[PSI3471 aula 9 parte 2. Início]

Aprendizado de máquina (classificar imagens, extrair características)

[É necessário explicar mostrando o conteúdo de variáveis intermediárias.]

Os programas usados nesta aula podem ser baixados em Linux com os comandos:

```
$ wget -U 'Firefox/50.0' http://www.lps.usp.br/hae/apostila/classif.zip
$ unzip classif
```

A biblioteca de funções *procimagem.h* deve estar localizada no mesmo diretório do seu programa OpenCV/C++.

0. Introdução

Na última aula, usamos o aprendizado de máquina clássica para projetar filtros. Nesta aula, vamos estudar alguns exemplos de aprendizado de máquina para classificar imagens simples. Vamos resolver esses problemas sem usar deep learning (que veremos a partir da próxima aula). Depois, vamos estudar algumas técnicas para extrair atributos de imagens, estudando detecção de pedestres e faces humanas.

Vamos acrescentar no arquivo *procimagem.h* as funções *tempo*, *argMax*, *xdebug* e a classe *MNIST* que usaremos nesta aula.

A função *tempo* devolve os segundos que se passaram desde *epoch* com casas decimais.

Nota: *Epoch* é normalmente tempo que passou desde “00:00:00 Coordinated Universal Time (UTC), 1 January 1970”.

Exemplo de uso:

```
//tempo.cpp
#include "procimagem.h"
int main() {
  double t1=tempo();
  this_thread::sleep_for(1s); //Troque por algum processamento
  double t2=tempo();
  printf("Tempo decorrido em segundos=%f\n",t2-t1);
}

Saída:
Tempo decorrido em segundos=1.000238
```

A função *argMax* devolve o índice do maior elemento de um *Mat_* considerado como um vetor unidimensional. Exemplo de uso:

```
//argmax.cpp
#include "procimagem.h"
int main() {
  Mat_<float> a(2,2);
  a << 6, 3, 9, 4;
  printf("Indice do maior elemento=%d\n",argMax(a));
}

Saída:
Indice do maior elemento=2
```

O macro *xdebug* facilita debugar o programa, imprimindo o nome do arquivo e o número da linha onde o macro está localizado. Exemplo de uso:

```
 1 //xdebug.cpp
 2 #include "procimagem.h"
 3 int main() {
 4   xdebug;
 5   Mat_<float> a(2,2);
 6   xdebug;
 7   a << 6, 3, 9, 4;
 8   xdebug;
 9   printf("Indice=%d\n",argMax(a));
10   xdebug;
11 }
```

```
File=xdebug.cpp line=4

File=xdebug.cpp line=6

File=xdebug.cpp line=8
Indice=2
File=xdebug.cpp line=10
```

1. MNIST

Muitos cursos sobre aprendizado de máquina em visão computacional começam apresentando o exemplo de classificação de dígitos manuscritos da base de dados MNIST. Faremos o mesmo. Figura 1 apresenta algumas imagens desse conjunto.

Na verdade, essa imagem mostra as imagens negativas (isto é, onde preto e branco foram trocados) pois as imagens originais têm letras brancas sobre o fundo preto.

O banco de dados MNIST de dígitos manuscritos possui 60.000 imagens de treino e 10.000 imagens de teste, todas com 28×28 pixels, juntamente com os rótulos verdadeiros (classificações das imagens em 0, 1, ..., 9). As imagens são em níveis de cinza (0 a 255) e estão centralizadas de forma que o centro de massa coincida com o centro da imagem.

Tabela 1 apresenta as taxas de erro de classificação de alguns métodos. A taxa de erro típico de um ser humano é algo entre 2% e 2,5%, segundo https://papers.nips.cc/paper/656-efficient-pattern-recognition-using-a-new-transformation-distance.pdf

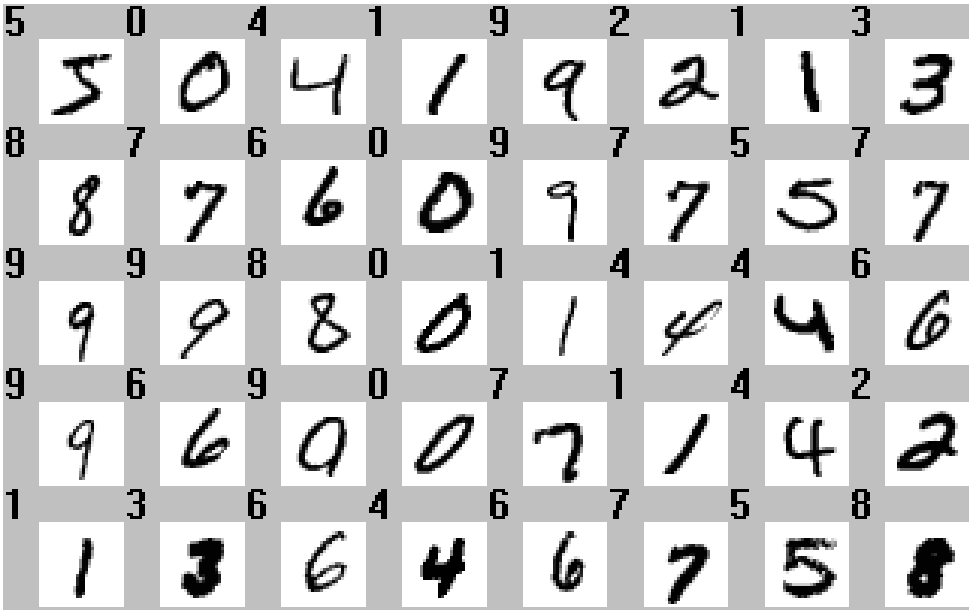


Figura 1: Alguns dígitos da base MNIST com os rótulos, com cores invertidas (as imagens originais possuem dígitos brancos sobre fundo preto).

Esse banco de dados podia ser baixada de:
http://yann.lecun.com/exdb/mnist/

O mesmo conjunto de imagens pode ser baixado de (clique em “Download Raw Dataset”):
https://git-disl.github.io/GTDLBench/datasets/mnist_datasets/
Ou pode baixar de Kaggle, mas é preciso criar uma conta Kaggle:
https://www.kaggle.com/datasets/hojjatk/mnist-dataset

Quem instalou Cekeikon, uma cópia desse conjunto está em:

```
(...)/cekeikon5/tiny_dnn/data
```

Em Python/Keras, este conjunto pode ser carregado com os comandos:

```
import tensorflow.keras as keras
mnist = keras.datasets.mnist
(AX, AY),(QX, QY) = mnist.load_data()
```

Em Python/PyTorch, o comando para carregar MNIST é:

```
A = datasets.MNIST(root="nome_diret", train=True, transform=None, download=True)
Q = datasets.MNIST(root="nome_diret", train=False, transform=None, download=True)
AX=A.data; AY=A.targets
QX=Q.data; QY=Q.targets
```

Tabela 1: As taxas de erro de alguns métodos quando classifica MNIST (retirado de http://yann.lecun.com/exdb/mnist/).

CLASSIFIER	PREPROCESSING	TEST ERROR RATE (%)	Reference
Linear Classifiers			
linear classifier (1-layer NN)	none	12.0	LeCun et al. 1998
linear classifier (1-layer NN)	deskewing	8.4	LeCun et al. 1998
K-Nearest Neighbors			
K-nearest-neighbors, Euclidean (L2)	none	3.09	Kenneth Wilder, U. Chicago
K-NN with non-linear deformation (P2DHMDM)	shiftable edges	0.52	Keysers et al. IEEE PAMI 2007
Boosted Stumps			
boosted stumps	none	7.7	Kegl et al., ICML 2009
product of stumps on Haar f.	Haar features	0.87	Kegl et al., ICML 2009
Non-Linear Classifiers			
40 PCA + quadratic classifier	none	3.3	LeCun et al. 1998
1000 RBF + linear classifier	none	3.6	LeCun et al. 1998
SVMs			
SVM, Gaussian Kernel	none	1.4	
Virtual SVM, deg-9 poly, 2-pixel jittered	deskewing	0.56	DeCoste and Scholkopf, MLJ 2002
Neural Nets			
2-layer NN, 300 hidden units, mean square error	none	4.7	LeCun et al. 1998
6-layer NN 784-2500-2000-1500-1000-500-10 (on GPU) [elastic distortions]	none	0.35	Ciresan et al. Neural Computation 10, 2010 and arXiv 1003.0358, 2010
Convolutional nets			
Convolutional net LeNet-1	subsampling to 16x16 pixels	1.7	LeCun et al. 1998
Convolutional net LeNet-4	none	1.1	LeCun et al. 1998
committee of 35 conv. net, 1-20-P-40-P-150-10 [elastic distortions]	width normalization	0.23	Ciresan et al. CVPR 2012

1.1 Diminuir a dimensão dos atributos

O número de pixels das imagens (28×28=784) é grande demais para a maioria dos algoritmos de aprendizado clássicos. Seria bom diminuir o número de atributos, pois isso facilitaria o aprendizado.

Como vimos, aumentando atributos aumenta a quantidade de amostras de treino necessárias, aumenta o tempo de processamento e diminui a taxa de acerto, fenômeno conhecido como a "maldição da dimensionalidade" [Shetty2022, wiki-curse, Raj2021, Yiu2019].

Há um bom artigo que fornece uma explicação matemática (mas intuitiva) da "maldição de dimensionalidade" e por que o aprendizado de máquina não funciona bem em alta dimensão:

https://towardsdatascience.com/the-math-behind-the-curse-of-dimensionality-cf8780307d74

Alguns argumentos desse artigo são:

1) Em alta dimensão, é necessário um número muito grande de dados de treino para preencher o espaço dos atributos e fazer aprendizado de máquina funcionar.

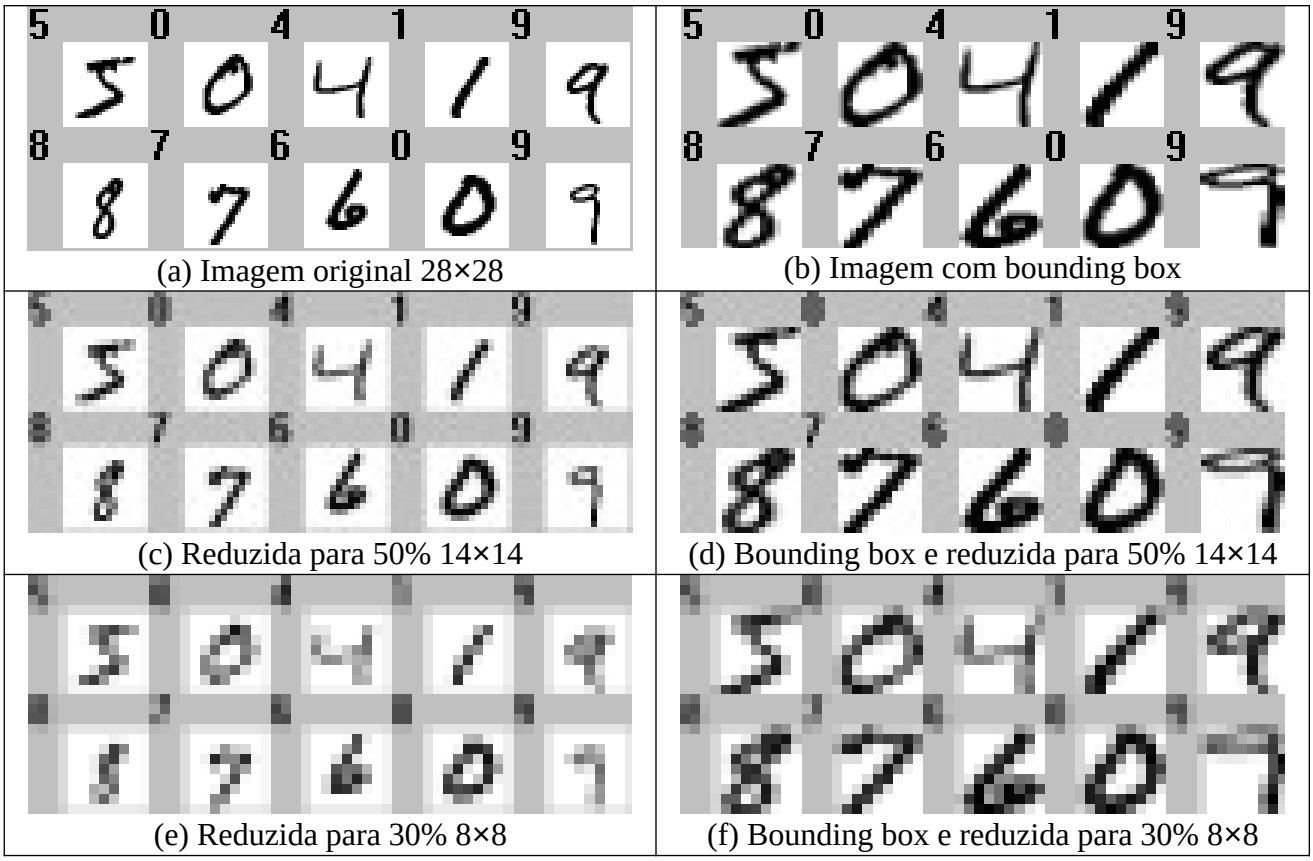
2) Em alta dimensão, a noção de distância perde sentido, fazendo com que algoritmos baseados em distância como *k*-NN deixem de funcionar.

Como poderia diminuir o número de atributos? A coluna "preprocessing" da tabela 1 lista algumas possibilidades, entre eles "Haar features" e "subsampling". Aqui, vamos considerar duas soluções simples e intuitivas:

a) Diminuir a resolução da imagem (é o mesmo que "subsampling" da tabela 1).

b) Calcular "bounding box", isto é, eliminar as linhas e colunas brancas em torno dos dígitos.

A intuição ao fazer isso é entregar ao algoritmo de aprendizado a menor quantidade de dados possível, mas onde ainda contenha informação suficiente que permita classificar os dígitos.



(a) Imagem original 28×28 (b) Imagem com bounding box

(c) Reduzida para 50% 14×14 (d) Bounding box e reduzida para 50% 14×14

(e) Reduzida para 30% 8×8 (f) Bounding box e reduzida para 30% 8×8

Figura 2: Dígitos de MNIST após sofrer reamostragem (coluna da esquerda) e eliminar linhas e colunas brancas (“bounding box”, coluna da direita).

Os dígitos originais (figura 2a) foram reduzidos para 50% (14×14 pixels) na figura 2c. Já não é possível reconhecer muito bem os dígitos “8” e “6”. Se calcular bounding box (figura 2d), é possível reconhecer bem todos os dígitos mesmo com redução para 50%. Se calcular bounding box mas reduzir resolução para 8×8 (figura 2f) fica difícil reconhecer vários dígitos. Assim, provavelmente, calcular bounding box e reduzir a resolução para 14×14 seria uma boa forma de diminuir a dimensão do espaço de entrada.

1.2 Leitura do MNIST

No arquivo *procimagem.h*, há a classe MNIST que faz a leitura desse conjunto de imagens, inverte branco com preto, ajusta bounding box, reduz a imagem e já coloca os dados resultantes no formato adequado para as rotinas de aprendizado de máquina do OpenCV. Copio abaixo os membros úteis dessa classe:

```
class MNIST {
 public:
  int na; vector< Mat_<GRY> > AX; vector<int> AY; Mat_<FLT> ax; Mat_<FLT> ay;
  int nq; vector< Mat_<GRY> > QX; vector<int> QY; Mat_<FLT> qx; Mat_<FLT> qy;
  Mat_<FLT> qp;
  MNIST(int _nlado=28, bool _inverte=true, bool _ajustaBbox=true);
  void le(string caminho="");
  int contaErros();
  Mat_<GRY> geraSaidaErros(int nl, int nc);
};
```

A seguinte sequência de comandos lê MNIST, calcula a imagem negativa (inverte preto com branco), elimina linhas/colunas brancas nas bordas, reduz o tamanho das imagens para 14×14 e coloca os dados resultantes em matrizes adequadas para alimentar as rotinas de aprendizado de máquina de OpenCV:

```
  // Reduz o tamanho das imagens para 14x14. inverte_cores=true, ajustaBbox=true
  MNIST mnist(14, true, true);
  mnist.le("/home/hae/cekeikon5/tiny_dnn/data");
```

Após a leitura, ficam disponíveis as seguintes estruturas dentro da classe MNIST:

```
  int na; // Numero de amostras de treinamento (60000)
  vector< Mat_<uchar> > AX;  // Vetor de 60000 imagens de treinamento
                             // redimensionadas, com cores invertidas, e com BBox.
  vector<int> AY; // Classificacoes das imagens de treinamento (0 a 9)
  Mat_<float> ax; // Imagens de treinamento convertidos para float, cada imagem por linha.
  Mat_<float> ay; // Classificacoes das imagens de treinamento convertidas para float (0 a 9)
  int nq; // Numero de imagens testes (10000)
  vector< Mat_<uchar> > QX; // Vetor de imagens de teste
                            // redimensionadas, com cores invertidas, e com BBox.
  vector<int> QY; // Classificacoes das imagens de teste (0 a 9)
  Mat_<float> qx; // Imagens de teste convertidos para FLT, cada imagem numa linha.
  Mat_<float> qy; // Classificacoes das imagens de teste  (0 a 9)
  Mat_<float> qp; // Matriz ja alocado para colocar as saidas dos algoritmos de aprendizagem.
                  // Matriz tem uma coluna e 10000 linhas.
```

1) *AX* e *QX* são vetores de imagens GRY (uchar) de treino e teste respectivamente. Dependendo das opções de leitura, já com cores invertidas e ajustadas por bounding box (linhas/colunas brancas eliminadas).
2) *AY* e *QY* são vetor de inteiros de rótulos (números de 0 a 9, as classificações corretas) de treino e teste respectivamente.
3) *na* é o número de elementos de treino (*AX* e *AY*, 60000). *nq* é o número de elementos de teste (*QX* e *QY*, 10000).
4) *ax* e *qx* são matrizes tipo float (respectivamente com *na* e *nq* linhas) que têm em cada linha um vetor com todos os pixels de uma imagem *AX*/*QX* (convertidos para 0=preto e 1=branco). Estas são estruturas adequadas para alimentar as rotinas de aprendizado do OpenCV.
5) *ay* e *qy* são matrizes tipo float (respectivamente com *na* e *nq* linhas) com uma única coluna. São cópias dos vetores *AY* e *QY*, convertidos de *int* para *float*. Contêm rótulos de 0 a 9.
6) *qp* é matriz tipo *float* com *nq* linhas e 1 coluna, com conteúdo indefinido. O seu algoritmo de aprendizado deve preencher *qp* com as classificações (números de 0 a 9).
7) O método `int e=mnist.contaErros()` devolve o número de elementos diferentes entre *qy* e *qp* (um número inteiro entre 0 e 10000).
8) Método `Mat_<GRY> a=mnist.geraSaidaErros(2,5)` gera imagem a com no máximo 2×5=10 imagens classificadas incorretamente, organizadas em 2 linhas e 5 colunas (2 e 5 são parâmetros que você pode mudar).

Note que AX, QX, AY e QY são as imagens e rótulos originais lidos. As matrizes ax, qx, ay e qy resultam da conversão dos dados anteriores para um formato que permite alimentar diretamente as rotinas de OpenCV (figura X). O vetor qp é onde o algoritmo deve colocar o resultado da inferência.

ax							ay
1ª imagem pixel (0,0)	1ª imagem pixel (0,1)	1ª imagem pixel (0,2)	...				3
2ª imagem pixel (0,0)	2ª imagem pixel (0,1)	2ª imagem pixel (0,2)	...				5
...							

Figura X: Os exemplos de entrada *ax* e *qx* da classe *MNIST* estão no formato float, no intervalo entre 0 e 1, pronto para alimentar as rotinas de aprendizado de OpenCV.

O programa abaixo gera uma imagem com as primeiras 20 imagens de teste de MNIST, organizadas em 5 linhas e 4 colunas:

```
//visualiza.cpp - imprime as primeiras 9 imagens de teste de MNIST 2024
#include "procimagem.h"
int main() {
  MNIST mnist(14,true,true); //nao redimensiona imagens
                             //inverte preto/branco=true,
                             //crop bounding box=false
  mnist.le("/home/hae/cekeikon5/tiny_dnn/data");
  mnist.qp.setTo(10); //Coloca uma classificacao errada de proposito
  Mat_<uchar> e=mnist.geraSaidaErros(5,4); //Organiza 10000 imagens em 100 linhas e 100 colunas
  imwrite("visualiza.png",e); //Imprime 10000 imagens-teste como visual.png
}
```

Programa: visualiza.cpp

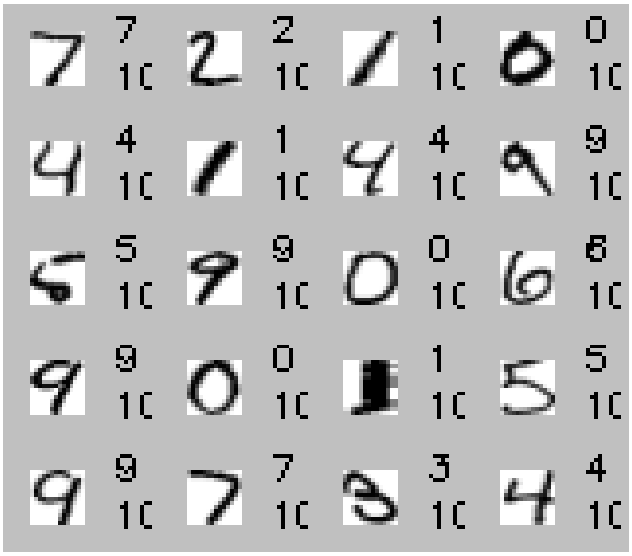


Figura: Visualiza.png

Agora, estamos prontos para testar os algoritmos de aprendizado de OpenCV para classificar os dígitos de MNIST.

[PSI3471 aula 9 parte 2 – fim]

[PSI3471 aula 10 – início]

1.3 Vizinho mais próximo força bruta

Vamos começar com o aprendizado vizinho mais próximo força bruta. Os algoritmos abaixo usam vizinho mais próximo do OpenCV versão 2 ou versões 3/4 para classificar MNIST.

Note que não existe tempo de treino para vizinho mais próximo força bruta, pois o treino consiste apenas em armazenar os exemplos de treinamento.

A taxa de erro das ambas versões é 2,5%. Porém, o programa em OpenCV v3/v4 é aproximadamente 7 vezes mais rápido que o programa em OpenCV2 (10s versus 77s).

Se escrevesse a rotina de vizinho mais próximo "manualmente" em C++, demora 360s, ou seja, 36 vezes mais lento que o programa em OpenCV v3/v4. Se escrevesse "manualmente" em Python, provavelmente demoraria algo como 10 vezes mais, algo da ordem de 3600s.

```
//knearest.cpp 2024
//Linkar com opencv2 (compila knearest -ocv -v2)
#include "procimagem.h"
int main() {
  MNIST mnist(14, true, true);
  mnist.le("/home/hae/cekeikon5/tiny_dnn/data");
  double t1=tempo();
  CvKNearest ind(mnist.ax,mnist.ay,Mat(),false,1);
  double t2=tempo();
  ind.find_nearest(mnist.qx, 1, &mnist.qp);
  double t3=tempo();
  printf("Erros=%10.2f%%\n",100.0*mnist.contaErros()/mnist.nq);
  printf("Tempo de treino: %f\n",t2-t1);
  printf("Tempo de predicao: %f\n",t3-t2);
}
```

```
Saída:

Erros=      2.50%
Tempo de treino: 0.019756
Tempo de predicao: 77.393110
```

Programa: Classifica MNIST usando função vizinho mais próximo força bruta de OpenCV2/C++. Demora 77s.

```
//knearest_v3.cpp 2024
//Linkar com opencv3
//compila knearest_v3 -ocv -v3 ou compila.sh knearest_v3
#include "procimagem.h"
int main() {
  MNIST mnist(14, true, true);
  mnist.le("/home/hae/cekeikon5/tiny_dnn/data");
  double t1=tempo();
  Ptr<ml::KNearest>  knn(ml::KNearest::create());
  knn->train(mnist.ax, ml::ROW_SAMPLE, mnist.ay);
  double t2=tempo();
  Mat_<float> dist;
  knn->findNearest(mnist.qx, 1, noArray(), mnist.qp, dist);
  double t3=tempo();
  printf("Erros=%10.2f%%\n",100.0*mnist.contaErros()/mnist.nq);
  printf("Tempo de treino: %f\n",t2-t1);
  printf("Tempo de predicao: %f\n",t3-t2);
}
```

```
Saída:

Erros=      2.50%
Tempo de treino: 0.018917
Tempo de predicao: 10.350899
```

Programa: Classifica MNIST usando função vizinho mais próximo força bruta de OpenCV v3/v4. Demora 10s.

Exercício: Complete o programa abaixo para classificar MNIST usando *k*-nearest neighbors, isto é, achar *k* vizinhos mais próximos e calcular a moda (a resposta mais frequente) dos *k* rótulos. Nota: Obtém 2,48% de erro usando *k*=3. Para outros valores de *k*, o erro aumenta. [A solução está em ~/algpi/classif/pos2024/11nearest34.cpp]

```
//11nearest34.cpp 2024
//Linkar com opencv 3/4 (compila knearest_v3 -ocv -v3 ou compila.sh knea-
rest_v3)
#include "procimagem.h"
int main() {
  MNIST mnist(14, true, true);
  mnist.le("/home/hae/cekeikon5/tiny_dnn/data");
  double t1=tempo();
  Ptr<ml::KNearest>  knn(ml::KNearest::create());
  knn->train(mnist.ax, ml::ROW_SAMPLE, mnist.ay);
  double t2=tempo();

  // Precisamos classificar 10000 imagens de teste mnist.qx (10000x196)
  // Precisamos colocar as 10000 classificacoes em mnist.qp (10000x1)
  Mat_<float> saidas,dists;
  int k=3;
  for (int l=0; l<10000; l++) {
    knn->findNearest(mnist.qx.row(l), k, noArray(), saidas, dists);
    //cout << saidas.rows << " " << saidas.cols << endl;
    //cout << saidas << endl;
    //cout << dists.rows << " " << dists.cols << endl;
    //cout << dists << endl;
    // Calcule a moda (o elemento mais frequente) de saidas.
    // Coloque moda em mnist.qp(l)
    (...)
    mnist.qp(l)=(...);
  }

  double t3=tempo();
  printf("Erros=%10.2f%%\n",100.0*mnist.contaErros()/mnist.nq);
  printf("Tempo de treino: %f\n",t2-t1);
  printf("Tempo de predicao: %f\n",t3-t2);
}
```

Exercício: Escreva o algoritmo vizinho mais próximo (1-NN) pela força bruta manualmente (sem usar uma rotina pronta de OpenCV ou de alguma outra biblioteca) para classificar MNIST e compare o seu tempo de processamento com o tempo de processamento das funções do OpenCV. Para ler MNIST, você deve usar os comandos abaixo, que reduzem imagens para 14×14 e eliminam linhas/colunas brancas.

```
MNIST mnist(14, true, true);
mnist.le("/home/hae/cekeikon5/tiny_dnn/data");
```

Imprima a taxa de erro.

Nota: O programa deve demorar algo como 10 minutos. Imprima quantas imagens já processou (por exemplo, 100, 200, 300, etc) para saber que o seu programa continua rodando e não travou. O seu programa está errado se não obtiver 2,5% de erro.

1.4 FlaNN (Fast Approximate Nearest Neighbor Search) do OpenCV

Agora, vamos testar *FlaNN* que faz busca do vizinho mais próximo aproximado usando *kd-árvore*. O programa abaixo (que pode ser linkado com OpenCV v2/v3/v4) constrói 4 *kd-trees* e a busca é realizada com parâmetro de backtracking 32. A taxa de erros obtida é 2,99% em OpenCV2 e 3.23% em OpenCV3. Os tempos de treino e teste são respectivamente 0,33s e 0,17s (muito melhor que 15s do algoritmo força bruta).

```
//flann.cpp 2024
//Linkar com opencv 2, 3 ou 4
#include "procimagem.h"
int main() {
  MNIST mnist(14, true, true);
  mnist.le("/home/hae/cekeikon5/tiny_dnn/data");
  double t1=tempo();
  flann::Index ind(mnist.ax,flann::KDTreeIndexParams(4));
  double t2=tempo();
  Mat_<int> matches(mnist.nq,1); Mat_<float> dists(mnist.nq,1);
  ind.knnSearch(mnist.qx,matches,dists,1,flann::SearchParams(32));
  for (int l=0; l<mnist.qx.rows; l++) {
    mnist.qp(l)=mnist.ay(matches(l));
  }
  double t3=tempo();
  printf("Erros=%10.2f%%\n",100.0*mnist.contaErros()/mnist.nq);
  printf("Tempo de treino: %f\n",t2-t1);
  printf("Tempo de predicao: %f\n",t3-t2);
}
```

```
Saída:

Erros=      3.23%
Tempo de treino: 0.360720
Tempo de predicao: 0.179155
```

Programa: Classifica MNIST chamando FlaNN de C++.

Aumentando o número de árvores para 64 e o parâmetro de backtracking para 256, obtive taxa de erro de 2,50%, com tempos de treino e teste de 4,91s e 2,24s. Isto é, a taxa de erro ficou igual ao algoritmo vizinho mais próximo força bruta, com menor tempo de processamento.

Diminuindo o tamanho das imagens para 12×12, obtive taxa de erro de 2,39% (em OpenCV3), com tempos de treino e teste de 4,50s e 2,14s.
Qual será a taxa de erro usando imagens 10×10?

Método	Treino	Teste	Erro
1-NN força bruta OpenCV2, imagem 14×14	0s	96s	2,5%
1-NN força bruta OpenCV3	0s	15s	2,5%
1-NN força bruta implem. manual C++	0s	360s (6 min)	2,5%
FlaNN 4 árvores, 32 backtrackings, C++	0,33s	0,17s	2,99%
FlaNN 64 árvores, 256 backtracking, C++	4,91s	2,24s	2,5%
FlaNN 64 árvores, 256 backtracking, C++, imagem 12×12	4,50s	2,24s	2,39%
FlaNN 4 árvores, 0 backtracking, Python, imagem 14×14	0,6s	0,25s	3,6%

A versão deste programa em Python3 (usando OpenCV4) está abaixo. Taxa de erro=3,6%, tempo de treino=0,6s e tempo de teste=0,25s. Aqui, programa Python é tão rápido quanto C++ pois ambos estão chamando funções da biblioteca FlaNN.

```python
# flann.py 2024
import tensorflow.keras as keras
from keras.datasets import mnist
import cv2, sys, time
import numpy as np

(AX, ay), (QX, qy) = mnist.load_data();

ax=np.empty((AX.shape[0],14,14))
for i in range(AX.shape[0]):
  ax[i]=cv2.resize(AX[i],(14,14),cv2.INTER_NEAREST);
ax = ax.reshape(ax.shape[0],ax.shape[1]*ax.shape[2]).astype("float32")/255

qx=np.empty((QX.shape[0],14,14))
for i in range(QX.shape[0]):
  qx[i]=cv2.resize(QX[i],(14,14),cv2.INTER_NEAREST);
qx = qx.reshape(qx.shape[0],qx.shape[1]*qx.shape[2]).astype("float32")/255

t1 = time.time()
FLANN_INDEX_KDTREE = 1;  # BUG: Faltam "FLANN enums" do OpenCV
flann_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 4);
flann = cv2.flann_Index(ax, flann_params)
t2 = time.time()
matches, dists = flann.knnSearch(qx, 1)
t3 = time.time()

# for l in range(matches.shape[0]):
  # i=matches[l]; qp[l]=ay[i]
qp = ay[matches].flatten()

# erros=0;
# for l in range(matches.shape[0]):
  # if qp[l]!=qy[l]: erros+=1
erros = np.count_nonzero(qp!=qy)

print("Erros=%5.2f%%" % (100.0*erros/qy.shape[0]) )
print("Tempo de treinamento: %f"%(t2-t1))
print("Tempo de predicao: %f"%(t3-t2))
```

```
Saída:

Erros= 3.60%
Tempo de treinamento: 0.352957
Tempo de predicao: 0.146556
```

Programa: Classifica MNIST chamando FlaNN/OpenCV4 de Python.

1.5 Árvore de decisão

Agora, vamos testar árvore de decisão em OpenCV2. O programa abaixo demora 6s para treinar e somente 0,0015s (sic) para classificar 10.000 imagens. Conforme vimos, a árvore de decisão é muito rápida na inferência. O problema é que a taxa de erro obtida foi 22,87%, pois a implementação de árvore de decisão em OpenCV não é muito boa.

```
//dtree.cpp - pos2020
//Linkar com OpenCV2
#include <cekeikon.h>
int main() {
  MNIST mnist(14, true, true);
  mnist.le("/home/hae/cekeikon5/tiny_dnn/data");
  TimePoint t1=timePoint();
  CvDTree ind; ind.train(mnist.ax,CV_ROW_SAMPLE,mnist.ay);
  TimePoint t2=timePoint();
  for (int l=0; l<mnist.nq; l++) {
    mnist.qp(l)=ind.predict(mnist.qx.row(l))->value;
  }
  TimePoint t3=timePoint();
  printf("Erros=%10.2f%%\n",100.0*mnist.contaErros()/mnist.nq);
  printf("Tempo de treinamento: %f\n",timeSpan(t1,t2));
  printf("Tempo de predicao: %f\n",timeSpan(t2,t3));
}
```

Programa: Classifica MNIST chamando árvore de decisão de OpenCV2 em C++. Taxa de erro=22,87%, pois a implementação de árvore de decisão de OpenCV é ruim.

Nota: Estou obtendo erros quando traduzo este programa para OpenCV v3/v4.

Vamos usar a implementação de árvore de decisão em SkLearn/Python, que é melhor que árvore de decisão em OpenCV. A taxa de erro muda a cada execução, mas é tipicamente 11,5%. O tempo de treino é 4s e teste 0,003s.

```
#dtree.py - 2024
import numpy as np
from sklearn import tree
import sys, time, cv2
import tensorflow.keras as keras

#<<<<<<<<<<<<<<< main <<<<<<<<<<<<<<<<
mnist = keras.datasets.mnist
(AX, ay),(QX, qy) = mnist.load_data()

ax=np.empty((AX.shape[0],14,14))
for i in range(AX.shape[0]):
  ax[i]=cv2.resize(AX[i],(14,14),cv2.INTER_NEAREST);
ax = ax.reshape(ax.shape[0],ax.shape[1]*ax.shape[2]).astype("float32")/255

qx=np.empty((QX.shape[0],14,14))
for i in range(QX.shape[0]):
  qx[i]=cv2.resize(QX[i],(14,14),cv2.INTER_NEAREST);
qx = qx.reshape(qx.shape[0],qx.shape[1]*qx.shape[2]).astype("float32")/255

t1 = time.time()
arvore= tree.DecisionTreeClassifier()
arvore= arvore.fit(ax, ay)
t2 = time.time()
qp=arvore.predict(qx)
t3 = time.time()

erros = np.sum( qy!=qp )

print("Taxa de erro = %2.3f%%"%(100*erros/qy.shape[0]))
print("Tempo de treinamento: %f"%(t2-t1))
print("Tempo de predicao: %f"%(t3-t2))
```

```
Saída:

Taxa de erro = 11.530%
Tempo de treinamento: 4.131044
Tempo de predicao: 0.003059
```

Programa: Classifica MNIST chamando árvore de decisão de SkLearn/Python. Taxa de erro = 11,5%.

1.7 Classificador Normal Bayes

O próximo algoritmo que testaremos é *Normal Bayes*. O programa abaixo cometeu 10,4% de erro e demorou aproximadamente 1,5s tanto para treino como para teste.

```
//bayes.cpp 2024
//Linkar com OpenCV2
#include "procimagem.h"
int main() {
  MNIST mnist(14, true, true);
  mnist.le("/home/hae/cekeikon5/tiny_dnn/data");
  double t1=tempo();
  CvNormalBayesClassifier ind;
  ind.train(mnist.ax, mnist.ay);
  double t2=tempo();
  ind.predict(mnist.qx, &mnist.qp);
  double t3=tempo();
  printf("Erros=%10.2f%%\n",100.0*mnist.contaErros()/mnist.nq);
  printf("Tempo de treino: %f\n",t2-t1);
  printf("Tempo de predicao: %f\n",t3-t2);
}
```

```
Saída:

Erros=     10.43%
Tempo de treino: 1.455647
Tempo de predicao: 1.524295
```

Programa: Classifica MNIST chamando Normal Bayes de OpenCV2/C++. Taxa de erro=10,4%.

Em OpenCV v3/v4, as saídas *ay*, *qy* e *qp* de Normal Bayes Classifier devem ser matriz de inteiros. A taxa de erro não muda.

```
//bayes34.cpp 2024
//Linkar com OpenCV 3/4
#include "procimagem.h"
int main() {
  MNIST mnist(14, true, true);
  mnist.le("/home/hae/cekeikon5/tiny_dnn/data");

  double t1=tempo();

  Ptr<ml::NormalBayesClassifier> ind = ml::NormalBayesClassifier::create();
  ind->train(mnist.ax, ml::ROW_SAMPLE, mnist.AY);
  double t2=tempo();

  Mat_<int> QP(mnist.qy.size());
  ind->predict(mnist.qx,QP);
  for (unsigned i=0; i<mnist.qp.total(); i++)
    mnist.qp(i)=QP(i);
  double t3=tempo();

  printf("Erros=%10.2f%%\n",100.0*mnist.contaErros()/mnist.nq);
  printf("Tempo de treino: %f\n",t2-t1);
  printf("Tempo de predicao: %f\n",t3-t2);
}
```

```
Saída:

Erros=     10.43%
Tempo de treino: 1.470124
Tempo de predicao: 0.225703
```

Programa: Classifica MNIST chamando Normal Bayes de OpenCV v3 v4/C++. Taxa de erro=10,4%.

1.8 Rede neural densa (MLP) do OpenCV2

Nota: Estou supondo que vocês já estudaram redes neurais.

Agora, vamos testar rede neural artificial densa do OpenCV2.

Para fazer com que rede neural faça classificação multi-classe, vamos usar a técnica chamada “one-hot encoding”. Vamos converter o rótulo de saída (um número entre 0 e 9) num vetor com 10 elementos preenchido com zeros, exceto na posição do rótulo que será preenchido com um.

Por exemplo, (AX, AY) = (“imagem 0”, 0) será convertido em (AX, AY2) = (“imagem 0”, (1, 0, 0, 0, 0, 0, 0, 0, 0, 0)). Estes exemplos convertidos irão treinar uma rede neural com 14×14 neurônios na entradas e 10 neurônios na saída.

A instância QX será classificada como o índice da saída que apresentar a maior ativação (usando a função argMax do *procimagem.h*). As duas camadas escondidas da rede têm 100 e 30 neurônios. Estes parâmetros foram selecionados empiricamente.

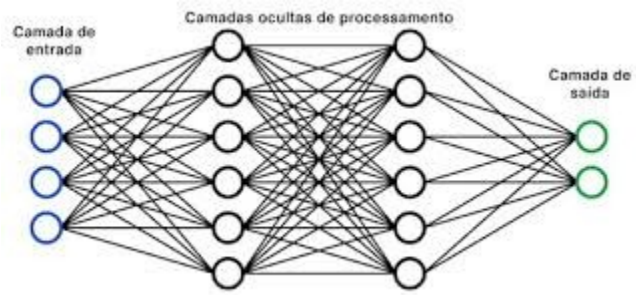


Figura: A imagem 14×14 de entrada é convertida num vetor com 196 neurônios (flatten). A rede possui duas camadas escondidas com 100 e 30 neurônios. Camada de saída tem 10 neurônios que categorizam as 10 classes.

O programa resultante fica:

```
1  //ann.cpp 2024
2  //Linkar com opencv2
3  #include "procimagem.h"
4
5  int main() {
6    MNIST mnist(14,true,true);
7    mnist.le("/home/hae/cekeikon5/tiny_dnn/data");
8
9    Mat_<float> ay2(mnist.ay.rows, 10); ay2.setTo(0.0);
10   for (int i=0; i<mnist.ay.rows; i++) {
11     int j=round(mnist.ay(i)); assert(0<=j && j<10);
12     ay2(i,j)=+1.0;
13   }
14
15   double t1=tempo();
16   Mat_<int> v = (Mat_<int>(1,4) << mnist.ax.cols, 100, 30, 10); //196
17   CvANN_MLP ind(v); ind.train(mnist.ax, ay2, Mat());
18
19   double t2=tempo();
20   Mat_<float> saida; //(1,mnist.qx.cols);
21   for (int l=0; l<mnist.nq; l++) {
22     ind.predict(mnist.qx.row(l),saida);
23     mnist.qp(l)=argMax(saida);
24   }
25   double t3=tempo();
26
27   printf("Erros=%10.2f%%\n",100.0*mnist.contaErros()/mnist.nq);
28   printf("Tempo de treino: %f\n",t2-t1);
29   printf("Tempo de predicao: %f\n",t3-t2);
30 }
```

```
Saída:

Erros=      6.52%
Tempo de treino: 52.918162
Tempo de predicao: 0.078301
```

Programa: Classifica MNIST chamando rede neural densa de OpenCV2/C++. Taxa de erro=6,52%.

```
1  //ann34.cpp 2024
2  //Linkar com opencv 3/4
3  #include "procimagem.h"
4  using namespace ml;
5  int main() {
6    MNIST mnist(14,true,true);
7    mnist.le("/home/hae/cekeikon5/tiny_dnn/data");
8
9    Mat_<float> ay2(mnist.ay.rows, 10); ay2.setTo(0.0);
10   for (int i=0; i<mnist.ay.rows; i++) {
11     int j=round(mnist.ay(i)); assert(0<=j && j<10);
12     ay2(i,j)=+1.0;
13   }
14
15   double t1=tempo(); Ptr<ANN_MLP> mlp=ANN_MLP::create();
16   Mat_<int> layersSize = (Mat_<int>(1,4) << mnist.ax.cols, 100, 30, 10); //196
17   mlp->setLayerSizes(layersSize);
18   mlp->setActivationFunction(ANN_MLP::ActivationFunctions::SIGMOID_SYM);
19   TermCriteria termCrit = TermCriteria(TermCriteria::Type::MAX_ITER, 100, 0.001);
20   mlp->setTermCriteria(termCrit);
21   mlp->train(mnist.ax, ROW_SAMPLE, ay2);
22
23   double t2=tempo();
24   Mat_<float> saida; //(1,mnist.qx.cols);
25   for (int l=0; l<mnist.nq; l++) {
26     mlp->predict(mnist.qx.row(l),saida);
27     mnist.qp(l)=argMax(saida);
28   }
29   double t3=tempo();
30
31   printf("Erros=%10.2f%%\n",100.0*mnist.contaErros()/mnist.nq);
32   printf("Tempo de treino: %f\n",t2-t1);
33   printf("Tempo de predicao: %f\n",t3-t2);
34 }
```

```
Saída:

Erros=      4.56%
Tempo de treino: 52.009850
Tempo de predicao: 0.082037
```

Programa: Classifica MNIST chamando rede neural densa de OpenCV v3/v4 e C++. Taxa de erro=4,56%.

As linhas 9-13 convertem os rótulos de saída AY em one-hot-encoding AY2.

Exemplos de one-hot encoding:
Classe 2 é convertido no vetor one-hot encoding abaixo:

0	0	1	0	0	0	0	0	0	0

Classe 7 é convertido no vetor one-hot encoding abaixo:

0	0	0	0	0	0	0	1	0	0

A linha 16 descreve a estrutura da rede neural (14×14, 100, 30, 10).

O comando *predict* faz inferência, onde *saida* é uma matriz com 1 linha e 10 colunas.
A função *argMax* acha o índice do maior elemento do vetor *saida*.

O programa apresentou erro de 4,5% a 6,5%, tempo de treino de 52s e tempo de teste de 0,08s. Veremos que a implementação de rede neural de Tensorflow/Keras é bem melhor.

1.9 Support Vector Machine

O aluno Nikolas M. Stankevicius observou que é possível atingir taxa de erro 1,95% usando Support Vector Machine em OpenCV2. Para atingir esta taxa, é necessário não eliminar as linhas/colunas brancas e usar imagens reduzidas para 10×10. É o menor erro obtido usando uma técnica que não seja rede neural convolucional. Este programa é bastante sensível ao ajuste dos parâmetros.

```
// svm.cpp 2024
// linkar com OpenCV2
#include "procimagem.h"
int main() {
  MNIST mnist(10, true, false); //10x10 e sem Bounding Box
  mnist.le("/home/hae/cekeikon5/tiny_dnn/data");
  CvSVM ind;
  double t1=tempo();
  ind.train(mnist.ax, mnist.ay); // Treinamento da SVM
  double t2=tempo();
  for (int l=0; l<mnist.nq; l++)
    mnist.qp(l)=ind.predict(mnist.qx.row(l));
  double t3=tempo();
  printf("Erros=%10.2f%%\n",100.0*mnist.contaErros()/mnist.nq);
  printf("Tempo de treino: %f\n",t2-t1);
  printf("Tempo de predicao: %f\n",t3-t2);
}
```

```
Saída:

Erros=      1.95%
Tempo de treino: 68.510494
Tempo de predicao: 10.446792
```

Programa: SVM em OpenCV2. Taxa de erro 1,95%.

```
// svm.cpp 2024
// linkar com OpenCV 3/4
#include "procimagem.h"
using namespace ml;
int main() {
  MNIST mnist(10, true, false); //10x10 e sem Bounding Box
  mnist.le("/home/hae/cekeikon5/tiny_dnn/data");
  Ptr<SVM> ind=SVM::create();
  double t1=tempo();
  ind->train(mnist.ax, ROW_SAMPLE, mnist.AY); // Treinamento da SVM
  double t2=tempo();
  for (int l=0; l<mnist.nq; l++)
    mnist.qp(l)=ind->predict(mnist.qx.row(l));
  double t3=tempo();
  printf("Erros=%10.2f%%\n",100.0*mnist.contaErros()/mnist.nq);
  printf("Tempo de treino: %f\n",t2-t1);
  printf("Tempo de predicao: %f\n",t3-t2);
}
```

```
Saída:

Erros=      1.92%
Tempo de treino: 51.627205
Tempo de predicao: 9.573501
```

Programa: SVM em OpenCV v3/v4. Taxa de erro 1,92%.

Veja, por exemplo, sites abaixo para mais explicações sobre SVM.
https://medium.com/swlh/support-vector-machine-from-scratch-ce095a47dc5c
http://web.mit.edu/6.034/wwwbob/svm-notes-long-08.pdf
https://www.newtechdojo.com/understanding-support-vector-machines-svm/
https://en.wikipedia.org/wiki/Support_vector_machine

1.10 Resumo dos testes

Tabela 2: Taxas de erro (%) e tempos de processamento usando bbox na classificação de MNIST.

Knearest OpenCV2 14×14	Erros	2,50%
	Treino	desprezível
	Predição	96s
Knearest OpenCV3/4 14×14	Erros	2,50%
	Treino	desprezível
	Predição	15s
Flann, OpenCV2 ou 3 4 árvores sem backtracking 14×14	Erros	2,99%
	Treino	0,52s
	Predição	0,25s
Flann, OpenCV 3/4 64 árvores com backtracking 256, 12×12	Erros	2,39%
	Treino	4,5s
	Predição	2,1s
Dtree 14×14 OpenCV2	Erros	22,98%
	Treino	6s
	Predição	0,0015s
Dtree 14×14 Python-SkLearn	Erros	12,3%
	Treino	4,3s
	Predição	0,003s
Bayes, OpenCV 2/3/4 14×14	Erros	10,43%
	Treino	1,8s
	Predição	1,9s
ANN OpenCV2 14×14	Erros	6,52%
	Treino	54s
	Predição	0,09s
ANN OpenCV 3/4 14×14	Erros	4,56%
	Treino	54s
	Predição	0,2s
SVM OpenCV 2/3/4 10×10 Sem bounding box	Erros	1,95%
	Treino	70s
	Predição	0,1s

Exercício interessante: Você consegue prever o que aconteceria com as taxas de erro, os tempos de treino e os tempos de teste se aplicássemos os algoritmos acima em imagens 28×28 originais (em vez de reduzidas para 14×14 ou 10×10)? Depois de fazer as previsões, execute os algoritmos para verificar se as suas previsões estão corretas.

[PSI3471 aulas 9/10. Lição de casa #2/2. Vale 5.] Implemente um programa de aprendizado de máquina clássico para classificar MNIST que atinge taxa de erro menor que 2,15% sem usar SVM ou taxa de erro menor que 1,85% usando SVM.

Nota: Não pode usar deep learning nem rede neural convolucional. Não pode usar bibliotecas de deep learning como Keras, Tensorflow ou PyTorch (exceto para ler MNIST).

Descreva (através de explicação falada no vídeo e também como comentários dentro do seu programa .cpp ou .py) a taxa de erro que obteve, o tempo de processamento e as alterações feitas para chegar ao seu programa com baixa taxa de erro.

Dica: É possível aumentar artificialmente os dados de treino deslocando as imagens originais um pixel para as direções norte, sul, leste e oeste, obtendo um conjunto de treino 5 vezes maior que o original. Com isso, cheguei à taxa de erro de 1,98% usando FlaNN e atingi taxa 1,78% com SVM (usando conjunto 2 vezes maior – em vez de 5 vezes – que a original). A técnica de gerar imagens de treino artificiais distorcidas chama-se “data augmentation”.

Nota: Você não é obrigado a usar imagens reduzidas para 14×14 nem é obrigado a eliminar linhas/colunas brancas. Você pode copiar os conteúdos de mnist.AX, mnist.AY, mnist.QX, mnist.QY (e outros membros da classe MNIST) para outras variáveis para que você possa alterá-las.

Nota: O aluno Ricardo Saito obteve o erro de 1.45% fazendo data augmentation com rotação e usando SVM de OpenCV/C++. ~/psi/disciplinas_velhas/SistInt-velho/SistInt241/solucao_aluno.

Nota: O aluno Guilherme Nicolino (2024) usou deskew para diminuiu o erro de 1.98% para 1.46% (usando SVM de SkLearn) ou para 1.91% (usando SVM de OpenCV). Adicionando HOG (Histogram of Gradient) aos atributos, diminuiu o erro de 1.46% para 0.87% (SVM de SkLearn). O cropping por bounding box diminuiu o erro de 0.87% para 0.81%. ~/psi/disciplinas_velhas/SistInt-velho/SistInt241/solucao_aluno. HOG leva em consideração as relações de vizinhança entre os pixels.

Exercício: Modifique o programa anterior para imprimir todos os dígitos classificados incorretamente (além de informar a taxa de erro).

2. Extração de Atributos

A quantidade de pixels de uma imagem típica costuma ser grande demais para alimentar diretamente os algoritmos de aprendizado de máquina clássicos. Assim, o programador deve extrair alguns atributos da imagem (*features* ou vetores com poucos números) para alimentar os algoritmos de aprendizado clássicos e classificar as imagens.

Estudamos acima o problema de classificação de imagens usando o conjunto de imagens MNIST. Nesse problema, o único pré-processamento que fizemos para diminuir a dimensionalidade foi reduzir a resolução da imagem e eliminar as bordas brancas.

Antes do aparecimento das redes neurais convolucionais profundas, o principal problema para classificar imagens era como extrair os atributos apropriados da imagem. Este processo tinha que ser feito manualmente.

Lembre-se: quanto menor a dimensão dos dados, mais facilmente o algoritmo de aprendizado consegue classificá-los. Só relembrar dos problema de achar ponta de reta (fácil) e detectar letras “a” ou “A” (difícil). Também já vimos o “ganho de informação” e “Gini” usados na árvore de decisão que consegue descobrir quais são os atributos úteis e inúteis para a classificação e que ajuda a diminuir os atributos.

Há uma variedade muito grande de técnicas de extração de atributos. Para servir de exemplos, vamos estudar rapidamente dois algoritmos que extraem atributos antes de classificar: detecção de rostos humanos usando filtros de Haar (algoritmo de Viola e Jones) e de pedestres usando Histograma de Gradiente Orientado.

Resposta de ChatGPT para a pergunta: “Quais eram os atributos usados antes do deep learning para classificar lesões em mamografia em benigno ou maligno?”

Antes do surgimento do deep learning, a classificação de lesões em mamografias era feita com base em características extraídas manualmente — descritores cuidadosamente projetados com base no conhecimento de radiologistas e em técnicas clássicas de processamento de imagens. Essas características eram extraídas das imagens e, em seguida, usadas em classificadores tradicionais de aprendizado de máquina (como SVM, k-NN, árvores de decisão, etc.).
Aqui está um resumo dos tipos de características mais usadas:

---

1. Características Morfológicas (baseadas em forma)

Descrevem a **geometria** da lesão, considerando que tumores benignos e malignos geralmente diferem na forma.

- **Área**
- **Perímetro**
- **Compacidade** (Perímetro² / Área)
- **Excentricidade** (quão alongada é a lesão)
- **Circularidade / Arredondamento**
- **Irregularidade / Espiculações** (bordas irregulares sugerem malignidade)
- **Nitidez das bordas**

---

2. Características de Textura

Descrevem os **padrões internos** dentro da lesão, capturando a heterogeneidade — um indicador importante de malignidade.

a. Textura Estatística (GLCM)

Baseadas em **matrizes de coocorrência de níveis de cinza (GLCM)**:

- Contraste
- Correlação
- Energia
- Homogeneidade

b. Baseadas em Histograma

- Média, variância, assimetria (skewness), curtose dos níveis de cinza

c. Laws' Texture Energy

- Mede padrões locais de textura, como ondulações ou manchas.

---

4. Características de Borda e Gradiente

Lesões malignas geralmente têm **bordas irregulares ou espiculadas**, então informações de gradiente ajudam a identificá-las.

- **Nitidez da borda** (magnitude do gradiente na fronteira)
- **Histograma da orientação dos gradientes**
- **Estatísticas da detecção de bordas (Canny)**
- **Perfil radial da borda**

---

6. Características no Domínio da Frequência

- **Transformada wavelet**: captura textura e forma em múltiplas escalas.
- **Descritores de Fourier**: descrevem a forma da lesão no domínio da frequência.

2.2 Detecção de rostos humanos em OpenCV2

Um problema muito estudado e útil é detectar rostos humanos, hoje presente em praticamente todos os smartphones.

A figura 7 mostra o resultado de detecção de rostos, usando o programa facedetect.cpp que vem como exemplo de OpenCV2. Quem instalou Cekeikon, esse programa está no diretório:

```
(Linux).../cekeikon5/opencv2cpu/sources/samples/c/facedetect.cpp
(Windows)...\cekeikon5\opencv2cpu\sources\samples\c\facedetect.cpp
```

Figura 7: Detecção de rostos em 012.jpg.

Para rodar o programa facedetect, é necessário compilá-lo e executá-lo especificando onde está o modelo e a imagem a ser processada. Tanto em Linux como em Windows, funcionam os seguintes comandos executados do diretório (...)/cekeikon5/opencv2cpu/samples/c:

```
facedetect --cascade=../../share/OpenCV/haarcascades/haarcascade_frontalface_default.xml 012.jpg
facedetect --cascade=../../share/OpenCV/haarcascades/haarcascade_frontalface_alt.xml 012.jpg
facedetect --cascade=../../share/OpenCV/haarcascades/haarcascade_frontalface_alt2.xml 012.jpg
facedetect --cascade=../../share/OpenCV/haarcascades/haarcascade_frontalface_alt_tree.xml 012.jpg
```

Em Linux, o seguinte comando no diretório (...)/cekeikon5/opencv2cpu/samples/c executa todos os testes acima:

```
~/cekeikon5/opencv2cpu/samples/c$ facedetect.sh
```

Exercício: Execute o programa *facedetect* em pelo menos 3 imagens que você escolher, diferentes das imagens ilustradas nesta apostila. Nos casos em que os programas falharem, tente achar uma explicação do motivo da falha.

2.3 Detecção de faces

Considere a detecção de faces por um telefone celular. A detecção deve ser quase instantânea, mesmo usando o processador pouco potente do celular, pois o usuário não pode esperar para tirar foto. O treino pode ser lento, pois será feito sem pressa num bom computador. O artigo [Viola2001] apresenta uma excelente técnica com estas características (sem usar deep learning).

Se conseguirmos criar um algoritmo rápido que classifica se o conteúdo de uma janela (por exemplo 24×24) é de um rosto humano ou não, podemos varrer a imagem usando esse modelo como um filtro móvel espacial para detectar as regiões da imagem onde aparecem rostos humanos.

2.4 Filtros de Haar

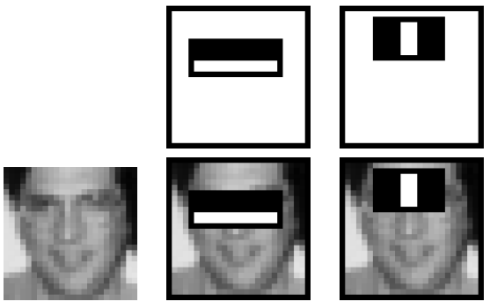
Como nós identificamos um rosto humano? Um rosto humano deve ter dois olhos, uma boca e um nariz. Esta intuição pode ser traduzida em um conjunto de filtros lineares (*template matchings*). Porém, para que a detecção seja ultra-rápida, os autores de [Viola2001] não usam qualquer filtro linear, mas somente aqueles chamados de filtros de Haar (Haar-like filters).

Um filtro de Haar (figura 8) é um filtro linear igual aos que já estudamos. A peculiaridade é que este filtro é formado somente por regiões retangulares alinhadas com os eixos com valores negativos (-1, pretos) ou positivos (+1, brancos). Os filtros de Haar podem ser aplicados muito rapidamente, em qualquer escala e em qualquer local da imagem, usando *imagem integral* (veja a apostila “integral”).

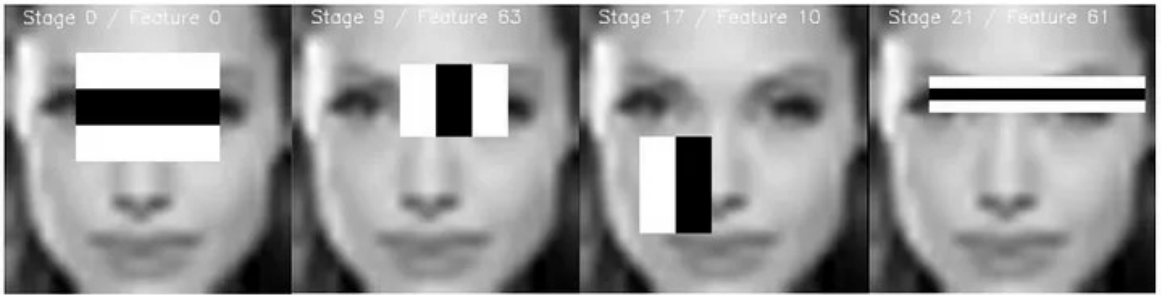
1, 1, 1, -1, -1, -1 1, 1, 1, -1, -1, -1 1, 1, 1, -1, -1, -1 1, 1, 1, -1, -1, -1 1, 1, 1, -1, -1, -1 1, 1, 1, -1, -1, -1	

Figura 8: Filtro de Haar.

Segundo [Viola2001], numa janela 24×24, podem ser construídos 180.000 filtros de Haar. Desses, os autores escolheram um conjunto de filtros que melhor conseguem detectar faces. A figura 9a apresenta os dois primeiros filtros de Haar utilizados pelos autores. O primeiro está usando o fato de que, numa face humana, a região dos olhos-sombrancelhas é mais escura do que a região logo abaixo (bochecha-nariz). O segundo está usando o fato de que as duas regiões dos olhos-sombrancelhas são mais escuras do que a região entre elas. A figura 9b mostra alguns filtros das etapas subsequentes.



(a)



(b)

Figura 9: Preto significa -1 e branco significa +1. (a) Dois primeiros filtros de Haar utilizados em [Viola2001]. (b) Alguns filtros de Haar dos estágios subsequentes [https://medium.com/@Andrew_D./computer-vision-viola-jones-object-detection-d2a609527b7c].

2.5 Adaboost

A técnica [Viola2001] utiliza uma técnica de Boosting (veja apostila “aprendizagem-ead”) aplicado em cascata para para classificar uma janela em face/não-face usando filtros de Haar (figura 10).

Cada estágio de Boosting (círculos da figura 10) consiste de um conjunto de árvores de decisão construído usando uma ou mais características de Haar. Cada estágio está programada para detectar corretamente praticamente 100% das faces (quase 0% de falso negativo) mesmo que responda que um número grande de não-faces é face (algo como 50% de falso positivo). Assim, se um estágio de Boosting responde que uma janela é não-face, não há necessidade de executar os estágios subsequentes, pois quase certamente aquela janela será não-face. Por outro lado, se responder que janela pode ser uma face, precisa fazer mais verificações para se certificar de que a janela realmente contém uma face. Uma janela é classificada como face somente se passar por todos os estágios. Como quase a totalidade das janelas de uma imagem são não-faces, normalmente somente poucos estágios serão aplicados a cada janela e a detecção de rostos será muito rápida em média.

O artigo [Viola2001] utilizou 6061 características de Haar, organizadas em 38 estágios. Os primeiros 5 estágios utilizam respectivamente 1, 10, 25, 25 e 50 características de Haar.

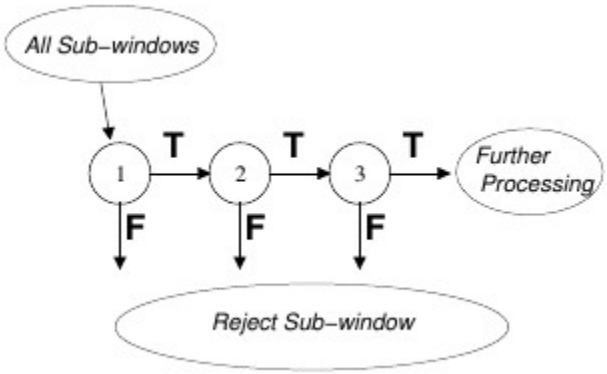


Figura 10: A classificação de uma janela em face/não-face é feita usando Boosting em cascata.

Exercício: Estude o que são imagens integrais para verificar como elas podem acelerar cálculo de filtro de Haar. Pode usar a apostila “integral” ou o artigo [Viola2001].

2.6 Detecção de pedestres pelo histograma de gradiente orientado (HOG)

Nota: O aluno Guilherme Nicolino (2024) usou várias técnicas junto com HOG (Histogram of Gradient) junto com outras técnicas para atingir erro de teste em MNIST de 0,81%. Isto é muito menos que 2,5% de erro do vizinho mais próximo, 1,9% de erro de SVM ou 4,5% de erro de rede neural artificial.

A figura 6 mostra o resultado de detecção de pessoas em pé, usando o programa *peopledetect.cpp* que vem como exemplo de OpenCV v2. Quem instalou *Cekeikon*, esse programa está no diretório:

```
(Linux)/cekeikon5/opencv2cpu/sources/samples/cpp/peopledetect.cpp
(Windows)\cekeikon5\opencv2cpu\sources\samples\cpp\peopledetect.cpp
```

Em Linux, execute o seguinte script:

```
(Linux)~/cekeikon5/opencv2cpu/samples/cpp$ peopledetect.sh
```

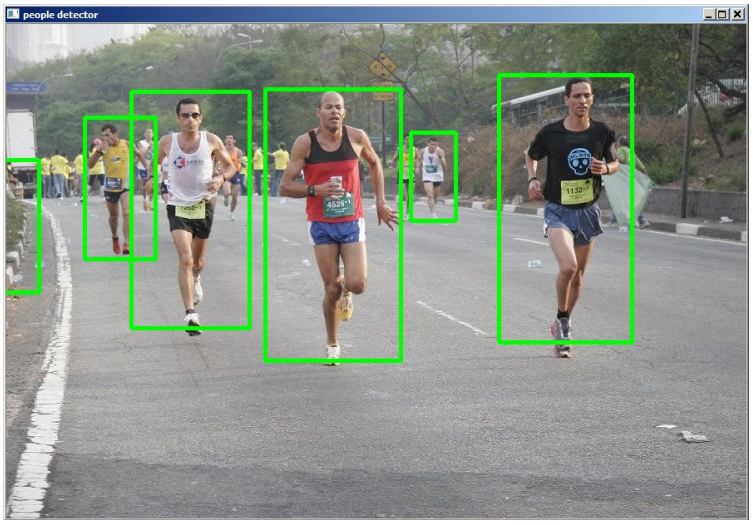


Figura 6: Detecção de pedestres, usando programa “peopledetect” do OpenCV

Para detectar pedestres, um algoritmo de aprendizado de máquina foi treinado com HOGs obtidos de imagens de pedestres e de não-pedestres. As imagens de não-pedestres são obtidas percorrendo com uma janela as imagens que não contém pessoas. Depois, este algoritmo faz inferências a partir dos HOGs das janelas da imagem para detectar pedestres [Dalal2005]. Veja a apostila “integral” para verificar como este processo pode ser acelerado usando imagem integral.

2.7 Histograma de Gradiente Orientado

Vamos tentar resolver uma versão simplificada do problema de detectar pedestres: vamos construir um programa que classifica imagens em pedestre ou automóvel, sendo que todas as imagens de treino e de teste estão na mesma escala (figura 3). Além disso, os pedestres estão "isolados" (não é um aglomerado de pessoas), sempre são vistos de frente ou por trás, e sempre estão de pé.

Um banco de dados (antigo) de MIT tem 924 imagens 128×64 de pedestres e 516 imagens 128×128 de carros[1]. Quebrei as imagens de carros em esquerda (carl*) e direita (carr*), para obter 2×516=1032 imagens 128×64 de metades dos carros. Esse material está no site "apostilas" como mit_cars.zip e mit_pedestrians.zip. Algumas dessas imagens estão na figura 3.

per00001.ppm

per00003.ppm

per00011.ppm

carl0001.ppm

carr0001.ppm

carr0002.ppm

Figura 3: Imagens de pedestres e carros.

O problema é, dada uma imagem 128×64 como na figura 3, classificá-la em pedestre ou a metade de um carro. Aqui é importante reduzir a dimensionalidade dos dados, pois alimentar um algoritmo de aprendizado com 128×64×3 = 24.576 atributos não irá funcionar bem.

Intuitivamente, os valores dos pixels das imagens (originais ou redimensionadas) não devem ser características muito boas para resolver este problema, pois há pedestres com roupas de todas as cores, assim como carros de cores variadas. O fundo também tem cores variadas.

O que você, um ser humano, faz para classificar uma imagem acima como de pedestre ou de carro? Utilizamos várias estratégias diferentes para classificar as imagens, difíceis de descrever verbalmente. Porém, podemos notar que todos os "contornos" dos pedestres são mais ou menos semelhantes entre si e bastante diferentes dos "contornos" dos carros. Esses contornos independem da cor do fundo, da cor da roupa, da cor do carro, etc. Porém, não é uma boa ideia usar os detectores de bordas (detector de Canny, máximo do módulo de gradiente, mudança de sinal do laplaciano, etc.) pois são operações altamente sensíveis a ruídos.

1 BD original estava nos endereços abaixo. Hoje (23/04/2020) não estão mais lá.
http://cbcl.mit.edu/cbcl/software-datasets/PedestrianData.html
http://cbcl.mit.edu/projects/cbcl/software-datasets/CarData1Readme.html

2.3.1 Histograma de gradiente orientado (HOG)

O artigo [Dalal2005] fornece uma solução interessante e robusta para este problema: histograma de gradiente orientado (HOG). Para vocês relembrarem, a figura 4 mostra o gradiente de uma imagem como flechas.

Se imaginarmos a imagem como um terreno onde os valores dos pixels representam alturas (pixels claros são montanhas e pixels escuros são vales), se colocarmos uma bola num pixel, ela vai descer o terreno no sentido contrário ao do gradiente. A magnitude do gradiente num ponto *p* corresponde à inclinação do terreno em *p*.

Em torno dos pedestres e carros devem existir gradientes de magnitudes grandes com direção perpendicular ao contorno.

O sentido desses gradientes (para dentro ou para fora do pedestre/carro) vai depender da cor do pedestre/carro e da cor de fundo.

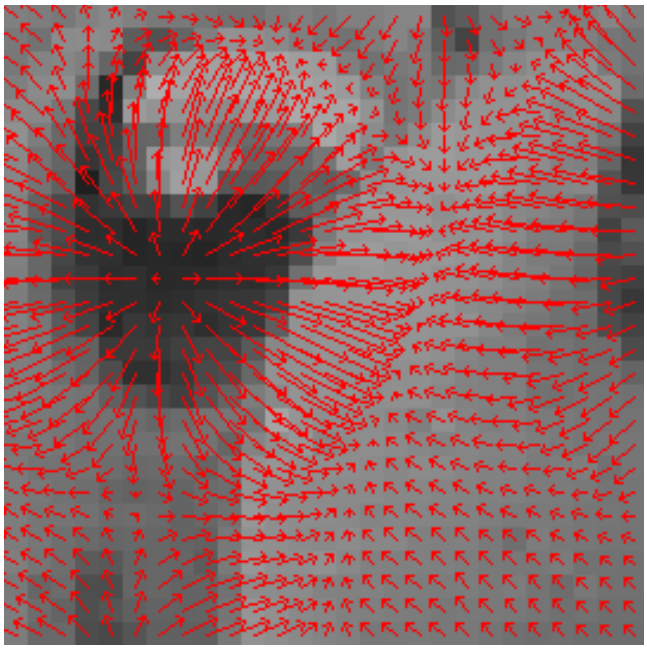
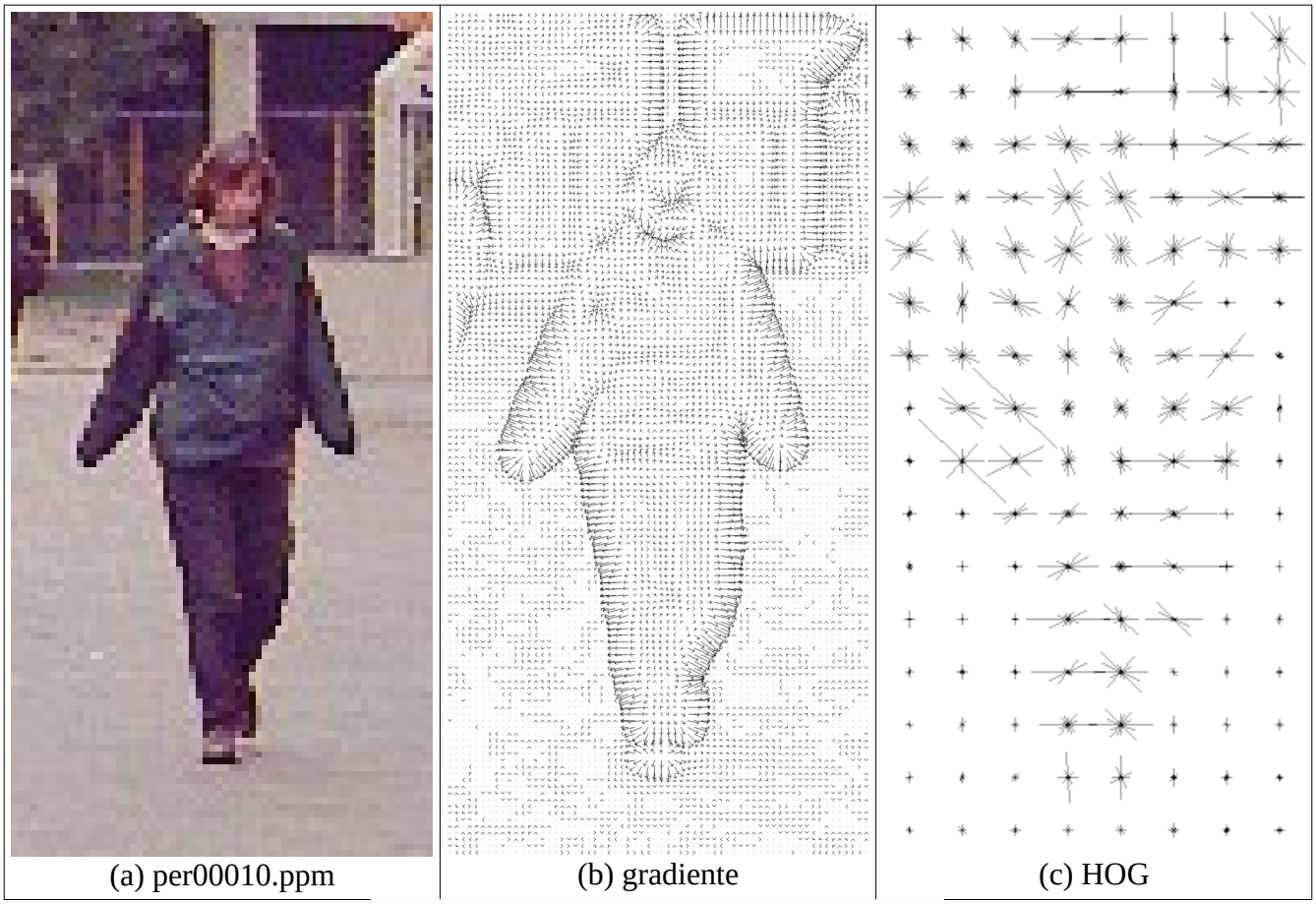


Figura 4: Gradiente de uma imagem é um campo vetorial que aponta para direção onde a imagem torna-se mais clara. A magnitude de gradiente é proporcionar à “inclinação do terreno”.

HOG primeiro calcula o gradiente em todos os pixels da imagem (figura Zb). Depois, HOG quebra a imagem em blocos de pixels, digamos, 8×8. Os 64 vetores de gradiente dentro de cada bloco são colocados em caixotes (bins) de acordo com o seu ângulo, digamos 8 caixotes: 0° a 22,5°; ...; 157,5° a 180° (figura Zc).



(a) per00010.ppm (b) gradiente (c) HOG

Figura Z: Imagem original, gradiente e HOG.

Nota: Para que o resultado seja sensível somente à direção e insensível ao sentido do gradiente, HOG subtrai 180° se ângulo do gradiente for maior que 180°. Para cada caixote, HOG soma os módulos dos gradientes que foram colocados nele. Assim, para cada bloco 8×8, HOG obtém 8 números, um para cada caixote.

Nota: Para imagens coloridas, HOG calcula os 3 gradientes, um para cada banda de cor RGB. Para juntar os gradientes das 3 bandas num único campo vetorial, HOG escolhe em cada pixel o vetor de maior módulo entre os 3 vetores das 3 bandas.

Nota: Se quisesse obter características sensíveis à direção e sentido do gradiente (isto é, que distingue objeto claro sobre fundo escuro do objeto escuro sobre fundo claro), os caixotes deveriam abranger o intervalo de ângulos de 0 a 360 graus e HOG não deveria efetuar subtração de 180 graus.

A figura 5 mostra algumas imagens de conjunto, com respectivas HOGs.

Cada “estrela” representa HOG de um bloco 8×8. O comprimento de um “raio da estrela” representa a soma dos módulos de gradiente no intervalo de ângulos daquele caixote.

Note que é possível observar os contornos dos pedestres nas HOGs das duas imagens inferiores, pois há “raios das estrelas” perpendiculares aos contornos dos dois pedestres. Como vimos, muitos algoritmos de aprendizado são bons em distinguir características inúteis das úteis. Isto é, conseguem identificar quais são os “raios das estrelas” úteis para distinguir pedestres de carros.

HOG é robusto a pequenos deslocamentos do objeto, pois HOG não se altera desde que a borda do objeto continue caindo dentro do mesmo bloco 8×8. Também é robusto a pequenas mudanças de ângulo de gradiente, desde que o ângulo continue caindo dentro do mesmo “caixote”.

Usando HOG e treinando *Boost* (conjunto de árvores de decisão) com 399 imagens de pedestres e 398 imagens de carros e classificando 525 imagens de pedestres e 634 imagens de carros, foi obtida taxa de acerto de 100%.

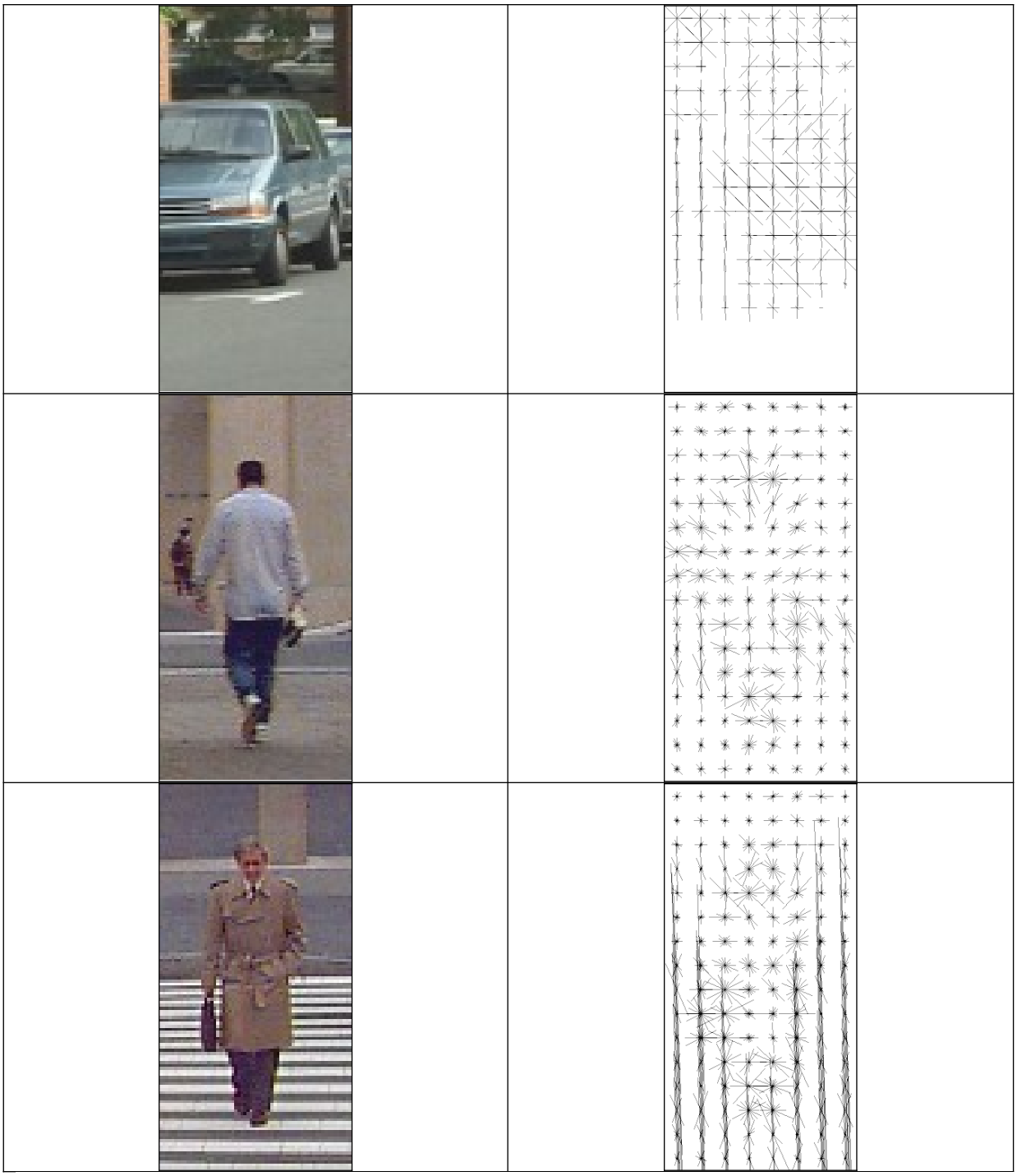


Figura 5: Imagens com respectivas histogramas de gradiente orientada (HOG), calculados em blocos 8x8, usando 8 “bins”.

Referências:

[Dalal2005] Dalal, Navneet, and Bill Triggs. "Histograms of oriented gradients for human detection." *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*. Vol. 1. IEEE, 2005.

[Viola2001] Viola, Paul, and Jones, Michael. "Rapid object detection using a boosted cascade of simple features." *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*. Vol. 1. IEEE, 2001.

[PSI3471 aula 10 – fim]