Capítulo 3

Desenvolvendo Modelos de Deep Learning para Aplicações Multimídia no Tensorflow

Antonio José G. Busson[1], Lucas C. Figueiredo[1], Gabriel P. dos Santos[2], André Luiz de B. Damasceno[1], Sérgio Colcher[1] e Ruy L. Milidiú[1]

[1]Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro (INF/PUC-Rio)

[2]Faculdade ISL | Wyden

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

gabrieainha@gmail.com

Abstract

The availability of massive quantities of data, combined with increasing computational capabilities, makes it possible to develop more precise Machine Learning algorithms. These new tools provide advances in areas such as Natural Language Processing and Computer Vision, allowing efficient processing of images, text and audio. Now, cognitive functionalities, such as learning, recognition and detection, can be used in multimedia applications to create mechanisms beyond traditional capture, streaming and presentation uses. Methods based on Deep Learning became state-of-the-art in several Multimedia challenges. This short course presents the grounds and ways to develop models using Deep Learning. It prepares the participant to: (1) understand and develop models based on Deep Neural Networks, Convolutional Neural Networks (CNN), Recurrent Neural Networks, including LSTM and GRU; (2) apply the Deep Learning models to solve problems within the multimedia domain like Image Classification, Facial Recognition, Object Detection, Video Scenes Classification. The Python programming language is shown alongside TensorFlow, a package for developing Deep Learning models.

Resumo

A disponibilidade de massivos volumes de dados somado ao aumento do poder computacional tornou possível a criação de métodos de Machine Learning mais precisos, provocando significativos avanços nas áreas de processamento de linguagem natural, visão e audição computacional. Tais avanços refletem em novas funcionalidades cognitivas

(e.g. aprendizagem, reconhecimento, detecção) que podem ser incorporadas em aplicações multimídia, dessa forma, permitindo a criação de novos mecanismos para uso das mídias, além do uso tradicional (e.g. captura, transmissão e apresentação). Métodos baseados em Deep Learning se tornaram o estado-da-arte em muitos problemas do domínio da Multimídia. Este minicurso tem como foco apresentar os fundamentos e tecnologias para desenvolver tais modelos de Deep Learning. Em especial, o minicurso prepara o participante para: (1) entender e desenvolver redes neurais profundas, redes neurais convolucionais (CNN), e redes neurais recorrentes (LSTM/GRU); (2) aplicar os modelos de Deep Learning para resolver problemas do domínio da multimídia: classificação de imagens, reconhecimento facial, detecção de objetos e classificação de cenas de vídeo. A linguagem de programação Python é apresentada em conjunto com a biblioteca TensorFlow para implementação dos modelos de Deep Learning ao decorrer do minicurso.

3.1. Introdução

Estamos vivendo em um período de produção de massivos volumes de conteúdo multimídia. Tomando como exemplo o YouTube[1] que, conforme mostra a Figura 3.1, em 2014 registrou uma taxa de *upload* de 72 horas de vídeo por minuto[2]. Já em 2018, esse número subiu para 400 horas[3].

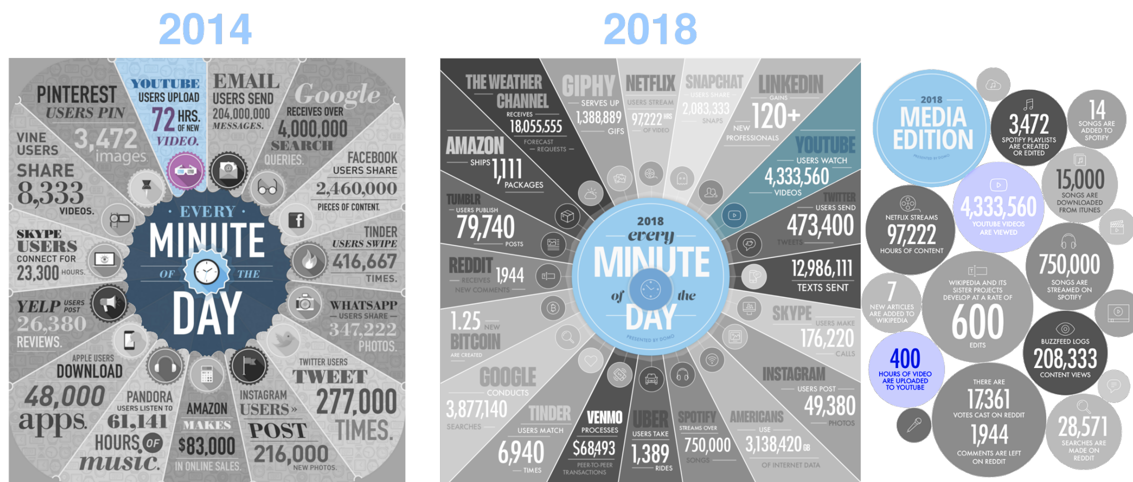


Figura 3.1. *Upload* de vídeos por minuto no YouTube (2014 e 2018).

Uma das razões para esse crescimento está relacionado: (i) a popularização e barateamento de equipamentos de produção de conteúdo multimídia, como *smartphones* e *tablets*; (ii) advento de serviços/plataformas para armazenamento e distribuição de conteúdo multimídia. Como consequência, o grande desafio do cenário atual consiste em extrair e organizar informações desse conteúdo multimídia de forma eficiente e prática. Isso implica na necessidade de estratégias e desenvolvimento de processos automáticos

[1] `https://www.youtube.com`
[2] `https://www.domo.com/learn/data-never-sleeps-2`
[3] `https://www.domo.com/learn/data-never-sleeps-6`

para análise do conteúdo multimídia, contemplando não somente a identificação de objetos mas também a compreensão de eventos mais complexos.

Nos últimos anos, o *Deep Learning* (DL) permitiu significativo avanço de vários segmentos da multimídia, principalmente em tarefas relacionadas a processamento de fala, audição e visão computacional [Ota et al. 2017]. Plataformas como IBM Watson[4] e Microsoft Azure ML[5] já oferecem DLaS (Deep Learning as a Service), permitindo que sistemas multimídia (e.g. Helpicto [Equadex 2018], PersonalizedTV [Thomas and Sadagopan 2016], ShrewsburyMuseum [Kearn and Beeby 2017]) possam incorporar novas funcionalidades baseadas em aprendizagem e reconhecimento de padrões, o que os leva para a categoria de IIMS (Intelligent Interactive Multimedia Systems).

Este capítulo apresenta os fundamentos e tecnologias para desenvolver modelos de *Deep Learning*. Em especial, o minicurso prepara o participante para: (1) entender e desenvolver redes neurais profundas, especialmente os modelos baseados em redes neurais convolucionais (CNN), e redes neurais recorrentes (LSTM/GRU); (2) aplicar os modelos de *Deep Learning* para resolver problemas do domínio da multimídia: classificação de imagens, reconhecimento facial, detecção de objetos e classificação de cenas de vídeo. A linguagem de programação Python é apresentada em conjunto com o *framework TensorFlow* para implementação dos modelos de *Deep Learning* ao decorrer do minicurso. Vale ressaltar que todos os projetos implementados neste minicurso estão disponíveis no Github[6].

O restante desse trabalho está organizado como a seguir. A Seção 3.2 apresenta o *framework TensorFlow*. A Seção 3.3 introduz os conceitos básicos de aprendizado de máquina. Em seguida, as Seções 3.4, 3.5 e 3.6 apresentam os fundamentos de Redes Neurais, Redes Neurais Convolucionais e Redes Neurais Recorrentes, respectivamente. As implementações para resolução de problemas de reconhecimento facial e detecção de objetos são descritos nas Seções 3.7 e 3.8, respectivamente. Por fim, a Seção 3.9 apresenta as considerações finais.

3.2. Framework TensorFlow

O *Tensorflow*[7] é um framework *open-source* para Python, Javascript, C/C++ e Go e tem o objetivo de auxiliar o processamento de dados em *machine learning* por meio de um modelo baseado em fluxo de dados. Este 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[8] (criado em 2011), com o intuito de melhorar seu desempenho.

Nesta Seção é apresentada uma visão geral do *framework Tensorflow*. A Subseção 3.2.3 descreve o processo de instalação. Em seguida, a Subção 3.2.2 apresenta os fundamentos e estruturas básicas do *framework*. Por fim, a Subseção 3.2.3 descreve como utilizar o *framework*.

[4] https://www.ibm.com/watson/
[5] https://azure.microsoft.com/pt-br/services/machine-learning-studio/
[6] https://github.com/Busson/curso_deep_learning_para_multimidia
[7] https://www.tensorflow.org/?hl=pt-br
[8] https://ai.google/research/pubs/pub40565

3.2.1. Instalação

No decorrer desta subseção é descrito como instalar o *Tensorflow* em sistemas Ubuntu e outras distribuições derivadas do Linux. Os projetos que são apresentados no decorrer do minicurso são implementados na linguagem Python.

O Python e Pip (sistema de gerenciamento de pacotes do Python) já estão presentes no Ubuntu e na maioria das outras distribuições Linux. Porém, com Python2.7 e não o mais recente, Python3.

Para instalação do Python2.7, execute:

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

Para instalação do Python3.x, 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 no Python2 e Python3 execute respectivamente:

```
pip -V
```

ou

```
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()
```

3.2.2. 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.

3.2.3. Utilizando o Tensorflow

Após entender as estruturas básicas do *framework*, para usar o *Tensorflow*, como mostra a Listagem 3.1, 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 tesor **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 3.1. 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 3.2 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 3.2. 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]]
```

3.3. Fundamentos de Aprendizado de Máquina

Aprendizagem de máquina, ou aprendizado automático é um subcampo da área de Inteligencia 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" [Samuel 1959]. 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 invez 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 cojunto 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.

3.4. 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 as áreas de visão computacional, reconhecimento de fala e processamento de linguagem natural.

Nesta seção são apresentados os conceitos básicos de redes neurais. Primeiro, na Subseção 3.4.1 são apresentados os modelos Perceptron e Perceptron de Múltiplas Camadas. Em seguida, na Subseção 3.4.2 é descrito o algoritmo de Retropropagação. Na Subseção 3.4.3 é apresentada 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.

3.4.1. Perceptron

O Perceptron [Rosenblatt 1957], inventado em 1957 por Frank Rosenblatt, é a estrutura mais básica de uma Rede Neural. A Figura 3.2 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: $z = w^t.x + b$.

Múltiplos Perceptrons podem ser usados para realizar tarefas de classificação múltipla. Nesse caso, como ilustra a rede A da Figura 3.3, 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 3.3, os neurônios também podem ser

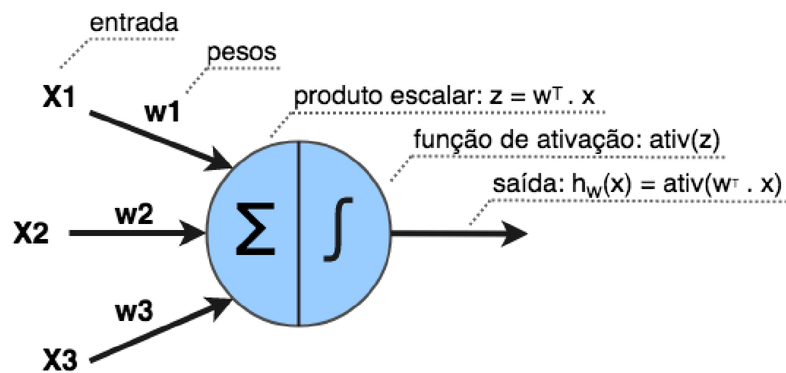


Figura 3.2. Estrutura de um Perceptron

estruturados em múltiplas camadas, onde cada neurônio das camadas intermediarias (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.

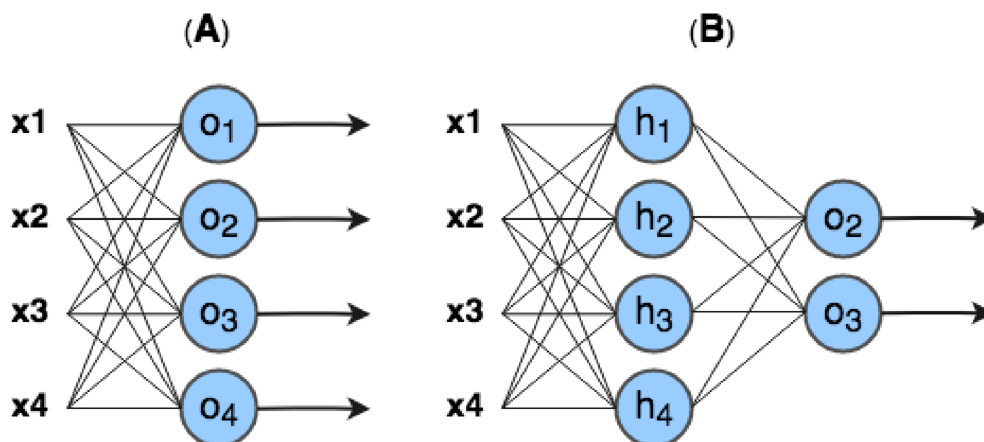


Figura 3.3. (A) Perceptron de múltiplas classes. (B) Perceptron múltiplas camadas.

A Figura 3.4[9] 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.

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

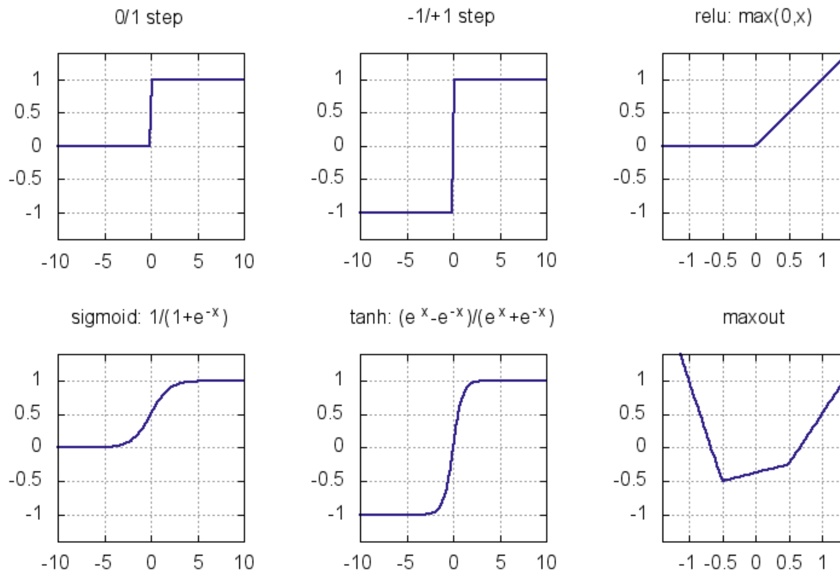


Figura 3.4. Funções de ativação.

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.

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

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

3.4.2. Algoritmo de Retropropagação

O *Tensorflow* encapsula o algoritmo de retropropagação, portanto não é necessário implementa-lo. No entanto, para entender redes neurais é inecessário entender com a retropapagação funciona. Basicamente, o algorítimo 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 $\frac{\partial C}{\delta a_j^l}$ 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:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{6}$$

Isto é, o erro δ_j^l é igual a mudança $\frac{\partial C}{\partial w_{jk}^l}$.

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \tag{7}$$

O termo $\frac{\partial C}{\partial w_{jk}^l}$ é calculado em relação ao erro δ_j^l e ativação da camada anterior a_k^{l-1}. Dessa forma, quando a ativação da camada anterior é pequena, espera-se que o termo do gradiente $\frac{\partial C}{\partial w_{jk}^l}$ 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 $\delta^L = \nabla_a C \odot \sigma'(z^L)$.

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

3.4.3. 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:

$$a^L_j = \frac{e^{z^L_j}}{\sum_k e^{z^L_k}} \tag{8}$$

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 3.5, se o vetor do produto escalar $(z^L_1, z^L_2, z^L_3, z^L_4)$ = $(0.1, 0.9, 0.4, 2.3)$, então o vetor de ativação $(a^L_1, a^L_2, a^L_3, a^L_4)$ = $(0.07, 0.16, 0.09, 0.66)$. Isso significa que dada entrada X tem 66% de chance de ser da classe 4.

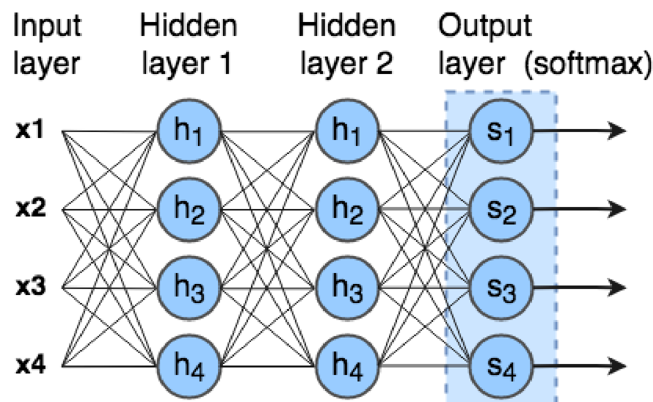


Figura 3.5. MLP Moderno com uma camada *softmax* para classificação.

3.4.4. Implementando um MLP

Nesta subseção é exemplificado o uso de um MLP para resolver um problema não linearmente separável. A Listagem 3.3 mostra o código que cria um pequeno dataset com

pontos distribuídos de forma circular usando a biblioteca scikitlearn[10], que pode ser visualizado na Figura 3.6. O dataset é composto por pontos de duas dimensões (duas *features*) que pertencem a dois conjuntos de classe, vermelho (0) e azul (1).

Listagem 3.3. 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 sequencia  conhecida
9 tf.set_random_seed(0)
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)
```

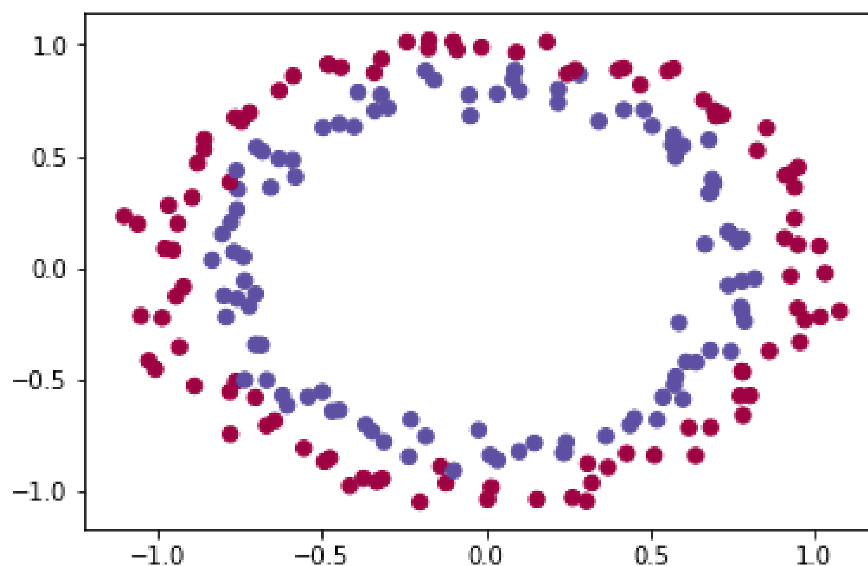


Figura 3.6. Visualização do dataset criado na Listagem 3.3

A Listagem 3.4 descreve a implementação de uma rede neural do tipo MLP.

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

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 3.4. 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 do 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 3.5 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 3.5. 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 variaveis
11 sess.run(tf.global_variables_initializer())
```

A Listagem 3.6 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 3.7. Por fim, as linhas 27-37 mostram a saída do programa.

Listagem 3.6. 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("accuracia 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 > accuracia do modelo: 0.97
38 > a classe do ponto [[0.4013312  0.88583093]] e: [0]
```

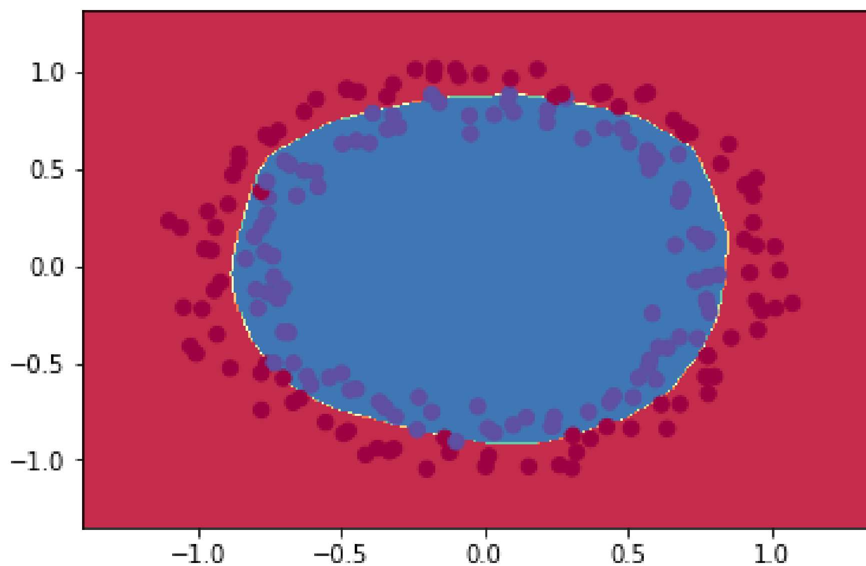


Figura 3.7. dataset separado pelo MLP.

3.5. 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 compõe 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 reconhecimento de padrões em imagens.

Nesta Seção são apresentados os fundamentos básicos e técnicas para implementação de CNNs. Na Subseção 3.5.1 a operação de convolução e *pooling* é apresentada. Em seguida, na Subseção 3.5.2 descreve a implementação de uma CNN para classificação de imagens de sinais de mão. Por fim, na Subseção 3.5.2.1 é apresentado o funcionamento e histórico de evolução da rede InceptionNet.

3.5.1. 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 continuas, 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 3.8 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 ...

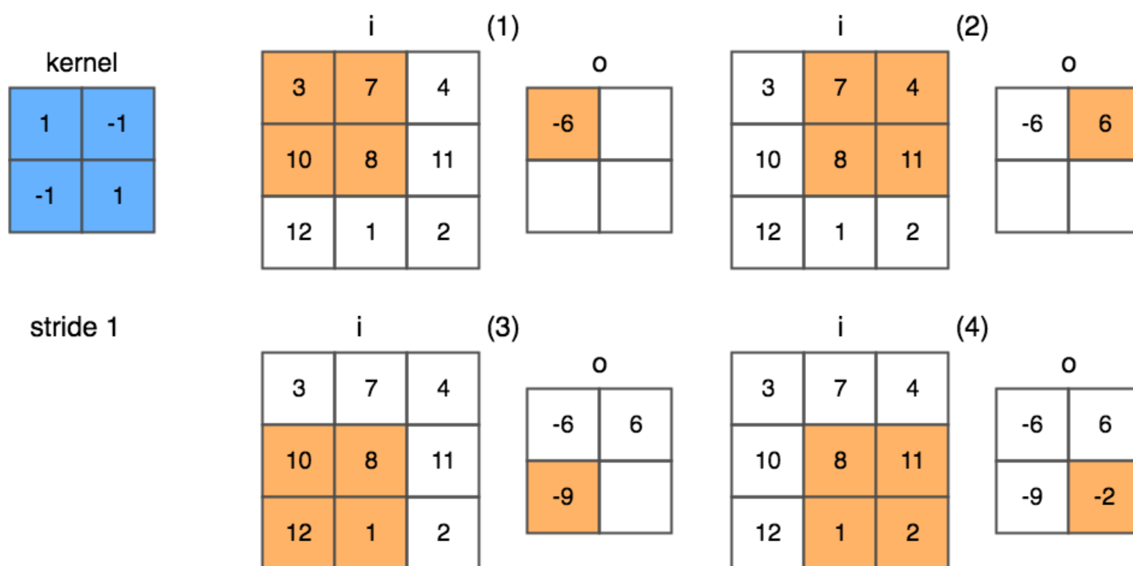


Figura 3.8. Processo de convolução.

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 distancia L^2 entre os elementos de uma área do tensor. A Figura 3.9 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.

A Figuta 3.10 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

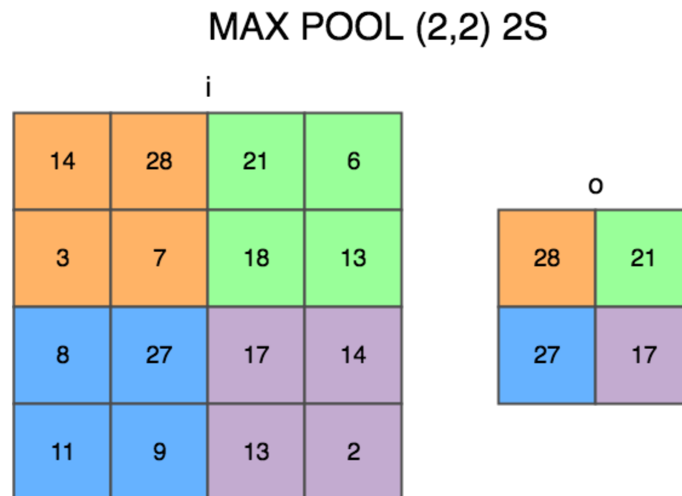


Figura 3.9. Exemplo de um processo de max *pooling* com *kernel* 2x2 e *stride* 2.

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 póxima 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).

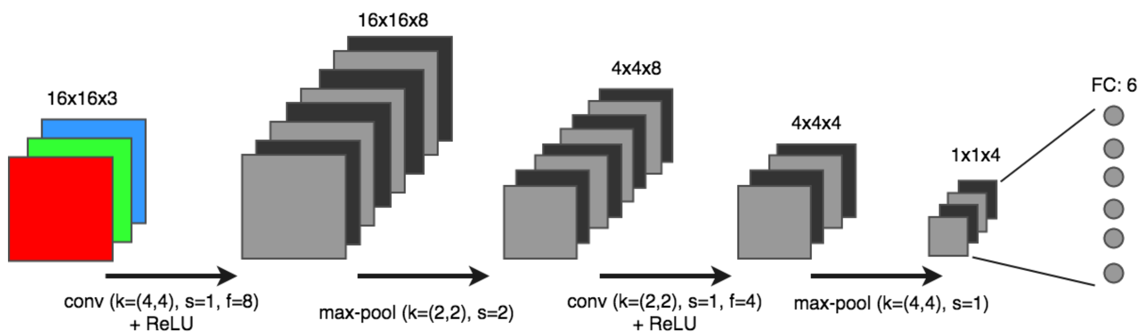


Figura 3.10. Exemplo de uma arquitetura de CNN.

3.5.2. 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) [11]. O *dataset* é composto por 1200 fotos de sinais de mão no formato RGB com dimensões 64x64. A Figura 3.12 ilustra os seis tipos de sinais de mãos encontrados no *dataset*, bem como os respectivos vetores de *labels* codificados no formato *OneHot*.

[11] `https://www.deeplearning.ai/`

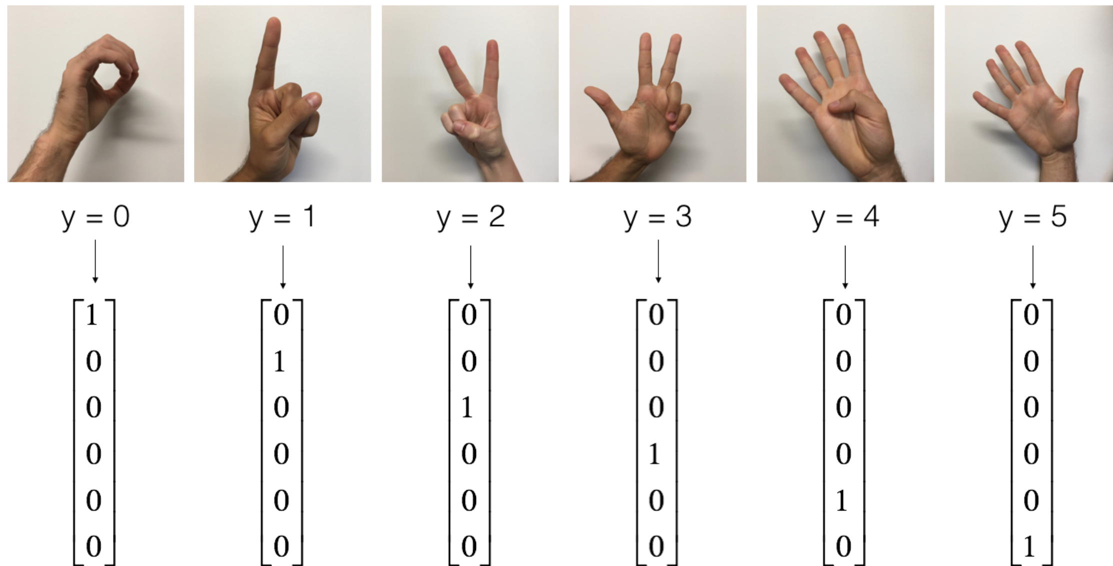


Figura 3.11. exemplos de cada classe do *dataset* de sinais de mão e suas respectivas codificações OneHot (by Prof. Andrew Ng., deeplearning.ai).

A Listagem 3.7 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 3.7. 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 sequencia conhecida
13 tf.set_random_seed(0)
14 np.random.seed(0)
15
16 #carregando o dataset (dividido em treino e teste)
17 X_train, Y_train, X_test, Y_test, classes = load_dataset()
18
```

```
19 #imprimindo as dimensoes dos conjuntos de treino e teste do dataset
20 print ("X_train shape: " + str(X_train.shape))
21 print ("Y_train shape: " + str(Y_train.shape))
22 print ("X_test shape: " + str(X_test.shape))
23 print ("Y_test shape: " + str(Y_test.shape))
24
25 #exibindo um exemplo
26 index = 6
27 plt.imshow(X_train[index])
28 print ("y =", Y_train[index])
29 ------------------------------------ OUTPUT ------------------------------------
30 > X_train shape: (1080, 64, 64, 3)
31 > Y_train shape: (1080,)
32 > X_test shape: (120, 64, 64, 3)
33 > Y_test shape: (120,)
34 > y = 2
```

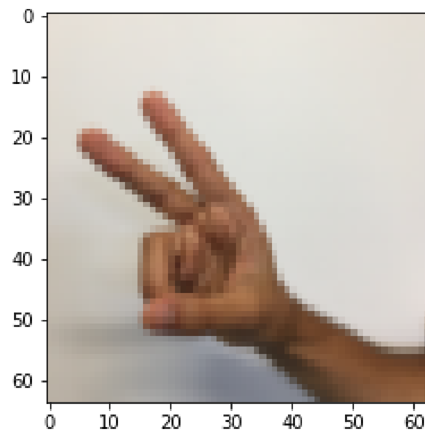


Figura 3.12. Exemplo de imagem renderizada na Listagem 3.7.

A Listagem 3.8 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 3.8. Construindo uma CNN.

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

```
46      softmax = tf.nn.softmax(out)
47
48      #Classe
49      class_ = tf.argmax(softmax,1)
50
51      #áAcurcia
52      compare_prediction = tf.equal(class_, Y)
53      accuracy = tf.reduce_mean(tf.cast(compare_prediction, tf.float32))
54
55      return X, Y, loss, opt, softmax, class_, accuracy
```

A Listagem 3.9 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 3.9. 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 3.10 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 3.10. 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 epocas
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})
23 print("accurcia do modelo:", acc)
24 ------------------------------ OUTPUT ------------------------------
25 > erro na epoca 0 : 1.79012
26 > erro na epoca 10 : 1.53382
27 > erro na epoca 20 : 0.809482
28 > erro na epoca 30 : 0.586155
29 > erro na epoca 40 : 0.505508
30 > erro na epoca 50 : 0.366477
31 > erro na epoca 60 : 0.29243
32 > erro na epoca 70 : 0.240354
33 > erro na epoca 80 : 0.21093
34 > erro na epoca 90 : 0.143943
35 > accurcia do modelo: 0.908333
```

A Listagem 3.11 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 3.13). Por fim, a linha 9 mostra a saída do programa.

Listagem 3.11. 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]
```

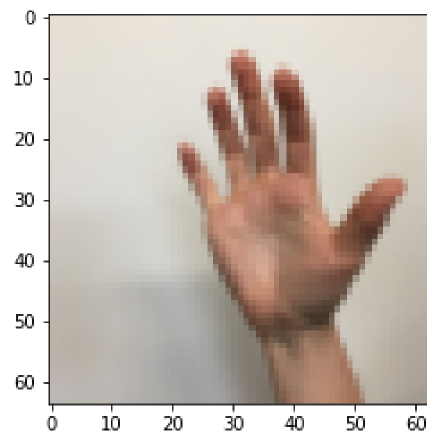


Figura 3.13. Imagem classificada como sinal 5 na Listagem 3.11.

3.5.2.1. 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 3.14, 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 esta distribuída globalmente, e um *kernel* curto quando a informação esta distribuída mais localmente.

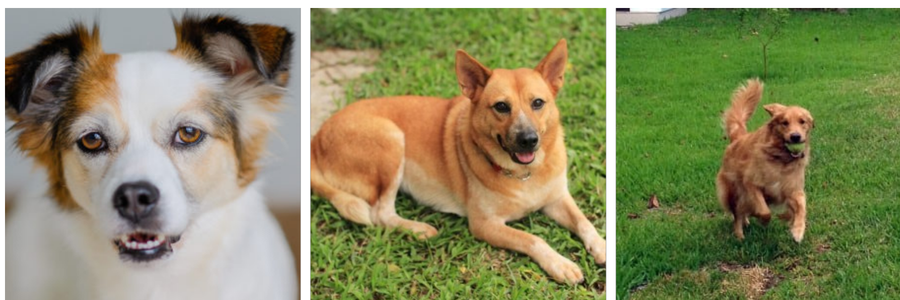


Figura 3.14. Esquerda: cão ocupando quase toda a imagem; Centro: cão ocupando uma parte da imagem; Direita: cão ocupando uma pequena parte da imagem.

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 3.15. 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.

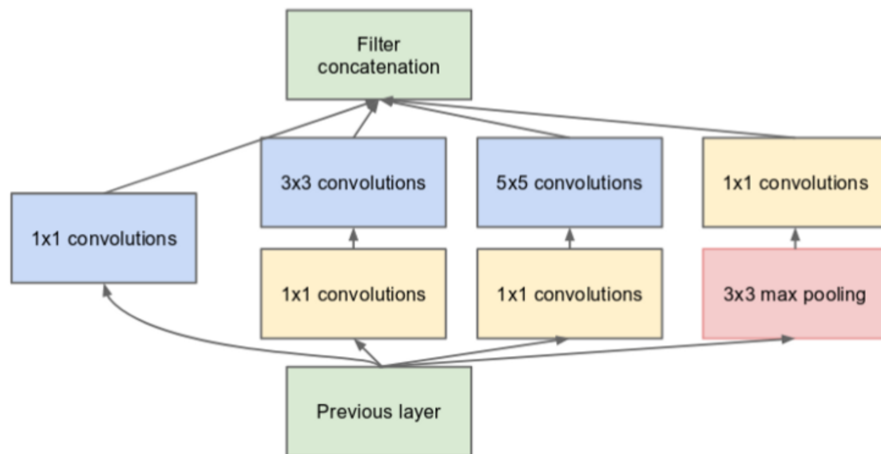


Figura 3.15. Módulo Inception da rede InceptionNet.

Como pode ser visto na Figura 3.16, 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.

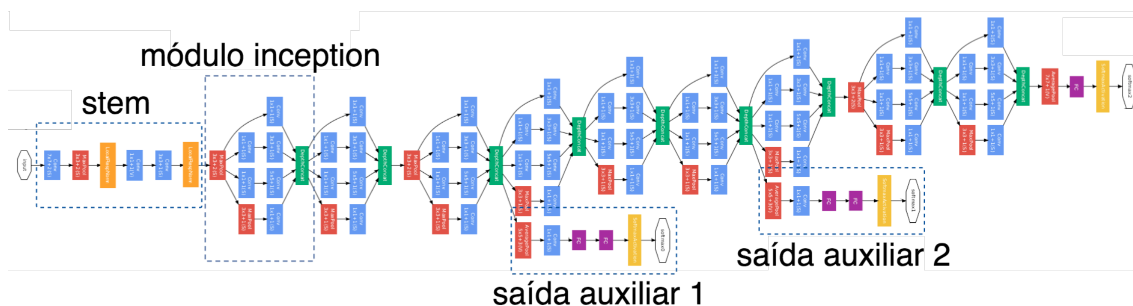


Figura 3.16. Arquitetura da rede InceptionNet (GoogleNet).

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 refatoram as convoluções 5x5 em duas convoluções menores. Como pode ser visto na Figura 3.17, no módulo A a convoluções 5x5 foi substituída por uma sequencia 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.

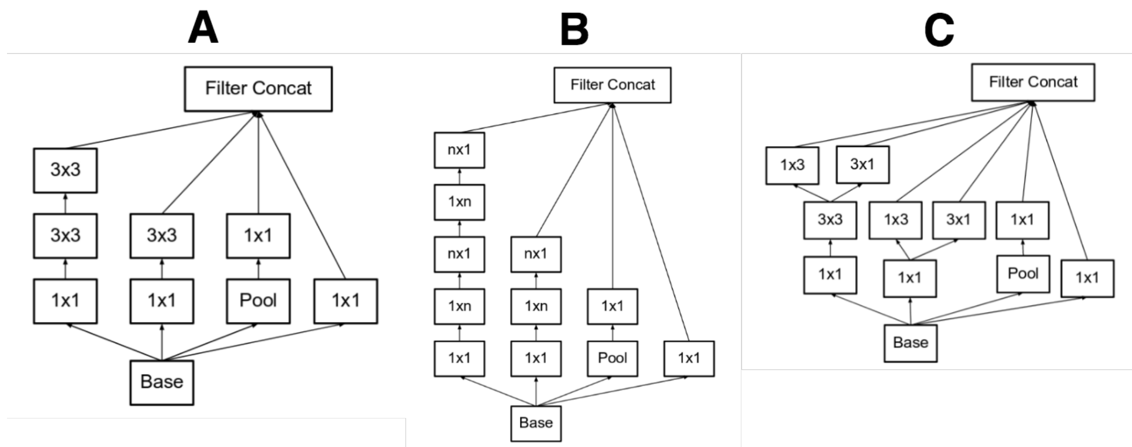


Figura 3.17. Três tipos de módulo inception da arquitetura InceptionNet v2.

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 algum módulos da arquitetura. A Figura 3.18 ilustra a arquitetura geral da rede. A Figura 3.19 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.

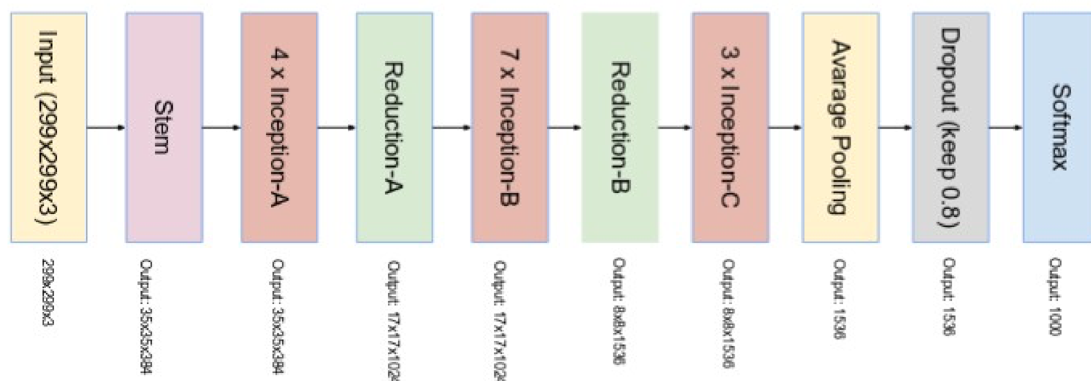


Figura 3.18. Arquitetura da rede InceptionNet v4.

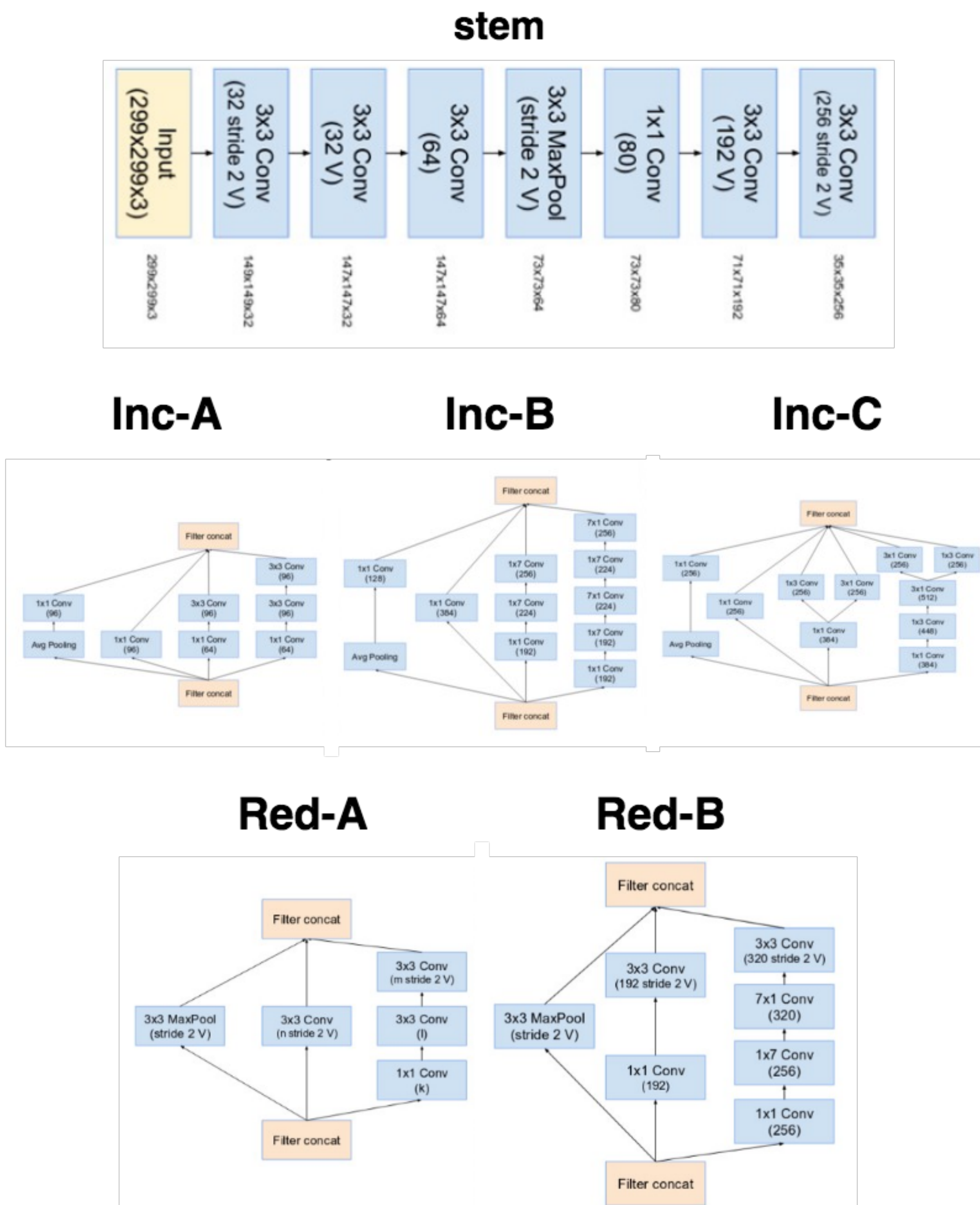


Figura 3.19. (1) Bloco *Stem* da rede InceptionNet v4. (2) IR-A, IR-B e IR-C: Três tipos de módulo Inception da rede InceptionNet v4. (3) Red-A e Red-B - Bloco de redução de 35x35 para 17x17, e 17x17 para 8x8, respectivamente.

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 3.20 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 3.21.

A principal ideia da arquitetura Inception-ResNet é a adição das conexões residuais propostas pela rede ResNet. A Figura 3.21 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 contataram que a rede tende a instabilidade quando o a rede excede mil filtros, para estabilizar a rede eles escalaram as ativações residuais por valores entre 0.1 e 0.3.

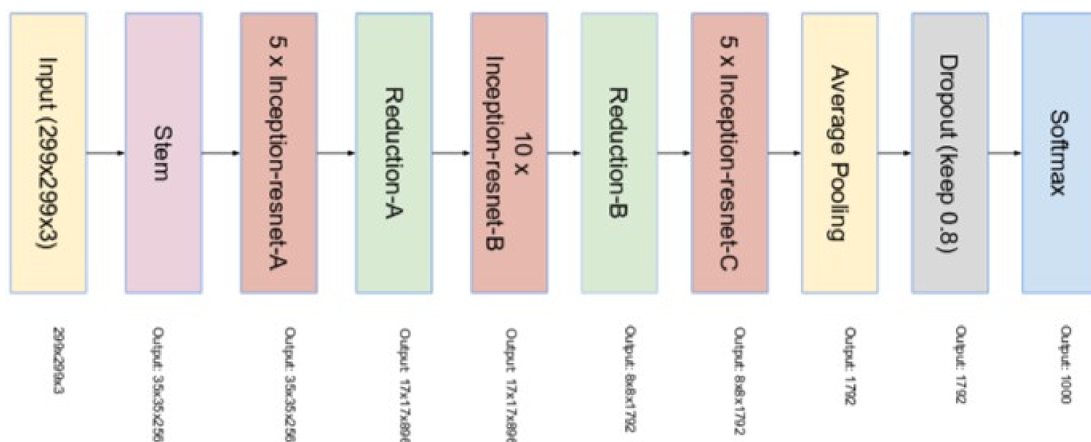


Figura 3.20. Arquitetura das redes Inception-ResNet v1 e v2.

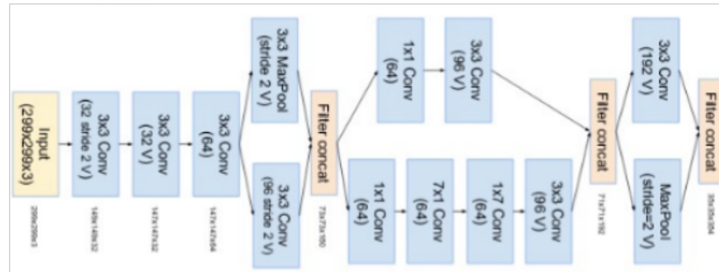
3.6. Redes Neurais Recorrentes

As redes neurais recorrentes, ou RNNs, do inglês *Recurrent Neural Networks* [Rumelhart et al. 1986], são um conjunto de redes neurais especializadas em processamento de dados sequenciais de diferentes tamanhos. Esse tipo de rede são escaláveis para grande sequencias de dados, diferentemente das redes que não são especializadas neste tipo de tarefa. Neste contexto, dados sequenciais são dados ordenados nos quais os valores estão relacionados, como dados temporais, textuais, uma sequência de DNA, dentre outros.

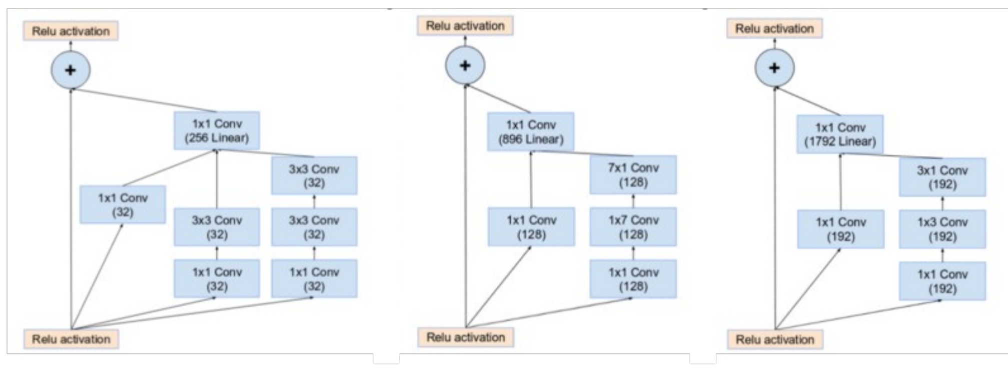
Esta família de redes neurais, diferente do que ocorre nas redes do tipo *feed-forward*, na qual o fluxo de informação tem apenas um sentido, passando de uma camada para a outra, se baseiam na passagem da informação do instante t para o instante $t+1$, assim, as redes recorrentes possuem uma memória do seu estado anterior que é utilizada no processamento do estado atual. Este conceito pode ser observado na Figura 3.22 (a). Esta característica é o que torna estes tipos de redes especiais, uma vez que a sequencia de dados contém informações essenciais sobre o que pode ser observado no próximo instante.

Uma outra maneira de enxergar esta passagem de informação é através do desdobramento da recursão. Considerando que o estado de um nó na rede pode ser dado

stem (Inception-Resnet v2)



IR-A **IR-B** **IR-C**



Red-A **Red-B**

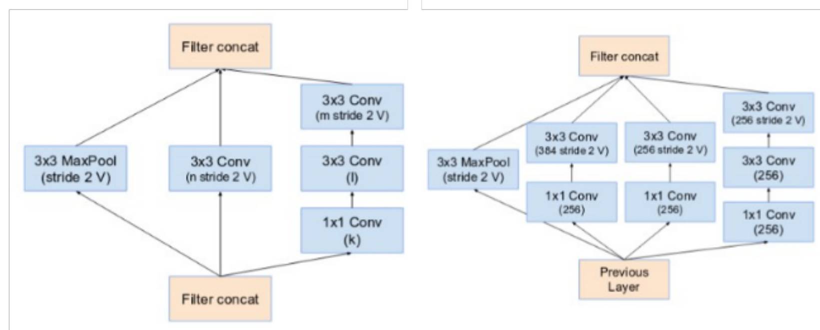


Figura 3.21. (1) Bloco stem da rede Inception-Resnet v2. (2) IR-A, IR-B e iR-C: Três tipos de módulo Inception-Resnet da arquitetura Inception-Resnet v1 e v2. (3) Red-A e Red-B - Bloco de redução de 35x35 para 17x17, e 17x17 para 8x8, respectivamente.

por:

$$s^t = f(s^{t-1}, x^t; \theta),$$

ou seja, o estado atual, s^t, é obtido através de uma função que depende do estado anterior, s^{t-1}, e um sinal externo x^t parametrizados por θ. Para obter o estado no instante T, é possível desdobrar a rede aplicando a operação $T-1$ vezes [Goodfellow et al. 2016]. Este processo de desdobramento é ilustrado na Figura 3.22 (b).

Utilizando os conceitos de passagem de informação entre os instantes e a repre-

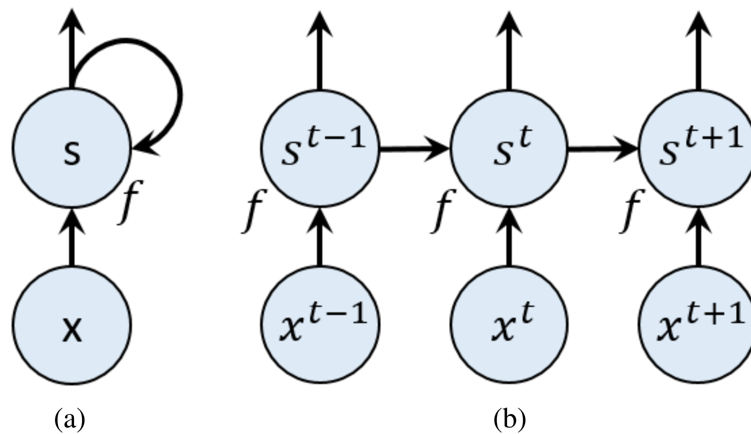


Figura 3.22. Um exemplo de RNN que utiliza a entrada x no instante t juntamente com o estado no instante $t-1$ para processar a saída no instante t. Em (a) a rede é representada com ciclo ou recursão e em (b) a mesma é visualizada com a recursão desdobrada.

sentação desdobrada é possível projetar diversos tipos de redes recorrentes, por exemplo uma rede que possua recorrência nas camadas escondidas e produza uma saída em cada instante (Figura 3.23 (a)) ou que processe todos os dados e gere uma única saída no final (Figura 3.23 (b)).

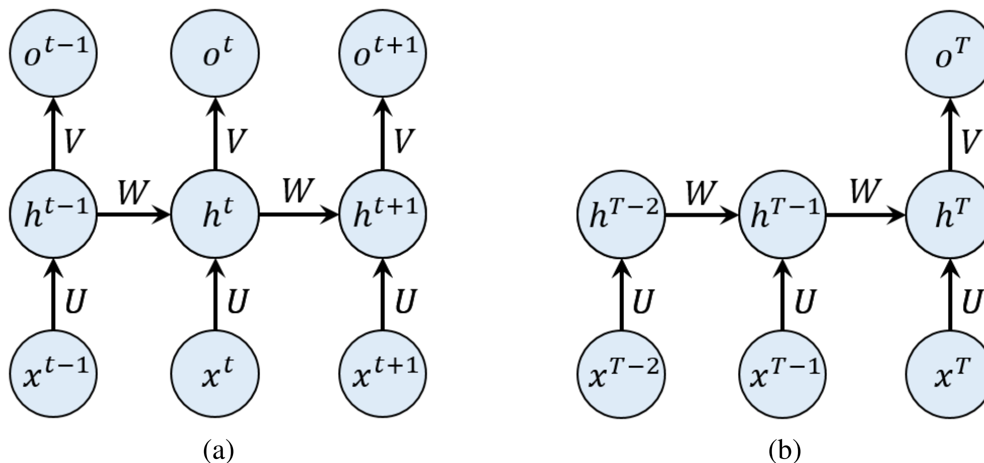


Figura 3.23. Exemplos de RNNs, onde uma sequencia de dados x são processados pela camada escondida h para obter a(s) saída(s) o. Nestas redes, U, W e V são as matrizes de pesos para a entrada, recursão e saída respectivamente. Em (a), é apresentada uma rede com uma saída em cada instante. Já em (b), é produzida uma única saída ao final do processamento.

O cálculo do gradiente neste tipo de rede se da através da aplicação do algoritmo geral de *backpropagation* na representação desdobrada da rede, sendo chamado assim de *Backpropagation Through Time*, ou BPTT. O gradiente então pode ser utilizado por qualquer técnica baseada em gradientes para treinar a RNN.

Os problemas de explosão e diluição de gradiente ao treinar a rede para aprender *long-term dependencies*, apesar de serem evitados por redes do tipo *feed-forward* pois utilizam matrizes de peso diferentes entre as suas camadas [Sussillo 2014], aparecem nas

redes recorrentes uma vez que a mesma matriz W é utilizada em cada passo [Goodfellow et al. 2016]. Diversas estratégias foram desenvolvidas para amenizar o problema de aprendizado de *long-term dependencies*, dentre elas as redes com sistema de portões, que até o presente momento se mostraram as mais eficazes na prática [Goodfellow et al. 2016], incluindo o modelo **LSTM** e redes baseadas em ***Gated Recurrent Unit*** (**GRU**).

As redes com sistema de portões se baseiam na criação de caminhos através dos instantes, nos quais os gradientes não explodem nem diluem, utilizando conexões com pesos que podem variar de acordo com o tempo. Essas conexões permitem que as redes agreguem informação ao longo do tempo e, após utilizarem essa informação, ela pode ser descartada, dessa forma, os portões são responsáveis pelo fluxo de informação dentro da rede. Esses detalhes podem ser observados em uma célula do tipo LSTM, por exemplo, na qual existem 3 portões: *Forget Gate*, *Input Gate* e *Output Gate*, com seu funcionamento esquematizado na Figura 3.24.

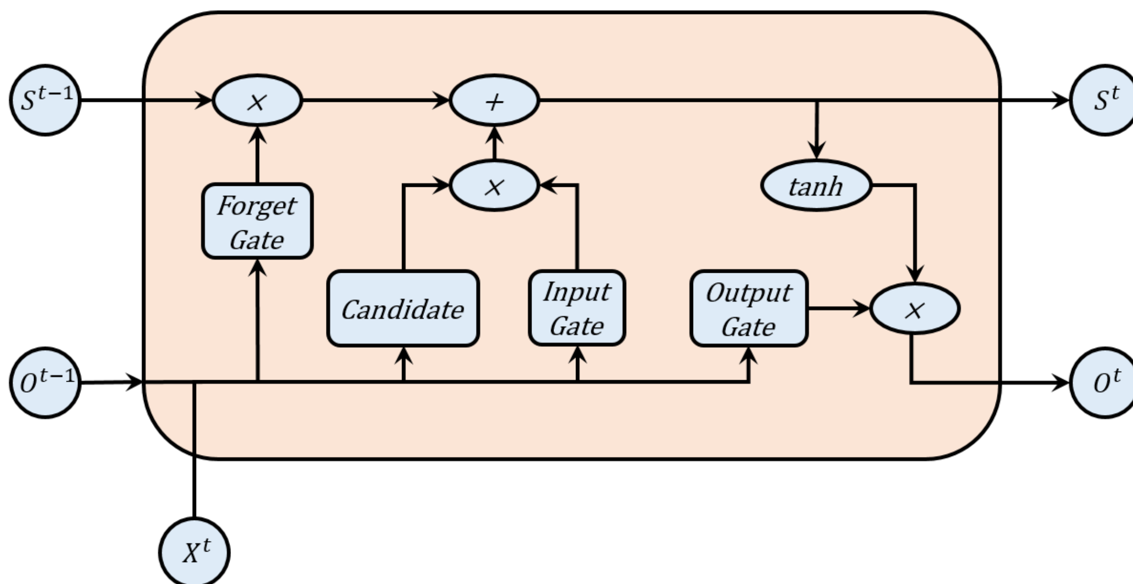


Figura 3.24. Representação de uma célula LSTM, onde S^t representa o estado da célula no instante t, X^t o vetor de entrada no instante t e O^t o vetor com a saída produzida pela célula no instante t.

Dentro de uma célula LSTM, o *forget gate* (f^t) é responsável por decidir o que deve ser removido do estado anterior $t-1$, mantendo apenas o que é relevante para o próximo passo, aplicando uma função *sigmoid* para estabelecer os pesos finais entre 0 e 1. Este portão pode ser dado por:

$$f^t = \sigma(\sum_j W_j^f O_j^{t-1} + \sum_j U_j^f X_j^t + b^f),$$

onde W^f é o vetor de pesos recorrente, U^f é o vetor de pesos para a entrada e b^f é um vetor de *bias*.

No esquema apresentado na Figura 3.24, é possível observar a presença de uma camada chamada *candidate*. Esta camada é responsável por produzir os novos possíveis

valores que serão adicionados ao estado da célula. De maneira similar ao *forget gate*, esta camada utiliza a entrada e a saída do instante anterior parametrizados por vetores de peso W^C e U^C, porém utiliza a função tanh para obter valores entre -1 e 1. Assim, o *candidate* (C^t) é obtido como:

$$C^t = \tanh(\sum_j W_j^C O_j^{t-1} + \sum_j U_j^C X_j^t + b^C)$$

Por sua vez, o *input gate* é responsável por decidir quais dessas novas informações que serão adicionadas no estado da célula, e, de maneira análoga ao *forget gate*, o *input gate* é dado por:

$$i^t = \sigma(\sum_j W_j^i O_j^{t-1} + \sum_j U_j^i X_j^t + b^i)$$

Dessa forma, o estado da célula é atualizado através da combinação dos valores produzidos pela camada *candidate*, parametrizados pelo *input gate*, com os valores do estado no instante anterior, parametrizados pelo *forget gate*, ou seja:

$$S^t = f^t S^{t-1} + C^t i^t$$

Por fim, para obter a saída da célula, a função tanh é aplicada no estado da célula (S^t), obtendo valores entre -1 e 1. Em seguida, o valor obtido é submetido ao *output gate*, que é responsável por decidir quais valores serão colocados na saída. De maneira similar aos outros portões, o *output gate* (q^t) é dado por:

$$q^t = \sigma(\sum_j W_j^q O_j^{t-1} + \sum_j U_j^q X_j^t + b^q),$$

assim, a saída da célula (O^t) é definida por:

$$O^t = \tanh(S^t) q^t$$

Em resumo, no modelo LSTM os portões *forget gate*, *input gate* e *output gate*, utilizam a saída do instante anterior juntamente com a entrada do instante atual para decidir quais valores serão, respectivamente, esquecidos, adicionados ou produzidos como saída. Vale ressaltar que os valores de W e U são independentes entre os portões e podem variar de acordo com tempo, o que diminui as chances do gradiente explodir ou diluir durante o BPTT.

Uma forma simplificada da célula LSTM é a GRU ([Cho et al. 2014, Chung et al. 2014]). A principal diferença com o modelo LSTM é que um único portão passa a controlar simultaneamente a decisão de esquecer e atualizar o estado da célula ([Goodfellow et al. 2016]), assim, a célula passa a ter dois portões, o *reset gate* e o *update gate*. Além disso, as células do tipo GRU não possuem um estado S associado, como nas células LSTM. O fluxo de informação dentro de uma célula do tipo GRU é esquematizado na Figura 3.25.

Dentro de uma célula do tipo GRU, o *reset gate* é responsável por decidir o quanto da informação do instante anterior deve persistir para a geração dos valores candidatos na

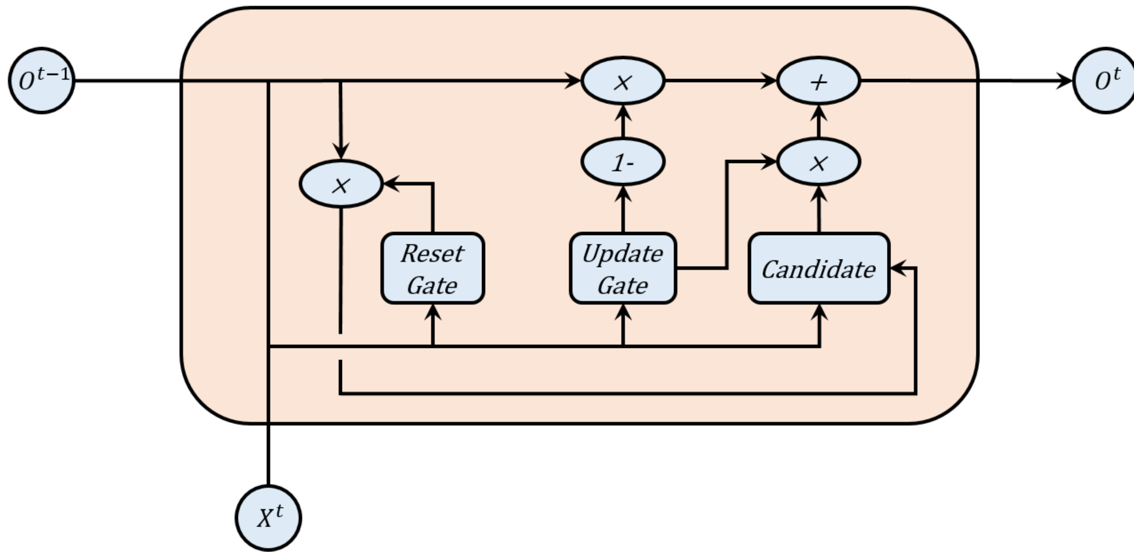


Figura 3.25. Representação de uma célula GRU, onde X^t representa o vetor de entrada no instante t e O^t o vetor com a saída produzida pela célula no instante t.

camada *candidate*. De maneira similar aos portões da célula LSTM, o *reset gate* (r^t) é dado por:

$$r^t = \sigma(\sum_j W_j^r O_j^{t-1} + \sum_j U_j^r X_j^t + b^r)$$

Dessa forma, utilizando o *reset gate*, os valores candidatos (C^t) podem ser obtidos por:

$$C^t = \tanh(\sum_j W_j^c r^t O_j^{t-1} + \sum_j U_j^c X_j^t + b^c)$$

Por sua vez, o *update gate*, é responsável por decidir qual parte da saída anterior vai persistir e qual vai ser atualizada. De maneira análoga ao *reset gate*, o *update gate* (q^t) é dado por:

$$q^t = \sigma(\sum_j W_j^q O_j^{t-1} + \sum_j U_j^q X_j^t + b^q)$$

Utilizando o *update gate* para parametrizar os valores candidatos e o complemento do *update gate* para parametrizar os valores do instante anterior, é possível obter a nova saída O^t através de:

$$O^t = q^t \cdot C^t + (1 - q^t) \cdot O^{t-1}$$

3.6.1. Implementando uma rede LSTM utilizando a base de dados YouTube8M

O YouTube8M [Abu-El-Haija et al. 2016] é um *dataset* de vídeo multi-etiquetado que contém cerca de 6.1 milhões de vídeos do Youtube[12] e um vocabulário com 3820 etiquetas. Para facilitar o desenvolvimento de projetos voltados ao entendimento de vídeo, o Youtube8M fornece as *features* áudio-visuais pré-computadas. As *features* visuais foram extraídas da última camada da rede Inception-Resnet-v2 [Szegedy et al. 2017]

[12] https://www.youtube.com/

pré-treinada com o *dataset* ImageNet [Deng et al. 2009]. Já as *features* de áudio foram extraídas da última camada de versão da rede VGG modificada para áudio [Hershey et al. 2017] pré-treinada com o *dataset* AudioSet [Gemmeke et al. 2017].

Após a extração das *features*, foi aplicado PCA e uma quantização de 8-bits sobre os vetores de *features* para redução de dimensionalidade, resultando em um vetor de 1024 dimensões de *features* visuais e um vetor de 128 dimensões de *features* de áudio. O *dataset* YouTube8M é fornecido em duas versões:

- *Frame-level* (1.71 TB): *features* áudio-visuais extraídas dos primeiros 360 segundos (6 minutos) do vídeo, a uma taxa de 1 *frame* por segundo.
- *Video-level* (31 GB): *features* áudio-visuais extraídas da média do RGB dos *frames* e áudio.

Para efeitos didáticos, apenas uma pequena parte do *dataset* em *frame-level* será utilizado e está disponível no Drive[13] dos autores juntamente com o vocabulário das etiquetas. Com o objetivo de classificar etiquetas, em um vídeo, a arquitetura geral do sistema utilizado é apresentado na Figura 3.26. Nesta arquitetura foi desenvolvido uma rede do tipo LSTM para agregar as *features* áudio-visuais e produzir uma lista de etiquetas para o vídeo. É importante notar que a utilização do *dataset* em *frame-level* é importante para o treinamento deste tipo de rede, uma vez que consiste de uma sequência de informações para cada vídeo, diferente do *dataset* em *video-level*.

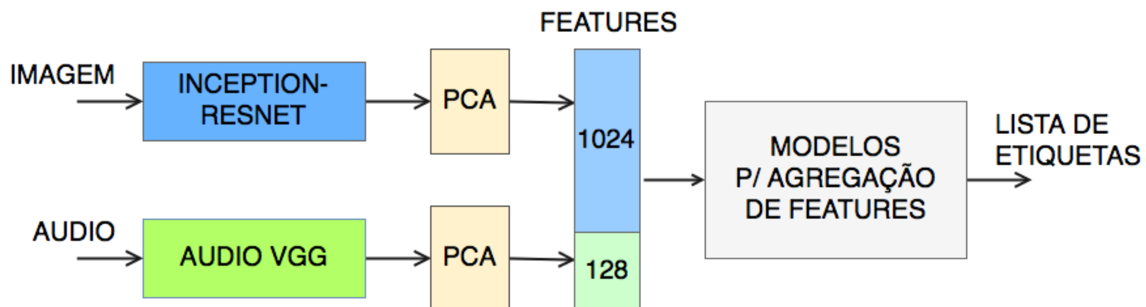


Figura 3.26. Arquitetura geral do sistema de classificação.

A Listagem 3.12 mostra a construção de uma LSTM. A função "build_lstm" recebe como parâmetro o número de camadas que a rede vai ter, o número de células que as camadas LSTM vão ter, o numero de *features* em cada *frame* do vídeo, bem como o número de etiquetas. Entre as linhas 3 e 8, são definidos os *placeholders* da LSTM, onde "X_" vai armazenar a matriz de *features* do vídeo (uma linha por *frame*), "Y_" é o vetor contendo as etiquetas do vídeo e "N_" o número de *frames* do vídeo. Na linha 10 e 11 as camadas de LSTM são construídas e empilhadas na linha 12. Na linha 16 essas camadas são então utilizadas para construir uma RNN desenrolando a matriz de *features*. Por fim o resultado da RNN é conectado a uma camada densa, produzindo um resultado equivalente a multiplicar a saída por pesos e adicionar um *bias*. Na linha 23 a função

[13]https://drive.google.com/open?id=14qkt9y2adFNPg16xpXG-lHoYLZTS_5iL

sigmoid é aplicada ao *logits* para obter a predição final da probabilidade de cada etiqueta estar no vídeo. O restante do código possui as definições da função de custo, otimizador já explicados nos exemplos anteriores deste capítulo.

Listagem 3.12. Construindo uma LSTM.

```
1 def build_lstm(n_layers, n_hidden_cells, n_features, n_labels):
2
3     # matriz de features:
4     X_ = tf.placeholder(dtype=tf.float32, shape=[None, n_features])
5     # vetor de etiquetas:
6     Y_ = tf.placeholder(dtype=tf.float32, shape=[n_labels])
7     # numero de frames:
8     N_ = tf.placeholder(dtype=tf.float32, shape=None)
9
10    lstm_layers = [tf.contrib.rnn.BasicLSTMCell(n_hidden_cells,
11    forget_bias=1.0) for _ in range(n_layers)]
12    lstm_stack = tf.contrib.rnn.MultiRNNCell(lstm_layers)
13
14    model_input = tf.expand_dims(X_, 0)
15    # rnn dinamico para batchs de tamanho variavel
16    outputs, state = tf.nn.dynamic_rnn(lstm_stack, model_input,
17    sequence_length=N_, dtype=tf.float32)
18
19    logits = tf.layers.dense(tf.layers.batch_normalization(outputs[:, -1, :]),
20    n_labels, activation=None,
21 kernel_initializer=tf.contrib.layers.xavier_initializer())
22
23    output = tf.nn.sigmoid(logits)
24
25    # loss function
26    loss = tf.reduce_mean(
27    tf.nn.sigmoid_cross_entropy_with_logits(
28    logits=logits, labels=tf.expand_dims(Y_, 0)))
29
30    # optimizer
31    opt = tf.train.AdamOptimizer(learning_rate=0.001).minimize(loss)
32
33    return opt, loss, output, X_, Y_, N_
```

A Listagem 3.13 mostra o código que carrega o *dataset*, inicia o *TensorFlow* e carrega a LSTM definida na listagem anterior. Na Linha 2 o *dataset* é carregado. Na Linha 8 e 9 o modelo é carregado, utilizando 2 camadas de LSTM com 512 células escondidas cada, o número de *features* é $1024 + 128$, correspondendo respectivamente as *features* visuais e de áudio, o número de etiquetas do *dataset*. Na linha 12 as variáveis do TensorFlow são inicializadas.

Listagem 3.13. Iniciando o TensorFlow e carregando o *dataset* e a LSTM.

```
1 # Carregando o dataset
2 load_dataset()
3
4 # Iniciando
5 with tf.Session() as sess:
6
7     # carregando o modelo LSTM
8     opt, cost, output, X_, Y_, N_ = build_lstm(n_layers=2,
9     n_hidden_cells=512, n_features=1024+128, n_labels=3862)
10
11    # inicializando as variaveis do tensorflow
12    tf.global_variables_initializer().run()
```

A Listagem 3.14 mostra o código que realiza o treinamento da LSTM com o *dataset*. Neste exemplo de implementação o modelo é treinado em 5 épocas. Em cada época, como definido nas linha 8, um *batch* é lido. Em seguida, a rede é treinada com o *batch*. O erro do treinamento é impresso a cada época.

Listagem 3.14. Treinando a LSTM.

```
1 #definindo numero de epocas
2 num_epochs = 5
3
4 for epoch in range(num_epochs):
5     last_cost = 0
6     #treinando a rede com cada batch
7     while True:
8         features, labels, n_frames = get_next_train_batch()
9         # se n_frames < 0, significa que os batchs acabaram
10        # e devemos comecar uma nova epoca
11        if n_frames < 0:
12            break
13
14        _, last_cost = sess.run([opt, cost],
15        feed_dict={X_: features, Y_: labels, N_: n_frames})
16
17        #imprimindo o erro a cada epoca
18    print("Erro na epoca", epoch, ":", last_cost)
19
20 ------------------------------ OUTPUT ------------------------------
21 > Erro na epoca 0 : 0.0039477115
22 > Erro na epoca 1 : 0.0037346634
23 > Erro na epoca 2 : 0.00365565
24 > Erro na epoca 3 : 0.0036228024
25 > Erro na epoca 4 : 0.0035409592
```

A Listagem 3.15 mostra como utilizar a LSTM recém treinada para realizar classificações multi-etiquetas nos vídeos. Na linha 3 é feita a predição das etiquetas para o

vídeo de entrada. Na linha 5 são obtidos as 5 etiquetas com maior probabilidade. Por fim, a linha 8 mostra a saída do programa.

Listagem 3.15. Usando a LSTM para obter as 5 primeiras *labels* de um vídeo.

```
1 features, labels, n_frames = get_test_example()
2
3 result = sess.run(output, feed_dict={X_: features, Y_: labels, N_: n_frames})
4
5 result_vocabulary = get_top_labels(5, result)
6 print("As 5 primeiras labels sao: ", result_vocabulary)
7 ------------------------------------ OUTPUT ------------------------------------
8 > As 5 primeiras labels sao:  ['Game', 'Cartoon', 'Food', 'Animation',
9 'Slam dunk']
```

3.7. Reconhecimento Facial

O Modelo FaceNet [Schroff et al. 2015] é o estado-da-arte para a tarefa de reconhecimento facial. Este modelo tem um desempenho 99.6% de acurácia no *dataset* LFW (Labeled Faces in the Wild) [Learned-Miller 2014]. A Figura 3.27 ilustra a arquitetura usada pelo FaceNet para realizar o reconhecimento facial. Dada duas imagens de face, o FaceNet extrai um vetor de *features* (*face embedding*) a partir da ativação linear da última camada densa da rede Inception-Resnet-V1 [Szegedy et al. 2017]. Em seguida, a similiraridade entre as imagens de face é calculada pela distancia euclidiana entre os seus vetores de *features*, se a distância for menor que um limiar, então presume-se que as imagens de faces sejam da mesma pessoa.

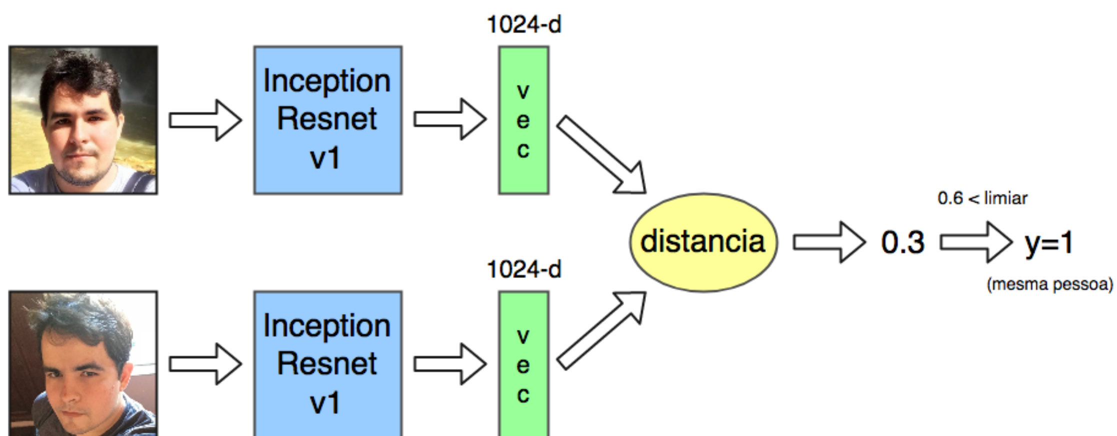


Figura 3.27. Arquitetura usada pelo FaceNet para reconhecimento facial.

Neste minicurso o modelo FaceNet é usado para reconhecimento facial em conjuntos de imagens de faces. Em um sistema empresarial para controle de acesso de funcionários, por exemplo, a câmera de segurança pode capturar a imagem da face do visitante, então o sistema pode verificar se o visitante está registrado como funcionário da empresa para liberar o acesso. Dessa forma, o sistema pode utilizar o FaceNet para extrair o *face embedding* da imagem do visitante, e em seguida, calcular a distância com

os *face embeddings* das imagens dos demais funcionários registrados, como ilustrado na Figura 3.28. A menor distância obtida corresponde a identidade do visitante, se a distância for maior que um limiar m, significa que o visitante não está registrado na base de funcionários.

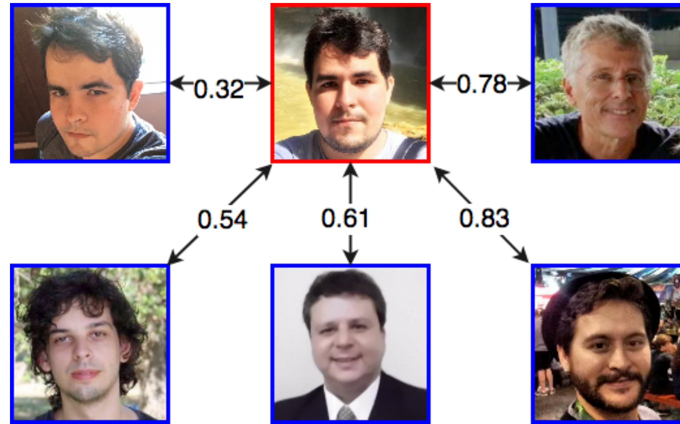


Figura 3.28. Comparação entre as distâncias das imagens do banco de dados (em azul) em relação a imagem de entrada (em vermelho).

A Sub-seção 3.7.1 descreve a função de perda utilizada para treinamento do FaceNet. Em seguida, a Sub-seção 3.7.2 apresenta um cenário de uso onde um modelo pré-treinado do FaceNet é usado na implementação de um sistema de reconhecimento facial.

3.7.1. Função de perda Tríplice

Dada uma imagem x, seu vetor de features faciais é dado por $f(x)$, onde f é a computação realizada pela CNN. A função de perda Trípicle usa um trio de imagens (A,P,N), onde:

- A (âncora) refere-se a uma imagem de face;
- P (positiva) refere-se a imagem de face do mesmo individuo da imagem A;
- N (negativa) refere-se a uma imagem de face de um individuo diferente da imagem A.

O objetivo é fazer a imagem $A^{(i)}$ ficar próxima da sua $P^{(i)}$ e mais distante da $N^{(i)}$ por uma margem α. Dessa forma, a função de perda Tríplice é descrita como:

$$\mathscr{J} = \sum_{i=1}^{m} [|| f(A^{(i)}) - f(P^{(i)}) ||_2^2 - || f(A^{(i)}) - f(N^{(i)}) ||_2^2 + \alpha]_+$$

A notação "$[z]_+$" corresponde a função $max(z,0)$. A contante α é usada para garantir que a rede não tente otimizar para $f(A) - f(P) = f(A) - f(N) = 0$. A Listagem 3.16 descreve a implementação da função tríplice no Tensorflow. Na linha 4 é calculado o primeiro termo da equação, que corresponde a distância entre os vetores de *features* das imagens A e P. Em seguida, na linha 7 é calculado o segundo termo, que corresponde a distância entre os vetores de *features* das imagens A e N. Na linha 10, é calculada a

diferença entre os dois termos anteriores e somado com a margem α. Por fim, na linha 13, todos os erros são somados, ressaltando que os erros menores que 0 são transformados em 0 pela função *max*.

Listagem 3.16. Implementação da função de perda tríplice.

```
1 def triplet_loss(anchor, positive, negative, alpha = 0.2):
2
3     #Passo 1: Calculo da distancia entre A e P
4     pos_dist = tf.reduce_sum(tf.square(tf.subtract(anchor,positive)), axis=-1)
5
6     #Passo 2: Calculo da distancia entre A e N
7     neg_dist = tf.reduce_sum(tf.square(tf.subtract(anchor,negative)), axis=-1)
8
9     #Passo 3: Subtracao dos dois termos anteriores e depois soma com alpha.
10    basic_loss = tf.add(tf.subtract(pos_dist, neg_dist), alpha)
11
12    # Step 4: Somatorio do max entre o basic_loss e 0
13    loss = tf.reduce_sum(tf.maximum(basic_loss, 0.0))
14
15    return loss
```

3.7.2. Cenário de Uso: Sistema de Controle de Acesso com Reconhecimento Facial

A tarefa de reconhecimento facial tenta responder a pergunta "Quem é essa pessoa?". Neste cenário de uso implementamos um sistema de controle de acesso que usa o FaceNet para realizar o reconhecimento facial dos funcionários de uma empresa, e então, permitir seu acesso.

O modelo FaceNet utilizado nesta implementação está disponível para download na sua página oficial no GitHub[14]. Adicionalmente, é possível utilizar modelos pré-treinados no FaceNet. Na página oficial há dois modelos pré-treinados disponíveis, um treinado no dataset CASIA-WebFace[15], e outro treinado no dataset VGGFace2[16].

A Listagem 3.17 mostra como o sistema de reconhecimento utiliza um modelo pré-treinado do FaceNet. Na linha 6 é feito o carregamento do modelo. Em seguida, nas linhas 9-11, são selecionados os tensores necessários para uso do modelo. Na linha 13 é definida a função que usa o FaceNet para extrair o *embedding* de uma imagem facial. Entre as linhas 16-18 é feito o redimensionamento da imagem para garantir a compatibilidade com as dimensões do tensor de entrada. Por fim, na linha 19 são atribuídos os valores dos tensores de entrada (placeholders) do modelo, e na linha 20, o *embedding* da imagem é obtido com a execução do FaceNet.

Listagem 3.17. Usando o FaceNet pré-treinado.

```
1 import facenet
2
```

[14]https://github.com/davidsandberg/facenet
[15]https://drive.google.com/open?id=1R77HmFADxe87GmoLwzfgMu_HY0IhcyBz
[16]https://drive.google.com/open?id=1EXPBSXwTaqrSC0OhUdXNmKSh9qJUQ55-

```
3 sess = tf.Session()
4
5 #Carregando do modelo pre-treinado
6 facenet.load_model("20170512-110547/20170512-110547.pb")
7
8 #Selecionando os tensores necessarios para execucao
9 images_placeholder = tf.get_default_graph().get_tensor_by_name("input:0")
10 embeddings = tf.get_default_graph().get_tensor_by_name("embeddings:0")
11 train_placeholder = tf.get_default_graph().get_tensor_by_name("phase_train:0")
12
13 def get_embedding(img):
14     img_size = 160
15     #Preparando a imagem de entrada
16     resized = cv2.resize(img, (img_size,img_size),interpolation=cv2.INTER_CUBIC)
17     reshaped = resized.reshape(-1,img_size, img_size,3)
18     #Configurando entrada e execucao do FaceNet
19     feed_dict = {images_placeholder: reshaped, train_placeholder: False}
20     embedding = sess.run(embeddings, feed_dict=feed_dict)
21     return embedding
```

Como primeira etapa da implementação do sistema, é necessário realizar o registro dos *embeddings* das imagens faciais dos funcionários. As linhas 2-6 da Listagem 3.18 mostram como usar a função descrita anteriormente para registrar os *embeddings*. Em seguida, nas linhas 9 é definida a função que calcula a distância euclidiana entre dois vetores. Essa função é utilizada para calcular a similaridade entre os *embeddings* das imagens.

Listagem 3.18. Usando o FaceNet para extrair os embeddings das imagens.

```
1 #Registrando os embeddings das pessoas
2 database = {}
3 database["antonio"] = get_embedding("faces/antonio.png")
4 database["lucas"] = get_embedding("faces/lucas.png")
5 database["gabriel"] = get_embedding("faces/gabriel.png")
6 database["sergio"] = get_embedding("faces/sergio.png")
7
8 #Funcao que calcula a distancia euclidiana entre dois vetores
9 def distance(vector1, vector2):
10         return np.sqrt(np.sum((vector1-vector2)**2))
```

Na segunda etapa, é implementada a função que identifica a foto do visitante a partir das fotos dos funcionários registrados. Como pode ser visto na Listagem 3.19, a função "who_is_it" recebe como parâmetro o caminho da imagem facial do visitante e a lista com os *embeddings* dos funcionários registrados. Na linha 6 é calculado o *embedding* da imagem de entrada. Em seguida, nas linhas 9-13 um laço percorre a lista de funcionários registrados, calculando a distância entre o *embedding* do visitante e cada funcionário. Por fim, nas linhas 15-18, é verificado se a menor distância encontrada é

maior que um limiar de 0.7, se sim, significa que a imagem de entrada é da face de uma pessoa não registrada, caso contrário, a identidade é de um funcionário é impressa.

Listagem 3.19. Função que identifica uma imagem de face.

```
1  def who_is_it(visitor_image_path, database):
2
3      min_dist = 1000
4      identity = -1
5
6      #Calculando o embedding do visitante
7      visitor = get_embedding(visitor_image_path)
8      #Calculando a distacia do visitante com os demais funcionarios
9      for index, employee in enumerate(database):
10         dist = distance(visitor, employee)
11         if dist < min_dist:
12                 min_dist = dist
13                 identity = index
14     #verificando a identidade
15     if min_dist > 0.7:
16         print("Essa pessoa nao esta cadastrada, soltem os caes!")
17     else:
18         print ("Bem vindo(a)",identity ,"!")
```

A Listagem 3.20 exemplifica o uso da função de reconhecimento facial. Na linha 1 foi colocada a imagem facial de uma pessoa que não foi registrada. Em seguida, na linha 2, foi colocada outra imagem facial de uma pessoa registrada. As linhas 5-6 mostram a saída do programa.

Listagem 3.20. Exemplo de uso da função de reconhecimento.

```
1 who_is_it("faces/carlos.png",database)
2 who_is_it("faces/antonio2.png",database)
3
4  ---------------------------------- OUTPUT ----------------------------------
5 > Essa pessoa nao esta cadastrada, soltem os caes!
6 > Bem vindo(a) antonio !
```

3.8. 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] obeteve 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[17].

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

A Subseção 3.8.1 descreve a arquitetura geral do YOLO. Em seguida, a Subseção 3.8.2 descreve a implementação de um cenário de uso que usa o YOLO para identificar objetos em imagens.

3.8.1. 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 Interseçao sobre União (explicada na Subseção seguinte) entre a BB predita e a BB do *ground truth*, $Pr(Object) * IOU^{truth}_{pred}$.

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 as 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:

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

.

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 3.29, 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 3.30 (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.

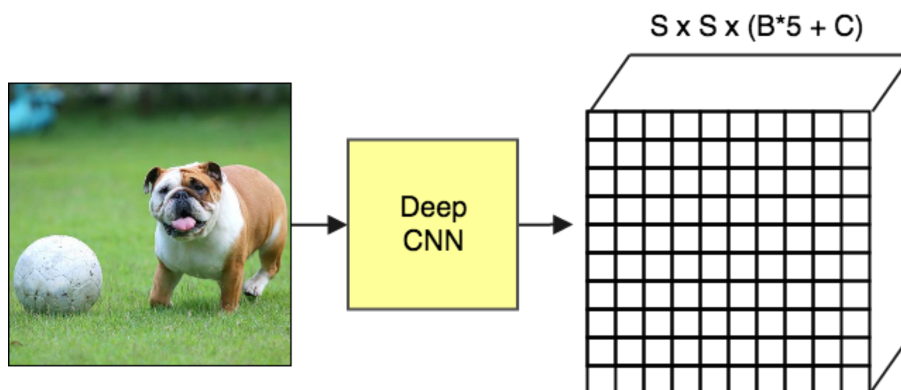
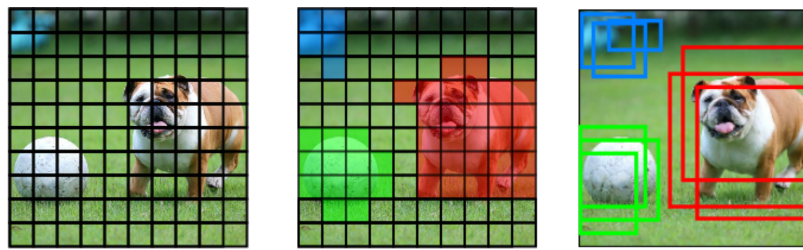


Figura 3.29. Esquema arquitetural do YOLO.



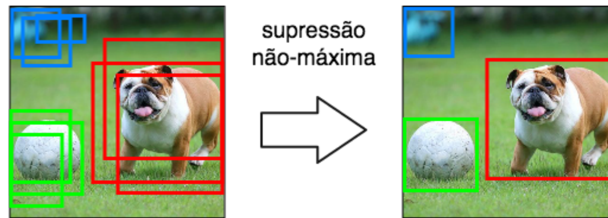


Figura 3.30. (cima) Visualização da saída do YOLO. (baixo) Remoção das BBs sobrepostas com o filtro de supressão não-máxima.

3.8.1.1. Supressão não-máxima

Mesmo com a filtragem pelo *score*, muitas BBs podem ficar sobrepostas uma as outras, como ilustrado na Figura 3.30 (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 3.31, essa técnica consiste basicamente em dividir a interseção pela união de duas BBs. A Listagem 3.21 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.

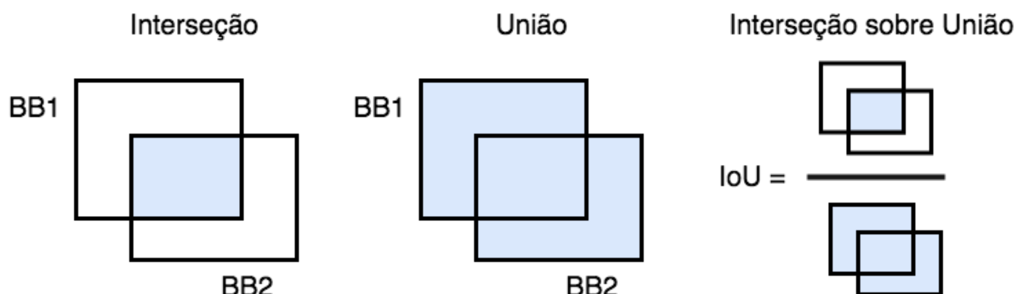


Figura 3.31. Visualização da operação de IoU.

Listagem 3.21. Função de cálculo do IoU.

```
def iou(box1, box2):
    #Calculando a intersecao entre as BBs
    xi1 = max(box1[0], box2[0])
    yi1 = max(box1[1], box2[1])
    xi2 = min(box1[2], box2[2])
    yi2 = min(box1[3], box2[3])
    inter_area = (xi2 - xi1)*(yi2 - yi1)

    #Calculando a uniao usando a formula: Union(A,B) = A + B - Inter(A,B)
    box1_area = (box1[2] - box1[0])*(box1[3] - box1[1])
    box2_area = (box2[2] - box2[0])*(box2[3] - box2[1])
    union_area = box1_area + box2_area - inter_area

    # Calculando o IoU
    iou = inter_area / union_area

    return iou
```

3.8.1.2. 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 preditar 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.

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

3.8.2. Cenário de Uso: Sistema de Detecção de Objetos

A forma mais fácil para usar o modelo YOLO é importando o *framework* Darkflow (Tensorflow + Darknet[18]). O *framework* Darknet é escrito em C e CUDA[19] 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[20]. E em seguida, realizar a instalação do Darknet usando o Pip. Vale ressaltar que é necessário ter o pacote Cython[21] instalado.

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

Após realizar a instalação, para utilizar o pacote basta importa-lo para o projeto, como mostra a Listagem 3.22. 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[22] 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

[18] https://pjreddie.com/darknet/
[19] https://developer.nvidia.com/cuda-zone
[20] https://github.com/thtrieu/darkflow
[21] http://cython.org/
[22] 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 tambem 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 3.22. 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)
```

A Listagem 3.23 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 3.32). 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 3.23. 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()
8 ------------------------------------ OUTPUT ------------------------------------
9 [{'label': 'person', 'confidence': 0.15689932, 'topleft': {'x': 314, 'y': 82},
10 'bottomright': {'x': 379, 'y': 253}},
11 {'label': 'person', 'confidence': 0.7260812, 'topleft': {'x': 268, 'y': 64},
12 'bottomright': {'x': 367, 'y': 358}},
13 {'label': 'person', 'confidence': 0.7622834, 'topleft': {'x': 143, 'y': 82},
14 'bottomright': {'x': 235, 'y': 369}},
15 {'label': 'handbag', 'confidence': 0.2231428, 'topleft': {'x': 198, 'y': 178},
16 'bottomright': {'x': 233, 'y': 239}},
17 {'label': 'skis', 'confidence': 0.12105702, 'topleft': {'x': 313, 'y': 103},
18 'bottomright': {'x': 375, 'y': 271}},
19 {'label': 'snowboard', 'confidence': 0.2203493, 'topleft': {'x': 342, 'y': 159},
20 'bottomright': {'x': 371, 'y': 257}}]
```

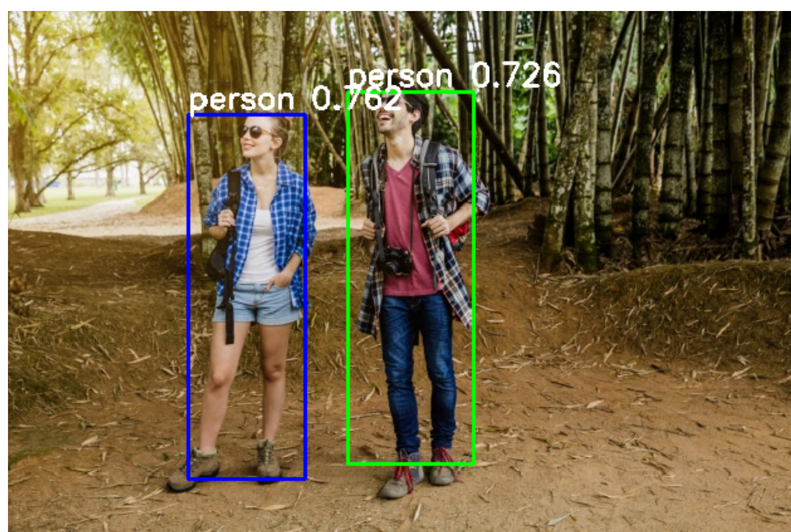


Figura 3.32. Imagem com as *bounding boxes* preditas pelo YOLO.

3.9. Conclusão

Este minicurso apresenta os fundamentos e tecnologias para desenvolver modelos de *Deep Learning*. Em especial, apresenta os modelos de redes neurais baseados em CNNs e LSTMs. Além disso, apresentamos técnicas e métodos de *Deep Learning* para resolução de problemas do domínio de sistemas multimídia. Esperamos que o participante do curso esteja apto para usar *Deep Learning* para realizar classificação de imagens, reconhecimento facial, detecção de objetos e classificação de cenas de vídeo.

Adicionalmente, o minicurso 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. As últimas versões da rede apresentada já ultrapassam as capacidades humanas (a acurácia humana no ImageNet está entre 5-10%).

Quanto ao planejamento do curso, foi apresentado um roteiro que 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 e LSTM. O minicurso é concluído com a apresentação de quatro projetos práticos, um de classificação de imagens com sinais de mão usando uma rede CNN, um de classificação de cenas de vídeo usando uma rede LSTM, um de reconhecimento facial usando o modelo FaceNet, e o último, de reconhecimento de objetos usando o modelo YOLO.

Referências

[Abu-El-Haija et al. 2016] Abu-El-Haija, S., Kothari, N., Lee, J., Natsev, P., Toderici, G., Varadarajan, B., and Vijayanarasimhan, S. (2016). Youtube-8m: A large-scale video classification benchmark. *arXiv preprint arXiv:1609.08675*.

[Cho et al. 2014] Cho, K., Van Merriënboer, B., Bahdanau, D., and Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. *arXiv*

preprint arXiv:1409.1259.

[Chung et al. 2014] Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.

[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.

[Equadex 2018] Equadex (2018). Helpicto. Accessed: 2018-05-18.

[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.

[Goodfellow et al. 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

[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)*.

[Kearn and Beeby 2017] Kearn, M. and Beeby, M. (2017). Using cognitive services to make museum exhibits more compelling and track user behavior. Accessed: 2018-05-18.

[Learned-Miller 2014] Learned-Miller, G. B. H. E. (2014). Labeled faces in the wild: Updates and new reporting procedures. Technical Report UM-CS-2014-003, University of Massachusetts, Amherst.

[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.

[Ota et al. 2017] Ota, K., Dao, M. S., Mezaris, V., and De Natale, F. G. (2017). Deep learning for mobile multimedia: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 13(3s):34.

[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.

[Rumelhart et al. 1986] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, 323(6088):533.

[Samuel 1959] Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229.

[Schroff et al. 2015] Schroff, F., Kalenichenko, D., and Philbin, J. (2015). Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823.

[Sussillo 2014] Sussillo, D. (2014). Random walks: Training very deep nonlin-ear feed-forward networks with smart ini.

[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.

[Thomas and Sadagopan 2016] Thomas, Janki Vora, J. W. and Sadagopan, S. (2016). Use cases for industry cognitive solutions. Accessed: 2018-05-18.

BIO

Antonio José Grandson Busson. Possui graduação (2012) e mestrado (2015) em Ciência da Computação pela Universidade Federal do Maranhão. Atualmente é doutorando em Informática pela Pontifícia Universidade Católica do Rio de Janeiro. Seus interesses de pesquisa incluem: sistemas multimídia/hipermídia, modelos de hiperdocumentos, reconhecimento de padrões e sistemas de TV Digital. Currículo Lattes: `http://lattes.cnpq.br/1857348479447184`.

Lucas Caracas de Figueiredo. Possui graduação (2014) em Ciência da Computação pela Universidade Federal do Maranhão e mestrado (2017) em Informática pela Pontifícia Universidade Católica do Rio de Janeiro. Atualmente é doutorando em Informática pela Pontifícia Universidade Católica do Rio de Janeiro e realiza pesquisa em parceria com o Instituto Tecgraf. Seus interesses em pesquisa incluem: processamento de imagens, sensoriamento remoto, LIDAR, computação gráfica e geometria computacional. Currículo Lattes: `http://lattes.cnpq.br/5353884964040661`.

André Luiz de Brandão Damasceno. Possui graduação (2012) e mestrado (2015) em Ciência da Computação pela Universidade Federal do Maranhão. Atualmente é doutorando em Informática pela Pontifícia Universidade Católica do Rio de Janeiro. Seus interesses de pesquisa incluem: sistemas multimídia, ciência de dados e análise visual. Currículo Lattes: `http://lattes.cnpq.br/0969337931297570`.

Gabriel Noronha Pereira dos Santos. É graduando em Engenharia de Eletricidade na faculdade Wyden. Atualmente trabalha como cientista de dados na Startup Niddu, onde desenvolve projetos de aprendizagem de máquina aplicada a microlearning e sistemas de recomendação. Currículo Lattes: `http://lattes.cnpq.br/8088572506239597`.

Sérgio Colcher. É professor do quadro principal do Departamento de Informática (DI) da PUC- Rio desde 2001 e coordenador do laboratório TeleMídia. Obteve os títulos de Engenheiro de Computação (1991), Mestre em Ciências em Informática (1993) e Doutor em Ciências em Informática (1999), todos pela PUC-Rio, além do Pós-Doutorado (2003) no ISIMA (Institute Supérieur D'Informatique et de Modelisation des Applications — Université Blaise Pascal, Clermont Ferrand, França). Trabalhou no Centro Científico da IBM - Rio e na divisão de desenvolvimento de hardware da COBRA (Computadores Brasileiros S/A). Foi também professor dos cursos de MBA em Gerência de Telecomunicações e MBA em e-Business da Fundação Getúlio Vargas. Suas áreas de interesse incluem redes de computadores, análise de desempenho de sistemas computacionais, sistemas multimídia/hipermídia e sistemas de TV digital. Currículo Lattes: `http://lattes.cnpq.br/1104157433492666`.

Ruy Luiz Milidiú. Bacharel em Matemática pela Universidade Federal do Rio de Janeiro (1974), Mestre em Matemática Aplicada pela Universidade Federal do Rio de Janeiro (1978), M.Sc. em Operations Research - University of California (1983) e Ph.D. em Pesquisa Operacional - University of California (1985). Atualmente, é professor associado da Pontifícia Universidade Católica do Rio de Janeiro e também consultor ad hoc do CNPq, da CAPES, da FAPERJ, da FAPESP e da FINEP. Tem experiência na área de Ciência da Computação, com ênfase em Algorítmica, Aprendizado de Máquina e Complexidade de Computação. Currículo Lattes: `http://lattes.cnpq.br/6918010504362643`.