Universiteit Antwerpen

Honours Program: Final Report

Identification of Smooth Newts Based on Spot Patterns

[1]

Authors: Jesse Daems & Rune De Coninck

Supervisor: Em. Prof. Dr. Nick Schryvers

Academic Years 2023–2025

Inhoudsopgave

Acknowledgement

We would like to thank our supervisor, Em. Prof. Dr. Nick Schryvers, for all his help and involvement throughout this project. We found it to be a pleasant collaboration from which we learned a great deal.

1 Background and Motivation

When the smooth newt, an amphibian species, wants to reproduce in a breeding pond, it often needs to cross a busy and dangerous road. To prevent many newts from being killed in traffic, relocation efforts are organized to safely transfer the animals to the other side. These relocations are carried out by volunteer teams who use buckets to carry the amphibians across the road and release them. This method significantly reduces road mortality. During these relocations, it would be useful to track which newts are present and which ones return each year. Our project aims to facilitate this tracking in an efficient way. The smooth newt can be uniquely identified based on its spot pattern. This feature is what we aim to exploit in our project to make individual newt identification possible.
The goal of this project is to deliver a user-friendly app. The app will be able to take a photo of the newt's belly—where the spot patterns are most visible—and match this to an existing individual in the database. If no matches are found, a new individual can be registered based on the captured photo.

To achieve this, we gathered a set of techniques we want to experiment with. The final program will integrate these methods to deliver reliable results.

The system needs to take into account the following factors that can complicate identification:

- Differences in lighting,
- Differences in scale, rotation, and position,
- Various body curvatures of the newts, and
- Variable resolution of input images.

We received hundreds of sample photos taken during relocation events over the past few years. Some of the photos turned out to be unusable, but the majority met our requirements (see Section 2.1).

As for the implementation, we structured it into two parts. On one hand, the logic and core functionality of the project is implemented in Python. This part is made accessible via a RESTful API. On the other hand, the user interface is implemented in Dart using the cross-platform Flutter framework, allowing us to build both an app and a website from a single codebase.
In Section 3, we go into more detail about the software.

This was an interesting project for both of us. For Jesse, as a computer scientist, it was valuable to approach a project like this in a more mathematical and formal way than usual. Additionally, it was a good opportunity to apply his knowledge of software design and AI in practice, while also learning to work with a new frontend platform. For Rune, as a mathematician, the project was highly valuable because it allowed him to apply mathematics to a real-world problem. Moreover, it involved a substantial amount of programming, which positively impacted his coding skills. Overall, this has been a valuable and educational project for both of us, where we were able to conduct research together to reach our final goal.

2 Results

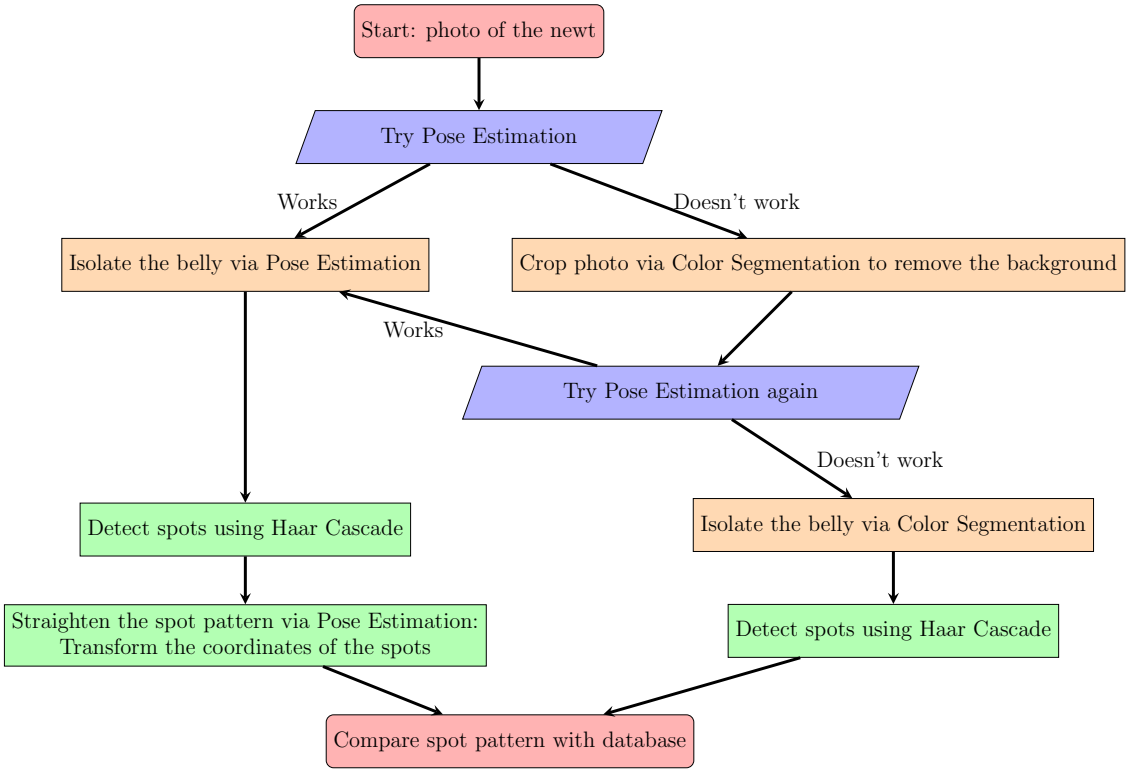
2.1 Assumptions about the Photos

The following assumptions were made in our project:

- Each photo contains only a single newt.
- The newt was photographed from below so that the belly is visible.
- The lighting in the photos is neither too dark nor too bright.

Figuur 1: Newt photo that meets the assumptions.

2.2 Overview of the Different Steps



Figuur 2: Diagram showing the different steps of the process.

We will now discuss the different steps from the flowchart.

2.3 Pose Estimation

Pose estimation is the first step we perform. The success of this step determines which steps will be taken next in the process (as shown in the diagram above). The results from pose estimation are also used in later steps of the process.

The goal of this step is, given an image, to determine the coordinates of specific points on the body of the newt (for example, the shoulders or the tail). If certain points are not present in the image, it is also important that we can detect this.

To achieve this, we use the DeepLabCut software package [7, 8]. This package allows us to train an effective pose estimation model with a limited amount of training data: between 100 and 200 images, with manually annotated points, are sufficient. When we feed a new image into the model,

we receive for each predefined point the estimated x- and y-coordinates and a confidence score. This score is used to decide whether or not to use a given estimated point. A low confidence can have two causes: either the image is too blurry or the point is hard to see due to other circumstances, or the point simply isn't present in the image. A high confidence means we can be reasonably sure that the estimated point is in the correct location.

2.3.1 Chosen Points

Before we can start training, we must first define which points are important to locate. In total, we use 16 different points (see also Figure 3):

- **head_tip**: the center at the tip of the head.
- **left_shoulder**: the intersection between the left side of the torso and the line connecting the torso with the left front leg.
- **left_hand_middle**: the middle of the left front leg.
- **right_shoulder**: the intersection between the right side of the torso and the line connecting the torso with the right front leg.
- **right_hand_middle**: the middle of the right front leg.
- **left_pelvis**: the midpoint of the connection between the left hind leg and the torso, on the curve defined by the left side of the torso.
- **left_foot_middle**: the middle of the left hind leg.
- **right_pelvis**: the midpoint of the connection between the right hind leg and the torso, on the curve defined by the right side of the torso.
- **right_foot_middle**: the middle of the right hind leg.
- **tail_connection**: the end of the cloaca on the side of the tail.
- **tail_end**: the tip of the tail.
- **spine_highest**, **spine_high**, **spine_middle**, **spine_low**, and **spine_lowest**: a series of points on the spine. The endpoints **spine_highest** and **spine_lowest** are defined respectively as the midpoints between the inner corners of the torso at the front and hind legs. The points in between are equally spaced across the body to ensure they lie in the middle of the torso. Note that we use the same spine points as described in [7].

Figuur 3: Illustration of the various points obtained through Pose Estimation.

2.3.2 Training Data

In the first version of the model for these points, we only used the 100 highest-quality photos we had. This already produced impressive results on other photos taken under roughly the same conditions. However, most of these photos featured the newt in roughly the same orientation: horizontal, with the head to the right. As a result, the model struggled more with images in different orientations.

In the second version, we not only added more photos, but also randomly rotated each image by 0°, 90°, 180°, or 270° before training. We limited ourselves to these angles to avoid adding extra black space or having to crop the images, which could risk cutting off parts of the newt. Since the newts were not usually perfectly straight in the original photos anyway, this decision didn't drastically affect the actual angles used. This new version was much more effective on rotated images.

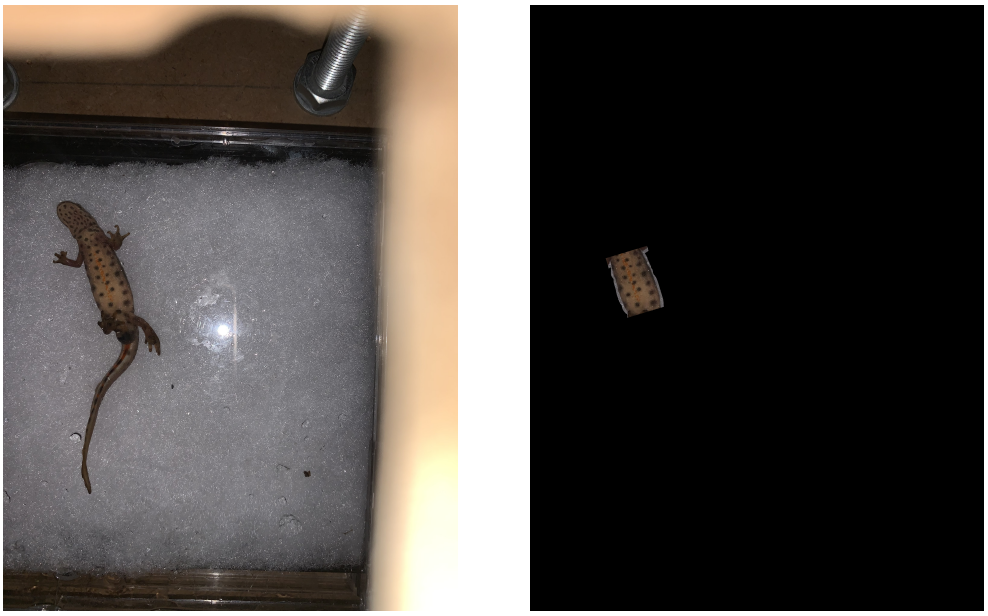
As we note in Section 5.3, we believe it could be beneficial to also train the model on the photos from the current year. This might improve the app's performance on the most recent data.

2.4 Isolating the Belly

The next step involves isolating the belly, which includes the following actions:

- Removing the background so that only the newt remains.
- Removing the limbs, tail, and head so that only the belly remains.

The goal is to isolate only the belly of the newt, as shown in Figure 4. This is crucial for the next step, where we detect the spots. The better we isolate only the belly, the fewer false positives will be detected as spots (remember, we only want to detect the spots on the belly).



(a) Original

(b) After isolating the belly via Color Segmentation

Figuur 4: Example illustrating what isolating the belly means.

Originally, we attempted this via Color Segmentation [6], and this method works well if the image meets a few conditions:

- The newt is positioned roughly in the center of the image.
- If the newt is horizontal, the head is on the right.
- If the newt is vertical, the head is at the top.
- There is no yellow or brown lighting in the background.

Due to these (sometimes strict) requirements, we switched to using Pose Estimation to isolate the belly, which works much more generally. This is also shown in Figure 2, where we prefer the Pose Estimation method [7, 8]. However, if that method fails, we can fall back on Color Segmentation, which also yields accurate results when the conditions are met. Additionally, the Pose Estimation method requires significantly more computational power and time than the Color Segmentation method. We thus choose longer computation times and more accurate results, because those accurate results reduce the number of false positives in the spot detection step. This is crucial, as we have not found a method that can reliably remove false positive spots. Finally, in Section 5.3, we also discuss a lighter version of the pose estimation model, which could be an improvement.

In the following sections, we give a detailed explanation of how each method works.

2.4.1 Isolating the Belly via Pose Estimation

We attempt to isolate the belly of the newt using Pose Estimation. The algorithm behind this process starts from the detected pose of the newt, as obtained in 2.3. It only works if the following (fairly relaxed) conditions are met:

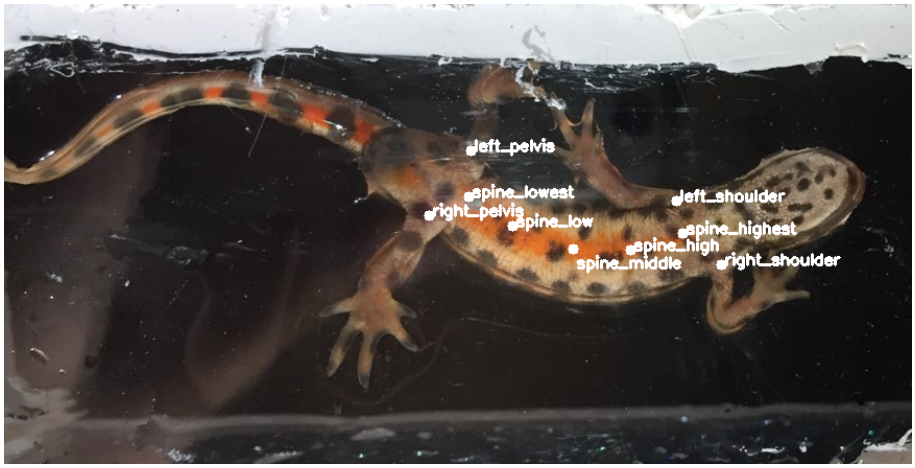
- The highest and lowest spine points have been correctly detected.
- One of the two pelvis points has been correctly detected.
- One of the two shoulder points has been correctly detected.

We found that these conditions are nearly always met in the photos we are working with. Once these checks pass, the algorithm follows a series of steps to isolate the belly. We explain these steps in the following sections.

2.4.1.1 Selecting the Detected Points

From all the reliably detected points (via Pose Estimation) on the newt's body, we retain only those that belong to the spine, shoulders, or pelvis. This results in an image like Figure 5.

Ultimately, we want to find points located on the torso of the newt so that we can draw a loop through these points to crop out the belly. We can locate these torso points by drawing a line segment from each spine point outward to the edge of the torso. This means we need to know how far the torso extends from the spine and at which angle to draw the line segment. Naturally, the angle differs for each spine point if the newt is curved, as in Figure 5.



Figuur 5: Newt with the relevant points indicated.

2.4.1.2 Calculating the distance to the torso for each point on the spine

In this step, we use all our assumptions to make the algorithm work. For each point on the spine, we want to know how far it is from the salamander's torso, but we do not know the location of the torso (otherwise, we could have segmented the belly immediately—we only know the spine). What we do know are the distances (expressed in pixels) between the left and right hips and the lowest point on the spine. Similarly, we also know the distances between the left and right shoulders and the highest point on the spine.

We first calculate the *lower distance*, which is the arithmetic mean of the distances between one or both hips and the lowest point on the spine. We also calculate the *upper distance*, which is the arithmetic mean of the distances between one or both shoulders and the highest point on the spine. In other words, the lower distance indicates how far the bottom of the spine is from the hips, serving as a local approximation of the distance to the torso. The upper distance indicates how far the top of the spine is from the shoulders, also serving as an approximation for the torso's location at that point.

Furthermore, we assume that the thickness of the salamander's body does not change significantly. We can now approximate the torso distance for each spine point using linear interpolation. We illustrate this with a simple example.

Suppose the lower distance is 5 and the upper distance is 6, and we have detected three points along the spine. Then the point closest to the hips gets a distance of 5.25, the next point gets 5.50, and the point closest to the shoulders gets a distance of 5.75.

2.4.1.3 Calculating the angle associated with each point on the spine

Once we have determined the distance from each point on the spine to the torso, we also need to find the angle (relative to the horizontal) associated with each point. We work in polar coordinates, where we need both an angle and a distance to draw a line segment from a spine point to a target point. This target point should, if we choose the angle correctly, lie on the torso and the line segment should be as perpendicular as possible to the torso. This method ensures that the endpoints of the segments provide an accurate approximation of distinct points on the salamander's torso. Figure 6 illustrates this idea.

Figuur 6: Indication of distances and angles for each point on the spine.

To realize this, we again rely on both assumptions. The angle at the two extreme points of the spine is equal to the angle of the segments defined by the detected hip and shoulder points. As shown in Figure 6, the segments through the extreme spine points are parallel to the (not drawn) lines through the hip and shoulder points. However, assumptions cannot prevent the detection of only one shoulder or hip. If only one shoulder is detected, for example, then the segment from the spine point closest to the shoulders will be drawn perpendicular to the line connecting this point and the next closest spine point. This way, we can always approximate the orientation using the extreme spine points.

We now systematically determine the angles for all spine points. Suppose we know the angles for the two leftmost points on the spine, and we want to find the angle for the middle point in Figure 6. We draw two lines through the middle point: one also goes through the point to its left, and the other through the point to its right. These lines intersect at a certain angle θ. Suppose the left point had an orientation angle φ. Then the angle for the middle point is given by $\theta + \varphi$.

In this way, starting from the point closest to the hips, we can iteratively compute all angles—always

using the previously calculated angle. If the belly locally curves more, our angles automatically adjust, following the curvature. This is clearly shown in Figure 6, where we see that the new points on the torso (i.e., the endpoints of the line segments) accurately follow the curved orientation of the salamander.

2.4.1.4 Detecting and isolating the belly

Once the technical steps are completed, the rest is straightforward. Using polar coordinates, we now have several points on the torso. By interpolating a closed curve through these points, we obtain the outline of the belly. Once we have the outline, we can remove everything outside of it, leaving only the isolated belly.

(a) Curve through torso points. (b) Belly extracted using the curve.

Figuur 7: Final result of isolating the belly using Pose Estimation.

Aside from the method being computationally expensive, we encountered no major issues as long as the pose estimation works properly and the assumptions are satisfied. Therefore, we strongly prefer this method over Color Segmentation, which can have more reliability issues. We discuss that method next.

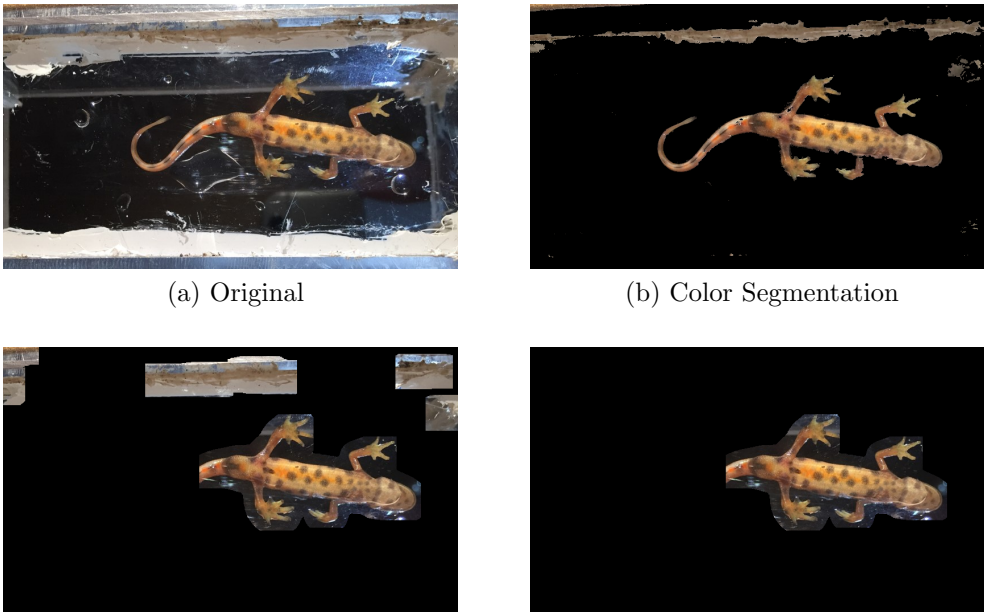
2.4.2 Isolating the belly via Color Segmentation

With Color Segmentation [6], one can isolate objects of distinct colors in an image. A color range is defined in advance, and all pixels within that range are detected. The salamanders in our images are usually yellow-orange-brown, while the background is mainly blue-gray-black. This makes it possible (assuming no brownish hue in the background) to distinguish the salamander body in most images. This method is only used when Pose Estimation fails, as it is less reliable. Nonetheless, it is worth noting that both methods yield very good results when their respective conditions are met.

Naturally, the salamander's black spots are not detected. Furthermore, filtering specific colors is a discrete process and often results in apparent "holes" in the body. To address this and reconstruct

a continuous body, we use several morphological operations combined with blurring techniques. A further issue is that some background parts may also have a brown color and be falsely detected. We later filter these out, but to prevent them from merging with the body, we apply techniques like dilation and erosion [10].

Figure 8 illustrates the idea behind Color Segmentation and the supporting steps.



(a) Original

(b) Color Segmentation

(c) Color Segmentation with post-processing

(d) Isolated salamander

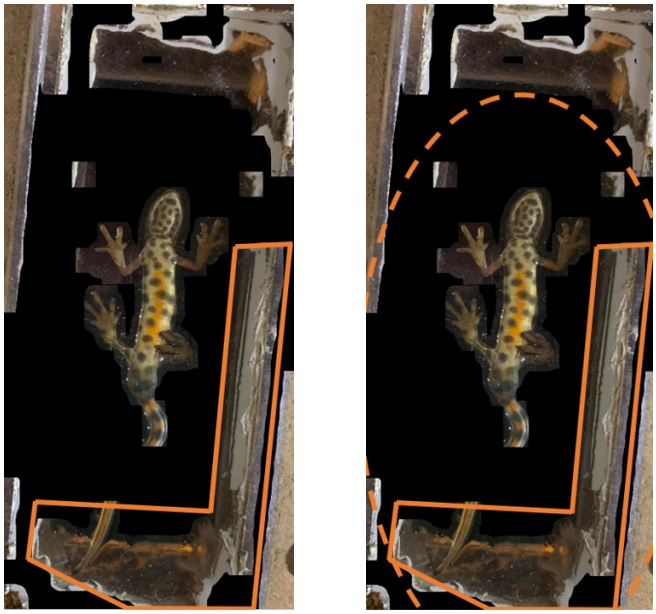
Figuur 8: Concept of isolating the salamander using Color Segmentation.

2.4.2.1 Filtering objects by location

In this first step, we remove all objects at the edges of the image and those not near the center. If no objects remain, we loosen the parameters iteratively until at least one object is retained.

2.4.2.2 Filtering by concentric objects

Many salamanders were photographed inside a box, meaning the box edge can be detected as an object. To remove these, we draw an ellipse around each object. If another object lies (partially) within this ellipse, the original object is removed. We avoid rectangles since ellipses fit more tightly around the salamander's body. This ensures that the box edge is removed without removing the salamander itself. An example is shown in Figure 9.



(a) Orange object belongs to the box.

(b) Salamander lies within the ellipse.

Figuur 9: Box part filtered out since the salamander lies within the ellipse.

2.4.2.3 Filtering by area

Next, we sort remaining objects by surface area. Since we previously removed edge and box-related objects, we can assume that the salamander is among the largest ones. We retain up to the three largest objects.

As seen in Figure 9, smaller nearby objects (like those above the forelimbs) may still remain. We always keep the largest object, and keep the second and third largest only if their area is at least 30% of the largest one. This filters out smaller artifacts.

2.4.2.4 Selecting the salamander

At this point, a maximum of three objects remain, including the salamander. We select the most centrally located object as the salamander, assuming it lies near the center.

However, we still need to remove limbs, tail, and head to isolate the belly. This is explained in the next section.

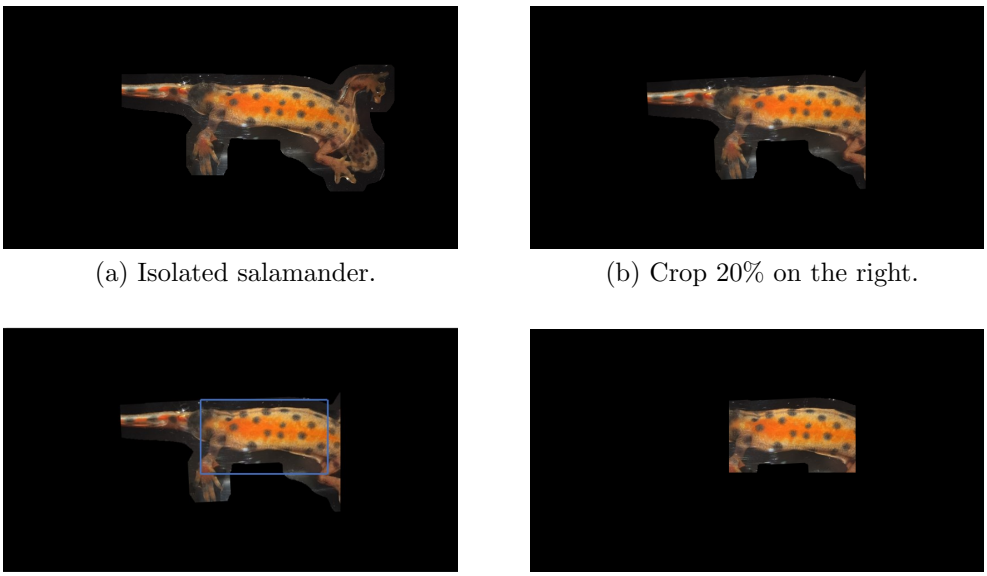
2.4.2.5 Removing limbs, tail, and head

We begin by cropping 20% off the right side (if the salamander lies horizontally) or the top (if it lies vertically), to ensure the head is excluded. (Recall our assumption that the head lies to the right or top depending on orientation.) Then we determine the object's contour.

To determine orientation, we fit an ellipse to the salamander object. From this, we extract the orientation angle.

We then use the *largest interior rectangle; lir* [11] software to find the largest possible rectangle within the contour. This rectangle approximates the belly. To ensure complete coverage, we expand the width by 5% on both sides and remove 5% from the length near the tail. This leaves only the belly.

An example is shown in Figure 10. Another example is shown in Figure 4.



(a) Isolated salamander.

(b) Crop 20% on the right.

(c) lir rectangle, sides adjusted by 5%.

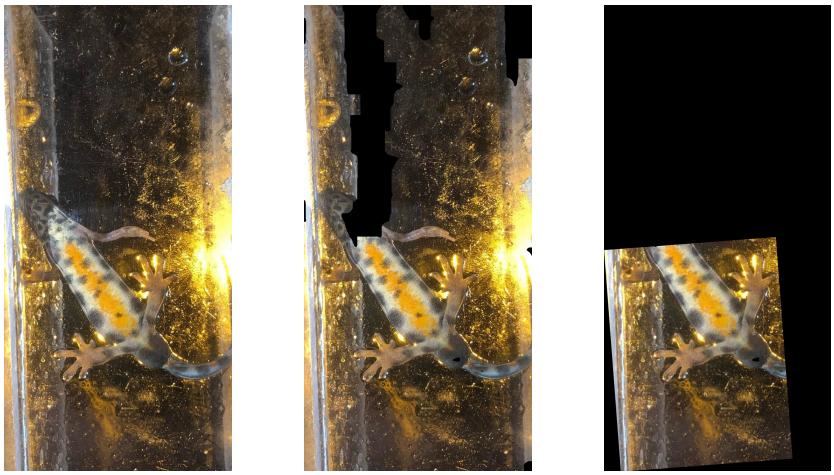
(d) Final result: the belly.

Figuur 10: Trimming tail, limbs, and head using cropping and lir.

2.4.2.6 Shortcomings

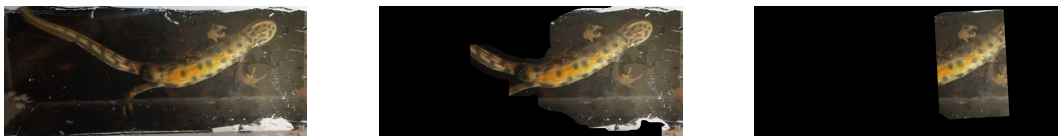
There are many conditions that must be met in order to use the Color Segmentation method. When these are violated, poor results may appear. Moreover, the method does not adapt if the salamander's belly is curved, whereas the Pose Estimation method does. Additionally, many of the

results depend on the lir software, which can sometimes be unpredictable. For example, in photos where the tail is wide and perfectly aligned with the belly, the largest rectangle may also contain part of the tail, which we obviously do not want. To make matters worse, this rectangle may contain a large portion of the tail and thus appear overly stretched, causing the width to be smaller than the actual belly width and leading to the loss of valuable information. Figures 11, 12 and 13 show examples where this goes wrong.



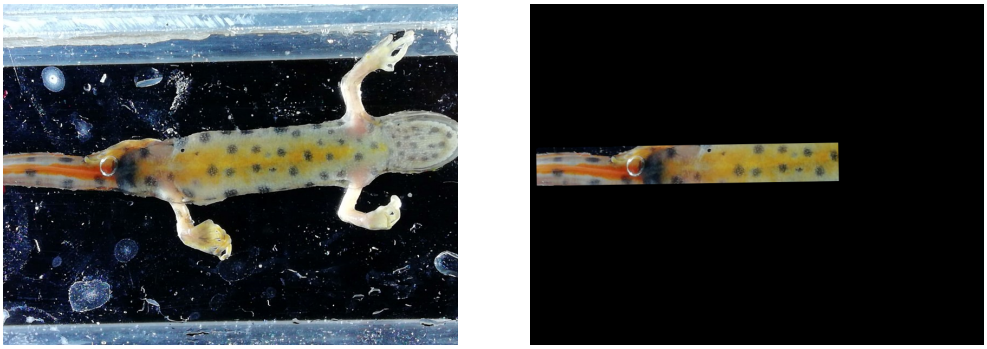
(a) Original (b) Isolated salamander (c) Isolated belly

Figuur 11: Yellow lighting in the background of the photo causes Color Segmentation to fail completely.



(a) Original (b) Isolated salamander (c) Isolated belly

Figuur 12: Brown mud in the background of the photo causes Color Segmentation to fail completely.



(a) Original (b) Isolated belly

Figuur 13: The tail lies perfectly in line with the belly, causing the lir rectangle to be overly stretched; the process fails completely.

2.5 Spot Detection

After isolating the belly, we proceed to detect the spots on it. We experimented with both classical methods and Haar Cascades. As we will explain, the classical methods were not suitable for our photos. Moreover, the Haar Cascade yields good results, making it the preferred method for us.

2.5.1 Classical Methods

As mentioned in the project proposal, we extensively experimented with various classical methods for spot detection:

- Global Thresholding
- Otsu's Method
- Adaptive Thresholding
- Blob Detection
- Color Segmentation
- One or more of the above, combined with different types of blurring and morphological operations.

Ultimately, the success of these techniques turned out to be too dependent on the lighting conditions and quality of the photos. As a result, we were unable to use these techniques for our purposes. We attempted to automatically estimate many of the parameters used by these techniques, which led to some limited progress, but this was still insufficient due to the major influence of false positives

(pixels detected as spots that are not actually spots) caused by varying lighting conditions and photo quality.

In the following sections, we provide examples showing how these methods failed.

2.5.1.1 Global Thresholding

Global Thresholding, as described in [2], is a method for extracting features from a grayscale image. The user sets a parameter in advance, known as the *threshold*. All pixels with a grayscale value below the threshold are turned black, and those above the threshold are turned white. We attempted to separate the spots from the rest of the salamander in this way, but the major issue is that the threshold must be manually set for each photo. Moreover, the correct threshold differs from photo to photo. This made it impossible to continue using Global Thresholding, as we could not scale it up in an automated way.

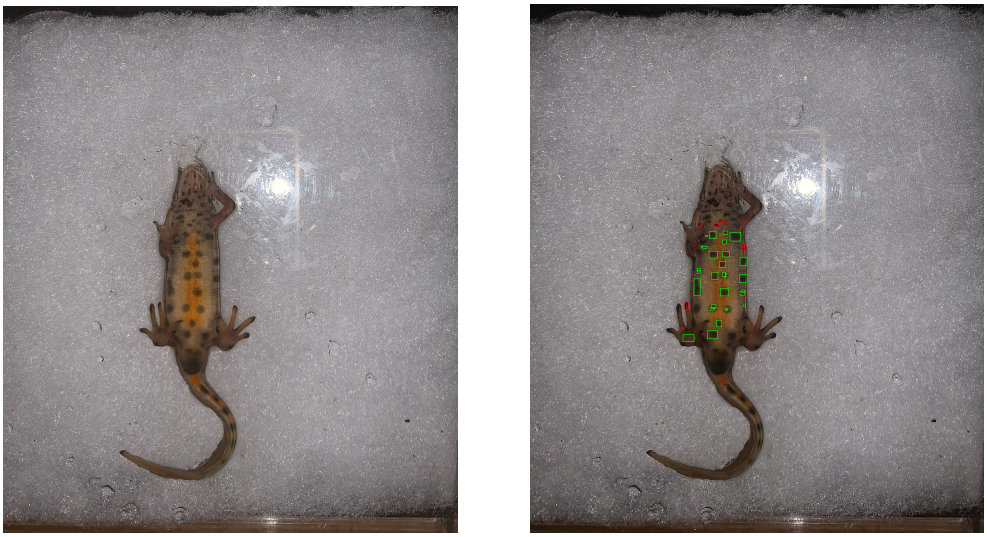
(a) Original (b) After thresholding

Figuur 14: Thresholding with a threshold of 60 can work for detecting spots.

(a) Original (b) After thresholding

Figuur 15: Thresholding with a threshold of 60 clearly does not work for detecting spots.

Below is another example where we were able to easily detect the spots after thresholding. In this case it worked, but we had manually tuned all the parameters specifically for the given photo.



(a) Original (b) After thresholding

Figuur 16: Thresholding works after manually setting the parameters. Spots are shown in green.

2.5.1.2 Otsu's Method

Otsu's Method [3] automates the Global Thresholding process (it chooses the threshold automatically) by using, among other things, the intensity levels in the image. To work well with Otsu's method, there should be a clear difference in intensity and contrast between the spots and the rest of the body, so the method can select a threshold that separates the two. In our case, this intensity difference is not always present, which made this method unsuitable for our purposes.

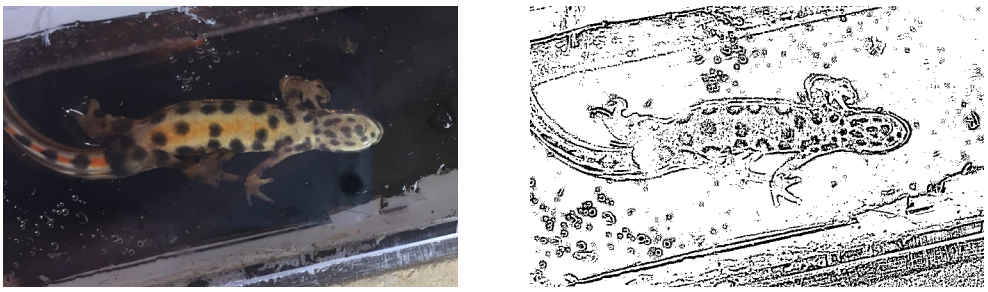
(a) Original (b) After thresholding

Figuur 17: Thresholding with Otsu's method fails.

2.5.1.3 Adaptive Thresholding

An alternative method is to use Adaptive Thresholding (sometimes also called Local Thresholding) [4]. Instead of choosing a fixed threshold (as before), we now automatically generate a separate, local threshold for each group of neighboring pixels. For example, the local threshold for a pixel can be taken as the average intensity of the surrounding pixels. This results in much more accurate outcomes, and we had high hopes that this would be a good method for us. We especially saw a lot of potential in Median Adaptive Thresholding. Gaussian Adaptive Thresholding seemed slightly less accurate for our applications.

The problem with these methods was that sometimes they were too detailed. For our application, we only wanted to detect spots, but parts of the salamander's body that were just slightly lighter than the surrounding areas were also detected. There was far too much noise detected in the photos, and we tried to reduce this by enlarging the area around the pixel used. We also attempted to apply blurring and morphological operations afterward to effectively detect good spots, but as before, we found that all these operations depend on the photo and thus are not scalable.



(a) Original (b) Mean Adaptive Thresholding

Figuur 18: Mean Adaptive Thresholding detects far too many details.

2.5.1.4 Blob Detection

Blob Detection [5] is a method to detect grouped objects (blobs). One can configure the types of these objects, such as minimum and maximum area and shape. However, there were some problems. The spots on a salamander are not always nice circles; sometimes they are greatly stretched. Also, some spots on the salamander's torso are partly outside the photo, resulting in half-circles in the image. Because of this, we could never make the blob detector specifically look for certain properties. So, this method also did not work for us.

2.5.1.5 Color Segmentation

Color Segmentation [6] is similar to thresholding. The difference is that in Color Segmentation, we segment based on a range of colors instead of a grayscale value as with thresholding. We only keep areas that have a color within a preset range. Our idea was to detect the spots by their dark color, since the body has a brown-orange color. But this also did not work: in some photos, the spots are very dark, while in others, the spots are rather light brown. Because of this, we had to include the light brown color (which, for some salamanders, is the color of the entire body) in the color range representing a spot. Obviously, it was impossible to use this method on a large scale.

(a) Salamander with bright black spots

(b) Salamander with light brown spots

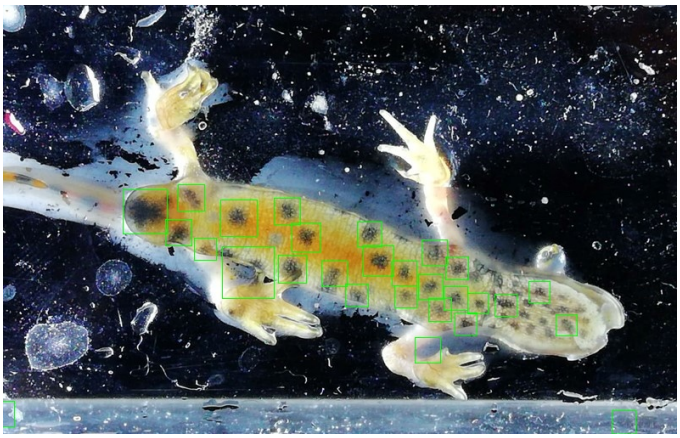
Figuur 19: Difference in contrast and color of spots in salamander photos.

With this, we conclude the classical methods for spot detection and switch in the next section to Haar cascades, which did work for our salamander photos.

2.5.2 Haar Cascades

As a middle ground between these methods and the heavier neural networks for object detection, we tried Haar Cascades [9]. This technique turned out to be surprisingly effective. The cascade is trained with a large number of "positive" images of the object to recognize (the spot, in our case). Note that these photos must contain only the object: the background, legs, tail, and head

of the salamander are cut out as much as possible beforehand using methods we discussed earlier. These are placed during training on top of the larger "negative" images, which are chosen so they absolutely do not contain the spot but do include backgrounds where the salamander is often found. Using the built-in implementation in the OpenCV library, we already achieved good results with about 500 positive images and 4 cascade stages. More stages quickly led to overfitting, where the model poorly generalized to new photos. To give an idea of how lightweight Haar Cascades are compared to heavier computer vision models: training the Haar Cascade took only about 30 seconds, while training the DeepLabCut model for pose estimation (see 2.3) took several hours.



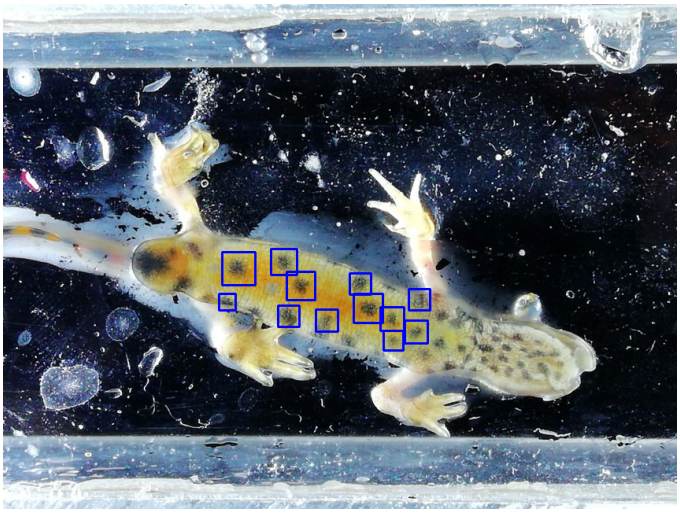
Figuur 20: Spot detection with an early version of the Haar cascade, with only a few false positives.

2.5.2.1 Evaluation of Haar Cascade Versions

To tune the parameters of the Haar cascade for the best possible result, it was important to objectively evaluate the results of each version to then compare them. To do this, we wrote a script that compares the automatic detections on several photos with manually indicated spots. This was done by measuring distances between locations and by the degree of overlap between areas marked as part of the spot.

2.5.2.2 Elimination of False Positives by Background Removal

The few false positives that still appeared during detection with the Haar Cascade were often located in the photo's background. By removing the background beforehand, we were able to reduce false positives to a minimum. As shown in Figure 21, isolating the belly resulted in minimal (in this case none) false positives; we detected only spots on the belly.



Figuur 21: Spot detection with Haar cascade, where we first isolated the salamander's belly.

2.6 Straightening Spot Pattern via Pose Estimation

Currently, the detected coordinates from different salamander photos cannot yet be compared with each other. They strongly depend on the salamander's pose. Indeed, the spot pattern of a curved salamander cannot be compared to that of a straight salamander. To solve this, we use the points on the spine detected by pose estimation (see 2.3).

As a first step, we interpolate the spine ourselves based on the 5 known points. A function $y = f(x)$ would not always work since there are no guarantees about the salamander's pose: the spine might not be expressible as a function. To bypass this, we use a parameter t. Thus, the x and y coordinates can be interpolated separately with the parametric functions $x(t)$ and $y(t)$. The meaning of t is as follows: at the detected point **spine_highest**, $t = 0$; the intermediate points have consecutive natural t-values; and at **spine_lowest**, $t = 4$.
Next, for the point to be straightened, called P, we determine several data:

- the closest point on the interpolated spine, called Q,
- the corresponding t-value t_Q so that $Q = (x(t_Q), y(t_Q))$,
- the distance d between P and Q,
- and on which side of the spine the point lies. This is done as follows: first define the t-values for the two nearest pose estimated points: $t_{prev} = \lfloor t_Q \rfloor$ and $t_{next} = \lceil t_Q \rceil$, using the floor and ceiling notations respectively. The corresponding points are Q_{prev} and Q_{next}. Then, we take the cross product of the vectors $Q_{next} - Q_{prev}$ and $P - Q_{prev}$. Depending on the sign of the cross product, we know whether the point lay to the "left" or "right".

Next, we straighten the spine itself. This means placing all spine points Q_t on the x-coordinate of the top point Q_0, and for the y-coordinate, accounting for the distance along the spine between the points. This distance L can be calculated as the arc length over the interpolated spine. Together this becomes: $Q_T = (x_0, y_0 + L)$ with

$$L = \int_{t_0}^{T} \sqrt{(x'(t))^2 + (y'(t))^2}\, dt.$$

Now, for the new straightened point P', the y-coordinate is calculated based on t_Q and the full length of the spine. The new x-coordinate equals that of the straightened spine, with the original distance to the spine either added or subtracted depending on which side of the spine the point lies, as determined above. Ultimately, in coordinates:

$$P' = (x_0 \pm d, y_0)$$

2.7 Comparing Spot Patterns

Once spots have been detected and straightened for each salamander, we can use a spot pattern matching algorithm to check whether salamanders in different photos are indeed the same. Our algorithm is based on [13] and [12] and works in several steps, often using the word "points" instead of spots. Below is a brief overview of all steps, followed by a detailed explanation.

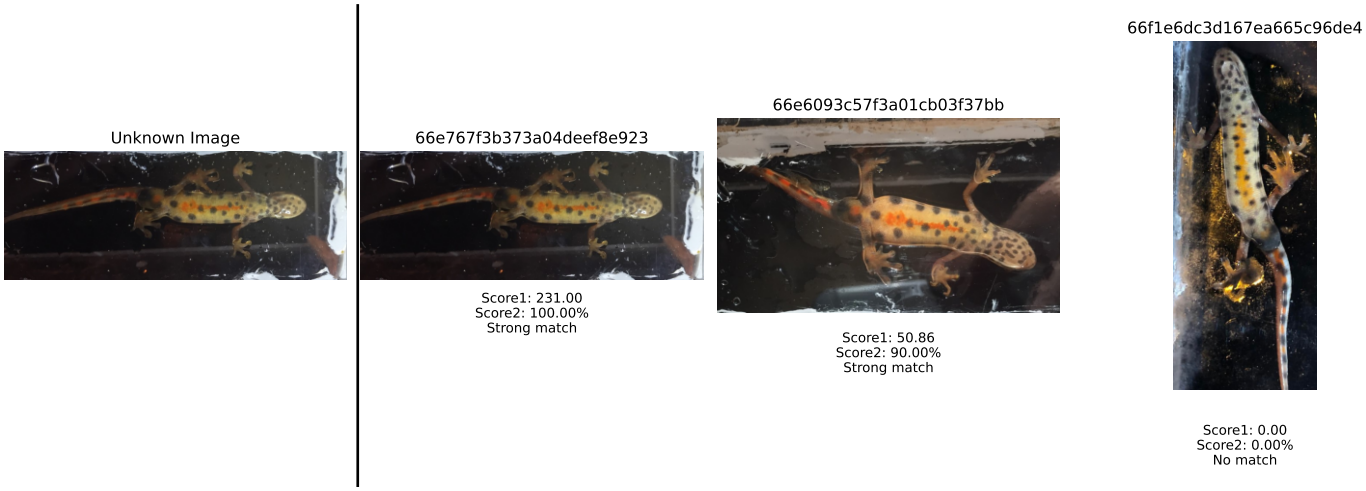
1. Start with a list of coordinates of the spots on the unknown salamander's belly.
2. Compare the unknown salamander one-by-one with salamanders from the database. For each database salamander, also start with a list of spot coordinates on the belly. Then execute steps (a) to (g) for this salamander, called the selected salamander. Afterward, move to the next salamander in the database and repeat steps (a) to (g) until every salamander has been checked.
   (a) Check if the unknown and selected salamander have approximately the same number of spots.
   (b) Generate all possible different triangles between the spots of both patterns and select the triangles relevant for matching.
   (c) Perform a matching algorithm on the triangles so that triangles from both photos can be linked.
   (d) Reduce the number of false matches.
   (e) Match points from both photos based on the matched triangles. Now, spots from both photos are linked.

(f) Repeat steps (a) to (e) once more, but using only the points matched in step (e) for a more accurate result (this is an additional way to filter out false matches).

(g) Calculate a score to determine how good the resulting match between both salamanders is.

3. Display the three selected salamanders on the screen with the highest scores from all salamanders in the database. The corresponding scores are also shown (which will be explained later). Additionally, it is indicated whether a strong, medium, weak, or no match occurs.

   In practice, however, only the matches with $S \geq 0.4$ (where S is the score, explained later) of the three best matches will be shown. This prevents confusing the user by proposing a poor match as a potential candidate and thus mistakenly assigning two wrong salamanders.



Figuur 22: Final result showing the three best matches on the screen.

As seen in Figure 22, the three best matches are located next to the vertical bar. It goes without saying that the leftmost photo is the best since it is the same photo as the unknown image. The second-best match also has a very high score and is considered a strong match. This is extremely good because (if one carefully examines the dot pattern) it is the same salamander as the salamander in the unknown image, but in a completely different position and setting. The third-best option is actually not a match, which means that none of the other salamanders in the database will match either. Therefore, we see that no false positives occur.

The goal is thus to match points of both salamanders by matching triangles. It is evident that a larger number of matched points will result in a better match (and consequently a higher score).

The major advantage of this approach is that the matching algorithm is independent of rotation, scaling, and reflection. This is necessary because we still want a salamander to be recognized if it is later photographed in a new image where the position is different. Moreover, it does not matter if the salamander is curved differently in a new photo, since we have already straightened the "curved dot pattern" in previous steps. In what follows, we will go into more detail about the previous steps.

2.7.1 List of coordinates

For each salamander, we determine the dots on the belly as previously described in this work. If multiple dots are within a distance ε of each other, only one is kept. This parameter ε is our tolerance. By default, it was set to $\varepsilon = 0.01$. Note that we implicitly normalize the coordinates of the dots so that these coordinates lie between 0 and 1. Each remaining dot is actually a rectangle (the Haar Cascade stores dots as rectangles), so to work more conveniently, we store the coordinates of the centers of the detected rectangles. We have chosen not to keep the size of the dots since the dots on a salamander usually all have approximately the same size.

2.7.2 Checking the number of dots

It is striking that the number of dots on salamanders can vary greatly. See for example Figure 23. It is therefore a logical step to first check whether the number of dots approximately matches (less than ten dots difference) before starting the entire matching procedure. This way, salamanders that would not match anyway can be immediately ignored, saving a great deal of computation time when working with a large database. When the following steps mention a database or all salamanders, we mean only those salamanders that have passed this check.

(a) many dots (b) few dots

Figuur 23: Two smooth newts

2.7.3 Selecting triangles to match

After generating all possible triangles (each being a collection of three points) for each dot pattern, we order each set of three points, which we will now call vertices, such that the shortest side of the triangle lies between vertices 1 and 2, the middle side between vertices 2 and 3, and the longest side between vertices 1 and 3. This will help us later in removing false matches. We furthermore denote r_2 as the shortest side and r_3 as the longest side of a triangle.
Next, for each set of three vertices, i.e. points $(x_1, y_1), (x_2, y_2), (x_3, y_3)$, we calculate several values. The goal is to generate a list per salamander whose elements are each triangle with the corresponding calculated values. These values are as follows:

$R = \dfrac{r_3}{r_2}$ ratio between longest and shortest side.
$s = \dfrac{r_3}{\max r_3}$ ratio between longest side and the longest side of all triangles.
$A = \dfrac{1}{2} \cdot (x_1 \cdot (y_2 - y_3) + x_2 \cdot (y_3 - y_1) + x_3 \cdot (y_1 - y_2))$.
$C = \dfrac{1}{r_3 \cdot r_2} \cdot ((x_3 - x_1) \cdot (x_2 - x_1) + (y_3 - y_1) \cdot (y_2 - y_1))$ cosine of the angle at vertex 1.
$S = \dfrac{1}{r_3 \cdot r_2} \cdot ((x_2 - x_1) \cdot (y_3 - y_1) - (y_2 - y_1) \cdot (x_3 - x_1))$ sine of the angle at vertex 1.
$F = \varepsilon^2 \cdot \left(\dfrac{1}{r_3^2} - \dfrac{C}{r_3 \cdot r_2} - \dfrac{1}{r_2^2} \right)$.
$\text{tol}_r = \sqrt{2R^2F}$ tolerance used for R.
$\text{tol}_c = \sqrt{2S^2F + 3C^2F^2}$ tolerance used for C.
$\log_p = \log(\text{perimeter of the triangle})$.

These calculated values allow us to remove some generated triangles (which we then do not use for matching). If $R > 8$ or $C > 0.99$ or $s > 0.85$, we discard the triangle. These thresholds were found experimentally and essentially remove large triangles. There are always many long triangles that almost resemble a straight line because they are so stretched out. These long triangles match too easily with long triangles from other salamanders, increasing the chance of false matches. By applying these criteria, we reduce false matches and retain many triangles for matching.

Next, we calculate the orientation of the triangle, i.e., we ask whether the vertices (from one to three) are traversed clockwise or counterclockwise. We do this using the *Shoelace formula* [14], so if $A > 0$, it is counterclockwise, and if $A < 0$, it is clockwise. If $A = 0$ (in practice if $A \approx 0$), the points are collinear and this 'triangle' is discarded.

Finally, for each remaining triangle, we store the three points, R, C, tol_r, tol_c, $\log_p$, and the orientation. Moreover, note that by knowing R and C, the triangle is uniquely determined, independent of position, orientation, rotation, and scaling.

2.7.4 Matching the triangles

After generating all interesting triangles of both salamanders, we proceed to match these triangles. For each triangle of the unknown salamander, we try to find a corresponding triangle of the selected salamander. This step is the most computationally expensive and causes the algorithm to take

a long time to run. Therefore, a different (faster) implementation was chosen than described in [12].

We perform the following process for each triangle of the unknown salamander; let R_a be the R-value of a triangle belonging to the unknown salamander. Hold this triangle fixed. We then order all triangles of the selected salamander based on the smallest value of $(R_a - R_b)^2$, where R_b is the R-value of a triangle of the selected salamander. After ordering, we check these triangles one by one (starting with those with the smallest $(R_a - R_b)^2$ value). As soon as a triangle satisfies the conditions

$$(R_a - R_b)^2 < \mathrm{tol}_{ra}^2 + \mathrm{tol}_{rb}^2,$$
$$(C_a - C_b)^2 < \mathrm{tol}_{ca}^2 + \mathrm{tol}_{cb}^2,$$

it is matched with the triangle of the unknown salamander. This ensures that only the best triangle b matches with the fixed triangle of the unknown salamander. Then we repeat the process with the next triangle of the unknown salamander until all triangles have been processed. For clarity, note that the values $\mathrm{tol}_{ra}, \mathrm{tol}_{rb}, \mathrm{tol}_{ca}$ and tol_{cb} are as expected (index a refers to the unknown salamander, and index b refers to the selected salamander).

When a match occurs between two triangles, say triangle a and triangle b, additional information is calculated for that specific match. We first calculate

$$\log M := \log_{p_a} - \log_{pb}$$

and call M the scaling factor between the two triangles. Then, we also calculate the *sense* of the match; if both triangles have the same orientation, the match is *same sense*, otherwise it is *opposite sense*.

2.7.5 Reducing the number of false matches

After matching the triangles, we have obtained many false matches. We want to reduce these without removing true matches. We look at the distribution of all $\log M$ values among all matches. Suppose our unknown salamander is indeed the same as a selected salamander. Then the $\log M$ values of all matched triangles should be approximately the same, because the scaling factor M between both photos is constant. Now suppose the selected salamander is not the same as the unknown salamander, then many triangles are falsely matched with random $\log M$ values.

Using this observation about the $\log M$ values, we can quickly separate false matches from good matches. We look at the mean and standard deviation of the $\log M$ distribution and iteratively remove outliers until nothing remains, a predetermined number of iterations (20 in our case) is reached, or all remaining values lie within a certain scaled standard deviation from the mean. For

more information on the specific calculations of the omitted values, please refer to [13] and [12].

Furthermore, we can look at the number of same sense and opposite sense matches remaining. Obviously, if we try to match two identical salamanders, the matched triangles will all be same sense or all opposite sense (depending on the relative orientation and reflection between the salamanders in the two photos). But if we look at two different salamanders and the matches happen by chance (false matches), the number of same and opposite sense matches will be approximately equal. We can therefore filter again by counting the number of same sense and opposite sense matches and removing all matched triangles in the minority group. This primarily removes many false matches.

2.7.6 Matching points based on matched triangles

At this point in the algorithm, we already have matched triangles. However, the goal is to match corresponding dots on the salamanders' bellies. Therefore, we convert matched triangles into matched points. Since there may still be false matches among the triangles, we use a more advanced method for point matching than simply assuming vertex a of one triangle matches vertex a of the other matched triangle.

Each pair of matched triangles contains exactly three pairs of vertices (vertex 1 of triangle a with vertex 1 of triangle b, and so forth). This way, we have a large number of vertex pairs. These vertex pairs are candidate matches. For example, vertex 1 of triangle a may be matched with vertex 1 of triangle b. To ensure that only true matches are labeled as matches, we use a voting system. Each pair of matched triangles casts one vote for each vertex pair. Vertex pairs appearing in multiple matched triangles are likely true matches and get multiple votes (because multiple matched triangles vote for the pair). After voting, we order all vertex pairs from most votes to least votes. Naturally, vertex pairs with the most votes are labeled as matching pairs. We then start labeling with the first vertex pairs in the ordered list and go through them one by one. We stop immediately if a label would be assigned to a vertex pair where one of the vertices is already labeled with another vertex, or if the number of votes of a vertex pair is less than half the votes of the previous pair, or if the number of votes is zero.

In this way, we match pairs of vertices, corresponding to matched dots between the two photos.

2.7.7 Calculating the score between two salamanders

Finally, we must calculate the score S between both salamanders so that (after going through all salamanders in the database) we can determine which salamanders best match the unknown

salamander. Our score S is determined as follows:

$$S = 0.4 \cdot \frac{S_1}{V_{\max}} + 0.6 \cdot S_2,$$

where

V = Total number of votes belonging to matched points.
$V_{\max}$ = Total possible votes; three times the number of matched triangle pairs.
f_T = Percentage of triangles that voted relative to the total triangles remaining after step (b); as in the overview.
S_1 $= V \cdot f_T$.
S_2 = Percentage of matched points relative to the number of points of the unknown salamander.

In other words, the higher the score S, the better the match. We can thus determine the three best matches using this parameter. Moreover, as in Figure 22, both S_1 and S_2 are displayed. More specific information about S_1 can be found in [13], and we mention that if $S_1 \geq 50$ and $S_2 \geq 0.5$, it is a strong match.

Summary of the algorithm

1. Determine the coordinates of the dots on the salamander.
2. Check the number of dots; if there is a large difference in the number of dots, the salamanders will not match.
3. Generate all triangles from the dot patterns and filter these triangles.
4. Match triangles from the unknown salamander with those of the selected salamander.
5. Remove false matches between triangles.
6. Match vertices based on the matched triangles using a voting system.
7. Calculate the score between the two salamanders.

3 App and Server Software

An important part of the project was making the software available in a way that is usable for end users. To achieve this, we developed an app and run the recognition software on a publicly accessible server.
Together, the app and server allow a salamander photo to be submitted, and in response return the salamanders that best match the submitted photo. The user can then either select one of the suggested salamanders or decide that this is a new salamander. The server will save this result in the database, and all information—including the submitted photos—remains accessible.
On the server side, we continued working in Python. Using Flask, we expose a REST API. The app is developed in Flutter, using the Dart programming language, which allows us to build both an app and a website from the same codebase.

For how the app/website works, see Appendix A.

For all underlying code, see the following:
`https://github.com/SpeedyCodes/salamander-tracking`
`https://github.com/SpeedyCodes/salamander-tracking-frontend`

3.1 Photo Quality

To ensure that users of the app do not try to upload and match poor quality photos, we implemented a check for this. When dealing with a new salamander photo, we assign a score based on how good the photo quality is. This can be either “good”, “average”, or “poor”.

Of course, photo quality depends on how well steps such as Pose Estimation, Color Segmentation, and the Haar Cascade perform; therefore, we express quality based on the outputs of these steps. We have chosen to use the following definitions:

1. Good: Pose Estimation works well, as defined in 2.4.1. The Haar Cascade detects at least three spots.
2. Average: Pose Estimation does not work, but the Color Segmentation method works well. The Haar Cascade detects at least three spots.
3. Poor: Pose Estimation does not work, and the Color Segmentation method also does not work and/or the Haar Cascade detects fewer than three spots.

If the photo quality is poor, we ask the user to take or upload a new photo of the salamander in question. When the quality is average, we warn the user that the photo quality is not ideal, but continue with the process nonetheless.

4 Problems Encountered

During the project, we had access to various *types* of photos, each with different lighting or backgrounds. This problem meant that we could not realize spot detection with classical methods like thresholding. To some extent, this also affects the Pose Estimation model, which we trained on photos from previous years. To avoid this problem, Prof. Dr. Nick Schryvers worked behind the scenes on a new setup to photograph salamanders each year in a similar way.

Figuur 24: Setup to take a salamander photo

5 Possible Extensions

In this section, we list possible extensions for this project. We start with some general extensions and then continue with more specific ones.

- Allow photos that have already been processed to be reprocessed and matched again. This is especially useful to test code adjustments with photos already stored on the server.
- Extension of the application to other salamanders or lizards: as long as the animal has similar spots, a process similar to what we used in this project should work to achieve the same goal.

5.1 Trainable Weka Segmentation

Trainable Weka Segmentation [15] is a method to detect spot patterns. We decided not to use this method for three reasons, but researchers in the future could try it out. First, considering the

many false positives that occurred in classical spot detection due to small shadows and various dirt on the salamander's skin, the segmentation approach (which divides the image into regions, which in our case would mean a large region for the skin, with small regions inside for the spots) also seemed particularly prone to false positives. Second, the software was tightly coupled to the ImageJ program: integration with another program was only possible by writing custom software to make the connection, which would have been unnecessarily time-consuming. Third, the results from the Haar Cascade were already sufficiently accurate for our purposes.

5.2 Most Prominent Spots

Our current setup for photographing salamander bellies uses a jewel case where the salamanders are placed. It may be that by flattening the salamanders, more spots on the side are visible than on older photos where we did not use this setup. We are not aware if this is a big or small problem, but a possible solution could be to use not all spots for matching but only the most prominent ones.

We already tried using the x spots closest to the average position of the spots. We tried different choices for x varying between 5 and 15. However, this approach did not yield better results. A more robust approach using the median (instead of the average) and the Mahalanobis distance (instead of the Euclidean distance) can certainly be tried, as well as other methods to detect the most prominent spots.

We also tried detecting the outer spots via the *convex hull* algorithm [16], where we could identify the spots on the side of the salamander as the vertices of the polygon around the spots. However, we encountered the same problem as before: the method seems to work on certain photos and gives good results but fails on others. Further research could definitely be done here.

5.3 Pose Estimation Model

There are several improvements that can be made regarding the Pose Estimation model.

- Retrain the pose estimation model with the new photos from 2025: in the newest setup used to take photos, the current model (trained with data up to 2024) sometimes performs less well. The better the training data matches the setup, the better the model will perform.
- A lighter version of the pose estimation model: pose estimation is by far the most computationally intensive part of the photo processing. Deeplabcut provides the possibility to use smaller types of models, which potentially require less powerful hardware and/or complete pose estimation faster.

5.4 Quality of Life Improvements in the App

There are several improvements that would make the app much more user-friendly.

- Make the properties (name, date, and location) of an individual or sighting editable in the app.
- Sort the display of individuals and sightings by name (alphabetical order, date last seen, etc.).
- Automatically extract location and date from the metadata of new images if present. For location data, this means searching for the already defined location for which the coordinates of the new photo lie within a certain radius of the coordinates of the defined location.
- Allow multiple photos of the same salamander to be submitted at the same time.

6 Conclusion

With this project, we deliver an app/website with which we try to identify individuals of the small water salamander based on the spot pattern on the belly. We do this through several steps. First, we determine a virtual skeleton of the image, which allows us to isolate the salamander's belly. Spot detection then takes place only on the isolated belly, after which we straighten the spot pattern and compare it with other spot patterns. The entire algorithm is integrated into a user-friendly app/website. As mentioned before, there are many possible improvements, but we believe our approach already provides a very good foundation to build on.

7 Referenties

[1] Ravon. (n.d.). Kleine watersalamander. Retreived on 29/02/2024.
https://www.ravon.nl/Soorten/Soortinformatie/kleine-watersalamander

[2] Wikipedia. (n.d.). Thresholding (image processing). Retreived on 09/04/2024.
https://en.wikipedia.org/wiki/Thresholding_(image_processing)

[3] Wikipedia. (n.d.). Otsu's method. Retreived on 09/04/2024.
https://en.wikipedia.org/wiki/Otsu%27s_method

[4] Encord (n.d.). Image Thresholding in Image Processing. Retreived on 11/07/2024.
https://encord.com/blog/image-thresholding-image-processing/

[5] LearnOpenCV (n.d.). Blob Detection Using OpenCV (Python, C++). Retreived on 18/07/2024.
https://learnopencv.com/blob-detection-using-opencv-python-c/

[6] Medium. (Februari 2021). Color Image Segmentation - Image Processing. Retreived on 20/07/2024.
https://mattmaulion.medium.com/color-image-segmentation-image-processing-4a04eca25c0

[7] Mathis, A., Mamidanna, P., Cury, K.M. et al.(2018) DeepLabCut: markerless pose estimation of user-defined body parts with deep learning. Nat Neurosci 21, 1281–1289
doi: 10.1038/s41593-018-0209-y

[8] DeepLabCut. (n.d.). Retreived on 20/03/2024.
https://www.mackenziemathislab.org/deeplabcut

[9] Jones. M. & Viola. P. (2001). Rapid Object Detection using a Boosted Cascade of Simple Features.
https://www.cs.cmu.edu/~efros/courses/LBMV07/Papers/viola-cvpr-01.pdf

[10] Medium. (Augustus 2023). OpenCV: Morphological Dilation and Erosion. Retreived on 22/07/2024.
https://medium.com/@sasasulakshi/opencv-morphological-dilation-and-erosion-fab65c29efb3

[11] Github OpenStitching. (Juni 2024). lir. Retreived on 06/08/2024.
https://github.com/OpenStitching/lir

[12] Groth, J. E. (1986). A pattern-matching algorithm for two-dimensional coordinate lists. The Astronomical Journal, Volume 91, Number 5.
doi: 10.1086/114099

[13] Arzoumanian, Z., Holmberg, J. & Norman, B. (2005). An astronomical pattern-matching algorithm for computer-aided identification of whale sharks Rhincodon typus. Journal of Applied Ecology, 42 , 999–1011.
doi: 10.1111/j.1365-2664.2005.01117.x

[14] Wikipedia. (n.d.). Shoelace formula. Retreived on 29/07/2024.
`https://en.wikipedia.org/wiki/Shoelace_formula`

[15] ImageJ. (n.d.). Trainable Weka Segmentation. Retreived on 10/04/2024.
`https://imagej.net/plugins/tws/`

[16] Wikipedia. (n.d.). Convex Hull. Retreived on 30/07/2025.
`https://en.wikipedia.org/wiki/Convex_hull`

A User Guide for the App/Website

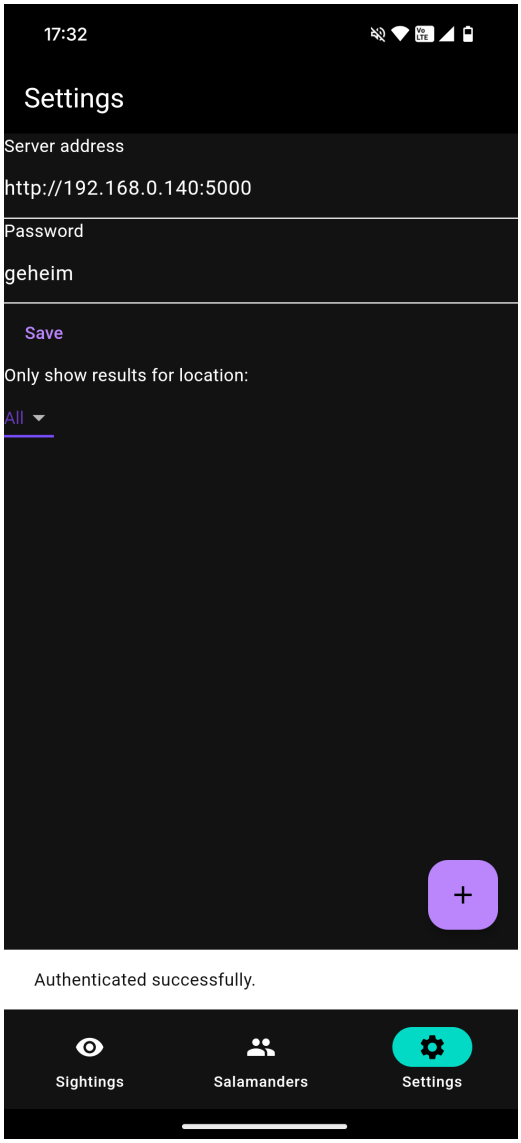
The main screen is shown in the image below: at the bottom are buttons to switch between Sightings (moments when any salamander was spotted), Salamanders (all individual salamanders in the database), and Settings.

(a) Sightings

(b) Individuals

A.1 Setup

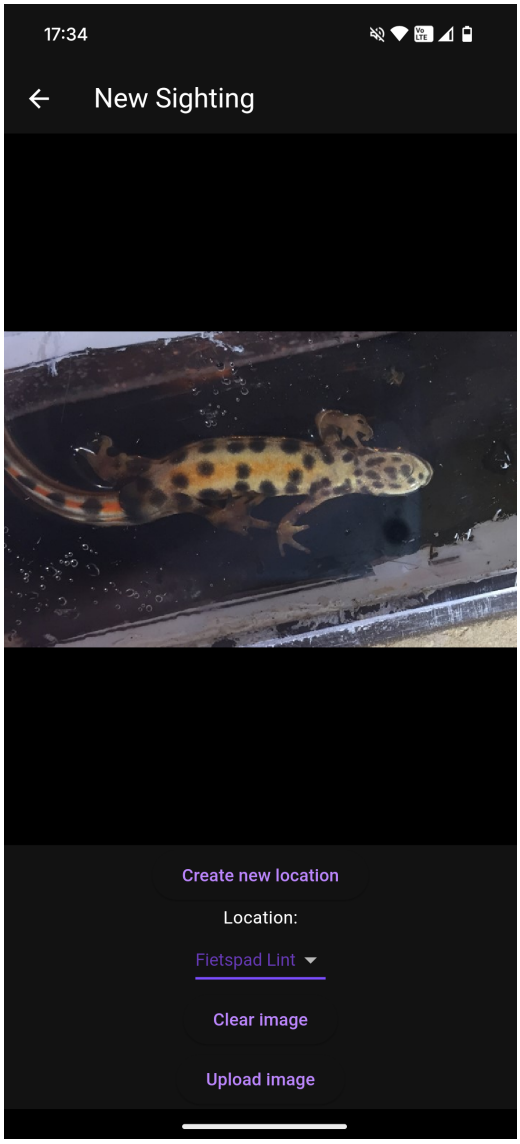
First, a connection to the server must be made. On the website this happens automatically, but on the app the IP address or domain name of the server must be set in the settings, as well as the network port. Additionally, to upload photos yourself, the password must also be entered. Both are stored on the user's device, so they don't need to be entered every time.



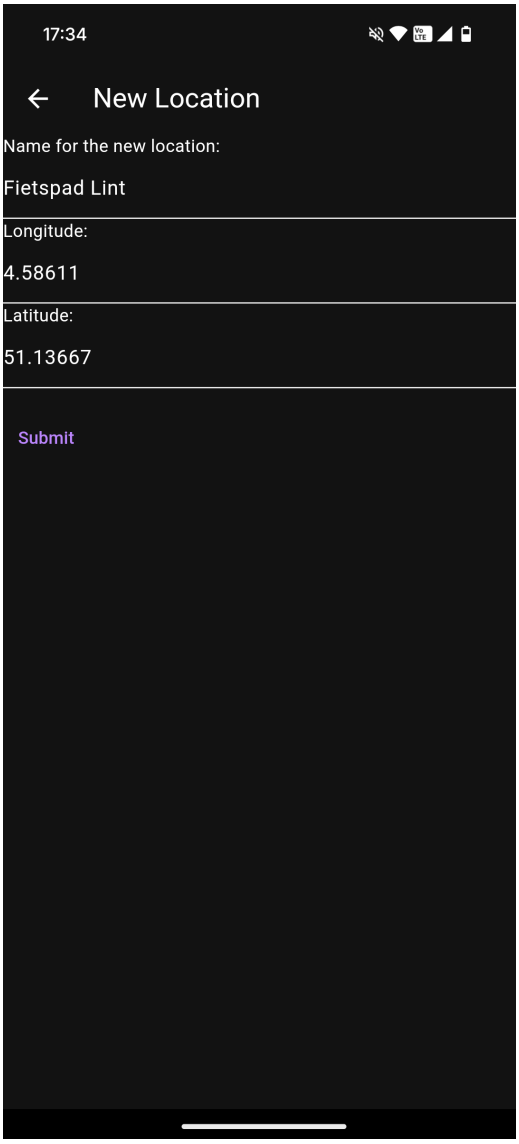
Figuur 25: After entering and submitting the correct information, authentication has succeeded.

A.2 Submitting a Photo

To submit a photo (with the purple + button) you can either take a new photo or upload an existing one. The photo can be assigned to a location: this can be created if it does not yet exist. After uploading, the photo is processed by the server. This usually takes about five to ten seconds.

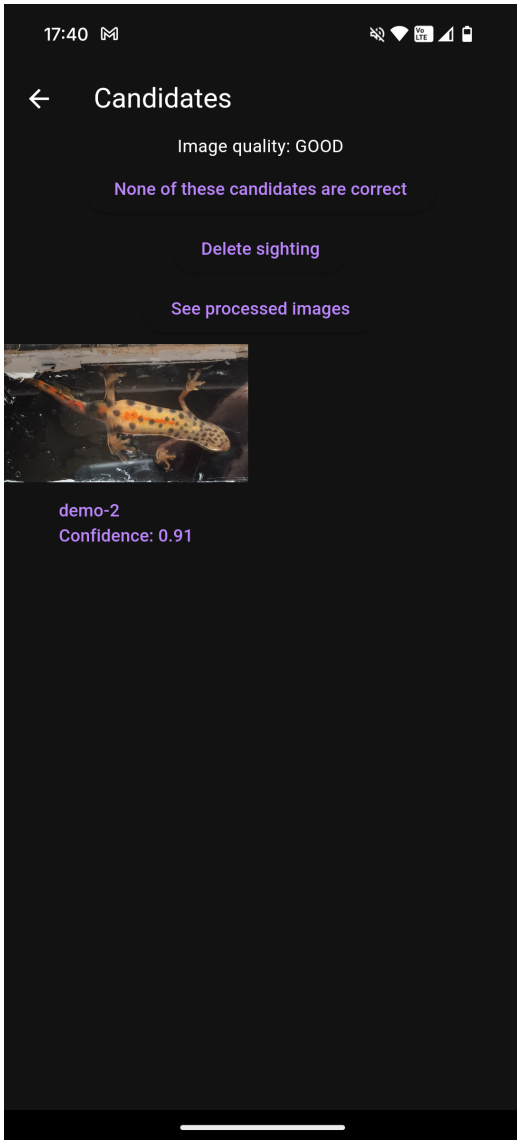


(a) A photo ready to upload

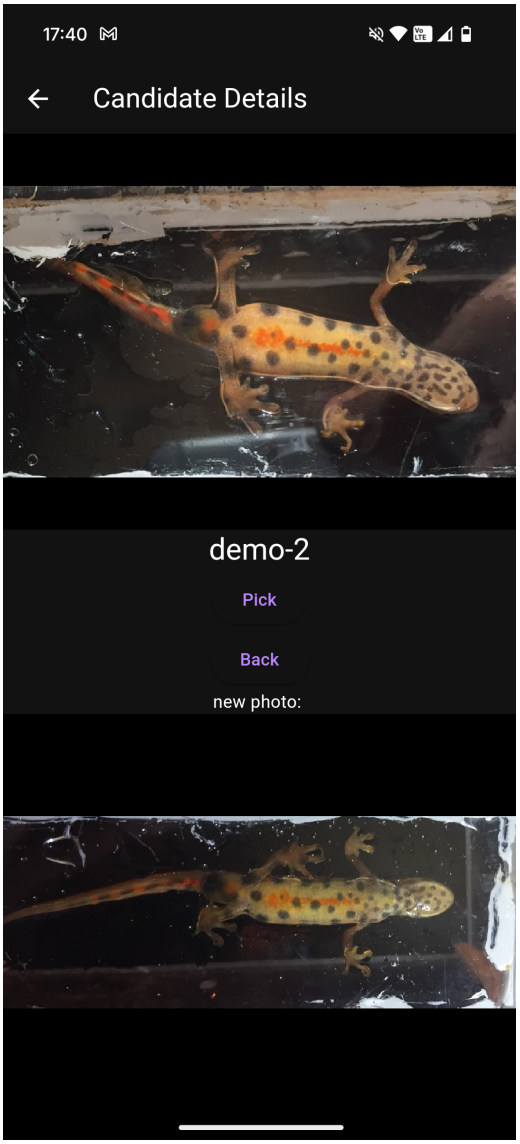


(b) Creating a new location

After processing, a number of existing individuals will be suggested, each with a score between zero and one indicating how confident the program is that this is the individual in the photo. The submitted photo is also shown to make manual comparison easier if needed. In case of an incorrect submission we can delete the submission, and we can always see the intermediate steps of the matching process under "See processed images."



(a) One possible match found with 91% certainty

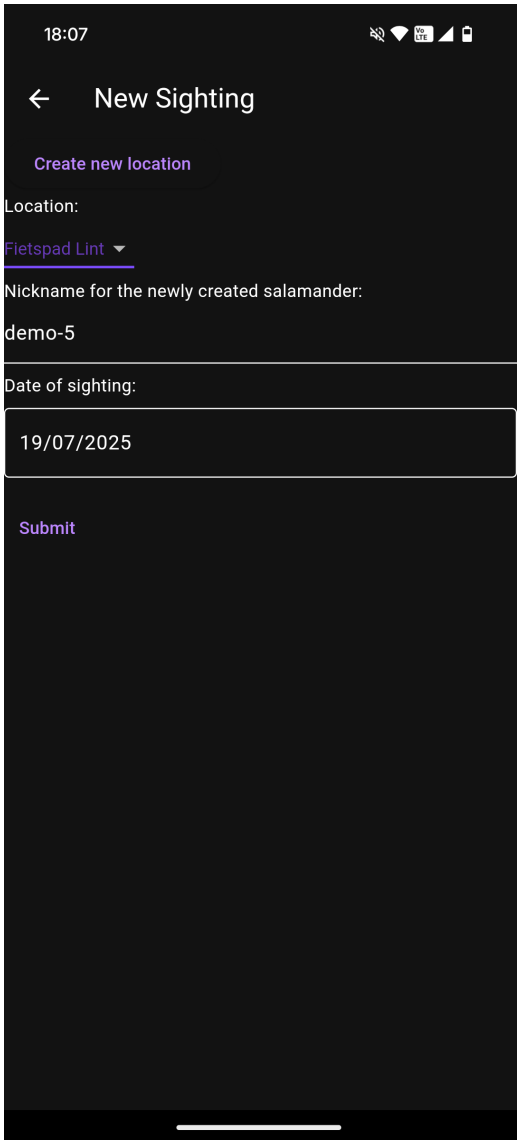


(b) We can manually check if the patterns match

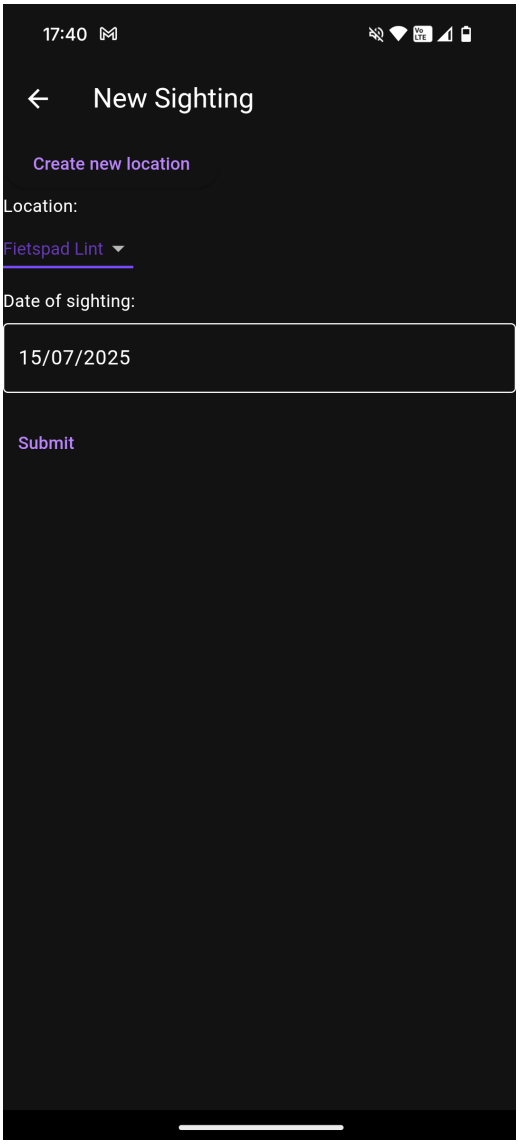
The user can then select one of the suggested individuals, or create a new individual.

A.3 Creating a New Individual / Matching with an Existing Individual

When making this decision, we can also provide the date when the salamander was spotted. A new individual also receives a name.

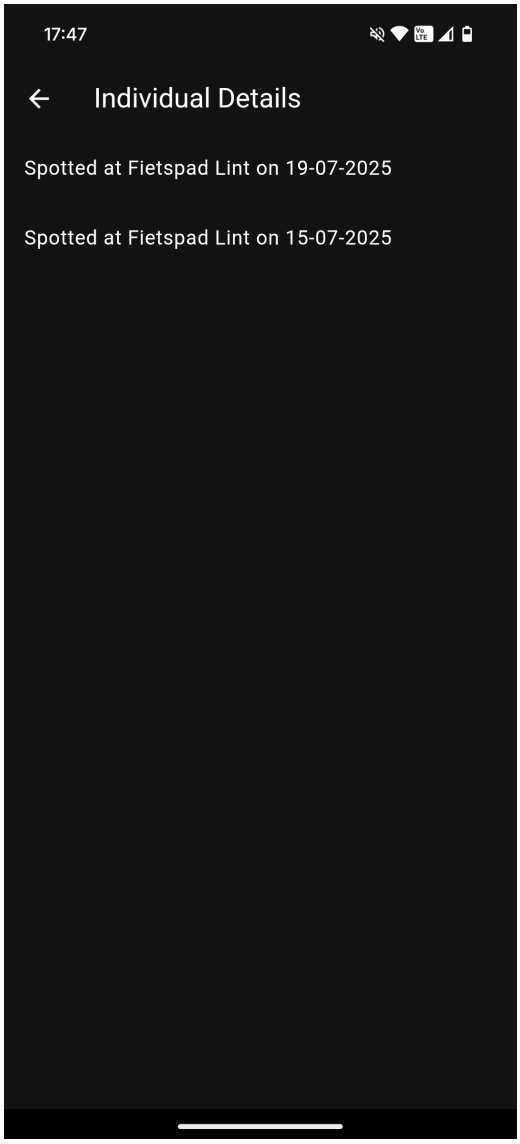


(a) No good match was found, so we create a new individual from this photo

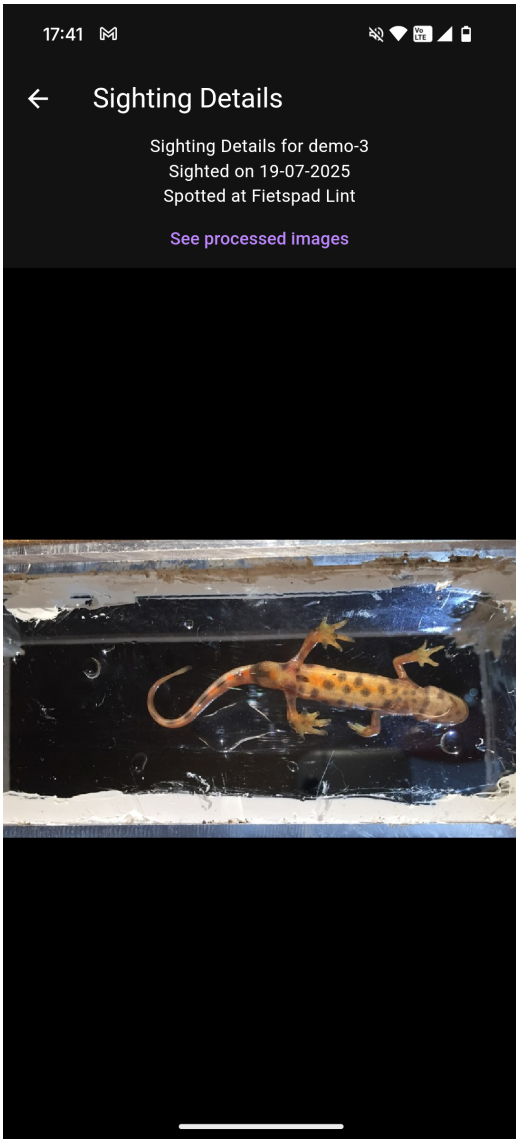


(b) A good match was found, so we link this photo to an existing individual

Finally, we return to the main screen. If we now select the individual to which a new photo was just matched, we indeed see that this individual has been spotted twice.



(a) This individual has been spotted twice



(b) One of the photos of this individual