Convolutional Neural Networks: In Search of Afterlife

Nasanbayar Ulzii-Orshikh, Final Project, 12/13/2019

MATH 215 Linear Algebra, Professor Tarik Aougab

Table of Contents:

Convolutional Neural Networks: In Search of Afterlife

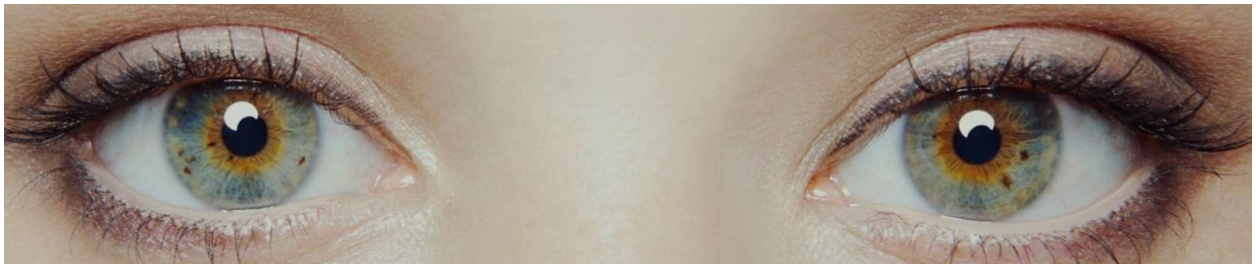


Figure 1. I Origins.

INTRODUCTION

Completely dazzled. Watching “I Origins” four years ago, I remember feeling exactly that by the beauty of our eyes, for the textures and patterns they show on the surface of our bodies and the meanings they entail. The movie theorizes that people who are reborn into new bodies have the same iris patterns as they did in their past ones, and researchers could scan the eyes of Earth’s population and potentially validate the hypothesis. Since then, I have been taking photographs of my friends’ eyes - collecting my “data set” - and now tried to write the code to see if the theory is true, since there are various modern computer vision techniques, including the Convolutional Neural Network (CNN) and other Machine Learning algorithms, that make it possible. Though the theory might sound naive, without touching on their practical applications, for instance, in diagnostics, I am inspired by it regardless. After all, it is not a coincidence that we have chosen an eye as our main subject. The human eye is, in fact, where the very story of Convolutional Neural Networks begins.

HISTORICAL BACKGROUND

In 1959, David Hubel and Torsten Wiesel published their paper on “Receptive fields of single neurons in the cat’s striate cortex” [1] that became the root concept behind CNNs. Their work differentiated two types of cells, which respond to edges, that are found in the brain's visual cortex: simple cells, whose receptive field has excitatory and inhibitory zones, and complex cells, whose receptive field is formed by summing and integrating the receptive fields of input simple cells and thus has no separate zones. Corresponding to this mechanism, Kunihiko Fukushima introduced in 1980 the concept of “neocognitron", which offers the two fundamental layers of CNNs [2]: convolutional and down-sampling. While the convolutional layer consists of units whose receptive fields cover a patch of the previous layer, the downsampling layers contain units whose receptive fields cover patches of previous convolutional layers. Ultimately, the construction of these two layers gave the vehicle for the further development of CNN algorithms, applied for various purposes, from audio recognition to computer vision [3].

MECHANISM BEHIND CONVOLUTIONAL NEURAL NETWORKS

Despite the mainstream rave about it, CNN, it turns out, is in fact a very complex architecture. Here is my shot at explaining it first vaguely and then for our specific case. As shown in the graph below, which depicts our network, the CCN starts with an input image, or, more likely, a set of input images, acting as a training set, in which each picture can be represented by a matrix, or a two-dimensional array data structure, and has a purpose of training the neural network to correctly recognize each image through its features.

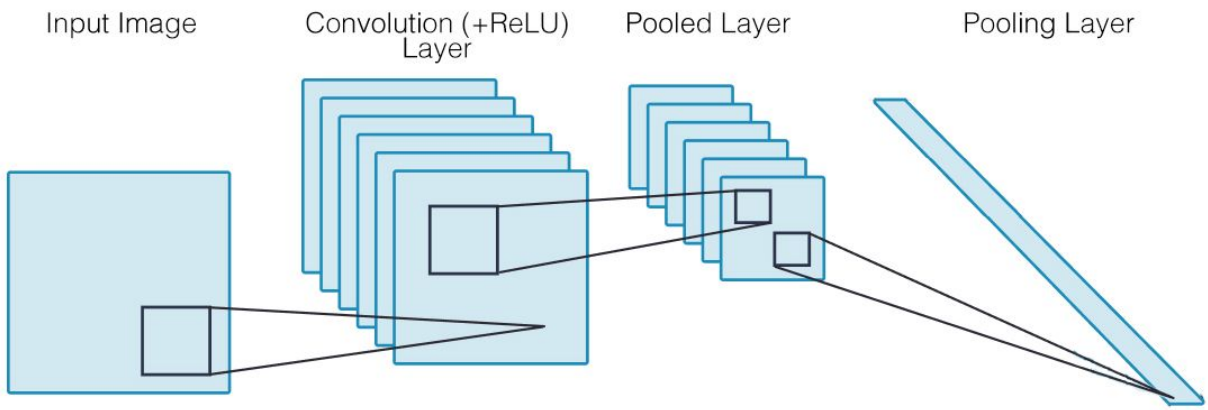


Figure 2. Convolutional Neural Network Graph.
(Source: Convolutional Neural Network Architecture: Forging Pathways to the Future)

To do so, we choose a set of relevant features, or kernels, we would like to categorize the images with, whether it is a patch of the input image that has a hand or a cloud in it [4]. Using the set of features, we then perform a convolution by dot-producting between the input image and the kernel, building a convolution layer, consisting of feature maps, which seem to carry the information of the presence of that specific feature in the input image (Figure 3) [5]. Depending on the feature map's pixel values and conditions, we run their matrices through a Rectified Linear Activation Function, where, mainly, each negative pixel value becomes zero and the positive ones are not touched: since "rectified linear units are nearly linear, they preserve many of the properties that make linear models easy to optimize with gradient-based methods" [6]. Then, the feature maps' matrices are pooled, creating a smaller matrix, whose units represent the maximum or the average value of each small patch of the feature map (Figure 4) [5]. This reduction in size not only makes the algorithm run faster, but also allows the network to disregard small positional variances in the input images [7]. Then, each matrix in the pooled layer is vectorized, or transformed into a feature vector [4], each representing one input image and whose dimension depends on how many features we use in convolution and what the area density of pooling is. Now that for each image we have a vector in a vector space, we can perform a simple function of finding the shortest distance between the existing vectors and the new input image's feature vector and display its closest match from our existing data set of iris images.

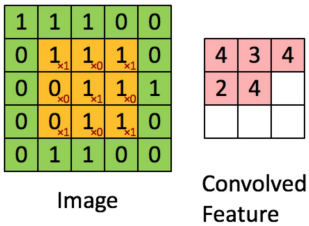


Figure 3. Convolution Dot-Product.

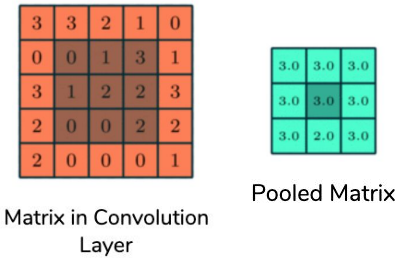


Figure 4. Pooling of max values.

CASE STUDY: *IRIS FORM*

Soliciting different resources, I implemented the code for CNN in my repository Iris_Form in GitHub, and this section thoroughly goes through the building process as well as the rationalizations behind it with direct excerpts from the code, the continuous version of which can be found in the attachments.

The training dataset for CNN consisted of five distinct iris images with various textures and brightness patterns. From each eye, I extracted a 24x24 dimensional pattern, which intuitively seemed to help differentiate each iris from the rest of the images and thus could be used as features, or kernels, for the convolution layer, allowing the network to recognize the irises.

Figure Collection 1. Training Dataset.

Figure Collection 2. Features, or Kernels.

Stage 1. Pre-Processing an Image.

Although we have the luxury of engaging with rich color data of RGB channels, to first learn how to recognize iris patterns, I decided to clear out the colors as a way of simplifying the calculations and focusing on texture. I had two choices: black and white or greyscale. Since greyscaling assigns a shade between black and white to each pixel instead either 1 or 0, intuitively, it seemed that it preserves more information both on the level of each pixel, namely its level of luminosity, and of relative difference in luminosity between pixels. Thus, I first greyscaled the images using the Pillow library in Python [8].

```
def orig_to_greysc():
    name_value = 1
    # For each file in the Traning Dataset folder, opens and greyscales the image. Then, saves it.
    for filename in glob.glob('Iris_Training_Dataset_Orig/*.JPG'):
        im = Image.open(filename)
        gs_im = im.convert('L')
        gs_im.save('Iris_Training_Dataset_GS/Image_GS_{}.JPG'.format(str(name_value)))
        name_value += 1
```

Figure 5. Training Input Image.

Figure 6. Training Input Image Greyscaled.

From the linear algebra perspective, this transforms the image to have pixels with the same three values for each of the RGB channels. That is, in a two dimensional array of pixels, each pixel-array of three values could now be represented by a single value. Hence, the two dimensional array of pixels could become an array of numbers, making it much easier to perform convolution and pooling on them.

Row before greyscaling:

```
[[[246 184 137]
  [243 181 134]
  [242 178 132]
  ...
  [ 84  46  27]
  [ 74  33  15]
  [ 78  37  17]]
```

Row after greyscaling:

```
[[197 194 191 ...  55  43  46]
```

In addition, since we are mainly focusing on the iris patterns and the images have various dimensions, they were cropped to 400x296 to include only the iris region. To make the algorithm run faster and allow more constructive iterations for myself, I decided to reduce the quality of each input picture by 50% [9].

```
def qual_reduction():
    #Reduces the quality of a picture by 50% in the cropped Training Dataset folder.
    name_value = 1
    for filename in glob.glob('Iris_Training_Dataset_GS_Cropped/*.JPG'):
        im = Image.open(filename)
        im.save('Iris_Training_Dataset_GS_CrandRedQ/Image_GS_{}.JPG'.format(str(name_value)), quality=50)
        name_value += 1
```

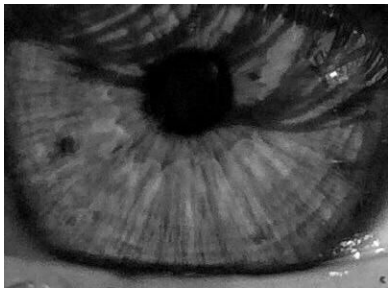


Figure 6. Cropped to 400x296. 23 KB.

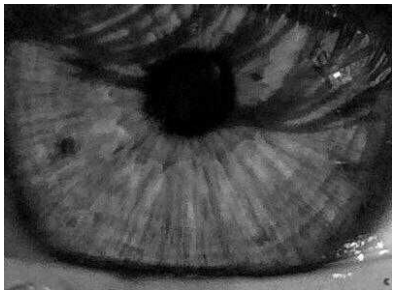


Figure 7. Still 400x296, but reduced quality. 14 KB.

Stage 2. Reduction and Reconstruction of an Image.

Once the image is greyscaled, I wanted to manually work with the greyscale array of pixels to transform it into an array of number that is easy to work with mathematically. The following method does exactly that, returning a matrix of numbers from a greyscale image.

```
def RGB_reduction(image):
    ''' Returns a matrix, where each pixel is represented by one number from the greyscale. '''
    # Checks if we are working with arrays.
    if not isinstance(image, np.ndarray):
        # If not, opens the image in an array type.
        opened_img = cv2.imread(image)
        opened_img = np.asanyarray(opened_img)
    else:
        opened_img = image
    # Checks if the image greyscaled.
    if opened_img[0][0][0] == opened_img[0][0][1] and opened_img[0][0][0] == opened_img[0][0][2]:
        # If they are, creates a new matrix (pixel -> number) without the first and second by index channel values,
        # since each pixel has the same three values for each of the RGB channels.
        new_image_matrix = []
        row_numb = 0
        # The construction of the new matrix is made row by row.
        for row in opened_img:
            blank_row = []
            # Pixel by pixel, or number by number.
            for pixel in opened_img[row_numb]:
                new_pixel = pixel[0]
                blank_row.append(new_pixel)
            new_image_matrix.append(blank_row)
            row_numb += 1
    # If not greyscaled, raises a TypeError.
    else:
        raise TypeError("Pixels must be greyscaled first.")
    # Returns the final matrix.
    return new_image_matrix
```

Correspondingly, there must be a method that does the opposite -- that is, reconstructs a greyscaled image from a matrix in order to flexibly convert images to matrices and back.

```
def image_from_array_reconstruction(matrix):
    # Pass on a matrix, and returns an image.
    blank_matrix = []
    # The construction is done row by row.
    for row in matrix:
        row_count = 0
        blank_row = []
        # Pixel by pixel.
        for i in row:
            pixel = [i, i, i]
            blank_row.append(pixel)
        row_count += 1
        blank_matrix.append(blank_row)
    img = np.asarray(blank_matrix).astype('uint8')
    img = Image.fromarray(img)
    return img
```

Stage 3. Convolution and ReLU Layers.

At this stage, the algorithm would perform a convolution between features and the training images to create matrices that represent how much of the features these pictures entail in themselves. In terms of matrices, a convolution is performed by dot-producting the feature matrix with small patches of the

training image, advancing one column at a time until the entire area is covered. The amount of advancing is called a stride, and in this case, I chose it to be one to maximize the amount of information I get from already small images [10]. One of the downsides of this could be the proneness to error due to a small positional variance. For example, if the image is fused and the pixels are slightly moved in random directions, the network might be more prone to making an erroneous guess, since its sensitivity to the pixels' relative positions seem to be maximized due to small strides.

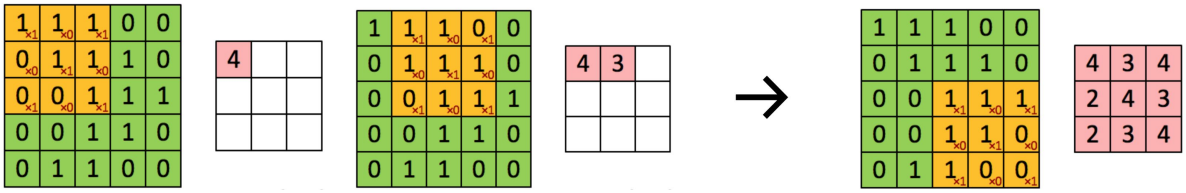


Figure 8. Scheme of the convolutions with a stride = 1 [5].

Another issue that we face with this convolution is the reduced dimension and loss of information of pixels closer to the edges of the image. Since our feature matrix is 24x24 and the dot-product is taken up until the edges of the image and feature matrix meet, the matrix in the convolution layer has 23 pixels (377x273) less in both width and height than the one originally used for convolution. One way to fix this is padding [10]. By adding zeros onto the edges of the main matrix, we allow a bigger room for the convolution operations, thus preserving the original dimensions of the image. However, since the iris pictures generally have irrelevant for the patterns regions such as lower eyelids and eyelash shadows around the edges, the convolution layer is created without padding.

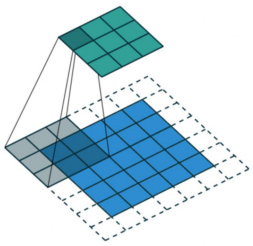


Figure 9. Convolution with padding [5].

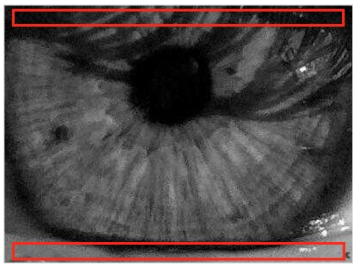


Figure 10. Irrelevant for the patterns regions.

Since we have five features, executing a convolution with each of the five training images will result in 5 new matrices. Hence, the convolution matrix consists of 25 matrices. As the following algorithm takes one matrix and 5 features as parameters, its output list will be a list of 5 feature maps:

```
def convolution_layer(matrix, features_folder):
    # Creates an empty list to hold the images of the features.
    features_list = []
    # Takes each feature, RGB-reduces it, and adds the resulting matrix to the features list.
    # Save sthe feature for future reference as well.
    for item in glob.glob(features_folder + '/*.JPG'):
        feature_img = cv2.imread(item)
        feature_matrix = RGB_reduction(feature_img)
        features_list.append(feature_matrix)
        feature_matrix = image_from_array_reconstruction(feature_matrix)
        feature_matrix.save('feature_matrix1.JPG')
    # Creates an empty list to hold the matrices of the convolution layer.
    output_list_of_matrices = []
    # Performs a convolution between each of the features and the main matrix using the previously defined
    # convolve() method and adds it to the output list.
    for feature_matrix in features_list:
        convolved_matrix = convolve(matrix, feature_matrix)
        output_list_of_matrices.append(convolved_matrix)
    return output_list_of_matrices
```

Convolve() as a helper method, in this case, will take a feature matrix on one hand and a grayscale image RGB reduced matrix on the other to perform a convolution. To both explore existing libraries and manually interact with matrices myself, I use the already existing convolve2d from scipy library in convolution layer and manually code similar operations for matrices for the pooled layer.

```
def convolve(matrix_1, matrix_2):
    # Convolve() as a helper method that takes two matrices and performs a convolution between them.
    array_1 = np.array(matrix_1)
    array_2 = np.array(matrix_2)
    convolved_matrix = convolve2d(array_1, array_2, 'valid')
    return convolved_matrix
```

For the ReLU layer, which turns the negative values in the matrix to zero, since the features are taken straight from the images, and thus their matrices have positive values only, there is no urgent need to apply the ReLU layer. In fact, if we plot the values of a feature map, it would correspond to the same exact plot that ReLU layer would output when applied to matrices. That is, “the properties that make linear models easy to optimize with gradient-based methods” [6] is already preserved by the feature maps.

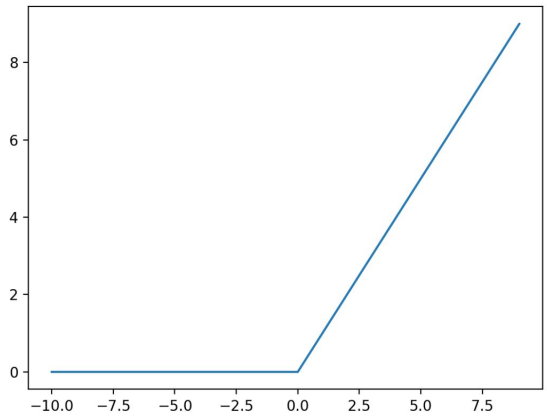


Figure 11. Line Plot of Rectified Linear Activation for Negative and Positive Inputs [6].

Stage 4. Pooling Layer.

The main goal of the pooling layer is to transform the matrix in the convolution layer to be more insensitive to slight variances in the pixels [7]. In contrast to other methods, the Pooling Layer method takes in a dimension as its parameter. This is the size of the square matrix that would be projected onto the convolution layer to calculate either the maximum and or the average value of the units in the receptive field. In this case, it made sense to take 13 as the dimension since it is the greatest common divisor between 377 and 273, since such a matrix would fully cover the entire feature map without leaving any units. Based on this concept, the method itself translates through the matrix with a period of the dimension, creating the pooling layer.

A decision still needed to be made when it came to taking the average or the max. Despite such aspects of noise in the data, it seemed meaningful to take the average instead of the max because the values of the matrices are all positive and exist within the range of 0-255. Thus, not only is it likely that there will certainly be a positive value in the receptive field of a pooled layer, but also those values were likely to be at around 200-255, since it takes only one value to be in that range for the entire unit of the matrix in the pooling layer to be in that same range. In contrast, taking the average, although the outliers in 200-255 skew it, allows a wider range for those units to take as well as serves as a representation that is more reflective of all the positive values that it averages, thus preserving more valuable information about the input matrix.

```
def pooling_layer(matrix, dimension):
    # Creates a blank_matrix, which will be the output matrix that is formed from pooling.
    blank_matrix = []
    # The nested loop makes use of the dimension to translate by through the matrix with a period that
    # is equal to it. Hence, we add a dimension to the counters each time we iterate through the loop.
    counter_1 = 0
    for i in range(int(len(matrix)/dimension)):
        blank_row = []
        counter_2 = 0
        # Similar to other methods, it creates the matrix row by row and value by value by taking the mean.
        for j in range(int(len(matrix[0])/dimension)):
            blank_row.append(int(np.mean(matrix[counter_1:(counter_1 + dimension), counter_2:(counter_2 + dimension)])))
            counter_2 += dimension
        blank_matrix.append(blank_row)
        counter_1 += dimension
    return blank_matrix
```

Stage 5. Matching in a Vector Space.

Now that we have 25 reduced matrices in the pooling layer, we can create a long vector that joins all of their columns in order by using a vectorize method for multiple matrices, which builds on a method for vectoring one matrix:

```
def vectorize_multiple_matrices(a_list):
    # By using the vectorize_matrix() helper method, takes in a list of matrices,
    # creates vectors by adding their columns one by one,
    # and then joins those vectors into a one long vector.
    blank_vector = []
    for matrix in a_list:
        micro_blank_vector = vectorize_matrix(matrix)
        blank_vector.extend(micro_blank_vector)
    return blank_vector
```

```
def vectorize_matrix(matrix):
    # Takes in a matrix and creates a vector by adding its columns one by one.
    blank_vector = []
    matrix = np.asanyarray(matrix)
    for i in range(len(matrix[0])):
        # In doing so, it uses the slicing in numpy arrays and the extend method, which takes in a list and
        # connects it to the existing list (while append adds a list into a list).
        blank_vector.extend(matrix[:, i])
    return blank_vector
```

By doing this, we are able to represent each input image as a feature vector in a vector space, the dimension which is dependent on the number of features (5) and rows and columns each matrix in the pooling layer has (21 rows and 29 columns). In this case, the vector space will thus be 5x21x29 = 3045-dimensional. Here is what the creation of the vector space looked like:

```
def vectors_in_vspace(training_dataset_address, identity_texts):
    # A list to hold all the feature vectors.
    vectors_in_vspace = []
    # For each image in the greyscale training dataset:
    counter = 1
    for file in glob.glob(training_dataset_address + '/Image_GS_*.JPG'):
        print("'{}/Image_GS_{}.JPG' being processed.".format(training_dataset_address, counter))
        # Opens the image.
        image_1 = cv2.imread(training_dataset_address + '/Image_GS_{}.JPG'.format(counter))
        # Creates an array from it.
        image_1_array = np.asarray(image_1)
        # RGB-reduces it into a matrix of values, since the image is already greyscale.
        image_1_RGB_reduced = RGB_reduction(image_1_array)
        # Performs convolution between each of the features and the matrix. Adds to the convolution layer.
        image_1_convolved = convolution_layer(image_1_RGB_reduced, features_folder='Features_for_Convolution')
        # Pools all the matrices in the convolution layer and adds the resulting smaller matrices to a pooled_list.
        pooled_list = []
        for item in image_1_convolved:
            pooled_list.append(pooling_layer(item, 13))
        # Creates a feature vector representing the training image.
        image_vector = vectorize_multiple_matrices(pooled_list)
        # Adds it to the vector space.
        vectors_in_vspace.append(image_vector)
        print("Iteration {} successfully completed.".format(str(counter)))
        counter += 1
    # Creates an empty list for the vectors, but adds in this case a tuple, consisting of an ID number that corresponds
    # to the training image, the feature vector itself, and the identifying text related to the person behind the iris.
    vectors_in_vspace_id = []
    id_numb = 0
    for vector in vectors_in_vspace:
        vectors_in_vspace_id.append((id_numb, vector, identity_texts[id_numb]))
        id_numb += 1
    # Sorts the list of tuples in case it is unordered, which is easily done, since the tuples start integers.
    vectors_in_vspace_id.sort()
    return vectors_in_vspace_id
```

As we place all the pictures within this vector space using their feature vectors, for a test image, we can simply find its corresponding feature vector as well and compare which of the existing vectors has the shortest distance from the test vector. The one that has the minimum distance must be either the absolute or closest match from the existing feature vectors. As we place huge-dimensional vectors into the vector space, we need to make sure any comparison or operation we do on them must have the smallest time complexity:

- np.linalg.norm(), which is used to find the distance between two vectors, has a time complexity relatively smaller than other methods [11].
- Once we find and store the distances with vectors in the dictionary, we can access the ID of a vector with the shortest distance at O(1) time complexity.

```
def find_match(vectors_in_vspace_id, some_vector):
    # Creates an empty list to store all the distances from the test feature vector to all the other existing ones.
    distance_list = []
    # In order to be able to access back the information about to whom a certain iris
    # that has the least distance form the test vector belongs, creates a dictionary
    # that has the distances as keys and vector IDs as the values.
    navig_dict = {}
    # For each of the vector in the vector space, accesses the vector's value in the list of tuples vectors_in_vspace_id
    # and finds the distance between that vector and the test one.
    for i in range(len(vectors_in_vspace_id)):
        dist = find_dist(vectors_in_vspace_id[i][1], some_vector)
        # Then, it adds the distance and its corresponding feature vector into the dictionary.
        navig_dict[dist] = vectors_in_vspace_id[i][0]
        # Adds the distance value into the distance_list.
        distance_list.append(dist)
    # Finds the minimum value in the distance_list.
    min_dist = min(distance_list)
    print("\nBest Match at a Distance Score: " + str(min_dist))
    # Accesses the corresponding ID of the image that gives the least distance.
    min_dist_id = navig_dict[min_dist]
    # Returns the tuple of the relevant vector by using min_dist_id as an index,
    # since the vectors are placed into the vector space, having
    # an ID that directly corresponds to their index/order.
    return vectors_in_vspace_id[min_dist_id]
```

```
def find_dist(v1, v2):
    # Helper method that takes two vectors and find the distance between them using np.linalg.norm().
    v1_t_a = np.array(tuple(v1))
    v2_t_a = np.array(tuple(v2))
    return np.linalg.norm(v1_t_a-v2_t_a)
```

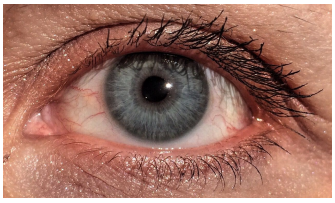
TESTING

There are four hypotheses I formulated that the algorithm, I think, should confirm for us to be sure in its accuracy:

1. **Ground-check:** If the test image is one of the images in the training set, the shortest distance, or the Distance Score, is zero, and it should match the corresponding (same) person.
2. **Modification-check:** If the test image is an image of the same human iris as in the Ground-check but taken from a slightly different angle and conditions, the Distance Score is not zero, but its closest match must be the right (same) person.
3. **Threshold-check:** If the test image is of somebody else's iris, not existent in the training set, yet matches the same person as in the previous hypothesis, the Distance Score must be bigger than the one in the previous hypothesis.
4. **Consistency-check:** If the test image is of somebody else's iris that clearly looks to the human eye the most similar to a specific one of the training images, then its closest match must be that person's iris.

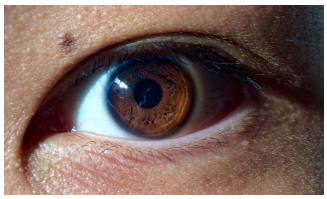
Training Dataset with their Identificator Labels:

Name: Anna Tenikhina
Age: 20
Location: Moscow, Russia
Image Date: Jan 20, 2016



Name: Alexandra Yuriyevna
Age: 42
Location: Saint Petersburg, Russia
Image Date: Jan 17, 2016

Name: Maria Last_Name_Unknown
Age: 18
Location: Moscow, Russia
Image Date: Jan 11, 2016

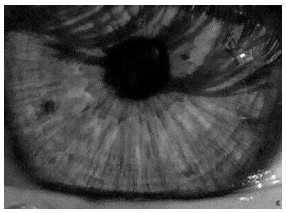


Name: Battemuulen Naranbat
Age: 19
Location: Amsterdam, Netherlands
Image Date: Jan 13, 2016

Name: Anastasiya Bukryeyeva
Age: 21
Location: Moscow, Russia
Image Date: Jan 11, 2016

Test for H1 - Ground-check:

Suppose that the test image is one of the images in the training set.

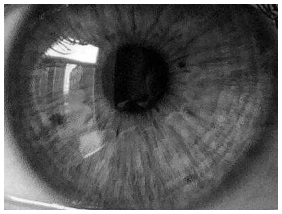


```
Best Match at a Distance Score: 0.0
Name: Anastasiya Last_Name_Unknown
Age: 21
Location: Moscow, Russia
Image Date: Jan 11, 2016
```

H1 Accepted: the Distance Score is zero and matches the correct person.

Test for H2 - Modification-check:

Suppose it is the same person as in H1 but a different condition.

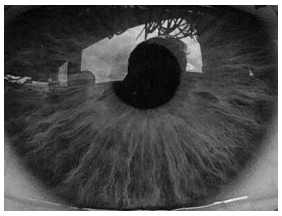


```
Best Match at a Distance Score: 436.0389890823985
Name: Anastasiya Bukryeyeva
Age: 21
Location: Moscow, Russia
Image Date: Jan 11, 2016
```

H2 Accepted: the Distance Score is not zero (≈436.0) and matches the correct person.

Test for H3 - Threshold-check:

Suppose it is a different person, not existent in the training set, yet matches the same person as in the previous hypothesis.



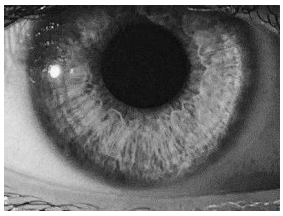
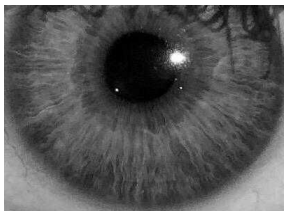
```
Best Match at a Distance Score: 439.94772416731513
Name: Anastasiya Bukryeyeva
Age: 21
Location: Moscow, Russia
Image Date: Jan 11, 2016
```

H3 Accepted: while the condition of a different person's iris being matched to an iris of the person in the previous hypothesis is met, its Distance Score (≈439.9) is bigger than that of the H2 case (≈436.0).

Since the images are consistent in the sense that the image of a different person' iris has a bigger Distance Score, I could put a **threshold** somewhere between 436.0 and 439.0, above which the iris would be considered as somebody else's, or unique.

Test for H4 - Consistency-check:

Suppose that the test image is of somebody else's iris that clearly looks the most similar to a specific iris out of the training images. The left picture below is the test image, while the image to the right seemed to be the one that is the most similar to it relatively to all the other images in the training dataset, considering the general uniformity of the texture and brightness.

```
Best Match at a Distance Score: 441.89704683330933
Name: Alexandra Yuriyevna
Age: 42
Location: Saint Petersburg, Russia
Image Date: Jan 17, 2016
```

H4 Accepted: its closest match is the hypothesized person's iris.

Interestingly, the Distance Score is ≈441.9 -- that is, higher than both of the previous cases. This means that if we indeed choose a threshold between ≈436.0 and ≈439.9, above which an iris is considered to be unique to the training data set, the case above would be both qualitatively and quantitatively **consistent** with the knowledge system of the algorithm.

NEXT STEPS

1. **Algorithm determines kernels for itself.** What makes me sure that the features I have chosen are the optimal way of differentiating the images? Visual systems of a biological body and human-constructed machines are fundamentally different in nature, despite the latter taking its roots from the former one. Even there, the ground of knowledge that humans possess can be thrown into doubt -- while we cannot imagine being incorrect about the knowledge that we are so sure is correct, we still could imagine coming to being incorrect about it. Without such foundation, knowledge must be a dynamic system of claims and inferences, so how much should the computer approach the system of human knowledge? That being said, to make sure the algorithm leverages on its own abilities to train itself, I think it would make sense for me as well as I am curious to further explore how the very algorithm can come up with significant features and self-feed to make better recognitions.

2. **Further training for knowledge depth.** The network was modeled by how the film uses these algorithms to scan an iris once the person is born and occasionally updating the data when they scan again in the span of their lifetime. While it might make sense economically, such frequency still leaves room for inaccuracy in recognizing irises. Using algorithms that create minor alterations in the same image, we could capitalize on one set of data to further train the model, adding a depth to its recognition abilities and optimizing its degree of variance.

3. **Transparency and understandability.** While an extensive training increases accuracy, we would also want to see what exactly the algorithm looks like inside and what the features are that it uses, so that we can preserve a deliberate approach in implementing and utilizing Machine Learning algorithms for practical applications.

BIBLIOGRAPHY

[1] Eric R Kandel, “An introduction to the work of David Hubel and Torsten Wiesel”, National Institutes of Health, 2009.

[2] Kunihiko Fukushima, “Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position”, Biological Cybernetics, 1980.

[3] Rostyslav Demush, “A Brief History of Computer Vision (and Convolutional Neural Networks)”, Hackernoon, 2019.

[4] Goodfellow, Bengio, & Courville, “Deep learning”, The MIT Press, 2016.

[5] Sumit Saha, “A Comprehensive Guide to Convolutional Neural Networks”, Medium, 2018.

[6] Jason Brownlee, “A Gentle Introduction to the Rectified Linear Unit (ReLU)”, Machine Learning Mastery, 2019.

[7] Jason Brownlee, “A Gentle Introduction to Pooling Layers for Convolutional Neural Networks”, Machine Learning Mastery, 2019.

[8] Simplilearn, Convolutional Neural Network Tutorial (CNN) | How CNN Works | Deep Learning Tutorial, YouTube, 2018.

[9] Pillow (PIL Fork), “Image Module”, 2016.

[10] Jason Brownlee, “A Gentle Introduction to Padding and Stride for Convolutional Neural Networks”, Machine Learning Mastery, 2019.

[11] https://stackoverflow.com/questions/7741878/how-to-apply-numpy-linalg-norm-to-each-row-of-a-matrix