using System;

using System.Diagnostics;

using Emgu.CV.Structure;

namespace Emgu.CV

{

/// <summary>

/// An object recognizer using PCA (Principle Components Analysis)

/// </summary>

[Serializable]

public class EigenObjectRecognizer

{

private Image<Gray, Single>[] \_eigenImages;

private Image<Gray, Single> \_avgImage;

private Matrix<float>[] \_eigenValues;

private string[] \_labels;

private double \_eigenDistanceThreshold;

/// <summary>

/// Get the eigen vectors that form the eigen space

/// </summary>

/// <remarks>The set method is primary used for deserialization, do not attemps to set it unless you know what you are doing</remarks>

public Image<Gray, Single>[] EigenImages

{

get { return \_eigenImages; }

set { \_eigenImages = value; }

}

/// <summary>

/// Get or set the labels for the corresponding training image

/// </summary>

public String[] Labels

{

get { return \_labels; }

set { \_labels = value; }

}

/// <summary>

/// Get or set the eigen distance threshold.

/// The smaller the number, the more likely an examined image will be treated as unrecognized object.

/// Set it to a huge number (e.g. 5000) and the recognizer will always treated the examined image as one of the known object.

/// </summary>

public double EigenDistanceThreshold

{

get { return \_eigenDistanceThreshold; }

set { \_eigenDistanceThreshold = value; }

}

/// <summary>

/// Get the average Image.

/// </summary>

/// <remarks>The set method is primary used for deserialization, do not attemps to set it unless you know what you are doing</remarks>

public Image<Gray, Single> AverageImage

{

get { return \_avgImage; }

set { \_avgImage = value; }

}

/// <summary>

/// Get the eigen values of each of the training image

/// </summary>

/// <remarks>The set method is primary used for deserialization, do not attemps to set it unless you know what you are doing</remarks>

public Matrix<float>[] EigenValues

{

get { return \_eigenValues; }

set { \_eigenValues = value; }

}

private EigenObjectRecognizer()

{

}

/// <summary>

/// Create an object recognizer using the specific tranning data and parameters, it will always return the most similar object

/// </summary>

/// <param name="images">The images used for training, each of them should be the same size. It's recommended the images are histogram normalized</param>

/// <param name="termCrit">The criteria for recognizer training</param>

public EigenObjectRecognizer(Image<Gray, Byte>[] images, ref MCvTermCriteria termCrit)

: this(images, GenerateLabels(images.Length), ref termCrit)

{

}

private static String[] GenerateLabels(int size)

{

String[] labels = new string[size];

for (int i = 0; i < size; i++)

labels[i] = i.ToString();

return labels;

}

/// <summary>

/// Create an object recognizer using the specific tranning data and parameters, it will always return the most similar object

/// </summary>

/// <param name="images">The images used for training, each of them should be the same size. It's recommended the images are histogram normalized</param>

/// <param name="labels">The labels corresponding to the images</param>

/// <param name="termCrit">The criteria for recognizer training</param>

public EigenObjectRecognizer(Image<Gray, Byte>[] images, String[] labels, ref MCvTermCriteria termCrit)

: this(images, labels, 0, ref termCrit)

{

}

/// <summary>

/// Create an object recognizer using the specific tranning data and parameters

/// </summary>

/// <param name="images">The images used for training, each of them should be the same size. It's recommended the images are histogram normalized</param>

/// <param name="labels">The labels corresponding to the images</param>

/// <param name="eigenDistanceThreshold">

/// The eigen distance threshold, (0, ~1000].

/// The smaller the number, the more likely an examined image will be treated as unrecognized object.

/// If the threshold is &lt; 0, the recognizer will always treated the examined image as one of the known object.

/// </param>

/// <param name="termCrit">The criteria for recognizer training</param>

public EigenObjectRecognizer(Image<Gray, Byte>[] images, String[] labels, double eigenDistanceThreshold, ref MCvTermCriteria termCrit)

{

Debug.Assert(images.Length == labels.Length, "The number of images should equals the number of labels");

Debug.Assert(eigenDistanceThreshold >= 0.0, "Eigen-distance threshold should always >= 0.0");

CalcEigenObjects(images, ref termCrit, out \_eigenImages, out \_avgImage);

/\*

\_avgImage.SerializationCompressionRatio = 9;

foreach (Image<Gray, Single> img in \_eigenImages)

//Set the compression ration to best compression. The serialized object can therefore save spaces

img.SerializationCompressionRatio = 9;

\*/

\_eigenValues = Array.ConvertAll<Image<Gray, Byte>, Matrix<float>>(images,

delegate(Image<Gray, Byte> img)

{

return new Matrix<float>(EigenDecomposite(img, \_eigenImages, \_avgImage));

});

\_labels = labels;

\_eigenDistanceThreshold = eigenDistanceThreshold;

}

#region static methods

/// <summary>

/// Caculate the eigen images for the specific traning image

/// </summary>

/// <param name="trainingImages">The images used for training </param>

/// <param name="termCrit">The criteria for tranning</param>

/// <param name="eigenImages">The resulting eigen images</param>

/// <param name="avg">The resulting average image</param>

public static void CalcEigenObjects(Image<Gray, Byte>[] trainingImages, ref MCvTermCriteria termCrit, out Image<Gray, Single>[] eigenImages, out Image<Gray, Single> avg)

{

int width = trainingImages[0].Width;

int height = trainingImages[0].Height;

IntPtr[] inObjs = Array.ConvertAll<Image<Gray, Byte>, IntPtr>(trainingImages, delegate(Image<Gray, Byte> img) { return img.Ptr; });

if (termCrit.max\_iter <= 0 || termCrit.max\_iter > trainingImages.Length)

termCrit.max\_iter = trainingImages.Length;

int maxEigenObjs = termCrit.max\_iter;

#region initialize eigen images

eigenImages = new Image<Gray, float>[maxEigenObjs];

for (int i = 0; i < eigenImages.Length; i++)

eigenImages[i] = new Image<Gray, float>(width, height);

IntPtr[] eigObjs = Array.ConvertAll<Image<Gray, Single>, IntPtr>(eigenImages, delegate(Image<Gray, Single> img) { return img.Ptr; });

#endregion

avg = new Image<Gray, Single>(width, height);

CvInvoke.cvCalcEigenObjects(

inObjs,

ref termCrit,

eigObjs,

null,

avg.Ptr);

}

/// <summary>

/// Decompose the image as eigen values, using the specific eigen vectors

/// </summary>

/// <param name="src">The image to be decomposed</param>

/// <param name="eigenImages">The eigen images</param>

/// <param name="avg">The average images</param>

/// <returns>Eigen values of the decomposed image</returns>

public static float[] EigenDecomposite(Image<Gray, Byte> src, Image<Gray, Single>[] eigenImages, Image<Gray, Single> avg)

{

return CvInvoke.cvEigenDecomposite(

src.Ptr,

Array.ConvertAll<Image<Gray, Single>, IntPtr>(eigenImages, delegate(Image<Gray, Single> img) { return img.Ptr; }),

avg.Ptr);

}

#endregion

/// <summary>

/// Given the eigen value, reconstruct the projected image

/// </summary>

/// <param name="eigenValue">The eigen values</param>

/// <returns>The projected image</returns>

public Image<Gray, Byte> EigenProjection(float[] eigenValue)

{

Image<Gray, Byte> res = new Image<Gray, byte>(\_avgImage.Width, \_avgImage.Height);

CvInvoke.cvEigenProjection(

Array.ConvertAll<Image<Gray, Single>, IntPtr>(\_eigenImages, delegate(Image<Gray, Single> img) { return img.Ptr; }),

eigenValue,

\_avgImage.Ptr,

res.Ptr);

return res;

}

/// <summary>

/// Get the Euclidean eigen-distance between <paramref name="image"/> and every other image in the database

/// </summary>

/// <param name="image">The image to be compared from the training images</param>

/// <returns>An array of eigen distance from every image in the training images</returns>

public float[] GetEigenDistances(Image<Gray, Byte> image)

{

using (Matrix<float> eigenValue = new Matrix<float>(EigenDecomposite(image, \_eigenImages, \_avgImage)))

return Array.ConvertAll<Matrix<float>, float>(\_eigenValues,

delegate(Matrix<float> eigenValueI)

{

return (float)CvInvoke.cvNorm(eigenValue.Ptr, eigenValueI.Ptr, Emgu.CV.CvEnum.NORM\_TYPE.CV\_L2, IntPtr.Zero);

});

}

/// <summary>

/// Given the <paramref name="image"/> to be examined, find in the database the most similar object, return the index and the eigen distance

/// </summary>

/// <param name="image">The image to be searched from the database</param>

/// <param name="index">The index of the most similar object</param>

/// <param name="eigenDistance">The eigen distance of the most similar object</param>

/// <param name="label">The label of the specific image</param>

public void FindMostSimilarObject(Image<Gray, Byte> image, out int index, out float eigenDistance, out String label)

{

float[] dist = GetEigenDistances(image);

index = 0;

eigenDistance = dist[0];

for (int i = 1; i < dist.Length; i++)

{

if (dist[i] < eigenDistance)

{

index = i;

eigenDistance = dist[i];

}

}

label = Labels[index];

}

/// <summary>

/// Try to recognize the image and return its label

/// </summary>

/// <param name="image">The image to be recognized</param>

/// <returns>

/// String.Empty, if not recognized;

/// Label of the corresponding image, otherwise

/// </returns>

public String Recognize(Image<Gray, Byte> image)

{

int index;

float eigenDistance;

String label;

FindMostSimilarObject(image, out index, out eigenDistance, out label);

return (\_eigenDistanceThreshold <= 0 || eigenDistance < \_eigenDistanceThreshold ) ? \_labels[index] : String.Empty;

}

}

}