OATAO is an open access repository that collects the work of Toulouse researchers and makes it freely available over the web where possible.

This is an author’s version published in:  
http://oatao.univ-toulouse.fr/22612

Official URL
DOI : https://doi.org/10.1145/3240508.3243656


Any correspondence concerning this service should be sent to the repository administrator: tech-oatao@listes-diff.inp-toulouse.fr
ABSTRACT
We introduce a new application for annotating images, with the purpose of constituting training datasets for machine learning algorithms. Our open-source software is meant to be easily used and deployed, configured to meet the annotation needs of any use case, and embeddable in crowdsourcing campaigns using the Amazon Mechanical Turk service.

KEYWORDS
annotation; open source software; dataset

1 INTRODUCTION
Image annotations are required in a wide range of applications including image classification (which requires textual labels), object detection (bounding boxes), or image segmentation (pixel-wise classification). The rise and success of deep learning lead to an increasing need for annotations, as training sets should be of a large size for these algorithms to be efficient. Yet, researchers still spend time and resources to create ad hoc tools to prepare those datasets. The application we present in this paper aims at providing a customizable tool to fulfill most image annotation needs.

Many image annotation applications already exist (Table 1). LabelMe [10], one of the most popular, provides an interface for drawing bounding boxes and polygons around objects in an image. It has been extensively used to create datasets for image segmentation. Some more recent softwares share the same goals, with their own specificities. For example, Labelbox [5] and Dataturks [3] provide annotation tasks management, particularly useful when crowdsourcing the annotations; these softwares are proprietary. The VGG Image Annotator (VIA [8]) is an open-source client application like ours, with the specificity of providing annotation attributes, editable in a spreadsheet format.

We release an open-source application [2], entirely client side, meaning that no data is uploaded to any server. Images are loaded from files and annotated locally, in the browser. The simplest tool, from a user perspective, should be immediately available i.e. should not require any additional installation to be fully functional. Our image annotation software is thus a Web-based application, easily configurable to fit users needs, as well as embeddable in the Mechanical Turk platform to design crowdsourcing campaigns.

We first present the features of our application, then describe its architecture. Finally, we explain how it can be used to start crowdsourcing experiments.

2 PRESENTATION OF THE APPLICATION
A screenshot of the application can be seen in Figure 1. The image to be annotated occupies the central part of the screen; a toolbar is located on top, object classes are available on the left and images to be annotated on the right.

Images. Multiple images can be loaded at the same time using the image icon on the top-right corner of the application. These images are not uploaded to the server, and can either be loaded locally from the client’s machine, or from a distant server.

Tools. Our application includes several tools to annotate images. Icons for these tools are depicted in Figure 2. From left to right, the first available annotation is the point, that can be useful to
The information we acquire are the left, right, top and bottom can be derived in virtually any number of variations. For example, listing the tools that should be available. In this case, they all are.

The application code is organized in two parts: a minimalist Node.js server, located in the server / directory. It is statically serving the content of server/dist/ with compression.

Table 1: Most relevant image annotation Web applications.

<table>
<thead>
<tr>
<th>Application</th>
<th>Year</th>
<th>Tools</th>
<th>Configurable interface</th>
<th>Tasks management</th>
<th>Type</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td>LabelMe</td>
<td>2008</td>
<td>bbox, polygon, iterative semi-automatic segmentation</td>
<td>no</td>
<td>Mturk integration</td>
<td>server</td>
<td>OSS</td>
</tr>
<tr>
<td>VIA</td>
<td>2016</td>
<td>bbox, polygon, point, circle, ellipse</td>
<td>no</td>
<td>no</td>
<td>client</td>
<td>OSS</td>
</tr>
<tr>
<td>Labelbox</td>
<td>2018</td>
<td>bbox, polygon</td>
<td>yes</td>
<td>yes</td>
<td>server</td>
<td>private</td>
</tr>
<tr>
<td>Dataturks</td>
<td>2018</td>
<td>bbox, polygon</td>
<td>yes</td>
<td>Mturk integration</td>
<td>client</td>
<td>OSS</td>
</tr>
<tr>
<td>Ours</td>
<td>2018</td>
<td>bbox, polygon, point, stroke, outline</td>
<td>yes</td>
<td>Mturk integration</td>
<td>client</td>
<td>OSS</td>
</tr>
</tbody>
</table>

Listing 1: A configuration file to annotate the PASCAL dataset.

```
{
  "classes": [
    { "category": "Person",
      "classes": [ "person" ]
    },
    { "category": "Animal",
      "classes": [ "bird", "cat", "cow", "dog", "horse", "sheep" ]
    },
    { "category": "Vehicle",
      "classes": [ "aeroplane", "bicycle", "boat", "bus", "car", "motorbike", "train" ]
    },
    { "category": "Indoor",
      "classes": [ "bottle", "chair", "dining table", "potted plant", "sofa", "tv/monitor" ]
    },
    { "category": "Object"
      "annotations": [ "point", "bbox", "stroke", "outline", "polygon" ]
    }
  ]
}
```

Listing 2: A configuration file to include two types of strokes.

```
{
  "classes": [],
  "annotations": [
    { "bbox":
      { "type": "bbox", "variations": [ "fg", "bg" ] }
    }
  ]
}
```

Figure 2: Annotation tools icons

Object classes. For most annotation tasks, we also need to differentiate objects in the images. Typically each annotated area is attributed a class, or label. The PASCAL VOC dataset [9], for example, is composed of 20 classes, grouped by categories:

- **Person**: person
- **Animal**: bird, cat, dog, horse, sheep
- **Vehicle**: aeroplane, bicycle, boat, bus, car, motorbike, train
- **Indoor**: bottle, chair, dining table, potted plant, sofa, tv/monitor

In our application, classes are specified in a JSON configuration file. A strict corresponding config for PASCAL VOC classes is presented in Listing 1.

To attribute a class to an annotation, a user should first select the class in the left sidebar, then use a tool to create an annotation. Selecting a class in the left sidebar also highlights the annotations corresponding to this class.

**Configuration file.** The five annotation tools are optionally made available by the configuration file. In Listing 1, the last line of the depicted configuration file contains an annotations field, listing the tools that should be available. In this case, they all are.

In addition to the five fundamental annotation types, each type can be derived in virtually any number of variations. For example, interactive segmentation algorithms often require foreground and background scribbles. In our application, this would mean the user would need to draw two types of strokes. This can be achieved using the configuration file, as in Listing 2. Such configuration would result in two stroke icons in the toolbar, of different colors, just as in Figure 1.

3 TECHNICAL CHOICES

The application code is organized in two parts:

- A minimalist Node.js server, located in the server / directory. It is statically serving the content of server/dist/ with compression.
- A complete Elm client application, located in the client/ directory. Elm [6, 7] isn’t a JavaScript framework, it is a functional programming language, compiling to JavaScript to run in browsers. Its syntax is inherited from Haskell but far simpler. The compiled application is 150 KB gzipped, which is great for low bandwidth connections.
3.1 The application architecture

The application architecture enforces a unidirectional data transformation flow, visualized in Figure 3. The central entity is the Model. It contains all and every information about our application state. The visual aspect of our application is called the View (basically an HTML rendered document) which is generated by the view function, from the Model. Finally, all events generate messages, of type Msg. The update function, updates the model by reacting to those messages, closing the loop.

All functions are pure, meaning there is no side effect, outputs of functions are entirely defined by inputs. There cannot be global variables mutations, real world events, network interaction etc. Basically such a program would be running in a predestined way from its start to its end, preventing us from loading images and interacting with them. This is why the application is attached to the Elm runtime, provided by the language, transforming all real world events (“side effects”) into our defined set of messages, of type Msg.

The main challenge with pure functions is to describe side effects without performing them. Those are described in three locations:

1. View attributes as DOM event listeners for pointer events.
2. Commands (Cmd) generated by the update function, like loading of images.
3. Subscriptions (Sub) to outside world events like the window resizing.

The Elm runtime takes those side effect descriptions, performs them, and, whenever there is a result / an answer, transforms it into one of our defined messages (Msg) and routes it to our update function.

3.2 The model states

The state is the main component of the Model. It contains the images and configuration loaded as well as the annotations performed. Its type is defined as in Listing 3 and can be modeled as a finite state machine, visualized in Figure 4.

The application available online starts in state 0 (NothingProvided) and enables you to reach state 2 (AllProvided) with buttons to load images and configuration. Two messages called LoadImages and ConfigLoaded produce transitions in the state machine.

3.3 The messages

All modifications of the model are understood by looking at the Msg type definition (Listing 4). The update function then performs the modifications described by those messages.

- The WindowResizes message is triggered when the application is resized. In the update function, it takes the new size and recomputes some view parameters.
- A PointerMsg message is triggered by pointer events (mouse, touch, etc.). In the update function, this is the message activating all the annotations logic code of our application.
The view of this application is based on four components, each
3.4 The view
as an API to create, modify and visualize geometric shapes, useful
3.5 Library and application duality
If you are interested in creating another rendering target than SVG,
under the
contains one module for each tool presented earlier. If you want to
ations are under the
in the context of image annotation.
sequently, the foundation of this application has been extracted in
thought of cases where advanced modifications are required. Con-
tated a configurable application solving most needs. But we also
depending on the current state of the application.
• The top action bar (src/View/ActionBar.elm).
• The center annotations viewer area (src/View/AnnotationsArea.elm).
• The right images sidebar (src/View/DatasetSideBar.elm).
• The left classes sidebar (src/View/ClassesSideBar.elm).

3.5 Library and application duality
In order to offer a turnkey solution to image annotations, we cre-
ated a configurable application solving most needs. But we also
thought of cases where advanced modifications are required. Con-
sequently, the foundation of this application has been extracted in
the independent package elm-image-annotation [4]. It is designed
as an API to create, modify and visualize geometric shapes, useful
in the context of image annotation.
Modules for manipulation and serialization (in JSON) of annota-
tions are under the Annotation.Geometry namespace. It already
contains one module for each tool presented earlier. If you want to
introduce a new tool, this is where you can create a new module.
This package also contains the following important modules, under the Annotation namespace:
• Annotation.Style: defines types describing appearance of
  points, lines and fillings of annotations.
• Annotation.Svg: exposes functions rendering SVG elements
  for each annotation kind.
• Annotation.Viewer: manages the central visualization area,
supporting zooming and translations, relative to an image
frame.
If you are interested in creating another rendering target than SVG,
like canvas, WebGL, …, it would require alternative modules to
Annotation.Svg and Annotation.Viewer. The rest of the code
stay unchanged.
4 CROWDSOURCING ANNOTATIONS
Image annotation interfaces are often used in the context of large
datasets of images to annotate. As such, tasks management for
crowdsourcing campaigns is an important feature. Labelbox and
Dataturks are all-in-one services providing tasks management di-
rectly in their applications. Just like LabelMe, we choose instead
to provide a configuration, ready to use with Amazon Mechanical
Turk (Mturb).
Mturb comes in two sides. A "requester" is defining a set of
tasks while a "worker" is performing them. Workers are payed by
requesters through Mturb service. The concept of a "HIT" (Human
Intelligence Task) characterizes the task unit. In our case, one HIT
means one image to be annotated. We describe in details how to
setup a campaign with our template in the application documenta-
tion.
5 CONCLUSION
In this paper we have introduced our web-based image annotation
application. More information is available in the online documenta-
tion [1]. The application is still actively developed, we welcome
all feedback and contributions.
ACKNOWLEDGMENTS
We would like to thank:
• @tforgione and @GarciaDelMolino for your wise feedbacks.
• @dneg for your Windows tests.
• The online Elm community for their help along the road:
  @evancz for the delightful Elm language, @ianmackenzie for
  your fantastic geometry library, @mdgriffith for your very
  refreshing layout library, @luke for the amazing tool Ellie,
  @norpan, @jessta, @loganmac, @antew, for your invaluable
  help on slack.
REFERENCES
[9] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and
  Andrew Zisserman. 2010. The pascal visual object classes (voc) challenge. Inter-