INSTITUTO FEDERAL
Goiano | Campus
Ceres

Instituto Federal Goiano - Campus Ceres
Bacharelado em Sistemas de Informação
Prof. Me. Ronneesley Moura Teles

Adallberto Lucena Moura
Andrey Silva Ribeiro
Anny Karoliny Moraes Ribeiro
Brener Gomes de Jesus
Davi Ildeu de Faria
Eduardo de Oliveira Silva
Gleyson Israel Alves
Gusttavo Nunes Gomes
Ianka Talita Bastos de Assis
Ígor Justino Rodrigues

Algoritmo Viola-Jones

Outubro
2017

Sumário

Algoritmo Viola-Jones

1 O artigo

Em 2001, *Paul Viola* e *Michael Jones*, dois pesquisadores de Cambridge publicaram uma artigo entitulado: "*Rapid Object Detection using a Boosted Cascade of Simple Features*" que demonstrava um novo método de detecção de faces. O artigo se diferencia e se paltava em 3 pontos importantes.

A primeira foi uma nova maneira de se representar uma imagem, a "imagem integral" (*Integral Image*, em inglês), que permitiu os detectores usados por eles, computarem a imagem de maneira mais rápida.

A segunda foi o algoritmo de aprendizado baseado no *AdaBoost*, que selecionava um número pequeno de características visuais críticas de um conjunto maior e com seus classificadores, extremamente eficientes.

O terceiro aspecto importante, foi o método de combinar e incrementar classificadores em "cascata", o que permitia regiões do fundo da foto de serem rapidamente descatadas, disponibilizando maior processamento computacional em posições com maior possíbilidade de ser o objeto no qual se está procurando, como um rosto.

Os autores aprofundaram seus métodos de como construiram esse algoritmo, de como funcionavam as equações e apresentaram os resultados encontrados e compararam com algoritmos similares da época, e como os algoritmos "*Rowley-Baluja-Kanade*", "*Schneiderman-Kanade*" e "*Roth-Yang-Ahuja*".

Devido sua implementação seguir uma abordagem diferente para construção de um sistema de detecção de face, aproximadamente 15 vezes mais rápida que os métodos anteriores, o algoritmo *Viola-Jones* revolucionou esse campo da computação se tornando uma referência.

2 Funcionamento do algoritmo

3 Vantagem

- 15 vezes mais rápido que o algoritmo "*Rowley-Baluja-Kanade*" no processamento da imagem.
- 600 vezes se comparado ao "*Schneiderman-Kanade*".

4 Desvantagem

- A detecção de faces, só é possível se o rosto estiver na posição frontal.
- A base de dados usada, precisa de faces em diferentes condições incluindo: iluminação, brilho, escala, pose e variações de câmera.
- Nível de detecção na literatura - 80% (FAUX,2012)
- É um algoritmo de detecção de face e não de reconhecimento facial.

5 Exemplos de Implementação

5.1 Python

`https://github.com/Simon-Hohberg/Viola-Jones`

5.2 Integral - Image

```python
import numpy as np


"""
In an integral image each pixel is the sum of all pixels in the original image
that are 'left and above' the pixel.

Original    Integral
+--------   +------------
| 1 2 3 .   | 0  0  0  0 .
| 4 5 6 .   | 0  1  3  6 .
| . . . .   | 0  5 12 21 .
            | . . . . . .

"""


def to_integral_image(img_arr):
    """
    Calculates the integral image based on this instance's original image data.
    :param img_arr: Image source data
    :type img_arr: numpy.ndarray
    :return Integral image for given image
    :rtype: numpy.ndarray
    """
    # an index of -1 refers to the last row/column
    # since row_sum is calculated starting from (0,0),
    # rowSum(x, -1) == 0 holds for all x
    row_sum = np.zeros(img_arr.shape)
    # we need an additional column and row
    integral_image_arr = np.zeros((img_arr.shape[0] + 1, img_arr.shape[1] + 1))
    for x in range(img_arr.shape[1]):
        for y in range(img_arr.shape[0]):
            row_sum[y, x] = row_sum[y-1, x] + img_arr[y, x]
            integral_image_arr[y+1, x+1] = integral_image_arr[y+1, x-1+1] + row_sum[y, x]
    return integral_image_arr


def sum_region(integral_img_arr, top_left, bottom_right):
    """
    Calculates the sum in the rectangle specified by the given tuples.
    :param integral_img_arr:
    :type integral_img_arr: numpy.ndarray
    :param top_left: (x, y) of the rectangle's top left corner
    :type top_left: (int, int)
    :param bottom_right: (x, y) of the rectangle's bottom right corner
    :type bottom_right: (int, int)
    :return The sum of all pixels in the given rectangle
    :rtype int
    """
    # swap tuples
    top_left = (top_left[1], top_left[0])
    bottom_right = (bottom_right[1], bottom_right[0])
    if top_left == bottom_right:
        return integral_img_arr[top_left]
    top_right = (bottom_right[0], top_left[1])
    bottom_left = (top_left[0], bottom_right[1])
    return integral_img_arr[bottom_right] - integral_img_arr[top_right]
    - integral_img_arr[bottom_left] + integral_img_arr[top_left]
```

recursos/codigo_python/Viola–Jones–master/violajones/IntegralImage.py

5.3 Haar Like Feature

```python
import violajones.IntegralImage as ii


def enum(**enums):
    return type('Enum', (), enums)

FeatureType = enum(TWO_VERTICAL=(1, 2), TWO_HORIZONTAL=(2, 1),
    THREE_HORIZONTAL=(3, 1), THREE_VERTICAL=(1, 3), FOUR=(2, 2))
FeatureTypes = [FeatureType.TWO_VERTICAL, FeatureType.TWO_HORIZONTAL,
    FeatureType.THREE_VERTICAL, FeatureType.THREE_HORIZONTAL,
    FeatureType.FOUR]


class HaarLikeFeature(object):
    """
    Class representing a haar-like feature.
    """

    def __init__(self, feature_type, position, width, height, threshold
    , polarity):
        """
        Creates a new haar-like feature.
        :param feature_type: Type of new feature, see FeatureType enum
        :type feature_type: violajonse.HaarLikeFeature.FeatureTypes
        :param position: Top left corner where the feature begins (x, y
    )
        :type position: (int, int)
        :param width: Width of the feature
        :type width: int
        :param height: Height of the feature
        :type height: int
        :param threshold: Feature threshold
        :type threshold: float
        :param polarity: polarity of the feature -1 or 1
        :type polarity: int
        """
        self.type = feature_type
        self.top_left = position
        self.bottom_right = (position[0] + width, position[1] + height)
        self.width = width
        self.height = height
        self.threshold = threshold
        self.polarity = polarity
        self.weight = 1

    def get_score(self, int_img):
        """
        Get score for given integral image array.
        :param int_img: Integral image array
        :type int_img: numpy.ndarray
        :return: Score for given feature
        :rtype: float
        """
        score = 0
        if self.type == FeatureType.TWO_VERTICAL:
            first = ii.sum_region(int_img, self.top_left, (self.top_left[0] + self.width, int(self.top_left[1] + self.height / 2)))
            second = ii.sum_region(int_img, (self.top_left[0], int(self.top_left[1] + self.height / 2)), self.bottom_right)
            score = first - second
        elif self.type == FeatureType.TWO_HORIZONTAL:
            first = ii.sum_region(int_img, self.top_left, (int(self.top_left[0] + self.width / 2), self.top_left[1] + self.height))
            second = ii.sum_region(int_img, (int(self.top_left[0] + self.width / 2), self.top_left[1]), self.bottom_right)
            score = first - second
        elif self.type == FeatureType.THREE_HORIZONTAL:
            first = ii.sum_region(int_img, self.top_left, (int(self.top_left[0] + self.width / 3), self.top_left[1] + self.height))
            second = ii.sum_region(int_img, (int(self.top_left[0] + self.width / 3), self.top_left[1]), (int(self.top_left[0] + 2 * self.width / 3), self.top_left[1] + self.height))
            third = ii.sum_region(int_img, (int(self.top_left[0] + 2 * self.width / 3), self.top_left[1]), self.bottom_right)
            score = first - second + third
        elif self.type == FeatureType.THREE_VERTICAL:
            first = ii.sum_region(int_img, self.top_left, (self.bottom_right[0], int(self.top_left[1] + self.height / 3)))
            second = ii.sum_region(int_img, (self.top_left[0], int(self.top_left[1] + self.height / 3)), (self.bottom_right[0], int(self.top_left[1] + 2 * self.height / 3)))
            third = ii.sum_region(int_img, (self.top_left[0], int(self.top_left[1] + 2 * self.height / 3)), self.bottom_right)
            score = first - second + third
        elif self.type == FeatureType.FOUR:
            # top left area
            first = ii.sum_region(int_img, self.top_left, (int(self.top_left[0] + self.width / 2), int(self.top_left[1] + self.height / 2)))
            # top right area
            second = ii.sum_region(int_img, (int(self.top_left[0] + self.width / 2), self.top_left[1]), (self.bottom_right[0], int(self.top_left[1] + self.height / 2)))
            # bottom left area
            third = ii.sum_region(int_img, (self.top_left[0], int(self.top_left[1] + self.height / 2)), (int(self.top_left[0] + self.width / 2), self.bottom_right[1]))
            # bottom right area
            fourth = ii.sum_region(int_img, (int(self.top_left[0] + self.width / 2), int(self.top_left[1] + self.height / 2)), self.bottom_right)
            score = first - second - third + fourth
        return score

    def get_vote(self, int_img):
        ”””
        Get vote of this feature for given integral image.
        :param int_img: Integral image array
        :type int_img: numpy.ndarray
        :return: 1 iff this feature votes positively, otherwise -1
        :rtype: int
        ”””
        score = self.get_score(int_img)
        return self.weight * (1 if score < self.polarity * self.threshold else -1)
```

recursos/codigo_python/Viola–Jones–master/violajones/HaarLikeFeature.py

5.4 AdaBoost

```python
from functools import partial
import numpy as np
from violajones.HaarLikeFeature import HaarLikeFeature
from violajones.HaarLikeFeature import FeatureTypes
import progressbar
from multiprocessing import Pool

LOADING_BAR_LENGTH = 50


# TODO: select optimal threshold for each feature
# TODO: attentional cascading

def learn(positive_iis, negative_iis, num_classifiers=-1, min_feature_width=1, max_feature_width=-1, min_feature_height=1, max_feature_height=-1):
    ”””
    Selects a set of classifiers. Iteratively takes the best classifiers based
    on a weighted error.
    :param positive_iis: List of positive integral image examples
    :type positive_iis: list[numpy.ndarray]
    :param negative_iis: List of negative integral image examples
    :type negative_iis: list[numpy.ndarray]
    :param num_classifiers: Number of classifiers to select, -1 will use all
    classifiers
    :type num_classifiers: int

    :return: List of selected features
    :rtype: list[violajones.HaarLikeFeature.HaarLikeFeature]
    ”””
    num_pos = len(positive_iis)
    num_neg = len(negative_iis)
    num_imgs = num_pos + num_neg
    img_height, img_width = positive_iis[0].shape

    # Maximum feature width and height default to image width and height
    max_feature_height = img_height if max_feature_height == -1 else max_feature_height
    max_feature_width = img_width if max_feature_width == -1 else max_feature_width

    # Create initial weights and labels
    pos_weights = np.ones(num_pos) * 1. / (2 * num_pos)
    neg_weights = np.ones(num_neg) * 1. / (2 * num_neg)
    weights = np.hstack((pos_weights, neg_weights))
    labels = np.hstack((np.ones(num_pos), np.ones(num_neg) * -1))

    images = positive_iis + negative_iis

    # Create features for all sizes and locations
    features = _create_features(img_height, img_width, min_feature_width, max_feature_width, min_feature_height, max_feature_height)
    num_features = len(features)
    feature_indexes = list(range(num_features))

    num_classifiers = num_features if num_classifiers == -1 else num_classifiers

    print('Calculating scores for images..')

    votes = np.zeros((num_imgs, num_features))
    bar = progressbar.ProgressBar()
    # Use as many workers as there are CPUs
    pool = Pool(processes=None)
    for i in bar(range(num_imgs)):
        votes[i, :] = np.array(list(pool.map(partial(_get_feature_vote, image=images[i]), features)))

    # select classifiers

    classifiers = []

    print('Selecting classifiers..')
    bar = progressbar.ProgressBar()
    for _ in bar(range(num_classifiers)):

        classification_errors = np.zeros(len(feature_indexes))

        # normalize weights
        weights *= 1. / np.sum(weights)

        # select best classifier based on the weighted error
        for f in range(len(feature_indexes)):
            f_idx = feature_indexes[f]
            # classifier error is the sum of image weights where the classifier
            # is right
            error = sum(map(lambda img_idx: weights[img_idx] if labels[img_idx] != votes[img_idx, f_idx] else 0, range(num_imgs)))
            classification_errors[f] = error

        # get best feature, i.e. with smallest error
        min_error_idx = np.argmin(classification_errors)
        best_error = classification_errors[min_error_idx]
        best_feature_idx = feature_indexes[min_error_idx]

        # set feature weight
        best_feature = features[best_feature_idx]
        feature_weight = 0.5 * np.log((1 - best_error) / best_error)
        best_feature.weight = feature_weight

        classifiers.append(best_feature)

        # update image weights
        weights = np.array(list(map(lambda img_idx: weights[img_idx] * np.sqrt((1-best_error)/best_error) if labels[img_idx] != votes[img_idx, best_feature_idx] else weights[img_idx] * np.sqrt(best_error/(1-best_error)), range(num_imgs))))

        # remove feature (a feature can't be selected twice)
        feature_indexes.remove(best_feature_idx)

    return classifiers


def _get_feature_vote(feature, image):
    return feature.get_vote(image)


def _create_features(img_height, img_width, min_feature_width, max_feature_width, min_feature_height, max_feature_height):
    print('Creating haar-like features..')
    features = []
    for feature in FeatureTypes:
        # FeatureTypes are just tuples
        feature_start_width = max(min_feature_width, feature[0])
        for feature_width in range(feature_start_width, max_feature_width, feature[0]):
            feature_start_height = max(min_feature_height, feature[1])
            for feature_height in range(feature_start_height, max_feature_height, feature[1]):
                for x in range(img_width - feature_width):
                    for y in range(img_height - feature_height):
                        features.append(HaarLikeFeature(feature, (x, y), feature_width, feature_height, 0, 1))
                        features.append(HaarLikeFeature(feature, (x, y), feature_width, feature_height, 0, -1))
    print('..done. ' + str(len(features)) + ' features created.\n')
    return features
```

recursos/codigo_python/Viola–Jones–master/violajones/AdaBoost.py

5.5 Utils

```
import numpy as np
from PIL import Image
from violajones.HaarLikeFeature import FeatureType
from functools import partial
import os


def ensemble_vote(int_img, classifiers):
    """
    Classifies given integral image (numpy array) using given classifiers, i.e.
    if the sum of all classifier votes is greater 0, image is classified
    positively (1) else negatively (0). The threshold is 0, because votes can be
    +1 or -1.
    :param int_img: Integral image to be classified
    :type int_img: numpy.ndarray
    :param classifiers: List of classifiers
    :type classifiers: list[violajones.HaarLikeFeature.HaarLikeFeature]
    :return: 1 iff sum of classifier votes is greater 0, else 0
    :rtype: int
    """
    return 1 if sum([c.get_vote(int_img) for c in classifiers]) >= 0 else 0


def ensemble_vote_all(int_imgs, classifiers):
    """
    Classifies given list of integral images (numpy arrays) using classifiers,
    i.e. if the sum of all classifier votes is greater 0, an image is classified
    positively (1) else negatively (0). The threshold is 0, because votes can be
    +1 or -1.
    :param int_imgs: List of integral images to be classified
    :type int_imgs: list[numpy.ndarray]
    :param classifiers: List of classifiers
    :type classifiers: list[violajones.HaarLikeFeature.HaarLikeFeature]
    :return: List of assigned labels, 1 if image was classified positively, else
    0
    :rtype: list[int]
    """
    vote_partial = partial(ensemble_vote, classifiers=classifiers)
    return list(map(vote_partial, int_imgs))


def reconstruct(classifiers, img_size):
    """
    Creates an image by putting all given classifiers on top of each other
    producing an archetype of the learned class of object.
    :param classifiers: List of classifiers
    :type classifiers: list[violajones.HaarLikeFeature.HaarLikeFeature]
    :param img_size: Tuple of width and height
    :type img_size: (int, int)
    :return: Reconstructed image
    :rtype: PIL.Image
    """
    image = np.zeros(img_size)
    for c in classifiers:
        # map polarity: -1 -> 0, 1 -> 1
        polarity = pow(1 + c.polarity, 2)/4
        if c.type == FeatureType.TWO_VERTICAL:
            for x in range(c.width):
                sign = polarity
                for y in range(c.height):
                    if y >= c.height/2:
                        sign = (sign + 1) % 2
                    image[c.top_left[1] + y, c.top_left[0] + x] += 1 * sign * c.weight
        elif c.type == FeatureType.TWO_HORIZONTAL:
            sign = polarity
            for x in range(c.width):
                if x >= c.width/2:
                    sign = (sign + 1) % 2
                for y in range(c.height):
                    image[c.top_left[0] + x, c.top_left[1] + y] += 1 * sign * c.weight
        elif c.type == FeatureType.THREE_HORIZONTAL:
            sign = polarity
            for x in range(c.width):
                if x % c.width/3 == 0:
                    sign = (sign + 1) % 2
                for y in range(c.height):
                    image[c.top_left[0] + x, c.top_left[1] + y] += 1 * sign * c.weight
        elif c.type == FeatureType.THREE_VERTICAL:
            for x in range(c.width):
                sign = polarity
                for y in range(c.height):
                    if x % c.height/3 == 0:
                        sign = (sign + 1) % 2
                    image[c.top_left[0] + x, c.top_left[1] + y] += 1 * sign * c.weight
        elif c.type == FeatureType.FOUR:
            sign = polarity
            for x in range(c.width):
                if x % c.width/2 == 0:
                    sign = (sign + 1) % 2
                for y in range(c.height):
                    if x % c.height/2 == 0:
                        sign = (sign + 1) % 2
                    image[c.top_left[0] + x, c.top_left[1] + y] += 1 * sign * c.weight
    image -= image.min()
    image /= image.max()
    image *= 255
    result = Image.fromarray(image.astype(np.uint8))
    return result


def load_images(path):
    images = []
    for _file in os.listdir(path):
        if _file.endswith('.png'):
            img_arr = np.array(Image.open((os.path.join(path, _file))), dtype=np.float64)
            img_arr /= img_arr.max()
            images.append(img_arr)
    return images
```

recursos/codigo_python/Viola–Jones–master/violajones/Utils.py

5.6 example

```
import violajones.IntegralImage as ii
import violajones.AdaBoost as ab
import violajones.Utils as utils

if __name__ == "__main__":
    pos_training_path = 'trainingdata/faces'
    neg_training_path = 'trainingdata/nonfaces'
    pos_testing_path = 'trainingdata/faces/test'
    neg_testing_path = 'trainingdata/nonfaces/test'

    num_classifiers = 2
    # For performance reasons restricting feature size
    min_feature_height = 8
    max_feature_height = 10
    min_feature_width = 8
    max_feature_width = 10

    print('Loading faces..')
    faces_training = utils.load_images(pos_training_path)
    faces_ii_training = list(map(ii.to_integral_image, faces_training))
    print('..done. ' + str(len(faces_training)) + ' faces loaded.\n\nLoading non faces..')
    non_faces_training = utils.load_images(neg_training_path)
    non_faces_ii_training = list(map(ii.to_integral_image, non_faces_training))
    print('..done. ' + str(len(non_faces_training)) + ' non faces loaded.\n')

    # classifiers are haar like features
    classifiers = ab.learn(faces_ii_training, non_faces_ii_training, num_classifiers, min_feature_height, max_feature_height, min_feature_width, max_feature_width)

    print('Loading test faces..')
    faces_testing = utils.load_images(pos_testing_path)
    faces_ii_testing = list(map(ii.to_integral_image, faces_testing))
    print('..done. ' + str(len(faces_testing)) + ' faces loaded.\n\nLoading test non faces..')
    non_faces_testing = utils.load_images(neg_testing_path)
    non_faces_ii_testing = list(map(ii.to_integral_image, non_faces_testing))
    print('..done. ' + str(len(non_faces_testing)) + ' non faces loaded.\n')

    print('Testing selected classifiers..')
    correct_faces = 0
    correct_non_faces = 0
    correct_faces = sum(utils.ensemble_vote_all(faces_ii_testing, classifiers))
    correct_non_faces = len(non_faces_testing) − sum(utils.ensemble_vote_all(non_faces_ii_testing, classifiers))

    print('..done.\n\nResult:\n      Faces: ' + str(correct_faces) + '/' + str(len(faces_testing))
          + '  (' + str((float(correct_faces) / len(faces_testing)) * 100) + '%)\n  non-Faces: '
          + str(correct_non_faces) + '/' + str(len(non_faces_testing)) + '  ('
          + str((float(correct_non_faces) / len(non_faces_testing)) * 100) + '%)')

```

```python
    # Just for fun: putting all haar-like features over each other generates a face-like image
    recon = utils.reconstruct(classifiers, faces_testing[0].shape)
    recon.save('reconstruction.png')
```

recursos/codigo_python/Viola–Jones–master/example.py

6 Referências Bibliográfica

VIOLA, P. e JONES, M. **Rapid object detection using a boosted cascade of simple features.** Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001, v. 1, p. I-511-I-518, 2001. Disponível em: <http://ieeexplore.ieee.org/document/990517/>.

IRGENS, Peter et al. **An efficient and cost effective FPGA based implementation of the Viola-Jones face detection algorithm.** HardwareX, v. 1, p. 68–75, 2017. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S2468067216300116>.

SANTOS, Ligneul. **Detecção de faces através do algoritmo de Viola-Jones**. Coppe/Ufrj, 2011.

FAUX, Francis e LUTHON, Franck. **Theory of evidence for face detection and tracking**. International Journal of Approximate Reasoning, v. 53, n. 5, p. 728–746, 2012. Disponível em: <http://dx.doi.org/10.1016/j.ijar.2012.02.002>.

BODHI, S. R. e NAVEEN, S. **Face detection, registration and feature localization experiments with RGB-D face database**. Procedia Computer Science, v. 46, n. Icict 2014, p. 1778–1785, 2015. Disponível em: <http://dx.doi.org/10.1016/j.procs.2015.02.132>.