Assignment 2

Making an eye tracker.

IAML 2020

Note: Please read the whole assignment carefully before attempting to it.

The purpose of this assignment is to give you practical experience with the use of computer vision and simple machine learning techniques. The goal is to create an eye tracker. You have to evaluate your solutions using the methods learned so far and make decisions based on your results. It consists of three exercises, each focusing on a specific part of the process. We suggest you solve one exercise per week, thereby distributing the workload.

1. **Pupil and glint detection (week 1):** You will implement methods to detect pupils and glints in images of eyes.
2. **Simple gaze estimation (week 2):** You will use the pupil detector to implement a gaze estimation system.
3. **Improvements (week 3):** Finally, you have to choose to implement one additional functionality to the system.

We expect that an average student need 10 hours to solve these exercises, although this might vary. This time is in addition to time used on lectures, reading, and exercises. This is why it is important that you use the resources available to you to get the help you need. Don't hesitate to contact us by email or on LearnIt - we are here to help you.

Hand-in

At the end of the course you have to hand in all the assignments combined. You have to hand in the following elements at the end of the course:

- **Code:** You have to hand in the complete source code used for the assignments **including any files provided by us**. Use a separate folder for each assignment. This makes it easy for us to test whether your implementations actually works. Upload the code as a **zip** archive (don't use rar).
- **Report:** You must combine your writings for each part into a single report document in **pdf** format. (Self contained and reproducible) The reports for each part should be approximately 2-3 pages in length (normal pages of 2400 characters). Use clear formatting for exam parts and exercises. Report structure and content requirements is listed for each exercise.
- **Results:** Include all pictures, videos, and other files produced in a separate **zip** archive. Use a separate folder for each assignment.

Requirements

A number of minimum requirements to pass the exam. To make these as clear as possible, we mark tasks with one of the following symbols:

- The task is required.
- You have to implement at least one of the specified tasks.
- The task is entirely optional. You should only focus on these after you completed the other tasks.
- M The task is required for master students but is of course allowed to be attempted by bachelor students as well.

These are content requirements and not a guarantee for passing. The quality of your answers will also be assessed. The final assessment is based on an overall evaluation of your assignment. By solving optional tasks, your assignment as a whole might get an overall better assessment. Do not expect that you will always get optimal results. Neither should you focus on optimising your code for speed unless explicitly specified. The assignments are generally open ended and we encourage you to improve your solutions as much as possible.

For questions regarding the general course requirements and differences between bachelor and master studens, refer to the official course descriptions.

Guidelines

We provide a few general guidelines here.

- **Be as precise as possible:** Be as precise as possible when explaining your approach, implementation, and results.
- **Reproducibility:** Your results should be reproducible from the report description alone.
- **Use the theory:** Relate your findings to theoretical concepts from the course when relevant. Don't repeat the theory unless specified, use references instead.
- **Remember that you are learning:** We don't expect you to be fluent in all the material. Solve the tasks to the best of your abilities and write what you did so others can reproduce your results from the descriptions.

Overview

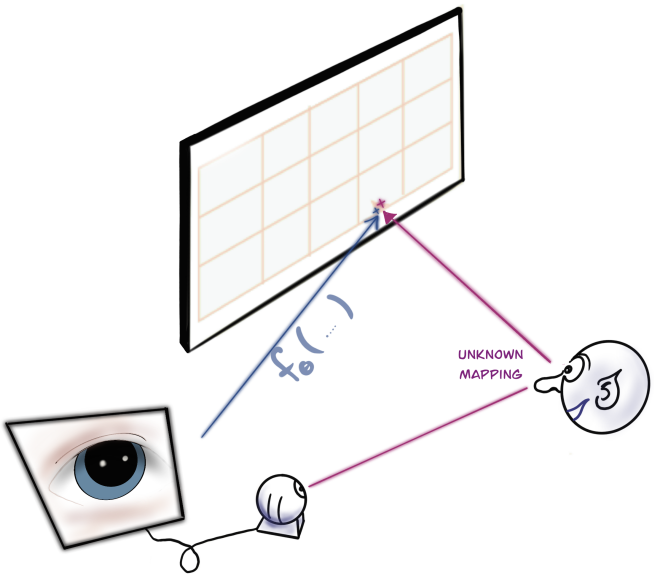

Figure 1: Visualisation of the gaze estimation task. The eye, which is directed at a specific point on the screen is captured by the camera at some unknown position. The two red lines represent this unknown function. We instead learn a function $f_\theta(x, y)$ with parameters θ which map pupil coordinates in images captured by the camera to screen coordinates.

An eye tracker refers to a system including camera, IR light sources and software that detects where the user is looking (gaze). The setup used in this assignment is shown in Figure 1. Without getting into too much detail, the gaze is directly related to the relative position of the screen, eye orientation, and camera orientation. When looking at an object, the person orients the eye so that the light of the object falls onto a small region on the retina, called the *Fovea* (shown in Figure 1). The position of the fovea varies between people and has to be determined by calibration (See Figure 2). In these exercises we will use the pupil to determine the rotation of the eye and implicitly learn the direction of attention (called the visual axis) by calibration. Eye tracking itself is not part of the course curriculum so don't worry if this seems like some strange and confusing facts. The tasks should be relatively intuitive once you get started. We provide all the data you need for calibration and testing and have also created some utilities to help with loading and handling the data.

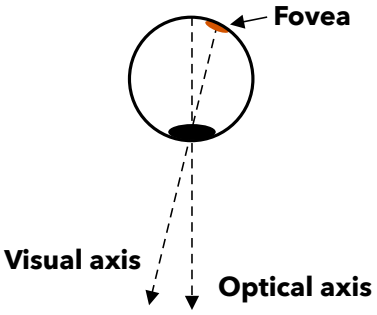

Figure 2: Shows the distinction between the visual and optical axes. The optical axis is defined as an axis perpendicular to the lens behind the pupil. The visual axis is personally dependent and is determined by the placement of the *fovea*.

Setup:

The setup used is shown in Figure 3. The head of the subject rests on a stand and the camera has a relatively fixed position on the table (although not completely as you will see). During recording, the subject is asked to look at a number of red dots on the screen moving in a specific pattern.

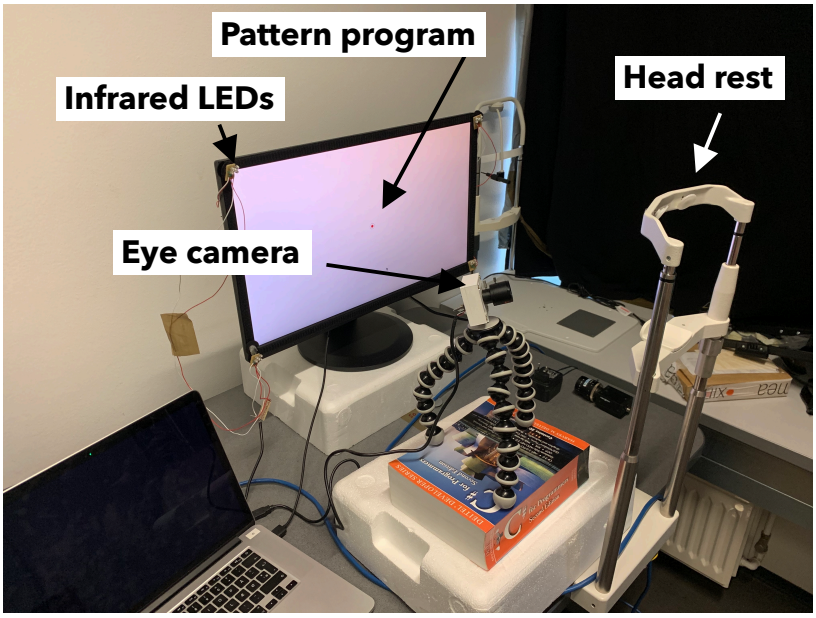

Figure 3: Setup used for all the included data.

Directory structure:

The recorded eye images and corresponding targets on the screen are saved in a number of subfolders in the `inputs/images` directory. Each subdirectory contain an image sequence of which the first 9 are calibration images. Each subdirectory additionally contains these three data files: `positions.json`, `pupils.json`, and `glints.json`:

- `positions.json` contains gaze coordinates (in pixel coordinates) for each image.
- `pupils.json` contains annotated ellipse parameters for the pupil in each image.
- `glints.json` contains annotated glint parameters for each image.

We additionally recorded two video sequences in `inputs/videos`, each with 9 calibration images as well but no annotated data for the actual videos.

Your task is to accurately infer the gaze (as position on the screen) of the user, e.g. the position on the screen given the eye image. This is achieved by detecting the pupil position in the image and the using a number of sample images with known gaze coordinates to create a model $f_\theta(x, y)$ (using regression) that maps image pupil positions x, y to screen positions x', y'. Figure 4 shows corresponding eye images and pupil positions. Figure 5 shows an example of how the x-coordinate of the pupil can be mapped to screen x-coordinates using a linear function. Additionally, you will use *glints*, which are reflections caused by infrared LEDs placed at the screen's corners, to normalise the eye position in the image and hence correct for errors related to movements of the subject. A sample image from the collected data is shown in Figure 6.

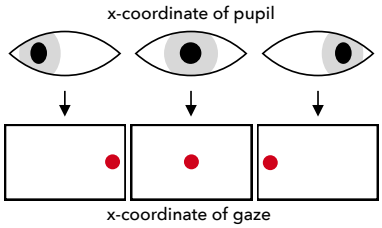

Figure 4: Eye positions and corresponding gaze points on screen.

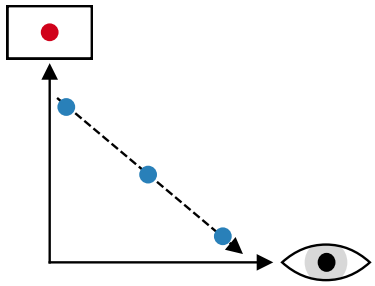

Figure 5: Linear regression model $f_\theta(x)$ for mapping pupil position to gaze coordinates (x-coordinate only).

Assignment 2.1 *(week 1)*

Pupil and glint detection

Your first task is to create robust detectors for pupils (1) and glints (2). The consistency and accuracy of the gaze estimation part depends directly on the quality of your detection algorithms.

Write your implementation for this exercise in `detector.py`. You will find that it already contains two function stubs `find_pupil` and `find_glints`. You have to implement these functions in this assignment. The script `detector_viz.py` lets you visualise your results. The test script takes one command-line argument which specifies which subfolder in the `inputs/images/` directory to use. Run it as `python detector_test.py <folder>`. The script displays a window with a slider which selects the image to use for testing. We have annotated all samples with the correct pupil and glint positions. Use the script regularly to test your solution as you develop it.

Task 1: *Pupil detection*

The first part of this exercise will guide you along the way and show what general approach you should take. At the end there will be a number of improvements that you have to make yourself. All the code for this exercise should be written in the `find_pupil` function in `detector.py`.

The pupil is roughly a circular hole that lets light into the eye. Most light is absorbed by the photosensitive cells (retina) on the back of the eye, giving the pupil its black appearance. The pupil appears approximately *elliptical* in images because of the projection onto the image. This effect is shown in Figure 7. We model the pupil shape as an ellipse with the following parameters (x, y, a, b, θ) (as shown in Figure 8) and assume the intensity to be darker than the surroundings.

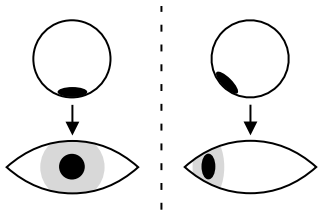

Figure 7: Visualisation of how pupil projections may lead to an elliptical pupil shape in an image.

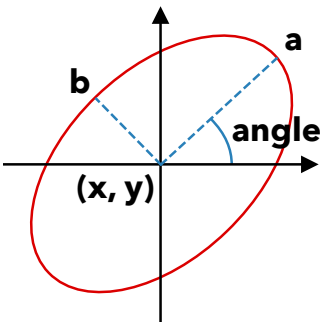

Figure 8: Ellipse parametrisation.

Approach

The general approach is to

1. Threshold the image to find BLOBs.
2. Find contours from BLOBs.
3. Select a single contour candidate as the best.
4. Fit ellipse to the selected contour candidate.

Detecting the pupil is fairly straight forward using thresholding since it is much darker than its surroundings. Notice that it may be difficult to use the same threshold for different videos and in different settings due to light changes. Thresholding should produce BLOB candidates which can then be fitted to ellipses. You will then experiment with using different threshold values and BLOB properties to select a final pupil candidate.

Tasks

The following steps will guide you through the process of detecting the pupil:

1. ⚠ **Convert color:** Convert the input color image to grayscale using `cv2.cvtColor`.
2. ⚠ **Threshold image:** Create a binary image using `cv2.threshold`. This is actually a pixel classification specifying pixels as either belonging to the pupil or background.

   Hint: *Use* `cv2.THRESH_BINARY_INV` *as the thresholding type, this inverts the output such that dark areas become white in the threshold output.*
3. ⚠ **Find contours:** Use the OpenCV function `cv2.findContours` to create a list of contours. Each contour is described as a list of points. The function returns a tuple of `(contours, hierarchy)`. Ignore the last element, it is used for detecting contours enclosed by other contours[1]

   [1] Example of use is available here.
4. ⚠ Use `cv2.imshow` to display an extra window with the thresholded image and possibly detected contours when running `detector_test.py` - this makes it easier to debug your implementation and choose appropriate values for the thresholding function.
5. ⚠ **Find largest contour:** We make the simple assumption that the pupil is the largest blob. For each contour, use `cv2.contourArea`[2] to calculate its area and select use the largest as the final candidate.

   [2] Find example of use here.
6. ⚠ **Fit ellipse parameters:** Fit an ellipse to the selected BLOB using `cv2.fitEllipse`. The function takes the contour as input and returns ellipse parameters as a tuple in the format `((x, y), (a, b), angle)`.

Optional additions:

Hint: *The following page (link) contains an overview of how to use contours in OpenCV. It is a very useful resource for understanding how to use the various functions provided by OpenCV.*

- ⚠ **Morphology:** Use morphology on the thresholded images to improve stability of your detectors.
- ⚠ **Features:** Experiment with more contour features, for example using OpenCV [3]. An obvious choice is circularity but many others may be appropriate. We provide a module `blob_properties.py` for easy contour property calculation.

  [3] Overview of contours in opencv (link)
- ⚠ **Determine distributions over pupil pixel intensities:** Use the provided annotations to find the distributions of pixels inside and outside annotated pupils or glints. Then use the simple pixel classification method to find a more well-informed threshold for the detector.

Task 2: *Glint detection*

This part of the assignment requires you to design your own approach for glint detection. You have to design a method for detect-

ing glints in eye images. Use the function `find_glints` for your implementation.

1. ⚠ **Detect glints:** Use the pupil detection method as inspiration for glint detection. Since there may be multiple glints in the image (there are four lights), you need to adapt the method to account for this. Use any method you like and use the visualisation script to test and debug your solutions.
2. ⚠ **Glint detection using pupil center:** Use the detected pupil center to improve the accuracy of your glint detector. Sort possible candidates by their distance to the pupil and return the top four.

Hint: *Remember that you can display windows from inside the function.*

Task 3: *Evaluation*

You now have to test the actual performance of your detectors. We suggest you use Jupyter or some other interactive platform for this part of the assignment as it makes iterating and recalculating faster and easier.

1. ⚠ **Create test file:** Create a file for your tests. Use either a plain script file `.py` or a Jupyter notebook `.ipynb`.
2. ⚠ **Load the data:** Load each available image sequence in the `inputs/image` from `utils.py` to make this easier. Notice that pupils are saved in `pupils.json` and glints are saved in `glints.json`.
3. **Calculate metrics:**[4]
   (a) ⚠ **Calculate distances for pupils:** For each image, calculate the distance (using the provided function `dist` from `utils.py`) between the ground-truth pupil centre and the one detected using your implementation. Store the distance errors in a list.
   (b) ⚠ **Calculate distances for glints:** For each detected glint, measure the distance to the closest ground-truth glint and record this as the error. Because we do not differentiate between the different glints, we choose the closest as the most likely candidate. Store the distances in a list.
4. ⚠ **Data analysis:** For both pupil and glints calculate at least the *mean* and *median* distance. Also produce a histogram plot for each (use `plt.hist` to automatically create a histogram from the lists).
5. ⚠ **Further analysis:** There may be other tests and metrics that would be useful in evaluating the detectors. One possibility is to use the ground-truth and detected pupil overlap, known as *intersection over union* (IOU)[5]. Another is to compare the number of detected glints to the number of ground-truth annotations. Explore different metrics for evaluation.

Hint: *The pupil data is a list of dictionaries with each element representing an ellipse parameter. We provide a helper function `pupil_json_to_opencv` in `util.py` to convert an entry to the format produced by `cv2.fitEllipse`.*

[4] **Hint:** *It might be helpful to finish the tasks for the pupil first and then expand to include the glints.*

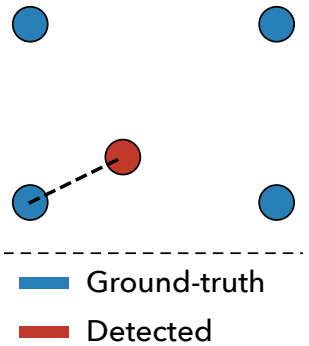

Figure 9: Visualisation of how glint distances should be calculated.

[5] Link to a tutorial on what IOU is and how to calculate it

Task 4: *Report*

- ⚠ **Implementation details:** Describe how your detectors work at a high level but still sufficient for reproduction. Describe the approach you took and what general methods you used (for the pupil detector this is only relevant if you made changes/additions).
- ⚠ **Theory:** Describe how contours are found from a binary image using course theory. Don't repeat the theory, use references.
- ⚠ **Evaluation:** Present a critical discussion of your results. This means you have to present the results and discuss the general precision and when the model works and doesn't work. How does this relate to your implementation? You also have to reflect on the data used to train and test the models (relate to bias/variance). Additionally, you have to include the following:
  - Use the produced metrics and plots to explain how well your detectors work.
  - Discuss and show example images of cases where detection fails (use the save function of the `detector_viz.py` script). Discuss and evaluate why they fail.
  - Discuss the ground-truth data. Are there any strange annotations that increase distance errors in your tests?

Hint: *Weekly exercise 3 shows how to save plots in the Matplotlib tutorial.*

Checklist

- The file `detector.py` with your implementation of the functions `find_pupil` and `find_glint`.
- The file you used to write your tests.
- Report section as specified.
- Plot images and sample detection images.
- All ⚠ has been sufficiently answered.

Assignment 2.2 *(week 2)*

Gaze estimation

The goal of this exercise is to estimate the gaze of image sequences using a regression model. As mentioned in the introduction, each image sequence contains 9 images for calibration and a varying number of images for inference. The calibration samples always represent the same 9 screen positions which form a simple 3 by 3 grid. An example of calibration images are shown in Figure 10. For each sequence, you will use the 9 calibration samples to train a regression model and then use the model to predict gaze positions for the rest of the images.

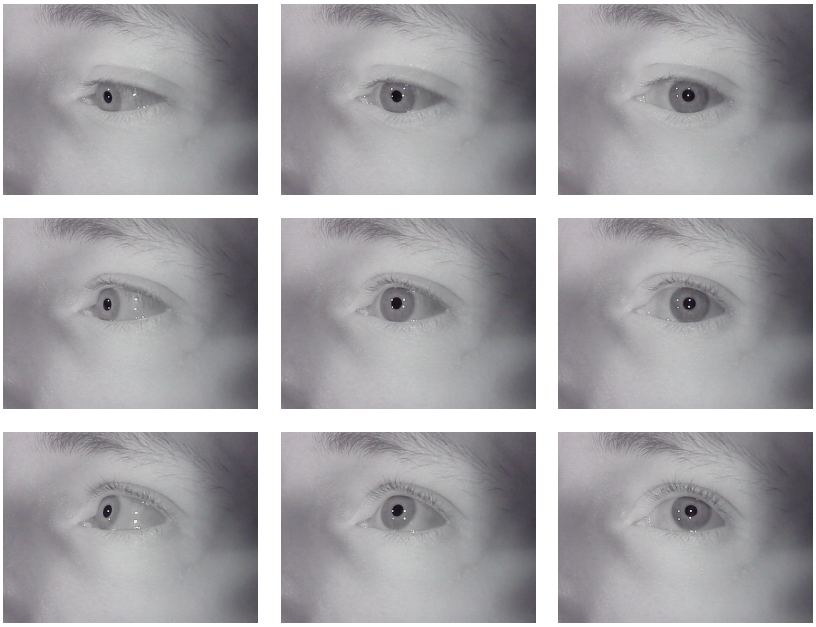

Figure 10: Calibration images. All image sequences contain 9 calibration images which all have equivalent gaze positions.

`positions.json` contains the ground-truth gaze positions for each image as an array (stored as y, x for each point). The included image sequences (found in `inputs/images`) are divided into two groups:

- **No head movement:** `pattern0`, `pattern1`, `pattern2`, `pattern3`
- **Head movement and rotation:** `movement_medium`, `movement_hard`

You may want to focus on the ones without head movement for now.

Use the provided script `position_vis.py` to visualise the image and corresponding screen point. It has one command-line argument which is the subfolder (in `inputs/images/`) to use, e.g. `python position_vis.py pattern1`.

Task 1: *Basic gaze estimator*

The mapping function $f_\theta(x, y)$ as shown in Figure 1 can take any form. Because the eye is spherical, the relationship between pupil

position in the image and gaze is non-linear. In this exercise, however, you will approximate the gaze mapping by a linear function. We assume the gaze coordinate x', y' to be independent variables. Therefore, we train a separate model for each. This is equivalent to training one model with a vector output. You do this as in the exercises but with one model for the x coordinate and one for the y coordinate. To get the screen coordinates x', y' we have

$$x' = ax + by + c$$
$$y' = dx + ey + f$$

The design matrices for both models are:

$$D_x = D_y = \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ & \vdots & \\ x_n & y_n & 1 \end{bmatrix}.$$

The principle is demonstrated ing Figure 11 to the right. Here, the x coordinate of the pupil maps to the x coordinate on the screen. In the real model, we use both x and y as inputs to both the model estimating the x position on the screen and the model estimating the y position.

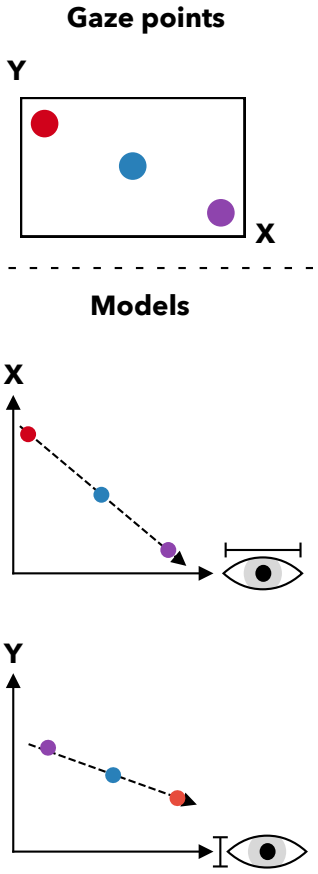

Figure 11: Demonstration of the use of two regression models for mapping gaze. To the right is shown three colour-coded gaze points on the screen. The left shows two linear regression models (lines) that map from pupil centre coordinates to screen coordinates. The marked points denote the points of the same colour on the right. Note that this example actually doesn't show the full model as both pupil coordinates are used for each!

Hint: *Note that points are saved in y, x format! Compared to the pupil and glint detection methods (and json files) which use x, y.*

1. **Getting an overview:** In the file `gaze.py` we have provided a stub class `GazeModel` for your implementation. It is used by the `position_vis.py` script to show the estimated gaze point when you actually implement it. Right now the `estimate` method always returns `0, 0`.
2. **Calibration:** Learn the parameters ϕ for the linear regression using the calibration images and points aved in `self.positions`. Use the detector from the assignment 2.1 to detect pupil points. Use the centres of the detected pupils to create a design matrix as shown above. Then create two models, one for the X coordinates of the calibration points and one for the Y coordinates.
3. **Estimation:** Implement the `estimate` function which predicts the gaze given an eye image using the learned model. First, detect the pupil centre in the input image. Then calculate and return the estimated screen coordinates using the models created during calibration.

Task 2: *Evaluation*

1. **Create test file:** Create a file for your tests. Use either a plain script file `.py` or a Jupyter notebook `.ipynb`.
2. **Load the data:** Load each available image sequence in the `inputs/image` from `utils.py` to make this easier. Notice that gaze points are saved in `positions.json`.

3. ⚠ **Calculate distance:** For each dataset, create and calibrate a `GazeModel` and use it to estimate the gaze position for the points not used for calibration. Calculate the euclidean distance between the estimate and ground-truth gaze position for each image and and save it.
4. ⚠ **Data analysis:** Perform the analysis for the sequences with and without head movement separately. Calculate the *mean* and *median* distance. Also produce a histogram, but this time use the `cumulative=True` and `density=True` arguments to make a cumulative histogram that is normalised, i.e. shows fractions instead of number of occurrences.
5. ⚠ Ⓜ **Correlation:** Record both the pupil detection distance and gaze distance errors. Measure their correlation.
6. ⚠ **Further analysis:** There may be other tests and metrics that would be useful in evaluating the detectors. Please explore possibilities as much as you'd like.

Task 3: *Report*

1. ⚠ **Implementation details:** Describe the approach you took in your implementation.
2. ⚠ **Evaluation:** Present a critical discussion of your results using the metrics. This means you have to present the results and discuss the general precision and when the model works and doesn't work. How does this relate to your implementation? You also have to reflect on the data used to train and test the models (relate to bias/variance). Additionally, you have to include the following:
   - ⚠ Use the metrics and plots produced to explain how well your gaze estimator works.
   - ⚠ Discuss and show example images of cases where gaze estimation is inaccurate.
   - ⚠ How is euclidean distance error related to the mean squared error?
   - ⚠ To what degree is the inaccuracy caused by the pupil detection? Explain how you may test the gaze accuracy without using the pupil detector (and thereby removing those errors) - you don't have to implement this in code.
     - Ⓜ Use the correlation measure to back up your arguments.
   - ⚠ How does the still data compare to the data with movement? Explain this behaviour.

Checklist

The file `gaze.py` with your implementation of the `GazeModel` class.

- The file you used to write your tests.
- Report section as specified.
- Plot images and sample detection images.
- All ⚠ has been sufficiently answered.

Assignment 2.3 *(week 3)*

Improving detection

For the final part you have to choose one of the following additions to your gaze estimation method. Please consider each one carefully before choosing one. We encourage you to try to solve several of these. Note that even though these tasks technically count equally in the exam, the last two are opportunities to further explore the material of the course and are recommended.

Task 1: *Estimating gaze in videos*

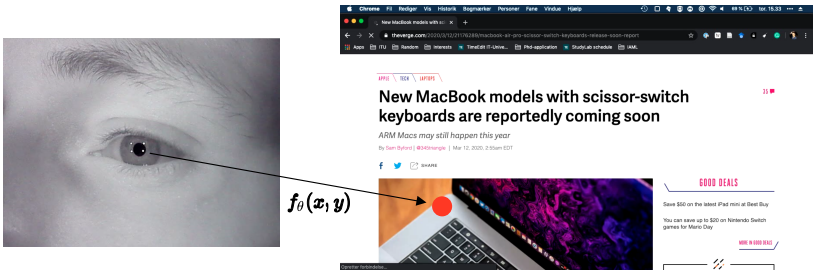

Figure 12: Example image pair from the eye and screen video from `video0/`. The mapping task is the same as for the sequences but with an actual usage example.

⚠ This improvement involves extending the gaze estimation method to work on videos (Figure 12 shows an example). You have to create a new script to perform this functionality.

1. **Create gaze model:** Load the required image and gaze position files using the helper functions `load_images` and `load_json` from `utils.py`. Create an instance of your `GazeModel` class using these as arguments.
2. **Video gaze:** Load and play the eye and screen video files in the chosen directory (in `inputs/videos`). Use the gaze model to estimate the gaze point and draw it onto the screen video.
3. **Save screen video:** Save the screen video in the `outputs`.

Task 2: *Polynomial gaze model*

⚠ In this task, we assume $f_\theta(x, y)$ to be a second-order polynomial. You have to implement this. When working with multiple inputs, the polynomial model for regression requires a bit more work than usual but it is still conceptually similar. When performing polynomial regression, we pre-calculated each power of an input value ahead of time, i.e. we performed the following transformation for each input value:

$$x_i \to [1, x_i, x_i^2, \ldots, x_i^n]. \tag{1}$$

With multiple values we stacked x_is to form the *design matrix*. For two input variables x_i, y_i, we would like to include all polynomial combinations of the inputs of a given order, i.e.

$$[x_i, y_i] \to [1, x_i, y_i, x_i^2, x_i y_i, y_i^2 \ldots, y_i^n]. \tag{2}$$

This allows the model to include interactions between the variables.

In this task, we suggest you use the library *scikit-learn* for creating the design matrix and for fitting the regression models as it is much easier to work with than doing things manually. [6] [7] [8]

[6] You can find a general overview of polynomial regression with scikit-learn here.
[7] The docs for the function for creating polynomial design matrices can be found here.
[8] The linear regression docs can be found here.

1. **Create new class** `PolynomialGaze`: Create the new class with the same methods as `GazeModel`.
2. **Add** `order` **parameter to constructor:** Add a parameter to choose polynomial order when initialising the model.
3. **Calibration:** Use scikit-learn to create the design matrix and train the model.
4. **Estimation:** The trained model has a `predict` function you can use when estimating gaze points.
5. **Evaluation:** Experiment with various values for the `order`. Discuss in the report which polynomial order works best (use the distance evaluation from Assignment 2.2) and why this is the case. Relate your discussion to the bias/variance dilemma.

Task 3: *Gaze estimation under head motion*

⚠ The model used so far assumes that the head is perfectly stationary for the whole image sequence. This is an unrealistic assumption in practice. The *glints* can be used to determine the position and orientation of the eye relative to the screen. Figure 13 demonstrates how the glints reflect from the screen to the eye and onto the camera.

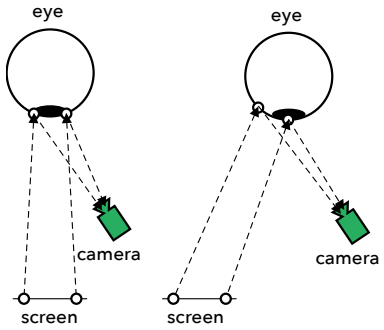

Figure 13: How glint reflections reveal the position of the screen relative to the eye.

The glint reflections can be used to form a planar coordinate system. Using it, it is possible to normalise the pupil position with respect to gaze. Recall the lecture on transformations. Depending on the number of glints detected, you can use the following transformation to normalise the pupil position:

- **One point:** Translation
- **Two points:** Similarity transformation
- **Three points:** Affine transformation
- **Four points:** Homography

You decide how many glints you want to support in your solution. Some important notes:

- A more stable solution is to use only glints close to the pupil due to the shape of the *cornea*[9].
- Note that all glints are not necessarily visible and it is therefore necessary to have fallback-methods if less glints are registered.

[9] The cornea is a part of the human eye which covers the iris and pupil and creates a protruding bulge from the eye.

1. **Create new class** `NormalizedGaze`: Create the new class with the same methods as `GazeModel`.

2. **Create normalisation method:** Add a method `normalize` to the class. It should perform the normalisation operation on detected pupils using glints. Infer the transformation from the detected glints to normalised coordinates (i.e. $(0,0), (0,1), (1,0), (1,1)$). You may want to use the transformation library you developed in Assignment 1.1.

Hint: *To determine which glints you have found, use the relative pupil position as an indicator.*

3. **Modify existing functions:** Modify the calibration and estimation functions to use normalisation.

Task 4: *Report*

⚠ After solving one (or more) of these tasks, add the following to your report:

1. ⚠ **Approach:** How did you approach the problem? Depending on the task chosen, this might be a complex or simple question to answer. Try to be as precise as possible but don't describe overly trivial decisions.
2. ⚠ **Results:** You must somehow demonstrate your results. For the video task you should add the video to your submission. For the other tasks you should provide test results that are comparable with the base models.
3. ⚠ **Evaluation:** For the video task you should comment on how precise the gaze estimation seems to be and whether it reveals some interesting results. For the other tasks comment on the difference (if any) in gaze estimation accuracy.

Checklist

- ☐ Any extra files you created, including test files (if any).
- ☐ Report section as specified.
- ☐ Sample video or plots depending on task (make sure to produce something demonstrating your results).
- ☐ All ⚠ has been sufficiently answered.