Shuffler: A Large Scale Data Management Tool for Machine Learning in Computer Vision

Evgeny Toropov  
Carnegie Mellon University  
Pittsburgh, Pennsylvania  
etoropov@andrew.cmu.edu

Paola A. Buitrago  
Pittsburgh Supercomputing Center  
Carnegie Mellon University  
Pittsburgh, Pennsylvania, USA  
paola@psc.edu

José M. F. Moura  
Carnegie Mellon University  
Pittsburgh, Pennsylvania, USA  
moura@ece.cmu.edu

ABSTRACT
Datasets in the computer vision academic research community are primarily static. Once a dataset is accepted as a benchmark for a computer vision task, researchers working on this task will not alter it in order to make their results reproducible. At the same time, when exploring new tasks and new applications, datasets tend to be an ever-changing entity. A practitioner may combine existing public datasets, filter images or objects in them, change annotations or add new ones to fit a task at hand, visualize sample images, or perhaps output statistics in the form of text or plots. In fact, datasets change as practitioners experiment with data as much as with algorithms, trying to make the most out of machine learning models. Given that ML and deep learning call for large volumes of data to produce satisfactory results, it is no surprise that the resulting data and software management associated to dealing with live datasets can be quite complex. As far as we know, there is no flexible, publicly available instrument to facilitate manipulating image data and their annotations throughout a ML pipeline. In this work, we present Shuffler, an open source tool that makes it easy to manage large computer vision datasets. It stores annotations in a relational, human-readable database. Shuffler defines over 40 data handling operations with annotations that are commonly useful in supervised learning applied to computer vision and supports some of the most well-known computer vision datasets. Finally, it is easily extensible, making the addition of new operations and datasets a task that is fast and easy to accomplish.

CCS CONCEPTS
• Information systems → Data management systems: Relational database model; • Computing methodologies → Machine learning: Computer vision; Image segmentation; Object detection; Object recognition; Matching.
INTRODUCTION

In the computer vision academic community, day-to-day work emphasizes primarily algorithms rather than data. From this point of view, annotated image datasets are ideally built once and remain fixed. This approach allows the community to use datasets as benchmarks. Researchers choose to store these datasets in formats that are most common and fast to load for machine learning (ML) packages. In contrast, for a data scientist in industry, the task is not necessarily to improve an algorithm, but rather to try different algorithms and tasks on various partitions and modifications of the same dataset. In this case, a dataset is not considered static, but rather constantly altered to fit the task at hand. In turn, multiple versions of the same dataset need to co-exist in a centralized or a distributed storage system. Ideally, a practitioner would want 1) a simple way to manipulate image data and its annotations, and 2) a file format that allows to store multiple copies of the annotation set in an organized and efficient way and to inspect them manually.

Data manipulation tools are sometimes packaged with a dataset, but they typically allow to perform only a limited number of operations only on that particular dataset and often for a single programming language. An example is the PASCAL VOC dataset [8] that had 10 releases in different years and with each release coming with a Matlab toolbox for that specific year. In alternative, researchers often write in-house small one-time scripts to quickly alter a dataset, for example, to change the size of object bounding boxes. This set of scripts usually contains duplicate code, tends to be error prone, and with time becomes increasingly difficult to maintain.

Datasets typically come in a custom format, which usually includes images and annotation files in one of the following formats: xml, txt, or json. Table 1 presents an overview of several popular object detection datasets in the area of autonomous driving and the formats of the associated annotation files. On the one hand, these formats are human-readable, but on the other hand, quite slow to load. Additionally, changing annotations and saving them as a copy means duplicating the whole directory with the annotation files, which is inconvenient and slow. Many development kits cache annotations by serializing them with formats such as pickle\(^1\) or protobuf\(^2\). Such formats are easy to load by a machine learning framework and convenient to store, but they are not interpretable by humans and can not be inspected or modified outside of a specialized programming environment.

To sum up, we consider a typical data preparation workflow of a computer vision practitioner to consist of three steps (Figure 2): 1) download or collect a dataset, 2) modify annotations, and 3) serialize the dataset. We further consider a common situation when multiple modifications of annotations are used. Modifications could be a chain of trivial tasks, for example, removing objects at image boundaries and then increasing the size of bounding boxes. We note 1) the lack of software for manipulating image data and annotations, and 2) a convenient format to store annotations.

In this work, we close this gap by proposing a software toolbox, Shuffler, designed specifically for manipulating annotations. It employs widely known relational databases and the associated SQL query language for storing and manipulating annotations. The proposed toolbox is heavily based on SQL and allows to chain multiple operations in a single command. Annotations are stored in an relational database (Sqlite, MySql, ...) with schema designed to cover the bulk of the common tasks in computer vision. The proposed solution satisfies the following properties:

- it has basic manipulation tools and allows to easily add new functions;
- annotations are fast to load and modify and convenient to store;
- annotations are stored in a format that can be manually inspected and edited;
- it is agnostic to the format of how images are stored on disk;
- it supports image classification, object detection, semantic segmentation, and object matching tasks in computer vision.

The toolbox, the installation instructions, the manual, and use cases are available at https://github.com/kukuruza/shuffler.

| Dataset          | Annotation file format      | Load + parse time |
|------------------|----------------------------|-------------------|
| PASCAL2012 [8]   | xml file per image         | 1 sec             |
| KITTI [9]        | txt file per image         | 6 sec             |
| Cityscapes [5]   | json file per image        | 20 sec            |
| BDD [18]         | a single json file         | 4 sec             |

Figure 2: Typical data preparation pipeline lacks convenient tools on step 2.

2 RELATED WORK

2.1 Relational Model format for annotations

Questions of data management have been studied in great detail since the early days of automated information processing. It proved convenient to store data using the Relational Model, where information is stored in a set of interconnected tables. Shortly after the Relational Model was proposed in 1970 [4], research was started on developing an appropriate language to handle it [2], until finally the International Organization for Standardization (ISO) accepted

---

1https://docs.python.org/3/library/pickle.html
2https://developers.google.com/protocol-buffers/?hl=en
the Structured Query Language (SQL) as a standard in 1987. Since then the Relational Model has remained dominant in industry.

Typically, a computer vision dataset can be well described by the Relational Model since a machine learning algorithm is trained on samples of training data with the same structure. In particular, in the computer vision field, an image classification algorithm can be trained on pairs {image, label}. Pairs {image, list of bounding box coordinates} are, for example, used to train in an object detection task. Semantic pixel-wise image segmentation is trained on pairs {image, label map}, where label map is of the same size as the image.

Storing annotations in a relational database has advantages over formats such as xml or json. First, the relational database is human-friendly: any SQL editor can be used to browse through the contents and to query for specific entries. Second, such database can be loaded in milliseconds, while it takes seconds to load and parse annotations in their original formats (Table 1). Importantly, as we show in the next chapters, the Relational Model proves to be a convenient format for the purpose of building a toolbox around it because of its inherent structure and the powerful SQL language.

So far, the Relational Model has not been popular in the modern computer vision field for several reasons. First, the Relational Model does not fit well in the academic workflow. For distributing a dataset, researchers choose universally known, human readable formats (Table 1). For training, the dataset can be serialized as files in Google protobuf format or similar in order to minimize the time to read the data from disk. In this work, we argue that a Relational Model is needed, not for dataset distribution or for data serialization, but rather for the intermediate step of modifying and filtering annotations, extracting a subset of the dataset, visualizing information, and other similar related tasks.

Second, different datasets define different information that needs to be stored. For example, angles are important for objects in images from traffic surveillance cameras but are irrelevant for generic images such as in the Pascal VOC dataset [8]. That makes any particular schema hard to generalize across datasets. Instead of providing a universal schema, we focus on setting up a base that can be customized for particular needs of a particular company, group, or user. At the same time, our design of the schema is generic enough to fit the tasks of image-level classification, object detection, semantic segmentation, and object matching, which cover a large part of the computer vision landscape.

### 2.2 Dataset management in Computer Vision

As computer vision becomes more useful for practical applications, a number of solutions facilitating the life-cycle of a project has emerged. These solutions address different challenges of the ML pipeline.

First, some systems are designed specifically for annotating data for computer vision applications. Examples include publicly available LabelMe [17], VGG Image Annotator [7], and CVAT, as well as commercial Supervise.ly, Playmate, and Labelbox. These systems offer sophisticated tools for human annotators to label images in order to prepare training data for different types of machine learning tasks. Though our proposed toolbox, Shuffler, offers basic functionality for image labelling, its primary focus is processing the output of such image annotation systems.

Second, an important part of a machine learning pipeline is loading and augmenting image data. Numerous libraries, such as DALI released by NVIDIA, have been proposed for this task. In Figure 2, we refer to this part of the pipeline as step 2. In turn, Shuffler is employed on step 2 to prepare a dataset of training data that can be further loaded and augmented during training.

Next, end-to-end product life-cycle management systems have been proposed, such as ModelHub [13], or commercial Allegro. These systems focus on managing experiments and trained models, while the goal of Shuffler is to provide instruments to manage training data.

As far as we know, there is no flexible, publicly available instrument to facilitate manipulating image data and their annotations in order to prepare data for a ML algorithm. The goal of Shuffler is to close this gap.

The rest of the paper is organized as follows. Section 3 describes the schema of the SQL database, the covered use cases, and the limitations of the schema. Sections 4 and 5 present the Shuffler toolbox and the operations it supports. This is further explored in Section 6 that illustrates an important feature – chaining multiple operations in a single command. Section 7 introduces the interface to Keras and PyTorch machine learning frameworks. Section 8 explains adding new functionality to the toolbox. We conclude the paper in Section 9.

### 3 DATABASE SCHEMA

The core of this work is the proposed Relational Model for common datasets that were collected to train for image classification [6, 12], object detection [8, 9, 12, 17], semantic segmentation [10, 12, 14, 18], and object matching tasks [16]. The proposed schema is presented in Figure 3 in the form of an Entity Relation (ER) diagram. The diagram presents five tables. Each table consists of several fields. Two fields in any table are mandatory to be filled in: the unique Primary Key (PK in bold font) and the Foreign Key (FK in italic font).

In this section, we describe a typical dataset in computer vision applications and show how it corresponds to this schema. An example is provided in Figure 4. It illustrates a traffic camera dataset and is focused on the tasks of object detection and matching.

Typically, a computer vision dataset consists of a number of images. Each image corresponds to an entry in images table and is uniquely defined by the `imagefile` field. Normally, this field contains the path to the image, but may also have other descriptors, such as the frame in a video. In the case when the dataset is focused on the image classification task, each image is associated with a label, such as “cat” or “dog.” The label is recorded in the `name` field.

---

1https://github.com/opencv/cvat
2https://supervise.ly
3https://playment.io
4https://labelbox.com
5https://github.com/NVIDIA/DALI
6https://allegro.ai
Figure 3: Database schema

Figure 4: A database populated according to the schema in Figure 3. Only non-empty columns are shown. The same bus is recorded under objectid=1 and objectid=2 in the two images and is matched via match=1. The two bus objects are assigned a bounding polygon, while the car is assigned a bounding box.
of the images table. In the case when the dataset focuses on the semantic segmentation task, each image comes with a segmentation map. In this schema, the segmentation map descriptor is recorded in the maskfile field of the table images.

A dataset may contain multiple objects in each image. In this schema, each object corresponds to one entry in table objects and has the unique objectid field. Each image may contain zero, one, or multiple objects, while any object must belong to some image. That is encoded as "mandatory one to many optional" relations between images and objects tables. In the detection setup, each object is characterized by its bounding box, which is encoded as $[x, y, width, height]$ in table objects. Besides, each object typically belongs to a single class out of the pre-defined set of classes, for example, "car" and "truck" in Figure 4. The class name, if present, is to be encoded in the name field of table objects.

Furthermore, an object may have auxiliary properties. For example, some cars in Figure 4 have the assigned color and the two angles, yaw and pitch, as seen by the camera. This extra information is recorded in the table properties, which is linked to the object table. Any object may have any number of different properties, thus this schema is applicable to datasets with unstructured information on their objects.

Some datasets, such as LabelMe [17], provide fine-grained boundaries of their objects using polygons instead of rectangular bounding boxes. Our proposed schema supports this via table polygons. The rows in polygons table with the same value of objectid field describe the points in a closed polygon. Thus the order of their id's matters: the points should be recorded either clockwise or counter-clockwise. It may happen that two polygons correspond to the same object. Then they should be differentiated with the name field.

Finally, the task of object matching is supported with table matches. The matched objects are recorded as separate entries in the matches table, and, as such, they have different id and objectid. However, they share the same value of the match field. That value uniquely identifies these particular matched objects among other matches. This idea is illustrated in Figure 4.

The proposed schema comes with certain limitations. For example, it does not support the increasingly popular format of video clips [10, 18]. It also does not inherently support 3D bounding boxes of objects [9, 10], though they can be encoded with our schema via the properties table. In both cases the schema can be trivially extended, but such extension is beyond the scope of this work.

4 TOOLBOX
The SQL schema alone would be useless without the tools that take advantage of it. We developed a toolbox that allows to 1) import annotations from other formats, 2) save annotations as an SQLite database, 3) modify them, and 4) export them into other formats.

A user interacts with the toolbox by executing the program shuffler.py from the command line. In a minimal working example below, Shuffler creates a new database and prints information about it to the standard output:

```bash
$ ./shuffler.py importKitti
--images_dir=kitti.db
--detection_dir=kitti.db
```

In this example, Shuffler calls the sub-command printInfo. In general, all the work with Shuffler is performed via sub-commands. The example below illustrates how the sub-command importKitti and command-line arguments --images_dir and --detection_dir are used to import annotations from the KITTI dataset [9]. It is assumed that KITTI has been downloaded and is located in the directory KITTI.

```bash
$ ./shuffler.py importKitti
--images_dir=kitti.db
--detection_dir=kitti.db
```

The next example shows how to load the recorded kitti.db and print basic information about it:

```bash
$ ./shuffler.py printInfo
```

Finally, when both --i and --o are specified, a database is loaded, modified, and saved under a different name:

```bash
$ ./shuffler.py -i=kitti.db -o=clean.db
```

In this example, Shuffler calls the sub-command filterObjectsAtBorder. In general, all the work with Shuffler is performed via sub-commands.
The effects of all the combinations of `-i` and `-o` command-line arguments are summarized in Table 2. Finally, for completeness, we present the Shuffler interface in Listing 1. In the next section, we focus on individual subcommands.

```bash
shuffler.py [-i IN_DB_FILE] [-o OUT_DB_FILE] 
[--relpath RELPATH] 
[--logging (10,20,38,40)] 
[-h] 
sub-command-1 [sub-arguments-1] 
[sub-command-2 [sub-arguments-2] ...]
```

Listing 1: Shuffler interface. Input/output is controlled by `-i/-o`, `--relpath` determines the root path that all imagefiles are relative to, `--help` prints out more information about arguments.

5 SUB-COMMANDS

Sub-commands are the workhorse of Shuffler. Their complete list is presented in the project’s official page, but can be printed out with:

```bash
$ ./shuffler.py -h
```

Besides the global command-line arguments, each sub-command defines its own arguments. One can get help on an individual sub-command and its arguments like in the example below:

```bash
$ ./shuffler.py printInfo -h
```

The sub-commands can be divided into several major groups:

1. **Import** group allows to add annotations from datasets with different formats to a new or existing database. At the moment, the functions `importPascalVoc2012`, `importKITTI`, `importLabelme`, among others, are implemented.

2. **Filter** group serves to remove images or objects from the database according to some criteria. The functionality of `filterEmptyImages` and `filterObjectsAtBorder` can be inferred from their names. The function `filterObjectsSQL` is a more flexible tool that filters out images or objects based on an SQL query. For example:

```bash
$ ./shuffler.py -i=my.db' 
filterObjectsSQL --where_object='width<64 AND name='car''
```

Under the hood, this sub-command opens `my.db` and run the `DELETE` SQL query on its tables. Its simplified version for the objects table may look like this:

```sql
DELETE FROM objects WHERE width<64 AND name='car'
```

3. **Modify** group changes entries in a database. For example, `expandBoxes` expands bounding boxes from each side, `filterObjectsSQL` can be used to exclude objects from the database. For example:

```bash
$ ./shuffler.py -i=predictions.db' 
evaluateDetection --gt_db_file=ground_truth.db'
```

6 CHAINING OPERATIONS

One main motivation for the toolbox design was the ability to conveniently chain operations. An example is shown in Listing 2. Commands are chained via the vertical bar symbol `|`, that must be escaped in a Unix shell as `\`, `'|', or `]]`. The program exports bounding boxes of cars into a new dataset that will be further sent to LabelMe annotators. First, expand bounding boxes by 20% from every side. Then select those cases that intersect with other cars by more than 30%, those at the image border, those with bounding boxes of size less than 64 pixels in either dimension, and those with names other than ”van,” ”taxi,” or ”sedan.”
Then the bounding boxes are cropped, scaled to $64 \times 64$, and written to a new dataset – a database with a video. Note that five calls are chained: expandBoxes, filterObjectsByIntersection, filterObjectsAtBorder, filterObjectsSql, and cropObjects. Without chaining, one would need to carefully store intermediate results – one after each operation.

| Input                      | Output                  | Description                                               |
|----------------------------|-------------------------|-----------------------------------------------------------|
| -i in.db                   | -                      | Create a new database in-memory. Discard it at the end.   |
| -                          | -o out.db               | Open in.db in read-only mode.                             |
| -i in.db                   | -o out.db               | Create a new out.db and commit transactions there.        |
| -i in.db                   | -o out.db               | Open in.db but commit transactions to out.db. Backup out.db if it already exists. |

### 7 INTERFACE TO KERAS AND PYTORCH

Apart from the functionality of `shuffler.py`, the toolbox also provides support for loading data in PyTorch [15] and Keras [3] directly from a database. Keras allows to use a custom `DataGenerator` class that loads data by batch. At the same time, a custom `ImagesDataset` class in PyTorch can be used to load individual items, which are further collected into batches by PyTorch’s native `DataLoader`.

File `interface.keras.generators.py` provides custom generator classes, which can load data for the tasks of image classification, semantic segmentation, and object detection. The advantage of this class is the ability it gives a user to choose data entries from the database with where_images and where_objects arguments. Similarly, file `interface.pytorch.datasets.py` contains classes inherited from `torch.utils.data.Dataset`. Arguments where_images and where_objects can be used in the same way to use only a subset of the dataset. An example of using this class for semantic segmentation is shown in Listing 3.

```python
import torch.utils.data
from shuffler.utils import ImageDataset

data = ImageDataset(db_file='my.db')
```

### 8 IMPLEMENTATION DETAILS

The program Shuffler is implemented as a Python script, which can call one of many functions. For example, imagine a user typed:

```
./shuffler.py -i my.db
```

Shuffler will parse the first command line arguments `-i my.db`, open the database `my.db`, and get the “cursor” that allows to send queries to the database. It then will find a subparser for the function `filterObjectsAtBorder`. The subparser will parse the remaining argument `--border_thresh_perc 0.01`. This functionality is implemented via the `argparse` package. Then Shuffler will call the function `filterObjectsAtBorder` passing it the cursor and the parsed arguments.

Shuffler analyzes command line arguments sequentially and executes the sub-commands when it runs across them in the command line. That allows to chain sub-commands inside a single call to `shuffler.py`. The database is opened or created by Shuffler in the beginning according to rules 2. The database cursor is passed to each function that is called by Shuffler. Therefore, each function has the possibility to perform transactions and introduce changes to the database. These changes may or may not be committed by Shuffler in the end, depending on whether the arguments `-o out.db` were passed to Shuffler.

All functions share the same interface:

```python
def myFunction(cursor, args)
```

where `cursor` is an SQLite3 cursor for the open database, and `args` is the named namespace with parsed arguments for `myFunction`. That makes adding sub-commands straightforward. To add a new function and its associated sub-command, one first needs to pick a file where the function would fit the best based on its functionality. For example, all operations of filtering reside in `filter.py`. Then one needs to 1) write its body, which implements the interface, 2) write the parser, and 3) register the parser in the `add_parsers` function. Listing 4 provides the skeleton for function `myFilter`.

```python
# Register your new parser in function add_parsers.
def add_parsers(subparsers):
    parser = subparsers.add_parser('myfilter',
                                   description='My new operation')
    parser.set_defaults(func=myFilter)
    parser.add_argument(' mandatory_arg', required=True)
    parser.add_argument(' extra_arg', type=int, default=1)
```
Listing 4: Adding a new sub-command `myFilter` to Shuffler.

9 CONCLUSION

Imagine a computer vision practitioner training a vehicle detector on the KITTI [9] (or another dataset) for an autonomous vehicle. As such, he/she wants to remove all labels except for “Car,” “Van,” “Truck,” and “Tram,” then to experiment if the bounding boxes around objects need to be expanded, if the objects on the image boundary should be removed, and whether it is better to also remove small objects. Furthermore, he/she would like to quickly see the distribution of objects by class and by size in the dataset. All in all, the researcher is using the workflow depicted in Figure 2.

Annotations in the KITTI dataset come as text files, one per each image (Table 1). Making each of the dataset modifications described above requires the researcher to use KITTI’s toolbox to load the data, write the custom code to filter/modify annotations, and write the annotations as the new set of text files, while somehow bookkeeping the paths to each annotation set.

The tool, Shuffler, that we developed allows to perform all these operations out of the box. Each of the original and modified annotations is stored as a SQLite database file. A researcher can directly use the provided `DataGenerator` class in Keras or `ImagesDataset` class in PyTorch during training and testing. The toolbox is made public at:

https://github.com/kukuruza/shuffler

In more general terms, we consider the workflow of a ML expert in the computer vision domain. Multiple modifications of the dataset are an important part of this workflow. No public tool or data representation is specifically designed to address this problem. In this work, we close this gap.

The design of our toolbox was motivated by the fact that data used in the computer vision field fits well the Relational Model. The data model or the toolbox that we presented in this paper do not target the questions of the convenient distribution of the dataset for public use or the efficient data storage for fast loading by machine learning packages. Instead, we have focused data preparation, labelling, exploration, and evaluation of ML models.

REFERENCES

[1] G. Bradski. 2000. The OpenCV Library.
[2] Donald D. Chamberlin and Raymond F. Boyce. 1974. SEQUEL: A Structured English Query Language. In 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control (SIGFIDET ’74). ACM, New York, NY, USA, 249–264. https://doi.org/10.1145/800296.811515
[3] François Chollet et al. 2015. Keras. https://keras.io
[4] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. Commun. ACM 13, 6 (June 1970), 377–387. https://doi.org/10.1145/362384.362685
[5] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. 2016. The Cityscapes Dataset for Semantic Urban Scene Understanding. In The IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 3213–3223. https://doi.org/10.1109/CVPR.2016.350
[6] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In The IEEE Conference on Computer Vision and Pattern Recognition. 248–255. https://doi.org/10.1109/CVPR.2009.5206848
[7] Abhishek Dutta and Andrew Zisserman. 2019. The VGG Image Annotator (VIA). arXiv preprint arXiv:1904.10699 (2019).
[8] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. 2015. The Pascal Visual Object Classes Challenge: A Retrospective. International Journal of Computer Vision 111, 1 (Jan. 2015), 98–136.
[9] A. Geiger, P. Lenz, and R. Urtasun. 2012. Are we ready for autonomous driving? The KITTI vision benchmark suite. In The IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW). 1067–10676. https://doi.org/10.1109/CVPRW.2012.6248074
[10] X. Huang, X. Cheng, Q. Geng, B. Cao, D. Zhou, P. Wang, Y. Lin, and R. Yang. 2018. The ApolloScape Dataset for Autonomous Driving. In The IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW). 2018.00141
[11] J. D. Hunter. 2007. Matplotlib: A 2D graphics environment. Computing In Science & Engineering 9, 3 (2007), 90–95. https://doi.org/10.1109/MCSE.2007.55
[12] Trung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. 2014. Microsoft COCO: Common Objects in Context. In The IEEE European Conference on Computer Vision (ECCV), David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars (Eds.). Springer International Publishing, Cham, 740–755.
[13] H. Miao, A. Li, L. S. Davis, and A. Deshpande. 2017. ModelHub: Deep Learning Lifecycle Management. In 2017 IEEE 33rd International Conference on Data Engineering (ICDE). 1393–1394. https://doi.org/10.1109/ICDE.2017.192
[14] G. Neuhold, T. Ollmann, S. R. Bulò, and P. Kontschieder. 2017. The Mapillary Vistas Dataset for Semantic Understanding of Street Scenes. In The IEEE International Conference on Computer Vision (ICCV). 5000–5009. https://doi.org/10.1109/ECCV.2017.534
[15] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In The International Conference on Neural Information Processing Systems (NIPS-W).
[16] J. Per, V. S. Krek, R. Mandjeck, M. Kristan, and S. Kovacic. 2012. Dana36: A Multi-camera Image Dataset for Object Identification in Surveillance Scenarios. In 2012 IEEE Ninth International Conference on Advanced Video and Signal-Based Surveillance. 64–69. https://doi.org/10.1109/AVSS.2012.33
[17] Bryan C. Russell, Antonio Torralba, Kevin P. Murphy, and William T. Free- man. 2008. LabelMe: A Database and Web-Based Tool for Image Annotation. Int. J. Comput. Vision 77, 1-3 (May 2008), 157–173. https://doi.org/10.1007/s11263-007-0090-8
[18] Huaize Xu, Yang Gao, Fisher Yu, and Trevor Darrell. 2017. End-to-End Learning of Driving Models from Large-Scale Video Datasets. In The IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 3530–3538. https://doi.org/10.1109/CVPR.2017.376