nesta seção, apresentamos os conceitos básicos de redes neurais. Primeiro, na Subseção 4.4.1 apresentamos os modelos Perceptron e Perceptron de Múltiplas Camadas. Em seguida, na Subseção 4.4.2 é descrito o algoritmo de Retropropagação. Na Subseção 4.4.3, apresentamos um tipo de camada chamada *Softmax*. Por fim, na Subseção é descrita uma implementação que utiliza um Perceptron de Múltiplas Camadas para classificar pontos de um *dataset* artificial.

4.4.1. Perceptron

O Perceptron [Rosenblatt 1957], inventado em 1957 por Frank Rosenblatt, é a estrutura mais básica de uma Rede Neural. A Figura 4.1 ilustra a estrutura do Perceptron, cada entrada x possui um peso w associado. Em seguida é calculado o produto escalar entre os dados de entrada e seus pesos ($z = w_1x_1 + w_2x_2 + ... + w_nx_n = w^t.x$). Então uma função de ativação é aplicada sobre o produto escalar, resultando na saída do perceptron: $a_w(x) = ativ(z) = ativ(w^t.x)$. Algumas fontes da literatura utilizam uma entrada com valor constante 1 para representar o viés (em inglês, *bias*) do neurônio. Em fontes mais recentes, o *bias* é considerado, por padrão, um dado interno do neurônio, resultando na equação: $z = w^t.x + b$.

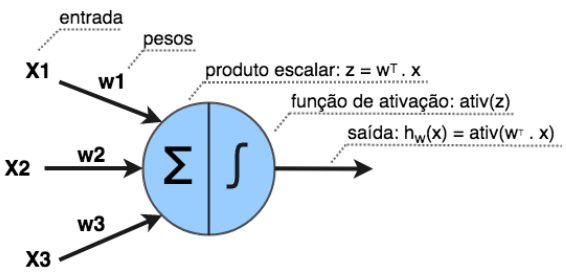


Figura 4.1: Estrutura de um Perceptron

Múltiplos Perceptrons podem ser usados para realizar tarefas de classificação múltipla. Nesse caso, como ilustra a rede A da Figura 4.2, os neurônios são organizados em paralelo e cada um fica responsável por aprender a ativar para uma classe específica. A classe predita é selecionada ao usar a função *argmax* para obter a maior ativação dentre todas as saídas dos neurônios. Esse modelo é conhecido como Perceptron Multiclasse. Adicionalmente, como ilustra a rede B da Figura 4.2, os neurônios também podem ser estruturados em múltiplas camadas, onde cada neurônio das camadas intermediárias (ou escondida) é conectado com todos os neurônios da camada anterior. Os dados da amostra de entrada são considerados os neurônios da camada de entrada, enquanto a última camada da rede é chamada de camada de saída. Nesse caso a rede aprende a aplicar uma hierarquia de transformações lineares ou não lineares (através das ativações) para gerar novas representações do dado de entrada, para que seja possível, por exemplo, realizar classificações. Esse modelo é conhecido como Perceptron de Múltiplas Camadas, ou MLP (do inglês, *Multilayer Perceptron*). Uma rede neural é considerada profunda quando ela possui mais de duas camadas escondidas.

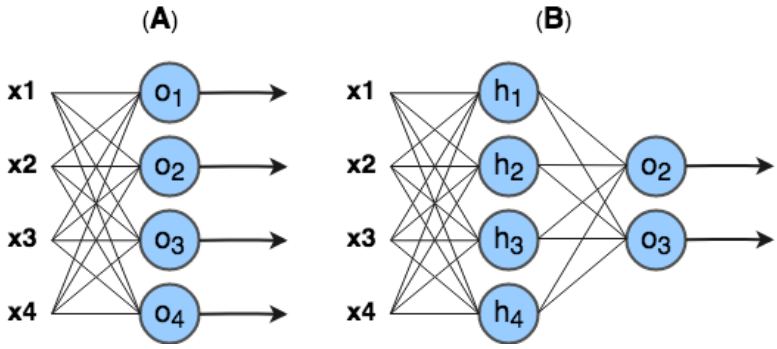


Figura 4.2: (A) Perceptron de múltiplas classes. (B) Perceptron múltiplas camadas.

A Figura 4.3[7] mostra as funções de ativação mais conhecidas. A função de ativação passo (*step*) foi utilizada na primeira versão do Perceptron. No entanto ela não oferece uma derivada útil para que possa ser usada para treinar modelos de múltiplas camadas. Funções de ativação logística (*sigmoid*) e tangente hiperbólica (*tanh*) tornaram-se populares nos anos 80 como aproximações mais suaves da função passo e permitiram a aplicação do algoritmo de retropropagação. Funções de ativação consideradas modernas como linear retificada (*ReLU*) e *maxout* são lineares por partes, computacionalmente baratas e funcionam bem na prática.

O algoritmo que permite o aprendizado da rede neural é chamado de retropropagação (em inglês, *backpropagation*). Basicamente o que se chama de "aprendizado" em redes neurais é o ajuste nos pesos (w) e *biases* (b) dos neurônios para aproximar a saída da rede de uma função $y(x)$ para toda entrada de treinamento x. Para quantificar o quão próximo a rede está do objetivo são utilizadas funções de custo (também conhecidas como funções de perda). Por exemplo, a Equação 1 descreve a função de custo quadrático, comumente utilizada em problemas de regressão. Já a equação 2, descreve a função de custo entropia cruzada, geralmente utilizada em problemas de classificação. No decorrer deste minicurso ambas as funções são utilizadas nas implementações dos projetos práticos.

[7]https://denizyuret.github.io/Knet.jl/latest/mlp.html

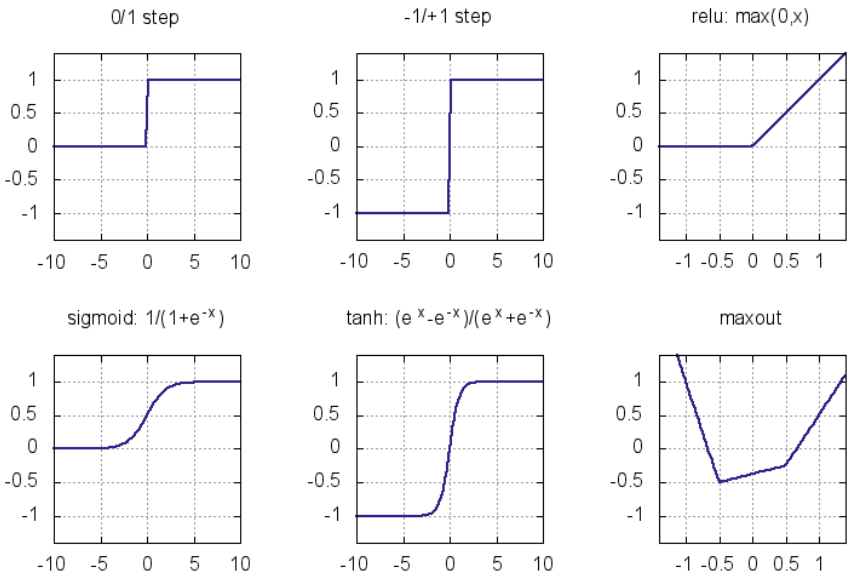


Figura 4.3: Funções de ativação.

$$\mathscr{J} = \frac{1}{2n}\sum_{x} || \, y(x) - a \, ||^2 \tag{1}$$

$$\mathscr{J} = -\frac{1}{n}\sum_{x}[y \log a + (1-y)\log(1-a)] \tag{2}$$

4.4.2. Algoritmo de Retropropagação

O *Tensorflow* encapsula o algoritmo de retropropagação, portanto não é necessário implementá-lo. No entanto, para entender redes neurais é inecessário entender com a retropropagação funciona. Basicamente, o algoritmo de retropropagação busca alterar os valores dos pesos e bias da rede para otimizar a função de custo. Este algoritmo realiza um procedimento para computar o δ_j^l (erro do j-ésimo neurônio da l-ésima camada) e então o relaciona com as derivadas parciais dos pesos e bias em relação a função de custo ($\frac{\partial C}{\partial w_{jk}^l}$ e $\frac{\partial C}{\partial b_j^l}$).

A equação do erro δ^L na camada de saída é dada por:

$$\delta_j^L = \frac{\partial C}{\delta a_j^l}\sigma'(z_j^L) \tag{3}$$

O primeiro termo $\frac{\partial C}{\delta a_j^l}$ mede quão importante é a saída de ativação do j-ésimo neurônio para a função de custo C. Se por exemplo, a saída de um neurônio não contribui para a função de custo, então δ_j^L será pequeno. De forma similar, o segundo termo, $\sigma'(z_j^L)$, mede quão importante é o produto escalar do j-ésimo neurônio para sua saída de ativação.

Ao reescrever a fórmula anterior para uma versão baseada em matriz, obtém-se:

$$\delta^L = \bigtriangledown_a C \odot \sigma'(z^L) \tag{4}$$

Onde, $\bigtriangledown_a C$ é um vetor das derivadas parciais $\frac{\partial C}{\delta a^l_j}$ e o símbolo $\odot$ denota multiplicação elementar entre vetores.

A equação do erro δ^l em relação ao erro na próxima camada (δ^{l+1}) é dada por:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{5}$$

Onde, $((w^{l+1})^T$ é a transposta da matriz de pesos e o δ^{l+1} é o erro da (l+1)-ésima camada. Essa equação permite que o erro seja retropropagado através da rede. Ao usar a equação (1) para computar o δ^L, e então usar a equação (2) em sequência para computar $\delta^{L-1}, \delta^{L-2}, \delta^{L-3}, ..., \delta^1$.

A equação para mudar o custo em relação a qualquer bias e peso são dadas respectivamente por:

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{6}$$

Isto é, o erro δ^l_j é igual a mudança $\frac{\partial C}{\partial w^l_{jk}}$.

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{7}$$

O termo $\frac{\partial C}{\partial w^l_{jk}}$ é calculado em relação ao erro δ^l_j e ativação da camada anterior a^{l-1}_k. Dessa forma, quando a ativação da camada anterior é pequena, espera-se que o termo do gradiente $\frac{\partial C}{\partial w^l_{jk}}$ tenda a ser pequeno.

Com base nas 4 equações descritas, o algoritmo de retropropagação é resumido nos cinco passos seguintes:

1. **Entrada**: Inserção do X (entrada) na rede e cálculo da ativação da camada de entrada.

2. **Propagação**: Para cada camada l = 2, 3, ..., L, calcular a ativação dos neurônios recebendo a ativação dos neurônio da camada anterior como entrada ($z^l = w^l a^{l-1}$ e $a^l = \sigma(z^l)$).

3. **Erro da saída**: Calcular o vetor de erro $\delta^L = \bigtriangledown_a C \odot \sigma'(z^L)$.

4. **Retropropagação do erro**: Para cada camada l = L-1, L-2, ..., 2, calcular o erro da camada $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.

5. **Atualização dos pesos e bias**: Atualizar os pesos e bias com os respectivos gradientes: $\frac{\partial C}{\partial b^l_j} = \delta^l_j$ e $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$.

4.4.3. Camada *Softmax*

Nesta subseção é apresentada a camada *softmax*, um importante recurso usado em redes que realizam classificação. A ideia do *softmax* é definir uma nova camada saída para a rede com neurônios que usam a função de ativação softmax, a qual é descrita como:

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} \tag{8}$$

Onde o denominador da equação é a soma do produto escalar de todos os neurônios da camada *softmax*. Como resultado, o vetor de saída da camada *softmax* pode ser interpretadas como uma distribuição probabilística. Por exemplo, considerando a camada *softmax* da rede ilustrada na Figura 4.4, se o vetor do produto escalar $(z_1^L, z_2^L, z_3^L, z_4^L) = (0.1, 0.9, 0.4, 2.3)$, então o vetor de ativação $(a_1^L, a_2^L, a_3^L, a_4^L) = (0.07, 0.16, 0.09, 0.66)$. Isso significa que dada entrada X tem 66% de chance de ser da classe 4.

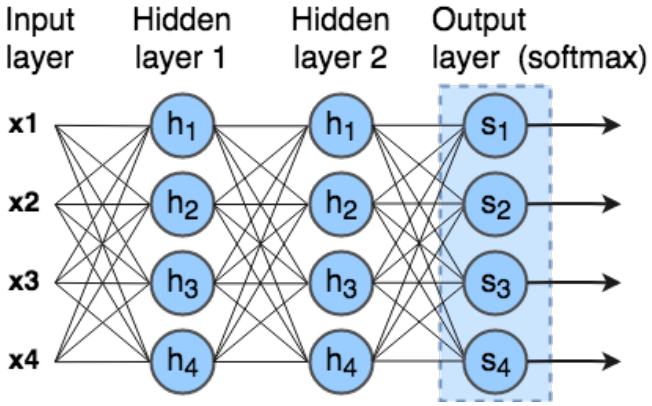


Figura 4.4: MLP Moderno com uma camada *softmax* para classificação.

4.4.4. Implementando um MLP

Nesta subseção é exemplificado o uso de um MLP para resolver um problema não linearmente separável. A Listagem 4.3 mostra o código que cria um pequeno dataset com pontos distribuídos de forma circular usando a biblioteca scikitlearn[8], que pode ser visualizado na Figura 4.5. O dataset é composto por pontos de duas dimensões (duas *features*) que pertencem a dois conjuntos de classe, vermelho (0) e azul (1).

Listagem 4.3: Criando um dataset não linearmente separável.

```
1 import tensorflow as tf
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import sklearn.datasets
5
6 from aux import draw_separator
7
8 # Setando o seed para gerar uma êsequncia  conhecida
9 tf.set_random_seed(0)
```

[8]http://scikit-learn.org/

```
10 np.random.seed(0)
11
12 # numero de classes do nosso problema
13 num_classes = 2
14
15 # gerando o dataset
16 dataset_X, dataset_Y = sklearn.datasets.make_circles(200, noise=0.05)
17 # plotando o dataset
18 plt.scatter(dataset_X[:,0], dataset_X[:,1], s=40, c=dataset_Y,
19              cmap=plt.cm.Spectral)
```

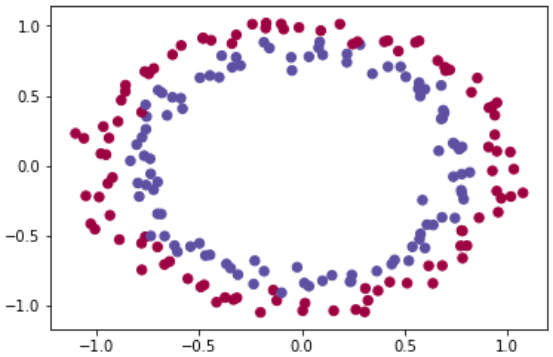


Figura 4.5: Visualização do dataset criado na Listagem 4.3

A Listagem 4.4 descreve a implementação de uma rede neural do tipo MLP. A função "build_net" recebe como parâmetro a quantidade de *features* e classes usadas no problema, neste caso, tanto a quantidade de *features* quanto a de classes são iguais a 2. Nas linhas 4 e 5 são definidos os *placeholders* do grafo de computação. O "X_placeholder" corresponde a camada de entrada da rede, enquanto o "Y_placeholder" é o vetor que contém os *labels* do dataset, e serão utilizados para comparação com a saída da rede. Na linha 8 é definida a camada escondida da rede, a qual possui cem unidades e usa a função de ativação ReLU. Na linha 11 é definida a camada de saída da rede, que neste exemplo possui apenas duas unidades. Na linha 15 o vetor de labels é convertido para o formato *OneHot* (por exemplo, a label 0 se torna o vetor [1,0] e o label 1 se torna o vetor [0,1]), dessa forma é possível a comparação com a saída da rede. Em seguida, na linha 17 é definida a função de perda, a função Entropia Cruzada é usada em conjunto com uma camada *softmax*. Na linha 20 é definido o otimizador da rede. Nas linhas 23 e 24 são definidos os tensores que classificam um exemplo de entrada da rede. Por fim, nas linhas 27 e 28, são definidos os tensores que calculam a acurácia da rede.

Listagem 4.4: Construindo um MLP.

```
1 def build_net(n_features, n_classes):
2
3     # Placeholders
4     X_placeholder = tf.placeholder(dtype=tf.float32, shape=[None,
          n_features])
5     Y_placeholder = tf.placeholder(dtype=tf.int64, shape=[None])
6
7     # camada oculta
```

```
8     layer1 = tf.layers.dense(X_placeholder, 100, activation=tf.nn.relu)
9
10    # camada de saida
11    out = tf.layers.dense(layer1, n_classes, name="output")
12
13    # adaptando o vetor Y para o modelo One-Hot Label
14    one_hot = tf.one_hot(Y_placeholder, depth=n_classes)
15
16    # funcao de perda/custo/erro
17    loss = tf.losses.softmax_cross_entropy(onehot_labels=one_hot,
          logits=out)
18
19    # otimizador
20    opt = tf.train.GradientDescentOptimizer(learning_rate=0.07).
          minimize(loss)
21
22    # classe de exemplo
23    softmax = tf.nn.softmax(out)
24    class_ = tf.argmax(softmax,1)
25
26    # acuracia
27    compare_prediction = tf.equal(class_, Y_placeholder)
28    accuracy = tf.reduce_mean(tf.cast(compare_prediction, tf.float32))
29
30    return X_placeholder, Y_placeholder, loss, opt, class_, accuracy
```

A Listagem 4.5 mostra como iniciar o TensorFlow e carregar o modelo de MLP criado. Na linha 2 é iniciada uma sessão interativa do TensorFlow. Em seguida, na linha 5 é obtida a quantidade de *features* do dataset. Na linha 8 o modelo de MLP criado na listagem anterior é carregado. Por fim, na linha 11, as variáveis do TensorFlow são inicializadas.

Listagem 4.5: Iniciando o TensorFlow e carregando o MLP.

```
1  # iniciando a sessao
2  sess = tf.InteractiveSession()
3
4  # obtendo o numero de features
5  n_features = dataset_X.shape[1]
6
7  # carregando o modelo
8  X_placeholder, Y_placeholder, loss, opt, class_,
9            accuracy = build_net(n_features,num_classes)
10 # inicializando as ávariveis
11 sess.run(tf.global_variables_initializer())
```

A Listagem 4.6 mostra o código que realiza treinamento da rede. Um laço executa o treinamento da rede mil vezes (mil épocas) usando todo o *dataset*. Vale ressaltar que devido ao tamanho pequeno do *dataset*, neste exemplo, o método de dividir o *dataset* em lotes não é usado. A cada 100 épocas o erro da rede é calculado e impresso. Ao fim do treinamento a acurácia da rede é calculada e impressa. A linha 21 exemplifica como utilizar o modelo para realizar uma classificação. Em seguida, na linha 24, a função "draw_separator" desenha o dataset separado pelo modelo, o qual pode ser visto

na Figura 4.6. Por fim, as linhas 27-37 mostram a saída do programa.

Listagem 4.6: Treinamento do MLP.

```
1  # definindo o numero de epocas
2  epochs = 1000
3  for i in range(epochs):
4
5      # treinamento (OBS: mini-batch nao usado por causa do tamanho
           pequeno do dataset)
6      sess.run(opt, feed_dict={X_placeholder: dataset_X,
7                   Y_placeholder: dataset_Y})
8
9      # a cada 100 epocas o erro e impresso
10     if  i % 100 == 0:
11         erro_train = sess.run(loss, feed_dict={X_placeholder: dataset_X
               ,
12                             Y_placeholder: dataset_Y})
13         print("O erro na epoca", i,":", erro_train)
14
15 # calculando a acuracia
16 acc = sess.run(accuracy, feed_dict={X_placeholder: dataset_X,
17                       Y_placeholder: dataset_Y})
18 print("acuracia do modelo:", acc)
19
20 cla = sess.run(class_, feed_dict={X_placeholder: dataset_X[:1]})
21 print("a classe do ponto", dataset_X[:1], "e:", cla)
22
23 # desenhando o separador
24 draw_separator(dataset_X, dataset_Y, sess, X_placeholder, class_)
25
26 ---------------------------- OUTPUT -----------------------------
27 > O erro na epoca 0 : 0.69493294
28 > O erro na epoca 100 : 0.67670804
29 > O erro na epoca 200 : 0.6603513
30 > O erro na epoca 300 : 0.643817
31 > O erro na epoca 400 : 0.6265911
32 > O erro na epoca 500 : 0.60751265
33 > O erro na epoca 600 : 0.58588564
34 > O erro na epoca 700 : 0.56282145
35 > O erro na epoca 800 : 0.5376183
36 > O erro na epoca 900 : 0.510537
37 > acuracia do modelo: 0.97
38 > a classe do ponto [[0.4013312  0.88583093]] e: [0]
```

4.5. Redes Neurais Convolucionais

Redes Neural Convolucionais (CNNs ou ConvNets) são redes especializadas no processamento de dados que são comumente organizados em topologia de grade (no caso mais comum, imagens). Esse modelo recebe este nome porque faz uso de uma operação matemática chamada convolução. As camadas de convolução possibilitam que uma CNN e encontre *features* de baixo nível nas primeiras camadas e então as compõem em *features* de mais alto nível ao decorrer da rede. A habilidade de encontrar uma estrutura hierárquica de *features* é o principal motivo pelo qual CNNs funcionam tão bem para

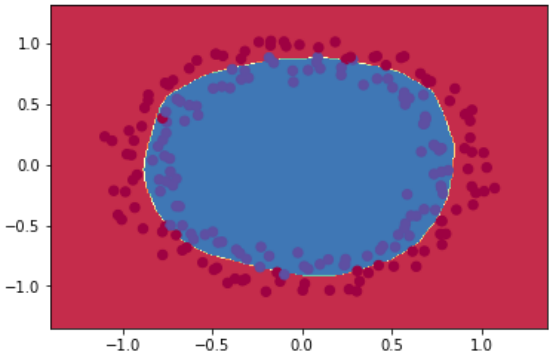


Figura 4.6: dataset separado pelo MLP.

reconhecimento de padrões em imagens.

Nesta Seção, apresentamos os fundamentos básicos e técnicas para implementação de CNNs. Na Subseção 4.5.1 a operação de convolução e *pooling* é apresentada. Em seguida, na Subseção 4.5.2 descreve a implementação de uma CNN para classificação de imagens de sinais de mão. Por fim, na Subseção 4.5.2.1 é apresentado o funcionamento e histórico de evolução da rede InceptionNet.

4.5.1. Camadas de Convolução e Pooling

A convolução consiste de um operador linear que, a partir de duas funções, resulta numa terceira que é a soma do produto dessas funções ao longo da região subentendida pela superposição delas em função do deslocamento existente entre elas. Para funções contínuas, a convolução é definida como a integral do produto de uma das funções por uma cópia deslocada e invertida da outra:

$$s(t) = (x * w)(t) = \int x(a)w(t-a)da$$

Para funções de domínio discreto, a convolução é dada por:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

Em CNNs a operação de convolução é feita em mais de uma dimensão por vez. Os pesos dos neurônios são representadas por um tensor chamado *kernel* (ou *filter*). O processo de convolução entre os neurônios e os *kernels* produzem saídas chamadas de mapas de *features*. Especificamente, baseado na equação discreta da convolução, a saída de um neurônio localizado na linha i, coluna j do mapa de *features* k em dada camada de convolução l é dada pela equação:

$$z_{i,j,k} = b_k + \sum_{u=1}^{f_h} \sum_{v=1}^{f_w} \sum_{k'=1}^{f'_n} x_{i',j',k'} . w_{u,v,k',k}$$

onde:

- $z_{i,j,k}$ é a saída do neurônio localizado na linha i, coluna j e no mapa de *features* k da camada convolucional l;
- $x_{i',j',k'}$ é a saída do neurônio localizado na linha i', coluna j' e no mapa de *features* k' da camada anterior (l -1);
- $w_{u,v,k',k}$ é o peso de conexão entre qualquer neurônio do mapa de *features* k da camada l e sua entrada localizada na linha u, coluna v e mapa de *features* k'.
- b_k é o bias para o mapa de *features* k na camada l
- Os parâmetros s_h e s_w representam os *strides* (deslocamentos) verticais e horizontais, f_h f_w são a altura e largura do *kernel*, e $f_{n'}$ é o número de mapa de *features* na camada anterior.

A Figura 4.7 mostra um exemplo de convolução entre dois tensores 2D. O *kernel* (em azul) tem dimensões (2,2), o tensor de entrada (i) tem dimensões (3,3) e o *stride* é igual a 1. Na primeira iteração, a saída (o) descrita pelo cálculo: $(1 \times 3) + (-1 \times 7) + (-1 \times 10) + (1 \times 8)$ = -6; Na segunda iteração, $(1 \times 7) + (-1 \times 4) + (-1 \times 8) + (1 \times 11)$ = 6; Na terceira iteração, $(1 \times 10) + (-1 \times 8) + (-1 \times 12) + (1 \times 1)$ = -9. E por fim, na quarta iteração, $(1 \times 8) + (-1 \times 11) + (-1 \times 1) + (1 \times 2)$ = -2. Note que o tensor de saída possui dimensões diferentes do tensor de entrada, isso ocorre porque ...

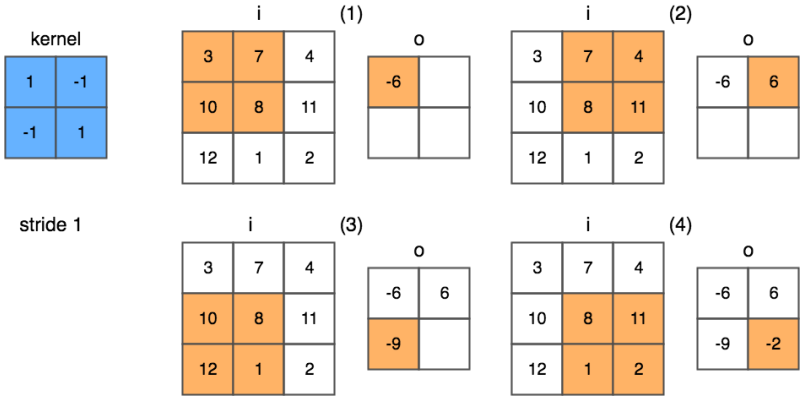


Figura 4.7: Processo de convolução.

Em CNNs a camadas de *pooling* tem a função de reduzir a dimensionalidade dos mapas de features para diminuir a carga computacional, uso de memória e número de parâmetros (dessa forma, reduzindo o risco de *overfitting*). Além disso, a redução de dimensionalidade permite que a rede tolere pequenas mudanças nos mapas de *features* (invariância de localização).

As camadas de *pooling* operam de forma semelhante as camadas de convolução, com a diferença que os *pooling kernels* não possuem pesos. Os *pooling kernels* agregam a entrada através de funções de agregação, como *max* ou *mean*. A função *max pooling*, por exemplo, retorna o maior valor dentro de uma área do tensor. Outras funções de *pooling* incluem, por exemplo, a média ou a distância L^2 entre os elementos de uma área do tensor. A Figura 4.8 ilustra um exemplo do processo de *max pooling*. Cada área colorida

representa uma etapa da operação que usa um *pooling kernel* com dimensões 2x2 e stride 2. Na área de cor laranja o maior valor é 28; Em seguida, na área de cor verde, 21; Na área azul, 27; E por fim, na área lilás, 17.

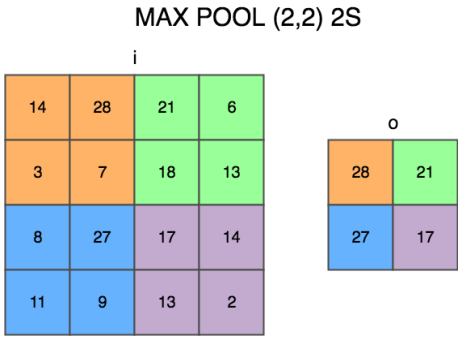


Figura 4.8: Exemplo de um processo de max *pooling* com *kernel* 2x2 e *stride* 2.

É importante também definir o conceito do que é um tensor nesse contexto. Uma rede neural é formada por um conjunto de matrizes, porém, estas podem ter mais de duas dimensões, então nos referenciaremos a elas como tensores. Operações são feitas em tensores para gerar os tensores seguintes, podendo essas reduzir ou aumentar seu numero de dimensões. Ao diminuir as dimensões em um tensor se espera representar um fator comparativo entre valores do tensor anterior para gerar o seguinte. Tal redução representa o aumento da semântica nos dados. Tal conjunto de regras registrada nos pesos, formada pela associa cão de valores, carrega informações sobre o contexto de forma que sejam uteis para diminuir o erro resultante da saída no final da rede.

A Figura 4.9 ilustra a arquitetura de uma CNN que usa camadas de convolução e *pooling*. A entrada da rede consiste de um tensor 16x16x3, correspondente a uma imagem com 16 de altura, 16 de largura e 3 canais (RGB). A primeira convolução usa 8 kernels com dimensões (4,4) e *stride* 1 seguido de uma função de ativação ReLu, resultando em 8 mapas de features com dimensões 16x16. Em seguida, é aplicada uma camada de *pooling* que usa um kernel com dimensões (4,4) e *stride* 4, resultando em mapas de *features* com dimensões reduzidas (4,4). A próxima camada de convolução usa 4 *kernels* (2,2) seguido de um ReLU, resultando em 4 mapas de features com dimensões 4x4. Por fim, uma última camada de *pooling* usa um kernel(4,4) e *stride* 1, resultando em 4 mapas de *features* 1x1. A última camada é então conectada a uma camada de saída *Fully Connected* (FC).

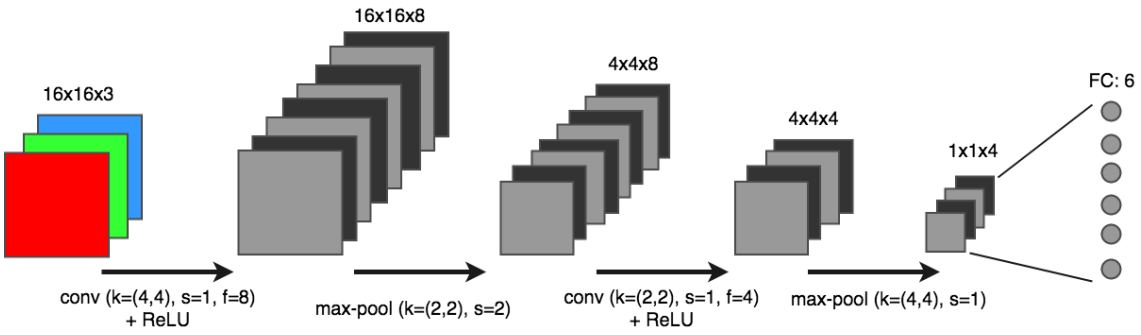


Figura 4.9: Exemplo de uma arquitetura de CNN.

4.5.2. Implementando uma Rede Neural Convolucional

Nesta subseção é exemplificada a implementação de uma CNN para reconhecer imagens de sinais de mãos. O *dataset* usado neste exemplo foi obtido no curso de especialização em Deep Learning do professor Andrew Ng. (deeplearning.ai) [9]. O *dataset* é composto por 1200 fotos de sinais de mão no formato RGB com dimensões 64x64. A Figura 4.10 ilustra os seis tipos de sinais de mãos encontrados no *dataset*, bem como os respectivos vetores de *labels* codificados no formato *OneHot*.

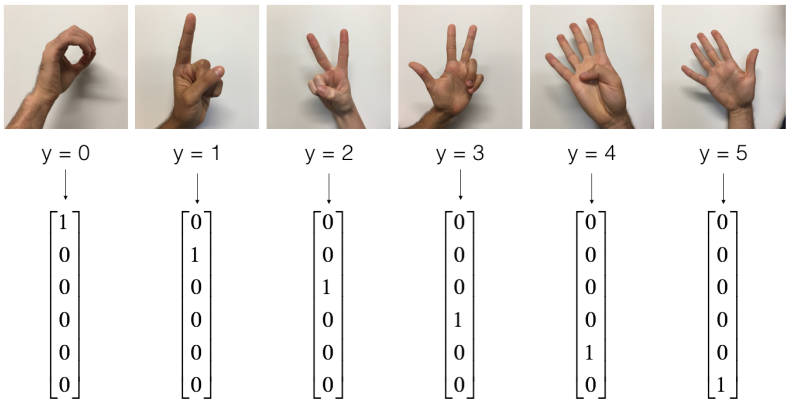


Figura 4.10: Exemplos de cada classe do *dataset* de sinais de mão e suas respectivas codificações OneHot (by Prof. Andrew Ng., deeplearning.ai).

A Listagem 4.7 mostra o código que carrega o dataset de sinais de mão. Nas linhas 1-10 são definidas as bibliotecas necessárias para implementação do exemplo. Na linha 17 o *dataset* é carregado. Em seguida, nas linhas 20-23 as dimensões dos conjuntos de treino e teste são impressas. Nas linhas 26-28 é chamada uma função que mostra uma imagem do conjunto de treino, bem como sua classe. Por fim, as linhas 30-34 mostram a saída do programa.

Listagem 4.7: Carregando o *dataset* de sinais de mão.

```
1  import math
2  import numpy as np
3  import h5py
4  import matplotlib.pyplot as plt
5  import scipy
6  from PIL import Image
7  from scipy import ndimage
8  import tensorflow as tf
9  from tensorflow.python.framework import ops
10 from cnn_utils import *
11
12 # setando o seed para gerar uma êsequncia conhecida
13 tf.set_random_seed(0)
14 np.random.seed(0)
15
16 # carregando o dataset (dividido em treino e teste)
```

[9] https://www.deeplearning.ai/

```
X_train, Y_train, X_test, Y_test, classes = load_dataset()

# imprimindo as õdimenses dos conjuntos de treino e teste do dataset
print ("X_train shape: " + str(X_train.shape))
print ("Y_train shape: " + str(Y_train.shape))
print ("X_test shape: " + str(X_test.shape))
print ("Y_test shape: " + str(Y_test.shape))

# exibindo um exemplo
index = 6
plt.imshow(X_train[index])
print ("y =", Y_train[index])
 ---------------------------- OUTPUT ----------------------------
> X_train shape: (1080, 64, 64, 3)
> Y_train shape: (1080,)
> X_test shape: (120, 64, 64, 3)
> Y_test shape: (120,)
> y = 2
```

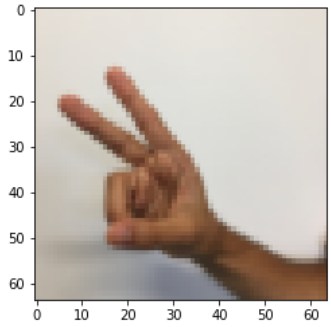


Figura 4.11: Exemplo de imagem renderizada na Listagem 4.7.

A Listagem 4.8 mostra a construção de uma simples arquitetura de CNN. A função "build_cnn" recebe como parâmetro a largura, altura e número de canais da imagem de entrada, bem como o número de classes do problema. Nas linhas 3 e 4 são definidos os *placeholders* da CNN. O "X" é o tensor que armazena as imagens de entrada da rede. Enquanto o "Y" é o vetor que contém as *labels* das imagens de entrada. Entre as linhas 11-26 estão definidas as camadas de convolução e *pooling* da rede. Todas as camadas usam *padding* SAME, ativação ReLU e são inicializadas com o método Xavier. A primeira e a segunda camadas de convolução (linha 11 e 15) tem um *kernel* de dimensões (4,4) e 32 *filters*. Em seguida, na linha 19 é definida a primeira camada de *pooling*, que possui um *kernel* de dimensões (8,8) e usa *stride* 4. A terceira camada de convolução (linha 22) tem um kernel de dimensões (2,2) e 16 *filters*. Por fim, a última camada de *pooling* possui um *kernel* de dimensões (8,8) e também usa *stride* 8. O restante do código possui as definições da função de custo, otimizador e demais tensores já explicados nos exemplos anteriores deste capítulo.

Listagem 4.8: Construindo uma CNN.

```
def build_cnn(input_width, input_height, input_channels, n_classes):

    #placeholders
```

```
4     X = tf.placeholder(tf.float32, shape=(None, input_width,
          input_height, input_channels))
5     Y = tf.placeholder(tf.int64, shape=(None))
6
7     initializer = tf.contrib.layers.xavier_initializer(seed = 0)
8
9   #camada convolucao 1
10    conv2d_1 = tf.layers.conv2d(inputs=X, filters=32, kernel_size
          =[4,4],
11    strides=1, activation=tf.nn.relu, padding = 'SAME',
          kernel_initializer=initializer)
12    #camada convolucao 2
13    conv2d_2 = tf.layers.conv2d(inputs=conv2d_1, filters=32,
          kernel_size=[4,4],
14    strides=2, activation=tf.nn.relu, padding = 'SAME',
          kernel_initializer=initializer)
15    #camada pooling 1
16    maxpool_1 = tf.layers.max_pooling2d(inputs=conv2d_2, pool_size=[8,
          8], strides=4, padding = 'SAME')
17  #camada convolucao 3
18    conv2d_3 = tf.layers.conv2d(inputs=maxpool_1, filters=16,
          kernel_size=[2,2]
19    ,strides=1, activation=tf.nn.relu, padding = 'SAME',
          kernel_initializer=initializer)
20    #camada pooling 1
21    maxpool_2 = tf.layers.max_pooling2d(inputs=conv2d_3, pool_size=[8,
          8], strides=8, padding = 'SAME')
22
23    #flatten
24    flatten = tf.contrib.layers.flatten(maxpool_2)
25
26    #output (fully_connected)
27    out = tf.contrib.layers.fully_connected(flatten, num_outputs=
          n_classes, activation_fn=None)
28
29    #adaptando o Label Y para o modelo One-Hot Label
30    one_hot = tf.one_hot(Y, depth=n_classes)
31
32    #funco de perda/custo/erro
33    loss = tf.losses.softmax_cross_entropy(onehot_labels=one_hot,
          logits=out)
34
35    #Otimizador
36    opt = tf.train.AdamOptimizer(learning_rate=0.001).minimize(loss)
37
38    #Softmax
39    softmax = tf.nn.softmax(out)
40
41    #Classe
42    class_ = tf.argmax(softmax,1)
43
44    #áAcurcia
45    compare_prediction = tf.equal(class_, Y)
46    accuracy = tf.reduce_mean(tf.cast(compare_prediction, tf.float32))
47
```

```
48      return X, Y, loss, opt, softmax, class_, accuracy
```

A Listagem 4.9 mostra o código que inicializa o TensorFlow e carrega a CNN definida na listagem anterior. Nas linhas 2 e 3 os valores dos pixels das imagens são normalizados para valores entre 0 e 1. Na linha 6 é iniciada uma sessão interativa do TensorFlow. Na linha 9 o modelo de CNN é carregado. E por fim, na linha 13 as variáveis do TensorFlow são inicializadas.

Listagem 4.9: Iniciando o TensorFlow e carregando a CNN.

```
1  #normalizando os dados de entrada
2  X_train = X_train/255.
3  X_test = X_test/255.
4
5  #Iniciando
6  sess = tf.InteractiveSession()
7
8  #carregando o modelo de CNN
9  X, Y, loss, opt, softmax, class_, accuracy =
10             build_cnn(64,64,3,6)
11
12 # inicializando as variveis do tensorflow
13 sess.run(tf.global_variables_initializer())
```

A Listagem 4.10 mostra o código que realiza o treinamento da CNN com o *dataset*. Neste exemplo de implementação o modelo é treinado em 100 épocas. Em cada época, como definido nas linhas 7 e 8, uma lista de *mini-batch* é gerada. Em seguida, a rede é treinada com cada *mini-batch*. O erro do treinamento é impresso a cada 10 épocas. Ao fim do treinamento, a acurácia da CNN treinada é impressa.

Listagem 4.10: Treinando a CNN.

```
1  #definindo o numero de epocas
2  epochs = 100
3
4  seed=0
5  for i in range(epochs):
6      #gerando um mini-batch aleatorio
7      seed = seed + 1
8      minibatches = random_mini_batches(X_train, Y_train, 64, seed)
9      #treinando a rede com cada minibatch
10     for minibatch in minibatches:
11         (minibatch_X, minibatch_Y) = minibatch
12         sess.run(opt, feed_dict={X_placeholder: minibatch_X,
13                  Y_placeholder: minibatch_Y})
14
15     # imprimindo o erro a cada 100 épocas
16     if  i % 10 == 0:
17         erro_train = sess.run(loss, feed_dict={X: X_train, Y: Y_train})
18         print("erro na epoca", i,":", erro_train)
19
20
21 #calculando a acuracia da rede
22 acc = sess.run(accuracy, feed_dict={X: X_test, Y: Y_test})
```

```
print("accurcia do modelo:", acc)
---------------------------- OUTPUT -----------------------------
> erro na epoca 0 : 1.79012
> erro na epoca 10 : 1.53382
> erro na epoca 20 : 0.809482
> erro na epoca 30 : 0.586155
> erro na epoca 40 : 0.505508
> erro na epoca 50 : 0.366477
> erro na epoca 60 : 0.29243
> erro na epoca 70 : 0.240354
> erro na epoca 80 : 0.21093
> erro na epoca 90 : 0.143943
> accurcia do modelo: 0.908333
```

A Listagem 4.11 mostra como utilizar a CNN recém treinada para realizar classificações de sinais de mão. Na linha 4 é feita a predição da classe de uma imagem de entrada. Em seguida a imagem que foi utilizada é desenhada na tela (Figura 4.12). Por fim, a linha 9 mostra a saída do programa.

Listagem 4.11: Usando a CNN treinada para classificar imagens.

```
1 index = 10
2 example = X_test[index:(index+1)]
3
4 cla = sess.run(class_, feed_dict={X: example})
5 print("classe:", cla)
6
7 plt.imshow(X_test[index])
8  ---------------------------- OUTPUT -----------------------------
9 > classe: [5]
```

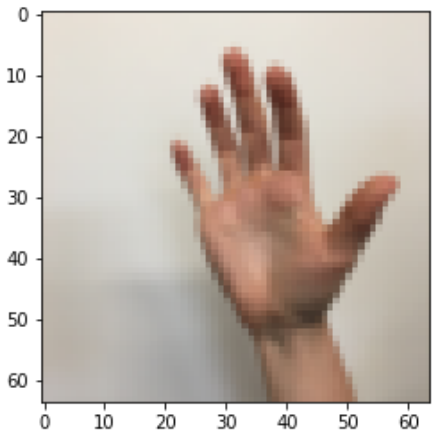


Figura 4.12: Imagem classificada como sinal 5 na Listagem 4.11.

4.5.2.1. Evolução da rede Inception

A primeira versão da rede InceptionNet (ou GoogleNet) [Szegedy et al. 2015] foi a campeã do desafio ImageNet 2014. Essa arquitetura é considerada importante porque

ataca o problema de localização da informação, pois elementos na imagem podem ter grande variedades de tamanhos. Como pode ser visto na Figura 4.13, por exemplo, a área ocupada por um cão é diferente em cada imagem. Por causa dessa variedade, escolher um tamanho de *kernel* apropriado se torna difícil. Um *kernel* largo é adequado quando a informação está distribuída globalmente, e um *kernel* curto quando a informação está distribuída mais localmente.

Figura 4.13: Esquerda: cão ocupando quase toda a imagem; Centro: cão ocupando uma parte da imagem; Direita: cão ocupando uma pequena parte da imagem.

A solução proposta pela rede InceptionNet é de usar *kernels* de diferentes tamanhos no mesmo nível, deixando a rede um pouco mais larga que profunda. Para isso, os autores da rede projetaram o modulo Inception, o qual é ilustrado na Figura 4.14. O módulo Inception realiza convoluções com três diferentes tamanhos de *kernel*, 1x1, 3x3 e 5x5. Adicionalmente é feito um *maxpooling* com um *kernel* 3x3. Para não deixar o processamento tão pesado, ele limita o número de camadas da entrada usando uma convolução 1x1 antes das convoluções 3x3 e 5x5. Apenas no *maxpooling* a convolução 1x1 é realizada posteriormente.

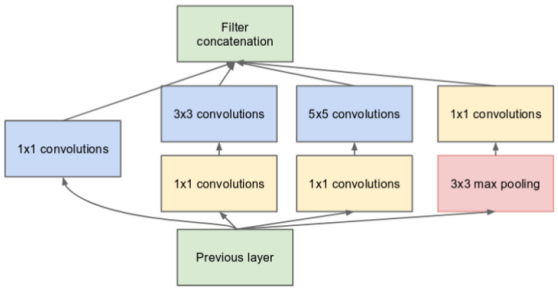


Figura 4.14: Módulo Inception da rede InceptionNet.

Como pode ser visto na Figura 4.15, a rede InceptionNet usa 9 módulos Inception em sequência, resultando em 27 camadas. Por ser uma rede profunda ela esta sujeita ao problema de desaparecimento do gradiente. Para resolver isso, os autores adicionaram duas saídas com classificadores auxiliares na saída de dois módulos Inception. O custo total da rede é dado pela soma ponderada entre o custo dado pelas três saídas da rede.

A arquitetura InceptionNet v2 [Szegedy et al. 2016] tenta reduzir o impacto de um problema conhecido como "gargalo representacional". CNNs funcionam melhor quando as convoluções não alteram a dimensão da entrada de forma drástica, pois essa redução pode causar perda de informação. Para isso, os autores criaram três novas versões do módulo Inception, que refatoraram as convoluções 5x5 em duas convoluções menores. Como

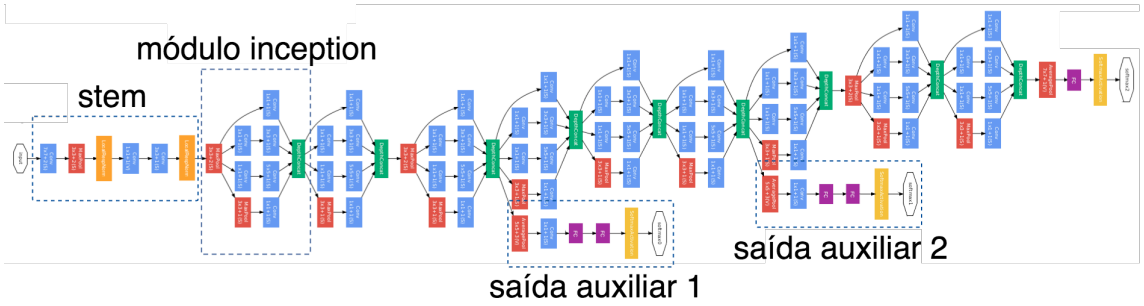


Figura 4.15: Arquitetura da rede InceptionNet (GoogleNet).

pode ser visto na Figura 4.16, no módulo A a convoluções 5x5 foi substituída por uma sequência de duas convoluções 3x3, o que implica em uma melhora de performance, já que uma convolução 5x5 é 2.78 vezes computacionalmente mais cara que uma convolução 3x3. Já no módulo B, os autores substituíram cada convolução 3x3 por uma sequência de 1xn seguida de uma nx1. Por fim, no módulo C a posição das camadas de convolução foram alteradas, de forma que ficaram mais esparsas do que profunda. Essa decisão de projeto tenta suavizar o gargalo representacional, pois se o módulo fosse mais profundo, haveria redução excessiva nas dimensões e, consequentemente, perda de informação.

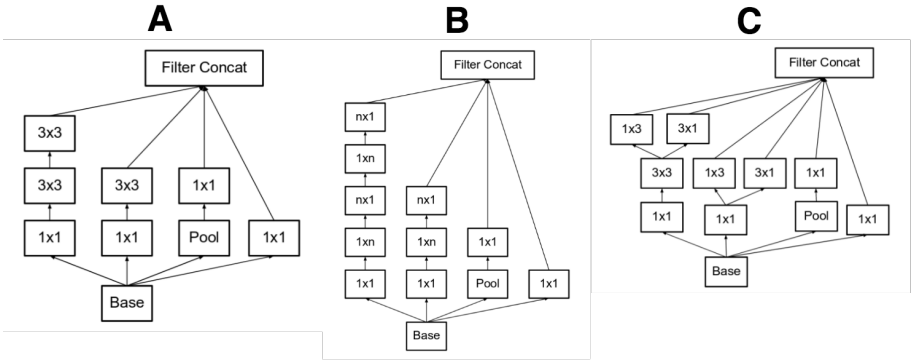


Figura 4.16: Três tipos de módulo inception da arquitetura InceptionNet v2.

A terceira versão, chamada de InceptionNet v3 [Szegedy et al. 2016] adatou o otimizador RMSProp, refatorou as convoluções 7x7 e aplicou a técnica de *Batch Normalization* (*BatchNorm*) as saídas auxiliares. Os autores da rede constataram que as saídas auxiliares não contribuíram muito até o final do processo de treinamento, quando as acurácias se aproximam da saturação. Eles argumentam que elas podem funcionar como regularizes, especialmente se eles tiverem operações *BatchNorm* ou *Dropout*.

A quarta versão, chamada InceptionNet v4 [Szegedy et al. 2017] reformulou alguns módulos da arquitetura. A Figura 4.17 ilustra a arquitetura geral da rede. A Figura 4.18 detalha cada bloco da rede. A InceptionNet v4 apresenta um bloco *Stem* modificado. Os módulo Inception A, B e C são similares aos das versões anteriores. Uma novidade proposta pelo InceptionNet v4 é a definição dos módulos de redução, que são usados para diminuir a dimensionalidade dos mapas de *features*. As versões anteriores já tinham essa funcionalidade, mas ela não estava explicitamente formalizada como um módulo da rede.

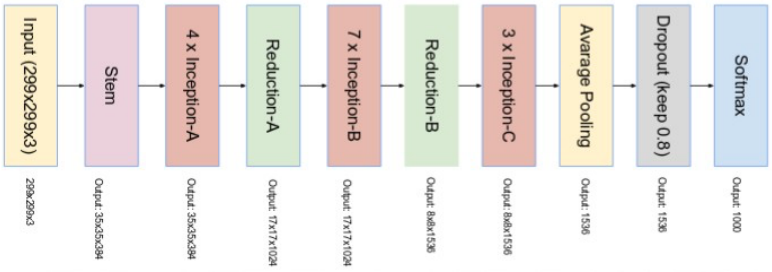


Figura 4.17: Arquitetura da rede InceptionNet v4.

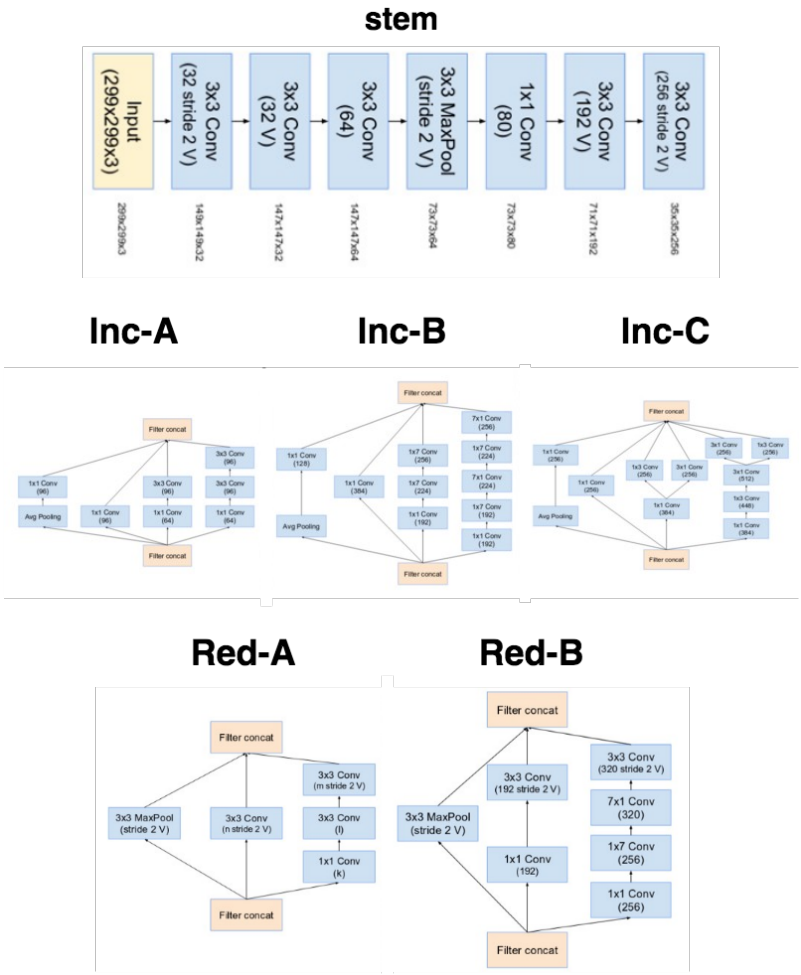


Figura 4.18: (1) Bloco *Stem* da rede InceptionNet v4. (2) IR-A, IR-B e IR-C: Três tipos de módulo Inception da rede InceptionNet v4. (3) Red-A e Red-B - Bloco de redução de 35x35 para 17x17, e 17x17 para 8x8, respectivamente.

Inspirados na performance da rede ResNet [He et al. 2016] (vencedora do desafio ImageNet 2015), os autores do InceptionNet criaram duas redes hibridas chamadas Inception-Resnet v1 e v2 [Szegedy et al. 2017]. A rede Inception-Resnet v1 tem custo computacional semelhante a rede Inception v3, enquanto a rede Inception-Resnet v2 tem custo computacional semelhante a rede Inception v4. A Figura 4.19 ilustra a arquitetura

das redes, ambas possuem a mesma estrutura para os módulos Inception-Resnet A, B e C e blocos de redução. As diferenças são os seus blocos *Stem* e hiper-parâmetros. O Inception-Resnet v1 usa o mesmo *Stem* da versão Inception v4, enquanto o Inception-Resnet v2 propõe o novo *Stem* que pode ser visto na Figura 4.20.

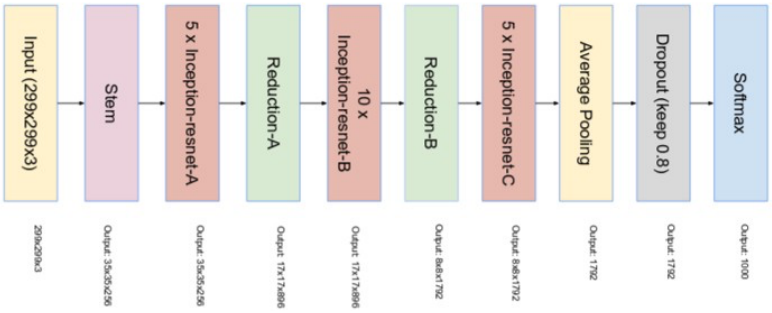


Figura 4.19: Arquitetura das redes Inception-ResNet v1 e v2.

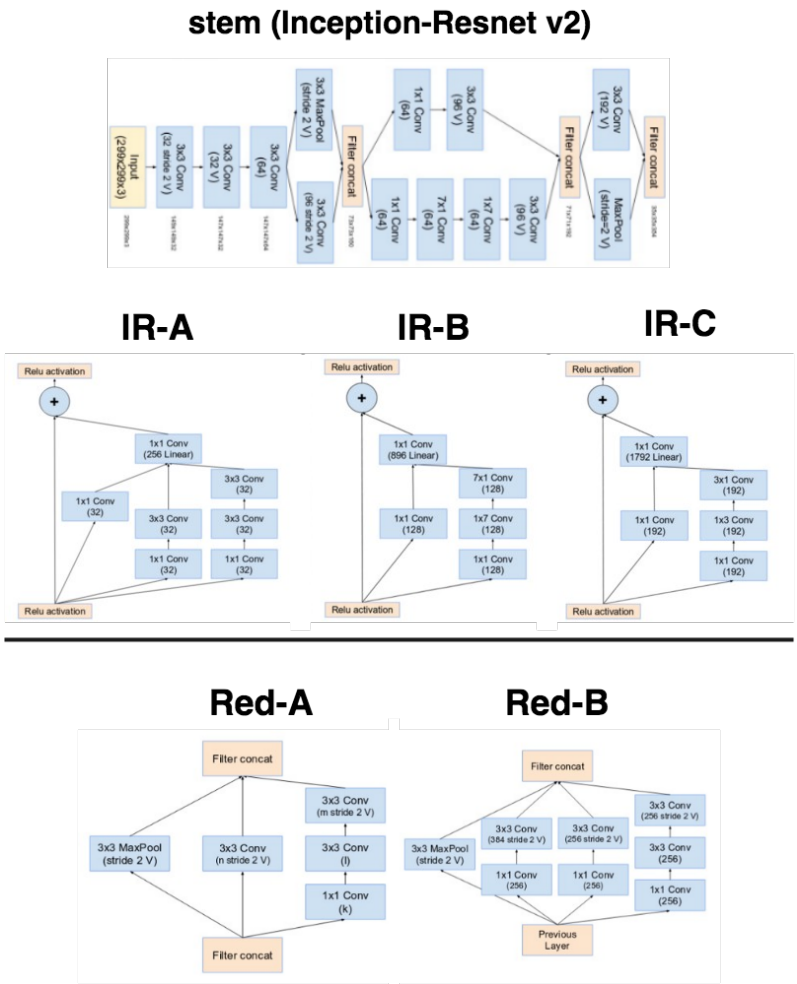


Figura 4.20: (1) Bloco stem da rede Inception-Resnet v2. (2) IR-A, IR-B e iR-C: Três tipos de módulo Inception-Resnet da arquitetura Inception-Resnet v1 e v2. (3) Red-A e Red-B - Bloco de redução de 35x35 para 17x17, e 17x17 para 8x8, respectivamente.

A principal ideia da arquitetura Inception-ResNet é a adição das conexões residuais propostas pela rede ResNet. A Figura 4.20 detalha os módulos da arquitetura Inception-Resnet. Para que a incorporação da conexão residual funcione é necessário que a entrada e a saída sejam concatenadas, e portanto que tenham a mesma dimensão. Para isso, foi adicionada uma convolução 1x1 após as convoluções tradicionais do módulo Inception para padronizar os tamanhos do mapas de *features*. A operação de *pooling* do Inception foi removido em favor da conexão residual. No entanto, essa operação ainda é presente nos blocos de redução A e B. Os autores constataram que a rede tende a instabilidade quando a rede excede mil filtros, para estabilizar a rede eles escalaram as ativações residuais por valores entre 0.1 e 0.3.

4.6. Classificação de vídeo

Nesta seção são apresentadas as técnicas para classificação de vídeo. Como ilustrado na Figura 4.21, a classificação de vídeo consiste de um processo bimodal. Primeiro as CNNs, chamadas de *backbones*, são usadas para extrair as *features* audio-visuais dos frames e áudio do vídeo.

Para extração das *features* visuais é possível utilizar uma das CNNs anteriormente citadas (e.g. Inception, ResNet) pré-treinadas no *dataset* ImageNet [Deng et al. 2009]. Já para extração de *features* de áudio, é possível utilizar uma CNN adaptada para o domínio de áudio, como AudioVGG [Hershey et al. 2017] ou WaveNet [Oord et al. 2016] pré-treinadas no *dataset* AudioSet [Gemmeke et al. 2017]. Após a extração, métodos para agregação de *features* como NetVLAD [Arandjelovic et al. 2016] e LSTM [Hochreiter and Schmidhuber 1997] são utilizados para minar as *features* audio-visuais e realizar a classificação.

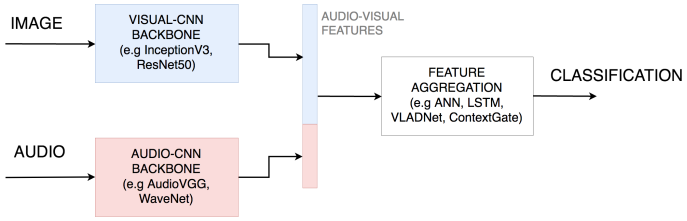


Figura 4.21: Arquitetura da ferramenta de classificação de vídeo.

Para apresentar o processo de classificação, organizamos esta seção como segue. Primeiro discutimos a extração de *features* visuais na subsection 4.6.1. Em seguida, apresentamos métodos de classificação na subsection 4.6.2.

4.6.1. Extração de features

As *features* são resultado da redução de dimensionalidade de camadas de uma rede, ou seja, suas camadas internas. Ao avançar nas camadas, a informação inserida no *input* é filtrada (ou "entrada", refere-se a primeira camada da rede), permanecendo informações que sejam úteis para a redução do erro em sua saída. Nas camadas seguintes da entrada, as informações presentes normalmente são mais simples e voltadas puramente para formas presentes na imagem, como vértices ou curvas. Em camadas mais internas a associação de tais informações podem indicar características como partes de um rosto, um carro ou

cenário; ou até mesmo informações mais complexas como o tipo de veículo presente na imagem ou se o animal presente na imagem é um gato ou não. Lembrando que nem sempre os valores das *features* representam algo discernível contextualmente, muitas vezes são relações entre inputs as quais não temos uma denominação.

As *features* concentram diversas características de um dado (no caso, imagem ou vídeo), as quais, vistas em conjunto, podem simbolizar um contexto maior ou contribuir com uma determinada inferência. Por exemplo, *features* que indicam alta iluminação, presença de água e céu, têm mais chances de estar representando uma praia do que uma apresentação musical.

Um exemplo de uso seria utilizar CNNs pré-treinadas para algum objetivo, como, classificar um animal presente na imagem. Essas redes concentram em suas camadas internas informações sobre sua entrada, logo, pode-se ignorar as camadas finais dessa rede e utiliza-las simplesmente como extratores de *features*. Estas *features* seriam utilizadas em outra rede com objetivo diferente (porém semelhante) da rede que as gerou. Tal técnica é chamada de *Transfer Learning*, quando o aprendizado é reaproveitado em um outro contexto.

4.6.2. Classificação

A normalização *softmax* transforma valores *logits*, ou seja, valores que variam entre mais e menos infinito, em valores que simbolizam probabilidades, que variam de 0 a 1; tendo sua soma igual a 1. Tal transformação é útil pois dadas várias possíveis classes simbolizadas por cada índice, tem-se a probabilidade da ocorrência de cada uma delas. Como mostra a Listagem 4.12

Listagem 4.12: Exemplo de uma função softmax.

```
import tensorflow as tf
import numpy as np

# gerando um input aleatorio
inp = [np.random.rand()*10 for i in range(5)]

# definindo a funcao softmax
softmax = tf.exp(inp)/tf.reduce_sum(tf.exp(inp),0)

with tf.Session() as sess:
    # executando o grafo
    out = sess.run(softmax)
    # imprimindo seus valores e somas
    print("\ninput: \n",inp)
    print("\nsoma input: \n",sum(inp))
    print("\nsoftmax: \n",out)
    print("\nsoma softmax: \n",sum(out))

------- OUTPUT -------
> input:
> [1.6542092596620717, 5.85058565829661, 1.807328616153372,
   9.651413386383854, 0.522533621278154]
>
> soma input:
```

```
> 19.486070541774062
>
> softmax:
> [3.28777882e-04 2.18456835e-02 3.83178820e-04 9.77336347e-01
   1.06028376e-04]
>
> soma softmax:
> 1.000000015636033
```

No código da Listagem 4.12 está a mesma rede feita na sessão 4.4.4 porém com um *dataset* diferente, sendo este, *features* extraídas de vídeos presentes em dois diretórios. Sendo cada diretório representante de uma classe, neste caso, vídeos próprios e vídeos impróprios.

Para a obtenção deste dado é utilizado um extrator de *features* baseado na *ResNet50* para extrair os *features* visuais dos vídeos.

```
from feature_extractor_image import extract_image_features
import tensorflow as tf
import numpy as np
import glob


PROPER_PATH = "./proper/*"
IMPROPER_PATH = "./improper/*"


def build_net(n_features, n_classes):
    '''
    Funcao criada na secao 4.4.4
    '''
    X_placeholder = tf.placeholder(dtype=tf.float32, shape=[None,
        n_features])
    Y_placeholder = tf.placeholder(dtype=tf.int64, shape=[None])
    layer1 = tf.layers.dense(X_placeholder, 100, activation=tf.nn.relu)
    out = tf.layers.dense(layer1, n_classes, name="output")
    one_hot = tf.one_hot(Y_placeholder, depth=n_classes)
    loss = tf.losses.softmax_cross_entropy(onehot_labels=one_hot,logits
        =out)
    opt = tf.train.GradientDescentOptimizer(learning_rate=0.07).
        minimize(loss)
    softmax = tf.nn.softmax(out)
    class_ = tf.argmax(softmax,1)
    compare_prediction = tf.equal(class_, Y_placeholder)
    accuracy = tf.reduce_mean(tf.cast(compare_prediction, tf.float32))
    return X_placeholder, Y_placeholder, loss, opt, class_, accuracy

def extract_features(filepath):
    return np.array(extract_image_features(filepath))


if __name__ == "__main__":

    # inicializando tags e datasets
    proper_tag = 1
```

```
36      improper_tag = 0
37      dataset_X, dataset_Y = [], []
38
39      # obtendo lista de arquivos por classe
40      proprios = glob.glob(PROPER_PATH)
41      improprios = glob.glob(IMPROPER_PATH)
42      print("proprios:",len(proprios),"improprios:",len(improprios))
43
44      # preenchendo dataset proprio
45      for ffile in proprios:
46          print("extraindo features de",ffile)
47          dataset_X.append(extract_features(ffile))
48          dataset_Y.append(proper_tag)
49
50      # preenchendo dataset improprio
51      for ffile in improprios:
52          print("extraindo features de",ffile)
53          dataset_X.append(extract_features(ffile))
54          dataset_Y.append(improper_tag)
55
56      #
57      dataset_X = np.array(dataset_X)
58      dataset_Y = np.array(dataset_Y)
59
60      # obtendo shape do input e output
61      n_features = dataset_X.shape[1]
62      num_classes = dataset_Y.shape[0]
63      print("n features:",n_features,"num classes:",num_classes)
64
65      # executando o modelo como feito na ãsesso 4.4.4
66      sess = tf.Session()
67
68      X_placeholder, Y_placeholder, loss, opt, class_, accuracy =
            build_net(n_features, num_classes)
69      sess.run(tf.global_variables_initializer())
70
71      epochs = 1000
72      for i in range(epochs):
73          sess.run(opt, feed_dict={X_placeholder: dataset_X,
                Y_placeholder: dataset_Y})
74          if i%30 == 0:
75              erro_train = sess.run(loss, feed_dict={X_placeholder:
                    dataset_X, Y_placeholder: dataset_Y})
76              print("erro na epoca",i,":",erro_train)
77
78      acc = sess.run(accuracy, feed_dict={X_placeholder:dataset_X,
            Y_placeholder:dataset_Y})
79      print("accuracia do modelo:",acc)
```

Neste exemplo iniciamos declarando a mesma função **build_net()** feita na seção 4.4.4, seguida da função **extract_features()** que encapsula o uso do extrator de *feature* citado anteriormente transformando sua saída em um vetor **numpy**.

A função **glob()** requisita o sistema em busca de arquivos que batem com o padrão fornecido, de acordo com as regras do sistema *Unix*, como *"./proper/*"* que corresponde

a todos os arquivos presentes no diretório chamado "proper".

Com as listas de arquivos obtidas com a função **glob()** extraimos as features visuais de cada arquivo, atribuindo as tags 0 e 1 dependendo de que diretório o arquivo foi lido. A partir da linha 65 todo o processo é exatamente igual ao que foi feito na seção 4.4.4.

4.7. Detecção de Objetos

Nesta seção descrevemos o modelo YOLO (You Only Look Once) [Redmon et al. 2016], considerado o estado-da-arte na tarefa de detecção de objetos. Sua última versão, chamada YOLOv3 [Redmon and Farhadi 2018] obteve um mAP de 57.9% no dataset COCO [Lin et al. 2014]. O YOLO é ideal para aplicações de tempo-real, visto que é o modelo de detecção de objetos baseado em CNN mais rápido da literatura, chegando a rodar próximo de 30 FPS na GPU Pascal Titan X[10].

A Subseção 4.7.1 descreve a arquitetura geral do YOLO. Em seguida, a Subseção 4.7.2 descreve a implementação de um cenário de uso que usa o YOLO para identificar objetos em imagens.

4.7.1. Arquitetura YOLO

O YOLO divide a imagem de entrada em uma grade de $S \times S$ dimensões. Cada célula pode conter *B bounding boxes* (BBs) e *scores* de confiança para cada uma. O score de confiança reflete o quão a rede tem certeza que a BB contém um objeto. Se não existe objetos na célula, então o score de confiança deve ser zero. Caso contrário, o score de confiança deve ser condicionada pela Interseção sobre União (explicada na Subseção seguinte) entre a BB predita e a BB do *ground truth*, $Pr(Object) * IOU_{pred}^{truth}$.

No YOLO, cada BB contém as seguintes informações: 1) score de confiança da célula conter um objeto; 2) coordenadas da BB (b_x, b_y, b_h, b_w), onde (b_x, b_y) representa o ponto central da BB relativa a uma célula da grade, enquanto (b_h, b_w) representa a altura e largura da BB relativa às dimensões da imagem de entrada; 3) Um vetor de probabilidades $(c_1, c_2, ..., c_n)$ para cada uma das n classes de objetos, onde cada probabilidade de classe é condicionada pela probabilidade da célula conter um objeto, $Pr(Class_i|Object)$.

O *score* da confiança de cada classe em cada BB é dada por:

$$Pr(Class_i|Object) * Pr(Object) * IOU_{pred}^{truth} = Pr(Class_i) * IOU_{pred}^{truth}$$

Os *scores* informam: (1) a probabilidade de um objeto de uma classe aparecer na BB, e (2), o quão ajustado está a BB ao objeto. Como ilustra a Figura 4.22, a saída do YOLO é um tensor de dimensões $S \times S \times (B * 5 + C)$. Onde S é a dimensão do *grid* de regiões, B é o número de *bounding boxes* em cada célula, e C é a quantidade de classes no problema. A Figura 4.23 (cima) mostra três visualizações da saída do YOLO. A imagem da esquerda mostra a entrada dividida em uma grade $S \times S$ regiões. A imagem do meio mostra o mapa de probabilidade de classes para cada célula da grade. Por último, a imagem da direita mostra as BBs.

[10] https://www.nvidia.com/pt-br/geforce/products/10series/titan-x-pascal

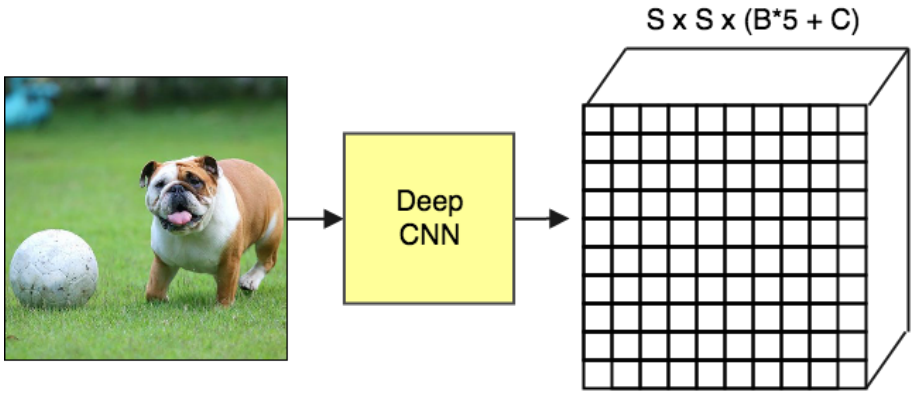


Figura 4.22: Esquema arquitetural do YOLO.

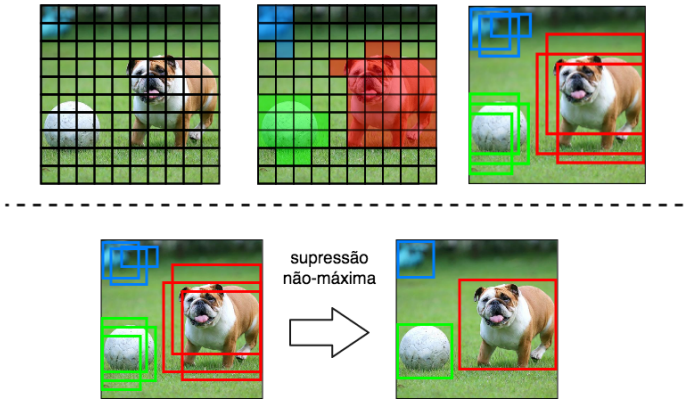


Figura 4.23: Visualização da saída do YOLO (parte superior) e remoção das BBs sobrepostas com o filtro de supressão não-máxima (parte inferior).

4.7.1.1. Supressão não-máxima

Mesmo com a filtragem pelo *score*, muitas BBs podem ficar sobrepostas uma as outras, como ilustrado na Figura 4.23 (baixo). Um segundo tipo de filtro chamado "supressão não-máxima" é necessário para remover as BBs sobrepostas. A supressão não-máxima utiliza uma técnica chamada "Interseção sobre União" (em inglês, *IoU - Intersection over Union*). Como pode ser visto na Figura 4.24, essa técnica consiste basicamente em dividir a interseção pela união de duas BBs.

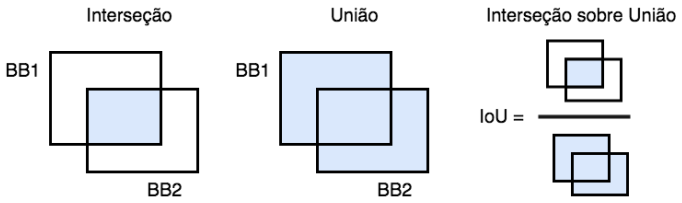


Figura 4.24: Visualização da operação de IoU.

A Listagem 4.13 mostra como implementar o IoU, nas linhas 3-7 é calculada a área de interseção entre as BBs. Em seguida, nas linhas 10-12 é calculada a área de união

entre as BBs. Por fim, na linha 16 é calculado o Iou pela divisão da área de interseção pela área de união.

Listagem 4.13: Função de cálculo do IoU.

```
1  def iou(box1, box2):
2      #Calculando a intersecao entre as BBs
3      xi1 = max(box1[0], box2[0])
4      yi1 = max(box1[1], box2[1])
5      xi2 = min(box1[2], box2[2])
6      yi2 = min(box1[3], box2[3])
7      inter_area = (xi2 - xi1)*(yi2 - yi1)
8
9      #Calculando a uniao usando a formula: Union(A,B) = A + B - Inter(A,
           B)
10     box1_area = (box1[2] - box1[0])*(box1[3] - box1[1])
11     box2_area = (box2[2] - box2[0])*(box2[3] - box2[1])
12     union_area = box1_area + box2_area - inter_area
13
14     # Calculando o IoU
15     iou = inter_area / union_area
16
17     return iou
```

4.7.1.2. Função de perda

Durante o treinamento o YOLO otimiza uma função composta por 5 partes. Cada parte é uma equação que realiza uma tarefa específica.

$$\mathscr{J} = eq1 + eq2 + eq3 + eq4$$

Para aumentar a sua estabilidade, o YOLO aumenta a perda da predições da coordenada das BBs e diminui a perda das BBs que não contém objetos. Para isso, dois parâmetros λ_{coord} e λ_{noobj} são definidos com valores 5 e 0.5, respectivamente.

A equação descrita abaixo calcula a perda relativa a posição **(x,y)** da BB. O 1_{ij}^{obj} denota se a j-ésima BB na i-ésima célula é responsável pela predição do objeto. o YOLO predita múltiplas BB por célula, durante o treinamento somente uma BB é responsável por cada objeto. Então uma BB recebe a responsabilidade de predizer baseado no maior IoU com a BB do *ground truth*.

$$eq1 = \lambda_{coord} \sum_{i=o}^{S^2} \sum_{j=o}^{B} 1_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2]$$

A equação descrita abaixo calcula a perda relativa a largura e altura **(w,h)**. A equação é similar a primeira, com a diferença do uso das raízes quadradas para fazer com que pequenas variações em BBs largas importem menos que em BBs pequenas.

$$eq2 = \lambda_{coord} \sum_{i=o}^{S^2} \sum_{j=o}^{B} 1_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2]$$

A equação abaixo calcula a perda associada ao *score* de confiança para cada BB, onde C é o score de confiança e $\hat{C}$ é o IoU entre a BB predita e a BB do *ground truth*. O termo 1_{ij}^{noobj} denota se a j-ésima BB na i-ésima célula não é responsável pelo objeto.

$$eq3 = \sum_{i=o}^{S^2} \sum_{j=o}^{B} 1_{ij}^{obj} (C_i - \hat{C}_i)^2 + \lambda_{noobj} \sum_{i=o}^{S^2} \sum_{j=o}^{B} 1_{ij}^{noobj} (C_i - \hat{C}_i)^2$$

A última equação calcula a perda da classificação. Ela é similar a equação de erro de soma quadrada tradicional, mas com a adição do termo 1_i^{obj}. Este termo denota se o objeto aparece na i-ésima célula, é usado para que o erro de classificação não seja penalizado quando o não houver um objeto em uma célula.

$$eq4 = \sum_{i=o}^{S^2} 1_i^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2$$

4.7.2. Cenário de Uso: Sistema de Detecção de Objetos

A forma mais fácil para usar o modelo YOLO é importando o *framework* Darkflow (Tensorflow + Darknet[11]). O *framework* Darknet é escrito em C e CUDA[12] e foi usado para a implementação oficial do modelo YOLO. O Darkflow é uma re-implementação em Python do Darknet usando o Tensorflow como base.

Como descreve os comandos abaixo, para instalar o Darkflow basta realizar o download do repositório no Github[13]. E em seguida, realizar a instalação do Darknet usando o Pip. Vale ressaltar que é necessário ter o pacote Cython[14] instalado.

```
1 git clone https://github.com/thtrieu/darkflow.git
2 cd darkflow
3 pip3 install .
```

Após realizar a instalação, para utilizar o pacote basta importa-lo para o projeto, como mostra a Listagem 4.14. Na linha 6, é criado um dicionário chamado *options*, que define os atributos necessários para executar o YOLO pré-treinado no *dataset* COCO. O atributo "model" especifica o caminho do arquivo "yolo.cfg", que define a arquitetura da CNN usada. O atributo "load" define o caminho para o arquivo "yolo.weights", que são os pesos da rede pré-treinada no *dataset* COCO. Esse arquivo pode ser baixado no Drive[15] do autor da rede. O atributo "threshold" define o percentual mínimo para confidencia de detecção de objetos, nesse caso só objetos com pelo menos 10% de *score* de confidencia

[11] https://pjreddie.com/darknet
[12] https://developer.nvidia.com/cuda-zone
[13] https://github.com/thtrieu/darkflow
[14] http://cython.org
[15] https://drive.google.com/drive/folders/0B1tW_VtY7onidEwyQ2FtQVplWEU

são retornados. O atributo "gpu" define se o programa pode fazer uso da GPU do sistema. Em seguida, na linha 10, é instanciada uma rede que recebe as opções definidas. É importante ressaltar que também é necessário ter o arquivo "coco.names" na pasta "cfg", esse arquivo é encontrado no repositório do Darknet e contém os nomes das classes do *dataset* COCO.

Listagem 4.14: Configurando a aplicação com os dados pré-treinados.

```
1  from darkflow.net.build import TFNet
2  import matplotlib.pyplot as plt
3  import cv2
4  from draw_boxes import *
5
6  options = {"model": "cfg/yolo.cfg",
7             "load": "cfg/yolo.weights",
8             "threshold": 0.1,
9             "gpu": 1.0}
10 tfnet = TFNet(options)
```

Agora, desejamos criar um *boundingbox* em objetos identificadas, como ilustrado na Figura 4.25.

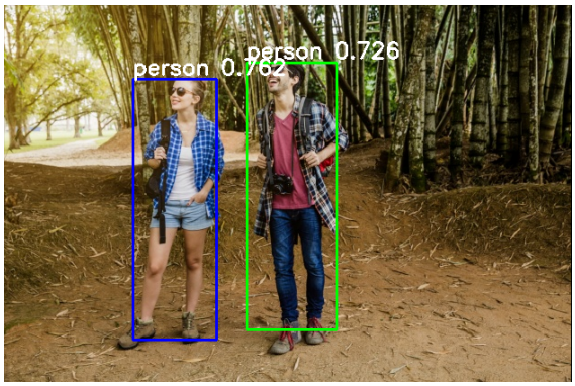


Figura 4.25: Imagem com as *bounding boxes* preditas pelo YOLO.

A Listagem 4.15 mostra como utilizar o modelo YOLO para detectar objetos. Nas linhas 1 e 2 um imagem é aberta e convertida para RGB. Em seguida, na linha 3, a imagem é usada como entrada da rede. Na linha 4 o resultado é impresso na tela. Por fim, nas linhas 6 e 7 a imagem de entrada é exibida com as BBs identificadas com pelo menos 30% de confidência (Figura 4.25). As linhas 9-10 mostram a saída do programa. Nota-se que o YOLO encontrou seis BBs, das quais apenas duas possuem score de confidência suficiente para ser desenhada.

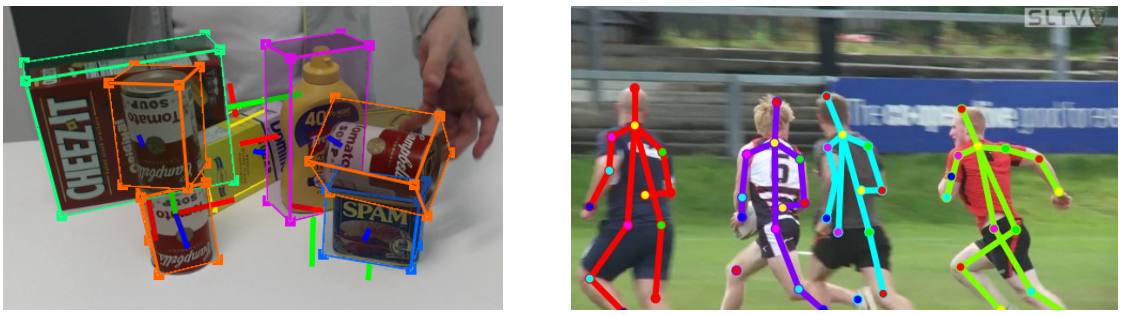
Listagem 4.15: Usando o YOLO para detectar objetos.

```
1  img = cv2.imread("images/sample1.png")
2  img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
3  result = tfnet.return_predict(img)
4  print(result)
5
6  plt.imshow(boxing(img, result, 0.3))
7  plt.show()
```

```
 ---------------------------- OUTPUT -------------------------------
[{'label': 'person', 'confidence': 0.15689932, 'topleft': {'x': 314, 'y': 82},
'bottomright': {'x': 379, 'y': 253}},
{'label': 'person', 'confidence': 0.7260812, 'topleft': {'x': 268, 'y': 64},
'bottomright': {'x': 367, 'y': 358}},
{'label': 'person', 'confidence': 0.7622834, 'topleft': {'x': 143, 'y': 82},
'bottomright': {'x': 235, 'y': 369}},
{'label': 'handbag', 'confidence': 0.2231428, 'topleft': {'x': 198, 'y': 178},
'bottomright': {'x': 233, 'y': 239}},
{'label': 'skis', 'confidence': 0.12105702, 'topleft': {'x': 313, 'y': 103},
'bottomright': {'x': 375, 'y': 271}},
{'label': 'snowboard', 'confidence': 0.2203493, 'topleft': {'x': 342, 'y': 159},
'bottomright': {'x': 371, 'y': 257}}]
```

4.8. Estimação de pose

O principal objetivo da estimação de pose é, a partir de dados obtidos por meio de um sensor óptico, seja ele um sensor de profundidade ou uma câmera, é determinar a posição espacial de um objeto ou ser. Naturalmente, estimar a posição de uma pessoa apresenta um desafio maior, pois além de localizar uma pessoa, também é interessante capturar a disposição de seus membros ou articulações. Existem muitas aplicações para a estimação de pose, entre elas, animação, jogos, monitoramento e interação natural. Nesta seção, trataremos especificamente da estimação de pose de pessoas. Na Figura 4.26 pode-se observar exemplos de estimação de pose para objetos e pessoas.



(a) [Tremblay et al. 2018]

(b) [Iqbal et al. 2017]

Figura 4.26: Estimação de pose para objetos (a) e pessoas (b). Na imagem (b) os pontos de interesse (keypoints) são apresentados como círculos coloridos.

Existem muitos pontos a ser levar em consideração na criação de modelos de deep learning para estimação de pose, por exemplo, a quantidade de frames ou imagens usadas para fazer uma predição, O tipo de sensor utilizado na captura dos dados, o numero de dimensões que serão levados em conta para a estimação, etc.

Quanto à quantidade de imagens levadas em conta para obtenção do resultado fi-

nal, temos duas opções: *Singleframe* ou *Multiframe*, na qual a primeira consiste no uso de um único frame ou imagem para realização da estimação. Já na segunda opção, a técnica utilizada leva em conta um ou mais frames, buscando-se manter a trajetória de indivíduos já detectados. O uso de múltiplos frames não implica que todos eles serão necessariamente usados diretamente na predição, mas que podem ser usados em conjunto com outras técnicas para extrair informação temporal. A desvantagem do uso de múltiplos frames é o custo computacional para a estimação, que geralmente implica é maiores tempos para predição e/ou a necessidade de hardware mais robusto.

Quanto aos tipos de dados a serem utilizados os mais utilizados são imagens RBG convencionais, existem também outros trabalhos que usam dois ou mais sensores, imagens de profundidade. Sendo imagens de profundidade os dados usados pelo Kinect, um dispositivo popular que usa estimação de pose, baseado em Randomized Decision Forests [Shotton et al. 2011] anterior à popularização do deep learning. Na estimação de pose pode-se partir de dados 2D (imagens RBG monoculares ou binoculares) ou de dados em 3D (imagens RGBD ou de sensores de profundidade) e estimar coordenadas em 2D ou 3D.

Um algoritmo para estimação de pose pode ser *single-person* ou *multi-person*, apenas uma pessoa é detectada na abordagem *single-person*, a presença de mais de uma pessoa na imagem geralmente causa erros na predição. Já a abordagem *multi-person* detecta todas as pessoas em uma imagem, agrupando seus *keypoints* (pontos de interesse representativos do corpo humano) por pessoa. Apesar de ser mais lenta, a abordagem *multi-person* é mais robusta e mais confiável que a abordagem *single-person*.

Geralmente, a entrada de uma rede convolucional para estimação de pose é uma imagem, ou dados de profundidade representados como uma imagem, de tamanho fixo. Uma rede convolucional treinada tem o número de parâmetros fixo, portanto imagens maiores ou menores que o tamanho padrão para que a rede foi treinada tem que ser escaladas. Já a saída dessas redes é um conjunto de mapas de calor (*heatmaps*) para cada *keypoint*, no qual cada heatmap contém a predição da probabilidade da presença desse *keypoint* para cada pixel. A partir desses mapas de calor é possível usar diferentes ténicas para obter as coordenadas dos *keypoints* preditos, sendo simplesmente medir a coordenada de máxima ativação (ponto de maior valor) uma das técnicas mais usadas. Exemplos de *heatmaps* para alguns *keypoints* podem ser observados na Figura 4.27.

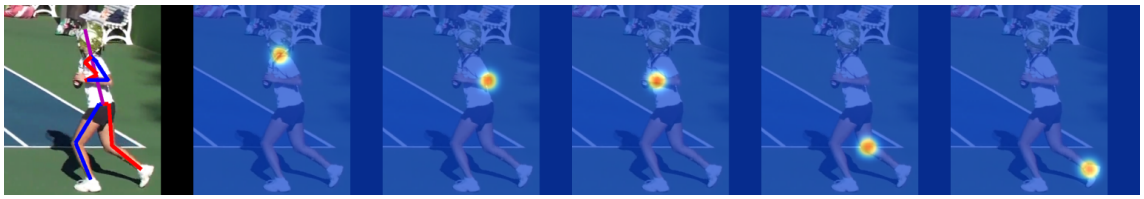


Figura 4.27: Exemplo de *heatmaps* preditos para diferentes *keypoints*. Fonte: [Newell et al. 2016]

Existem tambem duas grandes abordagens quanto ao agrupamento de *keypoints* em uma imagem. Na abordagem *top-down* primeiro se utiliza um algoritmo de detecção de pessoas na imagem original, obtendo-se assim imagens menores, recortadas, de cada pessoa detectada na imagem original. Para cada imagem gerada pelo detector de pessoas,

é executado algoritmo de estimação de pose e ao fim, se obtém todos os *keypoints* de cada pessoa mapeados na imagem original, agrupados por pessoa detectada, assim como pode ser visto pelo exemplo na Figura 4.28. A abordagem *bottom-up* por outro lado utiliza somente a imagem original. Se detecta todos os *keypoints* de todas as pessoas, sem identificação sobre qual *keypoint* pertence a qual pessoa e então se utiliza uma técnica de agrupamento para os *keypoints*, técnica essa que pode ou não ser baseada em machine learning. Por exemplo, [Cao et al. 2018] e [Insafutdinov et al. 2016] tratam o problema de atribuição de *keypoints* à pessoas distintas como um grafo.

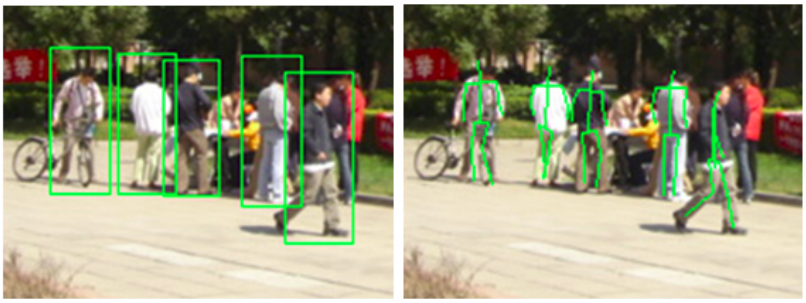


Figura 4.28: Exemplo de abordagem top-down para estimação de pose. Adaptado de [Wang et al. 2010]

Figura 4.29: Exemplo de abordagem bottom-up para estimação de pose. Fonte: [Insafutdinov et al. 2016]

4.8.1. Métricas de avaliação populares

Dentre as métricas de avaliação apresentaremos a Mean Average Precision (mAP) para estimação de pose *single-frame*, Multiple Object Tracking Accuracy (MOTA) para avaliação de estimação de pose para video e as métricas de similaridade/acerto Percentage of Correct Keypoint relative to head (PCKh) e Object Keypoint Similarity (OKS).

A métrica Mean Average Precision (mAP) consiste na media da pontuação Average Precision (AP) [Everingham et al. 2010] para todos os keypoints. Para se obter a pontuacao AP, primeiro se computa a curva precision/recall para todas as predições ordenadas. Recall é definido pela proporção de todos os Verdadeiros Positivos (TP) sobre todos os Positivos (TP + FP). Precision ou Precisão é a proporção Verdadeiros Positivos (TP) e os Verdadeiros Positivos e Falsos Negativos (TP + FN). A métrica AP resume a curva precision/recall é definida pela precisão média em 11 níveis de recall *r* igualmente espaçados [0.0,0.1,0.2,...,1.0]. Sendo AP definido por:

$$AP = \frac{1}{11} \sum_{r \in \{0, 0.1, \ldots, 1\}} P_{interp}(r)$$

Onde a precisão em cada nível de recall é interpolada pela precisão máxima correspondente a qualquer r maior que $\bar{r}$:

$$P_{interp}(r) = max_{\bar{r}:\bar{r}>r} p(\bar{r})$$

A métrica MOTA (Multiple Object Tracking Accuracy) é talvez a métrica mais utilizada para tracking [Stiefelhagen et al. 2006], pois combina três fatores de erro:

$$MOTA = 1 - \frac{\sum_t (FN_t + FP_t + IDSW_t)}{\sum_t GT_t}$$

- FN: Falsos Negativos, são os pontos *ground-truth* em que não houve *tracking*, ou seja, em que os pontos preditos estiveram fora dos limites de distancia do ponto do *ground-truth*;
- FP: Falsos Positivos, são os pontos preditos fora dos limites de distancia do ponto *ground-truth*;
- $IDSW$: Mudanças de id, ocorrem quando há fragmentação na predição de uma trajetória. Um algoritmo de *tracking* gera um novo id sempre que encontra um objeto não rastreado em um ou mais dos últimos frames. Quando o algoritmo perde *tracking* de um objeto e o reencontra (fragmentação), e trata-o como se fosse um novo objeto, é contabilizada uma mudança de id.

Onde t é o índice do frame e GT o número de pontos *ground-truth* (pontos da trajetória real). Na Figura 4.30 observa-se exemplos de fragmentação e mudanças de id, sendo a linha tracejada o trajeto *ground-truth* (GT). As linhas vermelhas e azuis são o trajeto interpolado a partir das predições, cada cor representa o tracking de um id diferente. Os pontos preenchidos de cor azul e vermelho são as predições corretas (True Positive). Os pontos vermelhos e azuis não-preenchidos são predições False Positive (FP). Os pontos cinza são pontos False Negative (FN). Por fim, os pontos preenchidos de preto mas realçados por vermelho ou azul representam os pontos verdadeiros que foram considerados rastreados mesmo que a predição não seja exata. O sombreado cinza em torno da linha tracejada representa os limites de distancia do trajeto *ground-truth* para se considerar um ponto como rastreado ou não.

Como proposto por [Andriluka et al. 2014] a métrica PCKh (Percentage of Correct Keypoint relative to head) é uma derivação da métrica PCK (Probability of Correct Keypoint) criada por [Yang and Ramanan 2012]. A métrica PCK avalia a acurácia da localização da coordenada predita em relação à coordenada verdadeira (*ground-truth*) como um limite de distancia do ponto predito ate o ponto *ground-truth* definido por $\alpha = max(h, w)$

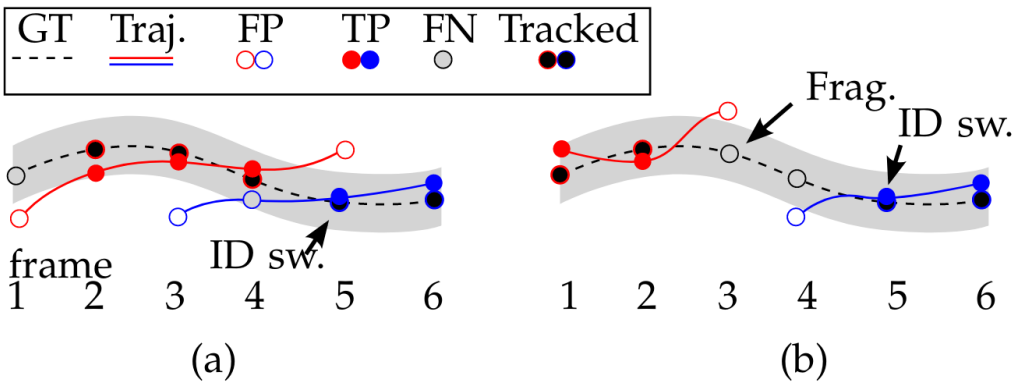


Figura 4.30: Exemplo de mudanças de id e fragmentação segundo a métrica MOTA. Imagem adaptada de [Milan et al. 2016].

onde α é uma constante percentual que controla o limite relativo para considerar um acerto e h e w são a altura e largura, respectivamente, da *bounding box* da pessoa. A métrica PCKh é definida como 50% do tamanho do segmento da cabeça, sendo o segmento da cabeça denotado pela distancia do keypoint que representa a cabeça até o keypoint que representa a clavícula/base do pescoço.

OKS ou Object Keypoint Similarity[Ruggero Ronchi and Perona 2017] é uma métrica de similaridade entre dois keypoints, é a métrica de similaridade utilizada para avaliar modelos baseado no dataset COCO (Common Objects in Context) [Lin et al. 2014] o qual abriga uma grande quantidade de imagens anotadas com keypoints de pessoas.

Simplificando, a OKS funciona como o IoU funciona na detecção de objetos. É calculada pela distância entre os pontos preditos e os pontos *ground-truth* com valores multiplicados por uma distribuição normal em torno dos pontos *ground-truth*. Essa distribuição de valores de OKS de acordo com a distância do ponto predito até o ponto real pode ser observado na Figura 4.31.

Para o cálculo do OKS são necessários os vetores preditos e *ground-truth* contendo as coordenadas de todos os keypoints, por exemplo temos um vetor de predição $x = [x_1, y_1, v_1, ..., x_k, y_k, v_k]$ onde x e y (podem ser mais dependendo da dimensão) são as coordenadas do keypoint k e v é uma flag de visibilidade do keypoint(também pode ser predita, sendo o equivalente à confiança da predição) com:

- $v = 0$: não detectado;
- $v = 1$: detectado mas não visível;
- $v = 2$: detectado e visível.

A equação que define a OKS é:

$$OKS = \frac{\sum_i \exp\left(-d_i^2/2s^2k_i^2\right)\delta\left(v_i > 0\right)}{\sum_i \delta\left(v_i > 0\right)}$$

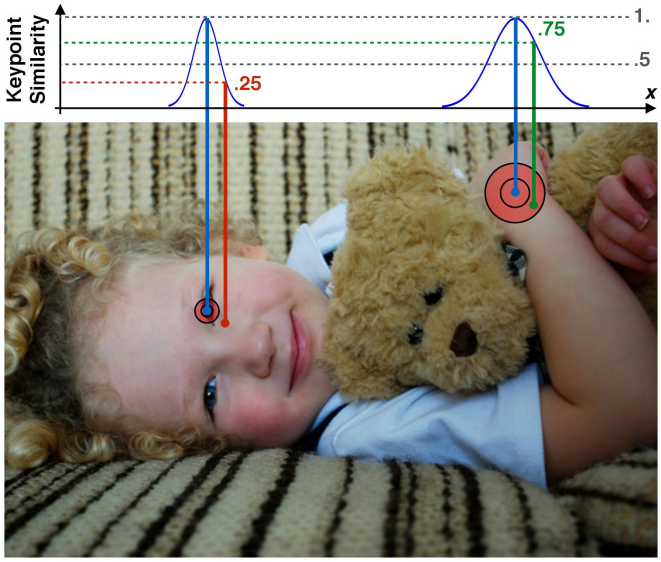


Figura 4.31: Exemplo da distribuição dos valores de OKS de acordo com a distancia entre os pontos preditos (vermelho e verde) e os pontos *ground-truth* (azuis). Fonte: [Ruggero Ronchi and Perona 2017].

Onde d_i são as distâncias Euclidianas entre cada *ground-truth* e keypoints preditos correspondentes e v_i são as flags de visibilidade do ponto *ground-truth*. As flags v_i preditas não são usadas. Para computar o OKS, passa-se d_i por uma gaussiana não normalizada com desvio padrão sk_i, onde s é a escala do objeto e k_i é uma constante por keypoint que controla a queda. Para cada keypoint é gerado um valor de similaridade entre 0 e 1. A média dessas similaridades é obtida sobre todos os keypoints cuja flag de visibilidade seja maior que zero ($v_i > 0$). Keypoints cuja flag de visibilidade seja 0 ($v_i = 0$) não influenciam na OKS. Predições perfeitas terão a $OKS = 1$ e predições nas quais todos os keypoints estão mais longe do que açguns desvios padrões sk_i terão $OKS \approx 0$. Dada a OKS, podemos calcular os valores AP assim como o IoU permite calcular essa métrica para detecção de objetos.

4.8.2. Stacked Hourglass Networks for Human Pose Estimation

Publicado em 2016, e estado-da-arte nesse ano, o artigo *Stacked Hourglass Networks for Human Pose Estimation*[Newell et al. 2016] se destaca por sua abordagem diferente em arquiteturas de redes convolucionais profundas. Através da sequenciação de vários módulos *hourglass* (Representado na Figura 4.32 (b)), os autores mostram que o principal diferencial de sua arquitetura está nas inferências sequenciadas no estilo "gargalo", e não meramente na profundidade, e portanto maior capacidade, da rede convolucional. O trabalho toma uma abordagem top-down, utilizando a rede *YOLO*, vista na Seção 4.7, para detectar e segmentar as imagens de cada pessoa em uma imagem e então executar a CNN *stacked hourglass*, para estimar os pontos de interesse de cada pessoa detectada.

O design do módulo *hourglass* foi motivado,segundo os autores, pela necessidade de capturar informação em cada escala. Enquanto evidencia local é essencial para identificar características como rostos e mãos, a predição final da pose requer um entendi-

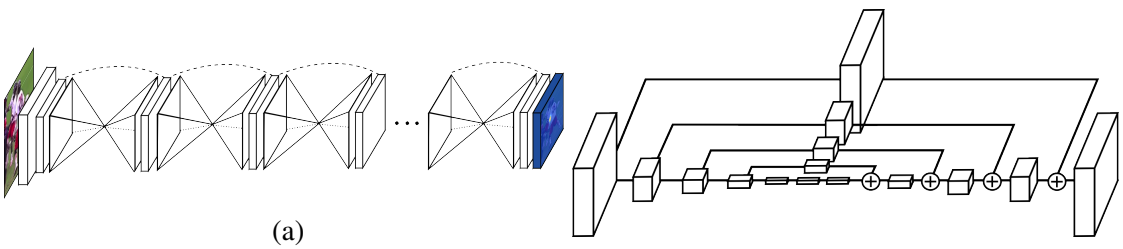
mento geral do corpo inteiro, por isso a necessidade da informação em todas as escalas. Com o objetivo de manter informação de características entre as escalas foram utilizadas *skip-layers* (Técnica de conexão entre escalas através de uma camada de convolução intermédiaria). A rede atinge a menor resolução em um tamanho de 4x4 pixels.

O módulo *hourglass* é definido da seguinte forma: Camadas convolucionais e *max pooling* são usadas para diminuir progressivamente a resolução da imagem de entrada até uma resolução muito baixa. A cada passo de *pooling* a rede ramifica e aplica mais convoluções à imagem antes do *pooling*. Após atingir a menor resolução, a rede começa uma sequencia de *upsamplings*(técnicas para aumento de resolução) e combinação de características entre escalas. Para unir informação entre duas resoluções adjacentes, os autores utilizaram o *upsampling* do vizinho mais próximo na menor resolução, como descrito em [Tompson et al. 2014], seguido de uma soma elemento-a-elemento dos dois tensores.

A topologia do modulo *hourglass* é simétrica para que cada camada de convolução e *max-pooling* tenha uma camada equivalente de *upsampling* para a combinação das características.

Ao fim do módulo *hourglass* duas convoluções 1x1 são aplicadas para se obter o conjunto de *heatmaps*. Na Figura 4.32 (b) é apresentada a organização do módulo *hourglass*. Já na Figura 4.32 (a), mostra uma visão geral de como funciona a arquitetura com os módulos hourglass empilhados. A entrada da CNN, uma imagem de dimensões 256x256, que é então reduzida para um tensor de tamanho 64x64, que é o tamanho do tensor de saída de cada módulo *hourglass*. Na saída da rede *stacked hourglass*, assim como na saída de cada módulo *hourglass*, temos um conjunto de *heatmaps*, de tamanho 64x64, um para cada *keypoint* a ser estimado. Segundo os autores, a explicação para a organização da arquitetura é a possibilidade de reconsiderar e/ou decidir quais *keypoints* ressaltar, em caso em que há, por exemplo, dois tornozelos de duas duas pessoas diferentes na mesma imagem.

A função de perda utilizada nessa arquitetura é a Mean-Squared Error (MSE), comparando o *heatmap* predito ao *heatmap ground-truth*, que consiste de uma gaussiana 2D (com desvio padrão de 1 pixel) centralizada na localização do *keypoint*.

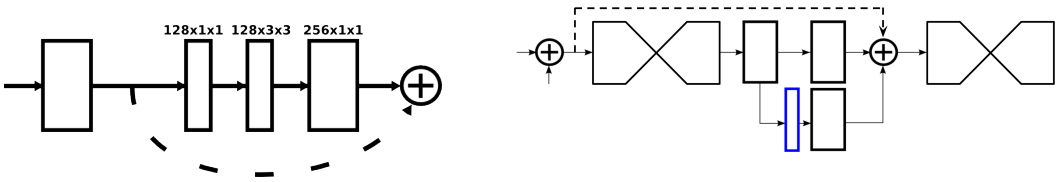


(a) (b)

Figura 4.32: (a) Organização da arquitetura *stacked hourglass*. (b) Estrutura de um módulo *hourglass*, Cada caixa corresponde a um módulo residual, como o visto na Figura 4.33 (a). Fonte: [Newell et al. 2016].

Os autores empregam também uma técnica chamada *supervisão intermediária*,

a qual consiste na geração de um grupo de *heatmaps* ao fim de cada módulo *houglass* (Destacado em azul na Figura 4.33 (b)), o que permite a aplicar uma função de *loss* entre cada conexão dos módulos *hourglass*. Esses *heatmaps* intermediários são convertidos de volta a *features* através de uma convolução 1x1 e então são somados aos *features* do módulo *hourglass* anterior e aos *features* de saída do *hourglass* atual. Uma representação gráfica pode ser consultada na Figura 4.33 (b), onde cada retângulo é um tensor, cada circulo com um simbolo de soma representa uma soma elemento-a-elemento entre tensores de mesmas dimensões, cada seta representa uma operação sobre os tensores anteriores, e por fim, cada seta tracejada representa uma conexão residual entre dois pontos. [He et al. 2016]



(a) (b)

Figura 4.33: Fonte: [Newell et al. 2016]

Na Figura 4.34 observa-se imagens geradas em diferentes etapas da predição no modelo *stacked hourglass*. Essas imagens podem ser usadas para representar os principais passos do processo de estimação do pose para qualquer abordagem top-down.

(a) (b) (c) (d)

Figura 4.34: Estágios da predição na arquitetura *stacked hourglass*. (a) Imagem original. (b) Detecção e segmentação de todas as pessoas na imagem. (c) Predição dos *heatmaps* (Aqui estão somados para aparecerem todos em uma só imagem). (d) Definição dos *keypoints* (na imagem também foram desenhados os segmentos, esses não são preditos, meramente ligam os *keypoints* preditos.) Fonte: Elaborado pelos autores.

No código[16] da Listagem 4.16 temos um exemplo para a geração das imagens da Figura 4.34, o código se baseia na classe *Inference*, que define diversas funções para a inferência com os pesos treinados da arquitetura *stacked hourglass*.

Listagem 4.16: Usando Stacked hourglass[Newell et al. 2016] para fazer estimação de pose em uma imagem.

[16]Disponível em https://github.com/pedropva/hourglasstensorlfow.git

```
import sys
sys.path.append('./') # Importar os outros arquivos na pasta
from inference import Inference
import cv2 # OpenCV, para trabalhar com imagens

# Caminho para o arquivo original
img_full_name = '0.jpg'
img_dir = './images/'
save_dir = './out/'
img_name, extension = img_full_name.split('.')
extension = '.' + extension
'''
 Instanciando um objeto da classe Inference
 Essa classe constroi o grafo de execucao dos modelos Stacked Hourglass e YOLO
 Tambem carrega os pesos treinados desses modelos.
'''
inf = Inference(config_file = 'config_tiny.cfg', model = 'hg_refined_tiny_200', yoloModel = 'YOLO_small.ckpt')

# Carregando uma imagem RGB
img = cv2.imread(img_dir+img_full_name)
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# Caso a imagem nao seja do tamanho 256x256, redimensionar.
if img.shape != (256,256,3):
    print('Wrong shape. Resizing.')
    img = cv2.resize(img, (256,256))

# Mostrando as dimensoes, deve ser (256X256X3) == (alturaXlarguraXnumero_de_canais_de_cor)
print('Img shape: ',img.shape)

'''
 Chamando a funcao que prediz a pose, onde os parametros sao:
 thresh : limiar de corte para o nivel de confianca para o keypoint
 pltJ : plotar keypoints (ou Joints)
 pltL : plotar segmentos (ou Limbs)
'''
new_img = inf.pltSkeleton(img, thresh = 0.5, pltJ = True, pltL = True)
new_img = cv2.cvtColor(new_img, cv2.COLOR_RGB2BGR)
cv2.imwrite(save_dir+img_name+'_prediction'+extension,new_img)

# Nessa funcao a predicao para na fase de çãdeteco de pessoas
bb = inf.pltBoundingBoxes(img)
bb = cv2.cvtColor(bb, cv2.COLOR_RGB2BGR)
cv2.imwrite(save_dir+img_name.split('.')[0]+'_bb'+extension,bb)

# Nessa funcao a predicao para quando temos os heatmaps, a funcao retorna eles
hm = inf.predictHM(img)
hm = cv2.cvtColor(hm, cv2.COLOR_RGB2BGR)
cv2.imwrite(save_dir+img_name.split('.')[0]+'_hm'+extension,hm)

print(f'Done! Check {save_dir} for the results!')
```

4.9. Considerações Finais

Este capítulo apresenta os fundamentos e tecnologias para desenvolver modelos de *Deep Learning* para analise de vídeo. O capítulo inicia com a explicação da instalação e uso básico do *framework* Tensorflow. Em seguida são apresentados os fundamentos de redes neurais e *Deep Learning*, focando principalmente em modelos do tipo CNN. Adicionalmente, o capítulo também aborda e evolução das técnicas de *Deep Learning*. Em especial, apresenta a evolução da rede InceptionNet, que é considerada um *milestone* da área e foi a vencedora do desafio ImageNet 2014. O capítulo é concluído com a apresentação de três projetos práticos, um de classificação de cenas de vídeo, reconhecimento de objetos e, por ultimo, estimação de pose

Nesse contexto, acreditamos que a proposta do capítulo é interessante para estudantes, pesquisadores e profissionais de TI. Em particular, esperamos que o leitor esteja apto para usar *Deep Learning* para desenvolver aplicações que envolvam tarefas como classificação de de cenas de vídeo, detecção de objetos e estimação de pose.

Referências

[Andriluka et al. 2014] Andriluka, M., Pishchulin, L., Gehler, P., and Schiele, B. (2014). 2d human pose estimation: New benchmark and state of the art analysis. In *Proceedings of the IEEE Conference on computer Vision and Pattern Recognition*, pages 3686–3693.

[Arandjelovic et al. 2016] Arandjelovic, R., Gronat, P., Torii, A., Pajdla, T., and Sivic, J. (2016). Netvlad: Cnn architecture for weakly supervised place recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5297–5307.

[Cao et al. 2018] Cao, Z., Hidalgo, G., Simon, T., Wei, S.-E., and Sheikh, Y. (2018). OpenPose: realtime multi-person 2D pose estimation using Part Affinity Fields. In *arXiv preprint arXiv:1812.08008*.

[Deng et al. 2009] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE.

[Everingham et al. 2010] Everingham, M., Van Gool, L., Williams, C. K., Winn, J., and Zisserman, A. (2010). The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338.

[Gemmeke et al. 2017] Gemmeke, J. F., Ellis, D. P., Freedman, D., Jansen, A., Lawrence, W., Moore, R. C., Plakal, M., and Ritter, M. (2017). Audio set: An ontology and human-labeled dataset for audio events. In *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on*, pages 776–780. IEEE.

[He et al. 2016] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

[Hershey et al. 2017] Hershey, S., Chaudhuri, S., Ellis, D. P. W., Gemmeke, J. F., Jansen, A., Moore, C., Plakal, M., Platt, D., Saurous, R. A., Seybold, B., Slaney, M., Weiss, R., and Wilson, K. (2017). Cnn architectures for large-scale audio classification. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*.

[Hochreiter and Schmidhuber 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8).

[Insafutdinov et al. 2016] Insafutdinov, E., Pishchulin, L., Andres, B., Andriluka, M., and Schiele, B. (2016). Deepercut: A deeper, stronger, and faster multi-person pose estimation model. In *European Conference on Computer Vision*, pages 34–50. Springer.

[Iqbal et al. 2017] Iqbal, U., Milan, A., and Gall, J. (2017). Posetrack: Joint multi-person pose estimation and tracking. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2011–2020.

[Lin et al. 2014] Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. (2014). Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer.

[Milan et al. 2016] Milan, A., Leal-Taixé, L., Reid, I., Roth, S., and Schindler, K. (2016). Mot16: A benchmark for multi-object tracking. *arXiv preprint arXiv:1603.00831*.

[Newell et al. 2016] Newell, A., Yang, K., and Deng, J. (2016). Stacked hourglass networks for human pose estimation. In *European Conference on Computer Vision*. Springer.

[Oord et al. 2016] Oord, A. v. d., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., and Kavukcuoglu, K. (2016). Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*.

[Redmon et al. 2016] Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. (2016). You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788.

[Redmon and Farhadi 2018] Redmon, J. and Farhadi, A. (2018). Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*.

[Rosenblatt 1957] Rosenblatt, F. (1957). *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory.

[Ruggero Ronchi and Perona 2017] Ruggero Ronchi, M. and Perona, P. (2017). Benchmarking and error diagnosis in multi-instance pose estimation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 369–378.

[Shotton et al. 2011] Shotton, J., Fitzgibbon, A., Cook, M., Sharp, T., Finocchio, M., Moore, R., Kipman, A., and Blake, A. (2011). Real-time human pose recognition in parts from single depth images. In *CVPR 2011*, pages 1297–1304. Ieee.

[Stiefelhagen et al. 2006] Stiefelhagen, R., Bernardin, K., Bowers, R., Garofolo, J., Mostefa, D., and Soundararajan, P. (2006). The clear 2006 evaluation. In *International evaluation workshop on classification of events, activities and relationships*, pages 1–44. Springer.

[Szegedy et al. 2017] Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. A. (2017). Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, volume 4, page 12.

[Szegedy et al. 2015] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9.

[Szegedy et al. 2016] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826.

[Tompson et al. 2014] Tompson, J. J., Jain, A., LeCun, Y., and Bregler, C. (2014). Joint training of a convolutional network and a graphical model for human pose estimation. In *Advances in neural information processing systems*, pages 1799–1807.

[Tremblay et al. 2018] Tremblay, J., To, T., Sundaralingam, B., Xiang, Y., Fox, D., and Birchfield, S. (2018). Deep object pose estimation for semantic robotic grasping of household objects. *arXiv preprint arXiv:1809.10790*.

[Wang et al. 2010] Wang, S., Ai, H., Yamashita, T., and Lao, S. (2010). Combined top-down/bottom-up human articulated pose estimation using adaboost learning. In *2010 20th International Conference on Pattern Recognition*, pages 3670–3673. IEEE.

[Yang and Ramanan 2012] Yang, Y. and Ramanan, D. (2012). Articulated human detection with flexible mixtures of parts. *IEEE transactions on pattern analysis and machine intelligence*, 35(12):2878–2890.

BIO

Gabriel Pereira dos Santos is an assistant researcher working under the guidance of Prof. Sergio Colcher at Pontifical Catholic University of Rio de Janeiro (PUC-Rio). He is graduating on Electrical Engineering at WYDEN Univerity on Brazil. His research interests are in multimedia systems and artificial intelligence, working mainly on the following topics: Video Classification and Economic Indicators Prediction. Currently, He is working on the VideoMR project, one of the selected projects in the Microsoft and RNP A.I. challenge, which aims to detect improper content in videos. Lattes CV: `http://lattes.cnpq.br/8088572506239597`.

Pedro Vinicius Almeida de Freitas is a MSc. candidate working under the guidance of Prof. Sergio Colcher at Pontifical Catholic University of Rio de Janeiro (PUC-Rio). He received a B.S. (2018) in Computer Science from the Federal University of Maranhão (UFMA) on Brazil. He is a assistant researcher at Telemidia Lab – PUC-Rio. His research interests are in multimedia systems and artificial intelligence, working mainly on the following topics: Video classification, Economic indicators prediction and Pose estimation. Currently, He is working on the VideoMR project, one of the selected projects in the Microsoft and RNP A.I. challenge, which aims to detect improper content in videos. Lattes CV: `http://lattes.cnpq.br/7566765486024024`.

Antonio Busson is Ph.D. candidate working under the guidance of Prof. Sergio Colcher at Pontifical Catholic University of Rio de Janeiro (PUC-Rio). He received a B.S. (2013) and M.S. (2015) in Computer Science from the Federal University of Maranhão (UFMA) on Brazil. He is researcher member at Telemidia Lab – PUC-Rio and research collaborator at the Laboratory of Advanced Web Systems (LAWS) – UFMA. His research interests are in multimedia systems, working mainly on the following topics: Coding and Processing of multimedia data; Hypermedia Document models, Pattern Recognition, and applications such as Web, iDTV and Games. Currently, He is working on the VideoMR project, one of the selected projects in the Microsoft and RNP A.I. challenge, which aims to detect improper content in videos. Lattes CV: `http://lattes.cnpq.br/1857348479447184`.

Álan Guedes holds a Ph.D. (2017) from the PUC-Rio, where he acts as post-Phd Researcher in TeleMídia Lab. He received his Bachelor (2009) and M.Sc. (2012) degrees in Computer Science from the UFPB, where he worked as researcher in Lavid Lab. In both labs, Álan worked in interactive video and TV research projects, and contribute to the Ginga and N CL specifications, which today are standards for DTV, IPTV and IBB. His research interests include Interactive Multimedia, Immersive Media, and Deep Learning for Multimedia. Lattes: `http://lattes.cnpq.br/1481576313942910`.

Ruy Luiz Milidiú received B.S. (1974) in Mathematics and M.S. (1978) in Applied Mathematics from Federal University of Rio de Janeiro. He also received a M.S. (1983) and Ph.D. (1985) in Operations Research

from University of California. Currently, he teaches in Informatics Department at Pontifical Catholic University of Rio de Janeiro (PUC-Rio). He has experience in Computer Science, focusing on Algorithmics, Machine Learning and Computational Complexity. Lattes CV: `http://lattes.cnpq.br/6918010504362643`.

Sérgio Colcher received B.S. (1991) in Computer Engineer, M.S. (1993) in Computer Science and Ph.D. (1999) in Informatics, all by PUC-Rio, in addition to the postdoctoral (2003) at ISIMA (Institute Supérieur d'Informatique et de Modelisation des Applications - Université Blaise Pascal, Clermont Ferrand, France). Currently, he teaches in Informatics Department at Pontifical Catholic University of Rio de Janeiro (PUC-Rio). His research interests include computer networks, analysis of performance of computer systems, multimedia/hypermedia systems and Digital TV. Lattes CV: `http://lattes.cnpq.br/1104157433492666`.