AISS-CV Pandemic Package Documentation

Release 0.0.1

Luca Deck, Johannes Jestram, Joel Oswald
Alexey Rosenberg, Manuel Sauter, Bolatito Zäch

Jul 23, 2021

PROJECT

Welcome to our project page! Our work took place during Summer Term 2021 at KIT in the course “Artificial Intelligence in Service Systems - Applications in Computer Vision” . This Document consists of three sections. First, we explain the use case and further details of the project itself. The second part consits of a detailed explanation and discussion of the system architecture. Lastly, we provide a documentation of our code base.

CHAPTER

ONE

SUMMARY

We built a system that utilizes a NVIDIA Jetson to visually detect packages and - if existent - damaged areas on the detected packages.

> **Warning:** This is not production ready software yet

CHAPTER

TWO

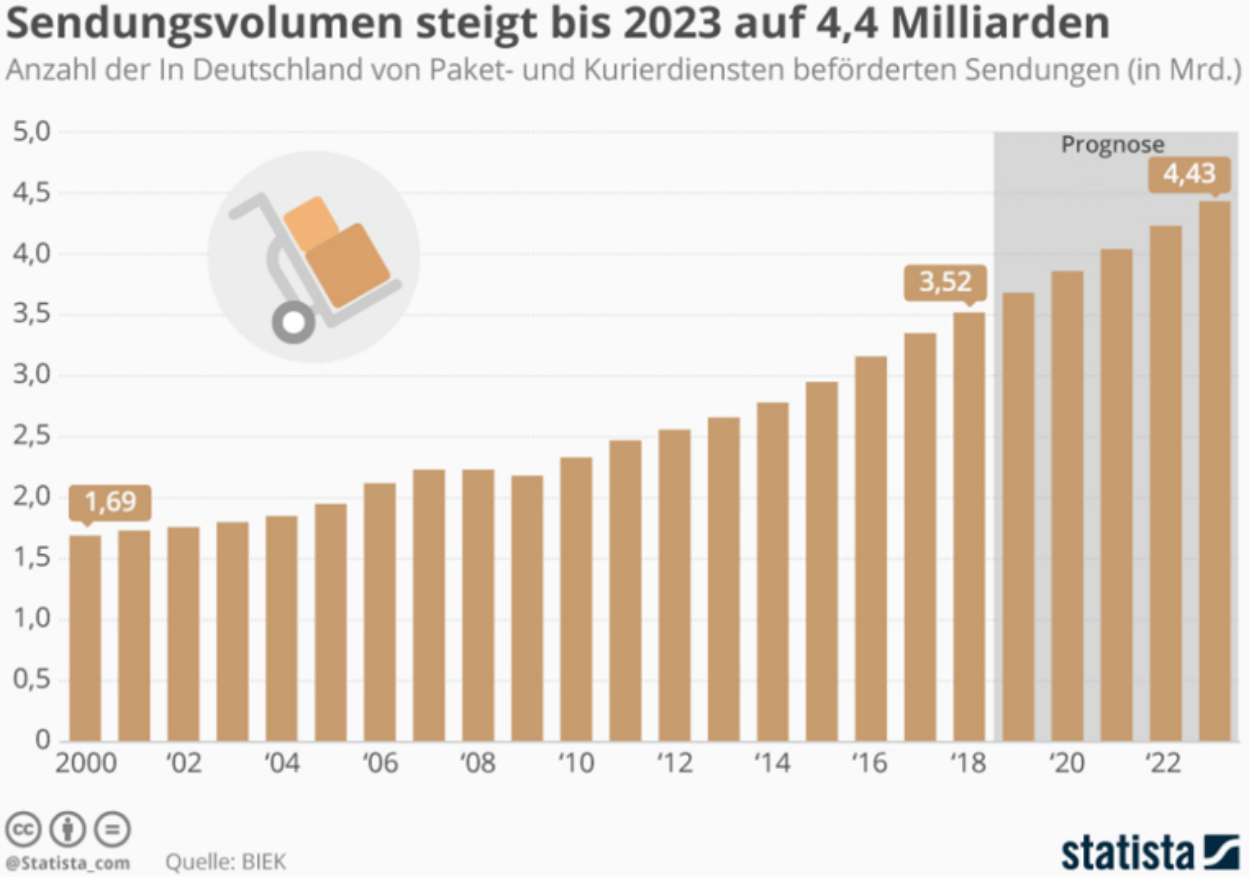
PROJECT DESCRIPTION

2.1 Use Case

Written by Luca Deck

2.1.1 Problem Setting

With online trade becoming more and more popular, the number of shipments (in Germany alone) rose to more than **3.5 billion** in 2018, expected to reach 4.4 billion in 2023. Handling this immense daily volume requires logistics and **distribution centers** to consistently increase efficiency while maintaining a sufficient level of quality. One crucial detrimental factor in this context are **damaged or unsealed packages**. Primarily, for **quality control** and customer satisfaction intensive and expensive **manual effort** is required in order to evaluate the intactness of outgoing shipments. Moreover, if not identified in time, they can cause interruptions and traffic jams within the material flow.



Source:[1]

2.1.2 Real-Life Value

We want to resolve this issue by implementing an **automated warning system** for logistics or distribution centers that informs the control center, whenever some quality specification of a package is violated. This can be set up and used in receiving areas, before shipping or basically everywhere within the supply chain. In this sense it might also be of interest for package delivery services like DHL, Hermes etc. to avoid handing out corrupted parcels. Our computer vision model firstly **detects packages** and secondly **localizes and classifies critical areas of a package**. The picture including the critical information is immediately transmitted to the control center to intervene and resolve the issue in time. This way, quality of packages within a flow system can be guaranteed and corrupted packages can be filtered out quickly while saving **manual effort**.

[1] https://de.statista.com/infografik/9992/in-deutschland-von-den-paket-und-kurierdiensten-befoerderten-sendungen/

For collecting training data and providing a proof of concept, we organized a cooperation with an industry partner who operates a distribution center and is confronted with all sorts of critical packages every day. Unfortunately, due to Corona restrictions, our visit to their logistics center had to be cancelled at short notice.

2.1.3 General Approach

Step 1: Identifying Packages

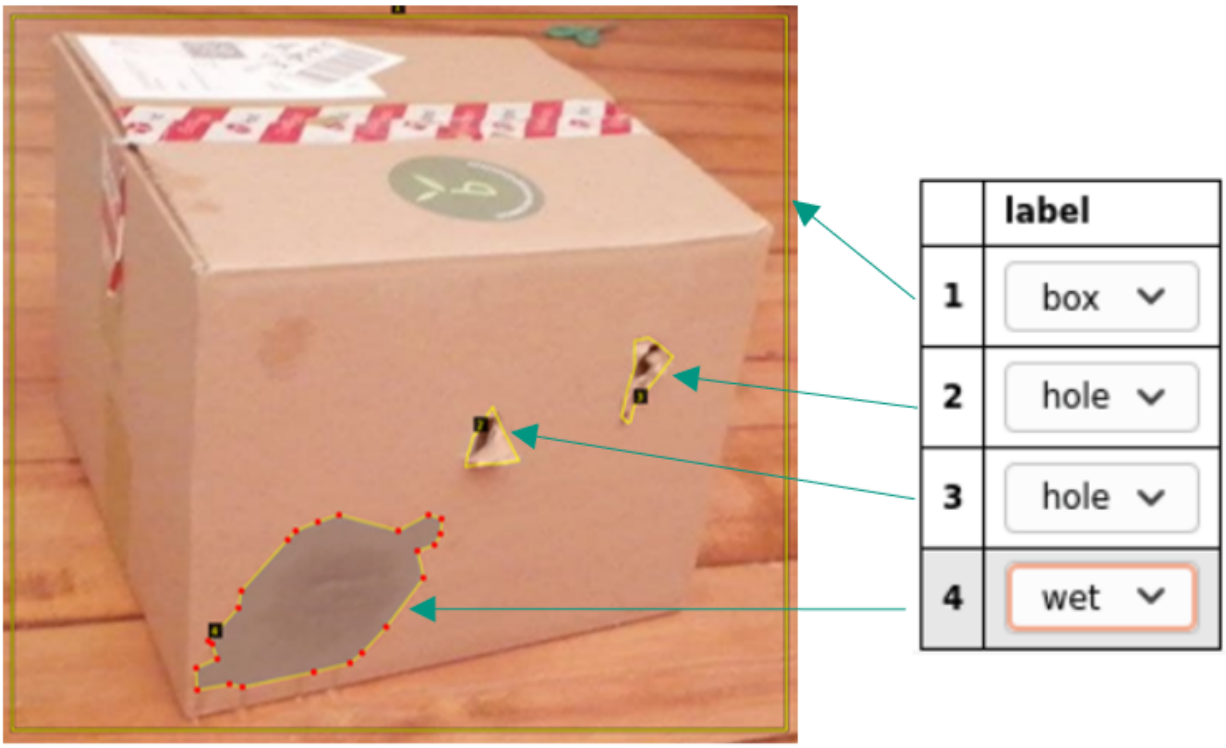
First of all, our object recognition model should identify cardboard packages within the entire image. This a quite simple task for modern object recognition applications and provides the basis for subsequent analyses.

Step 2: Localizing and Categorizing Critical Areas

Within the bounding box of identified packages, our object recognition model further aims to localize and categorize critical areas indicating violations of a quality specification. For our use case we decided to focus on four different critical properties:

- Holes
- Dents
- Wet areas
- Unsealed

We claim that these properties are potential threats within the material flow (e.g. content spill-out) and primarily responsible for superficial quality insufficiencies (e.g. customer complaint about damaged package). However, this list is not comprehensive and can be adjusted depending on the specific use case.



Step 3: Interpretation and Warning Message

Finally, based on this information, we want to come to reasonable conclusions for the control center. If some critical criterion is fulfilled (e.g. unsealed) or the extent or sum of damages is concerning (e.g. giant hole), the system sends out a warning message to the control center in real-time. This message includes the picture as well as the potentially critical areas. Within seconds, the control center can evaluate the image and initiate an intervention (e.g. stopping the conveyor or filtering out the package) to manually handle the problem. In our project, we simulated the message forwarding by transmitting the information to a mobile application.

Image Sources

2.2 Collaboration

Written by Luca Deck

This section introduces our group members and gives an overview about the tools we used for organizing our basic workflow.

2.2.1 Our Team

- **Alexey Rosenberg**
- Industrial Engineering (1st M.Sc.)

- **Luca Deck**
- Industrial Engineering (4th M.Sc.)

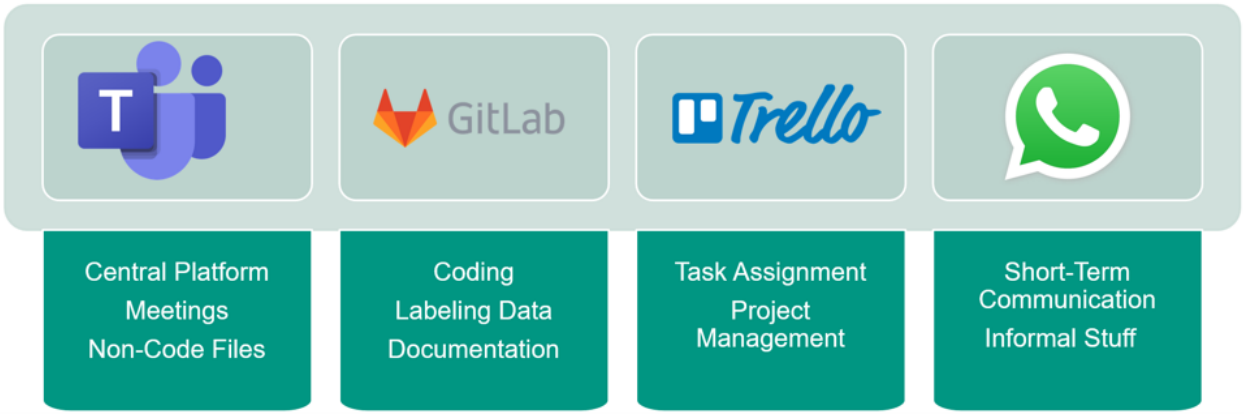
- **Bolatito Zäch**
- Industrial Engineering (3rd M.Sc.)

- **Joel Oswald**
- Information Systems (2nd M.Sc.)

- **Manuel Sauter**
- Information Systems (2nd M.Sc.)

- **Johannes Jestram**
- Information Systems (2nd M.Sc.)

2.2.2 Co-Working Environment



- **MS Teams** We used Microsoft Teams as our central platform for weekly meetings. This is also where we shared our minutes of meeting and further non-coding related files like presentations.
- **SCC GitLab** The university's GitLab instance was our central platform for all coding related topics. The key activities here were version control, code documentation, and a series of short Readmes and tutorials for internal communication.
- **Trello** We used Trello as a project management tool and for task assignment. In every meeting we created tasks to solve upcoming challenges, assigned group members and highlighted important dates. This not only helped structuring the meetings and individual workload, but also ensured us meeting all the deadlines.
- **WhatsApp** Lastly, we used WhatsApp as a messenger for short-term communication and coordination (e.g., postponing meetings or discussing recent developments).

2.3 Timeline

Written by Alexey Rosenberg

This section covers the progress of this project in chronological order. For an overview of the project, we will first present a compressed timeline. Following the compressed version, a monthly timeline with detailed descriptions of the team activities is presented.

2.3.1 Brief Timeline

- **End of April:** Creating a team and brainstorming of ideas for use cases.
- **Beginning of May:** Definition of use cases and start of training data collection.
- **End of May:** End of training data collection, beginning of model prototyping, system architecture design and data augmentation definition.
- **Beginning of June:** First successes in model prototyping with Tensorflow and YOLO, continued data augmentation, successful testing of SSD on Jetson Nano
- **End of June:** Implementation of data augmentation script and creation of an augmented training data set, set-up of Microsoft Azure for online training, decision to continue development of both SSD and YOLO models until the end of the project, successful test of YOLO on Jetson Nano
- **Beginning of July:** Deployment of both models on Jetson Nano, evaluation and fine-tuning of model, appointment with company for validation of use case (canceled on short notice), prototype, and collection of real-life testing data, additional training data collection with static background
- **End of July:** Final training of YOLO, SSD with MobileNet and SSD with ResNet with newly collected data, final documentation, presentation of results and end of project

2.3.2 Detailed Timeline

April

After team creation, the project began with brainstorming of possible use cases at the end of April. After a meeting with our supervisor the possible use cases were reworked, specified, and reconsidered.

The teams decided on weekly sprints as workmode and the infrastructure of an IT project is set-up, such as a GitLab repository, Microsoft Teams for meetings and Trello for project management. Additionally, we researched availability of image data for the different proposed use cases.

May

By the 5th of May we chose the recognition of damages on packages as the use case. The main arguments for the use case were the high practical relevance of the use case in logistics. As many, especially foreign, packages are often delivered with extensive damages, automated recognition of packages could reduce the number of personell necessary for quality management in package reception and shipping. Additionally, the possibility of cooperation with a company for real-life data and validation of the prototype at the company's premises were important arguments in the use-case decision.

A week later, on the 12th of May, we defined the common standards for training data collection up to this meeting and chose a tool for image annotation. Also, the to-be-detected types of damages were collaboratively defined, enabling the start of the training data collection phase.

For image annotation the VGG Image Annotator was chosen. At least 50 images were expected from each team member resulting in 300+ images for training the models. We describe the training data collection in detail in section *Training Data Collection*.

Further, the Erwin Müller Mail Order Solutions GmbH (EMMOS) was contacted by one of the group members to initiate talks for a possible cooperation, with the goal to collect training data in a "real-life" environment. This data was intended to be used to evaluate the model created on homemade training data and increase its applicability by using it in the final training of the model. We also hoped to gain valuable insights for our use case, for example, how often they receive complaints for damaged packages or how they usually handle damage detection.

We finished data collection by 19th of May. Initially, we wanted to save the training images on GitLab using LFS (Large File Storage), but this function is not supported by KIT's SCC. Therefore, we chose *BW Sync-and-Share* as an alternative for training data storage. Also, the training data collected by the different group members was checked for consistency and conformity to the predefined standards.

Next, we received the Jetson Nano hardware, set it up and tested it. To enable later deployment by all members of the group, a SSH access to the Jetson Nano was proposed. For documentation of the project a Sphinx Python Documentation Generator was chosen and added to the GitLab repository. Additionally, the model prototyping began with an overview of the different available models and a test of a pretrained model, as seen in the image below. Lastly, we discussed and finalized the intermediate presentation.

Fig. 2.1: Example of first test inference on the team mascot.

June

During the following two weeks, we focused on testing different libraries for data augmentation and assessing the use of different object detection frameworks, namely YOLO and the Tensorflow Object Detection API. Regarding data augmentation, we analyzed different image augmentation libraries, such as *imgaug*, and the preprocessing capabilities included in Darknet and Tensorflow. After analysis of the offered options and the augmentations necessary for our use case, we deemed the preprocessing capabilities included in Darknet and Tensorflow insufficient. Hence, we decided on using an additional library. We defined a list of augmentations for the training data and created a set of augmented data. One of the major obstacles in data augmentation was the augmentation of the bounding boxes along with the images. Details can be found in section *Data Augmentation*.

Further, both object detection frameworks (Tensorflow, Darknet), were successfully implemented, resulting in working prototypes for both models. At first, we researched the possible integration of YOLO in the Tensorflow object detection API, in order to enable quick switching and evaluation of different detection models. Unfortunately, we did not find a viable implementation of YOLO in Tensorflow. Therefore, the team decided on the parallel implementation of both frameworks, to later decide for one. Further, we set up Microsoft Azure to enable online training of models. We also decided to begin writing the documentation.

On 23 of June, the data augmentation had been finished and a set of augmented training data was created for training. We successfully trained the SSD model online using Azure. Additionally, the team prepared the appointment at the company for a field test of the Jetson with one of the models and interviews with employees of the company. Finally, we added an extensive number of negative examples to the training data set, to increase model performance.

July

A week later, the appointment with the company was cancelled due to COVID-19 restrictions. This was unfortunate, as we planned to validate the use-case and collect additional data for testing purposes. YOLO model was successfully trained with all training data locally. We first discussed an outline of the application logic beyond pure object detection required for the use case. In particular, we discussed a functionality to notify the user about detected damages.

On the 14 of July, the user notifications were implemented using the Pushover API. The user receives a notification and a labelled image, if a packages with damages is detected. Further, the weights of the YOLO model and config file were converted to TensorRT for deployment on the Jetson Nano. Due to the mediocre performance of all models, the team initiated a second round of training data collection with a static background. The static background was intended to increase model performance and provide a closer resemblance for the intended use case with a static device above or besides a conveyor belt.

On the 21 of July, the second round on of training data collection was finished, i.e., taking, labelling and augmenting all new images. We performed a final training of YOLO, SSD with MobileNet and SSD with ResNet on the new, augmented dataset. Also, the project documentation and final presentation was finalized, leading to the end of the project.

2.4 Training Data Collection

Written by Alexey Rosenberg

We performed two rounds of training data collection, the first one being in May and the second one in July, towards the end of the project. First, we give general information about the data collection. Then the classes of damages are described in greater detail and examples are given.

2.4.1 Overall Information on the First Round of Training Data Collection

The task of collecting training data was shared among all team members. Each team member was tasked with finding several cardboard boxes and taking at least 50 images in a homemade environment.

Fig. 2.2: Example training image.

The goal of collecting at least 300 training images was fulfilled and exceeded as most members provided more data than necessary. We deemed 300 unique images enough as this number can be further increased through data augmentation methods, as described in section *Data Augmentation*.

2.4.2 Classes of Damages

As the use case of the project is the identification of damages on cardboard boxes, 60% of the provided images contain damages on the depicted boxes and 40% depict boxes without damages. The frequency of each damage class is defined to be about 25% to avoid class imbalance in the training data. We chose four classes of damages:

- Holes: Packages with holes or cuts in the sides or lid of the box
- Open: Packages with an open or insufficiently closed lid
- Dents: Packages with small- or large-scale dents or broken edges
- Wet: Packages with obvious damage from exposure to water

The following figures are examples of training images with different damages.

We discussed several further classes, such as overfull packages or packages with damaged labels. These additional classes were discarded mainly for reasons of simplicity in the initial development process, as they can be added at a later stage of development.

Fig. 2.3: Box with a dent.

Fig. 2.4: Box with a hole and damage from water.

2.4.3 Standards and Tools for Training Data Collection

To increase consistency among the collected training data, we defined standards for collection prior to the start of the collection process. A mobile phone camera with an aspect ratio of 4x3 and a HD resolution of at least 1080p was sufficient for our use case. While most object detection frameworks work with a resolution of 300x300px, we chose a higher resolution to enable later down-scaling and cropping of images, without the loss of necessary resolution. Additionally, all team members named their images distinctly to enable unique identification of the training data.

2.4.4 Image Annotation

For image annotation we chose the VGG Image Annotator. The main arguments for the use of the tool were it being open-source and free of charge. Additionally, the tool and its functionality were presented in the lecture, hence all team members were familiar with its functionality.

For annotation we defined the boxes to be annotated using rectangles, while the different damages were annotated by polygons. The use of different annotation types was decided to balance workload for the annotation against future prediction precision. As damages are often small scale and difficult to distinguish from the boxes, the finer, polygon annotation method was initially chosen. Multiple *helper functions* are created to handle and convert label types.

Later in the project, we converted the polygons used in this first annotation to rectangles, as the used models are not able utilize polygon annotations.

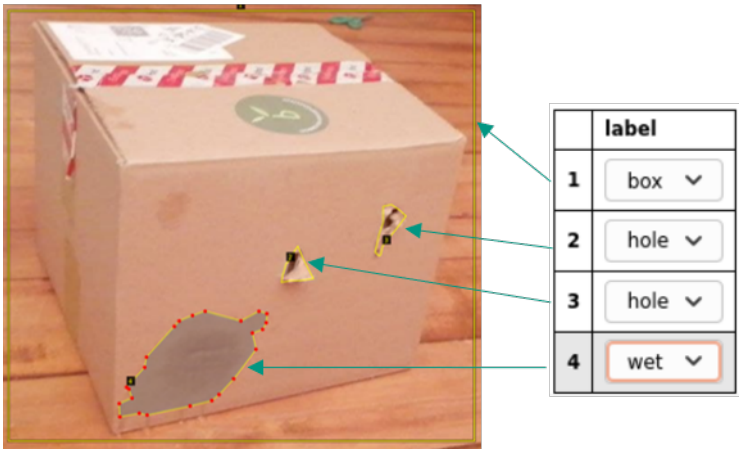


Fig. 2.5: Example of annotations of box and damages.

2.4.5 End of First Round of Training Data Collection

At the end of the data collection phase, we reviewed our dataset to avoid bias and check for consistency of labeling among all team members. Explicitly, we reviewed a correlation between the presence and the number of damages, to check if the presence of one type of damage implied the presence of another type of damages, either a random or a specific type. Such correlation was not found among the data after a review with random sampling of the training data.

We stored the training data on BW Sync-and-Share, as the GitLab instance of the SCC does not support the use of LFS (Large File Storage) at this point. The Pascal VOC exports of the labels reside were stored in the GitLab repository.

2.4.6 Additional Phase of Training Data Collection in July

Due to deficits in the training data collected in the first round and the mediocre performance of all models, we decided on collecting additional training data. This time, we used a static background, as this should increase model performance and provide a closer resemblance for the intended use case. To provide a more homogeneous set of images, we solely used the Jetson Nano camera. This way, we could reduce the differences between the images taken with our own cameras and the data the Jetson performs inference on in production. In this round, we collected *230* images with the Jetson Nano, thus resulting in a total of *720* unique, un-augmented images for augmentation and subsequent training.

Additionally, we added *842* images of negative examples, such as images of household floors, grass, and pathways, to the training set.

This led to a total of *5162* images (including all augmented images and negative examples) to be used for our final model trainings.

2.5 Data Augmentation

Written by Luca Deck

2.5.1 Scientific Inspiration

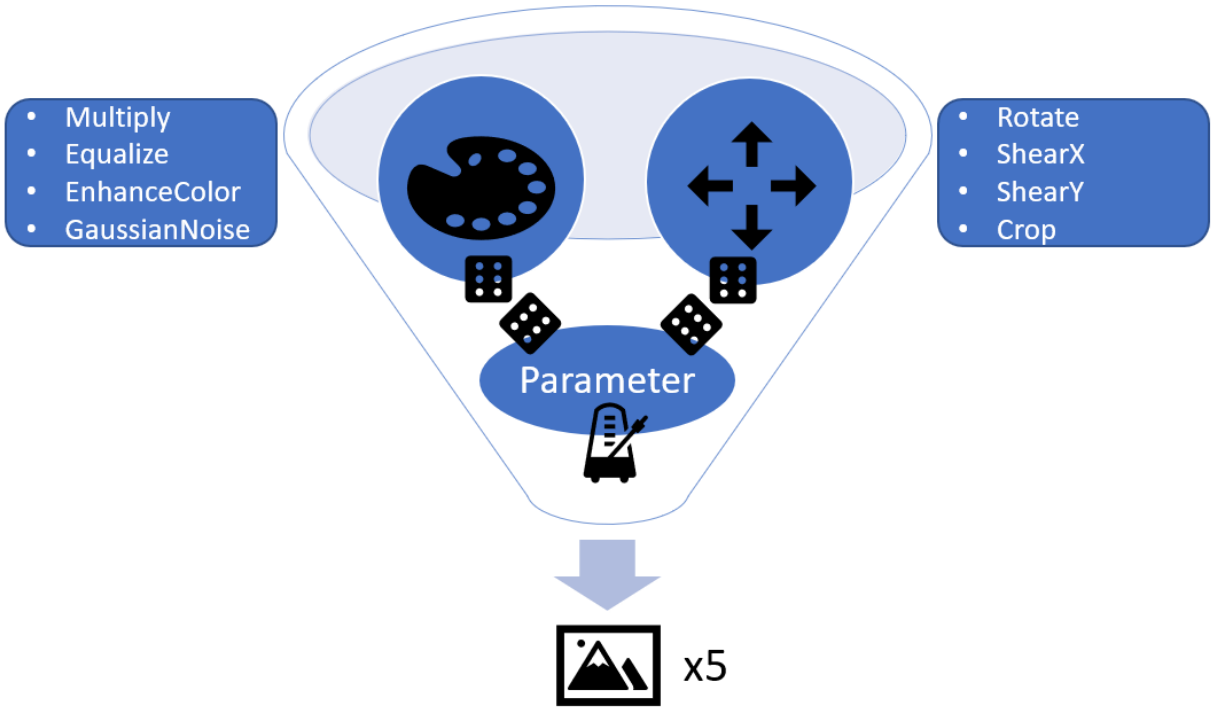
In order to increase performance and avoid overfitting, we increased data volume and diversity by means of data augmentation. For this, we developed a function to automatically apply a randomized combination of image augmentations. One important prerequisite for our library choice was the consideration of our manual labels, i.e. bounding boxes and polygons. Our augmentation "policy" is inspired by the findings from the paper "Learning Data Augmentation Strategies for Object Detection" which identified optimal data augmentation strategies through learning algorithms. Based on these findings, the popular library bbaug including 4 augmentation policies was published by Google's Brain Team. These policies each consist of a set of tuples representing a specific augmentation (e.g. Rotate and GaussianNoise). The actual augmentations are imported from the imgaug library.

2.5.2 Augmentation Script

Due to use case-specific considerations and compatibility issues with the bbaug library, we decided to come up with our own policy. Similarly to bbaug, we used imgaug. First of all, our code is very flexible regarding kinds of augmentation, number of augmentation effects per augmented image as well as number of augmented images per raw image. Thus, the trial and error for different parameter combinations can be easily implemented.

Our function accesses the directory of raw images and its labels, goes through a number of augmentation iterations and saves the augmented images as well as the updated list of labels into a folder. Firstly, these augmentation effects are chosen randomly from the entire pool of specified augmentations. Secondly, the parameters for the augmentation effects are also chosen randomly from a specified interval. This way, we ensure a broad and randomized variety of effects without inducing overfitting hazards and without effect overload. One augmentation iteration consists of two superficial "color augmentations" (Multiply, Equalize, EnhanceColor, AdditiveGaussianNoise) and two "geometrical" augmentations (Rotate, ShearX, ShearY, Crop). The selection of these effects and their parameters is the result of an extensive trial and error procedure across the imgaug library. For example, we removed a perspective transformation because it altered the bounding box "fit" too strongly and decided against. Also, we decided against a cutout feature which replaces pixel areas with random unicolor boxes because it was prone to entirely covering smaller damages.

The entire approach is depicted in the following framework:



2.5.3 Result

We went with 5-7 augmented images per raw image, so that we could increase our dataset from unique 756 images and *842* negative images to a total of *5162* images for training.

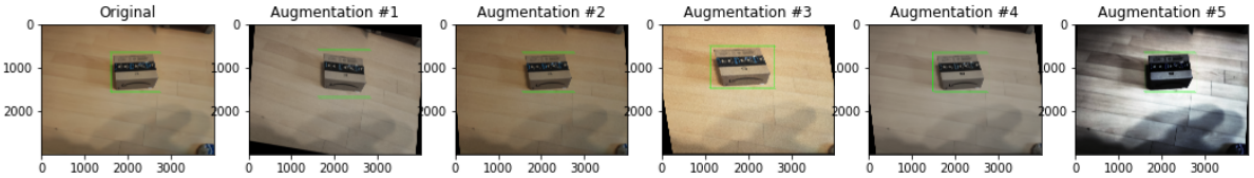


Fig. 2.6: Example augmented images.

2.5.4 Side Notes

Note that for simplicity reasons and time restrictions, we decided to transform the polygons into bounding boxes for the subsequent processing of the annotations. However, the implementation of polygons (and thus more precise labels) is theoretically possible with our augmentation script and only requires some adjustments. We also faced the challenge that some augmented labels are "out of range", e.g., when rotation leads to bounding box coordinates that are not within the actual image pixels. This issue is resolved in the training pipeline and not in the augmentation script.

2.6 Model Comparison and Choice for the Project

Written by Manuel Sauter

In this section we highlight the decision making process regarding object detection architectures. In general there are two state-of-the-art types of CNN object detection architectures, namely two-stage detectors and one-stage detectors, which we both introduce briefly.

2.6.1 Two-Stage Detectors

Two-stage object detection architectures are solving two separate tasks consecutively. The first task is to propose regions that possibly contain objects that are relevant for the later classification. The first versions of two-stage object detection architectures proposed regions through exhaustive search or sliding windows. In more recent architectures, methods like selective search or regional proposal networks became more relevant.

The second task is to classify the objects inside the regional proposals. Therefore, the proposals will get fed into a CNN, where features get extracted. Based on these features, the objects inside the regional proposals are classified. An early implementation of the two-stage approach is *R-CNN*. Based on that *Fast R-CNN*, *Faster R-CNN* and *Mask R-CNN* were developed and have greatly improved the performance.

2.6.2 One-Stage Detectors

Object detection algorithms from this type of architecture are different from the two-stage approaches. Instead of using regions to localize the object within the image first and using a separate algorithm for this task, one-stage algorithms are only using a single convolutional network. While training on the full image this network can predict bounding boxes and the class probabilities simultaneously.

A famous one-stage detection algorithms is YOLO (*You Only Look Once*). YOLO splits the image into a grid of size SxS. Inside every cell of the grid YOLO predicts bounding boxes and a box confidence score for each bounding box. Every cell predicts a probability for every different class and is responsible for the one with the highest probability (in the first YOLO implementation). More recent YOLO version not only greatly improved the performance and the error rate, but also accounted for different other weaknesses of earlier versions. In YOLOv2 the detection of several objects per grid cell was introduced through the usage of anchor boxes. Anchor boxes get defined by exploring the training data. Anchor boxes with different height/width ratios representing the different classes are used. YOLOv3 introduced multi-label classification for objects that belong to different classes at the same time. YOLOv4 is the most recent version and reaches state-of-the-art performance in object detection tasks.

Another famous and state-of-the-art group of one-stage object detection architectures besides YOLO are the *Single Shot MultiBox Detectors* (SSDs). One of the main differences compared to YOLO is the possibility for predictions of detections at multiple scales. This is achieved by a different architecture that contains series of convolution filters, which decrease in size progressively. This addresses the reduced performance of YOLO when working with relatively small objects. Further, SSD contains a base network to extract feature maps. In this project MobileNet and RestNet are used for feature extraction. This base network can be easily switched and many base networks are available.

2.6.3 Comparison and Project Choice

Having the two different architectures at hand, the question arises which one to use for our use case. In general two-stage detectors seem to reach a high accuracy and precision and have been observed to be slower in comparison to the one-stage detectors. Even though many inference and training improvements were made, this still holds true. One of the limitations in the project is the Jetson Nano. Looking at his computing power it becomes clear that a one-step approach is more likely to yield the desired results. Also, frames per second (FPS) was an important factor for in choosing our model, which was also an argument for a one-stage detector.

Therefore, we focus on using one-stage detection architectures. As no team experience with the use of the different one-stage object detection architectures on the Jetson Nano hardware was available, we decided to not only focus on YOLO but also try out a SSD architecture with MobileNet and ResNet networks.

2.7 Outlook & Possible Enhancements

Written by Luca Deck

2.7.1 Real-Life Insights and Fine-Tuning for Productive Environment

We would have hoped to base our interpretations on real-life experiences from our industry partner. By tapping into the customer sphere and collaboratively designing our service, we would have been able to make at least two central adjustments. Of course, our model is open for such adjustments. Unfortunately, we missed out on the valuable information due to Corona restrictions. These two aspects also provide a crucial basis for future concept or model drift considerations.

Relevant Critical Properties

Our list of critical properties (hole, dent, wet, open) is based on initial considerations. Talking with employees in the return order department of our industry partner would have revealed authentic insights into the real-life situation. In a short preliminary talk with the supervisor we learned that there are all sorts of damages and disfigurements one can imagine. For example, many packages are scribbled on heavily or strappings are damaged. By updating and prioritizing the list of critical properties, one could adapt the training data set and prioritize it according to the respective damage frequency.

Interpretation of Critical Properties

Exchange with domain experts would have also enabled a more sophisticated interpretation logic. Information on which kind or which extent of damage usually leads to manual interventions could be implemented into the interpretation of the model output. For example, we imagine that a package with a giant hole usually should be filtered out while some minor dents are unproblematic. Concluding, this would reduce the number of warning messages to a relevant minimum and thus increase the operational efficiency of our overall system.

2.7.2 Two-Stage Model Architecture

One of our initial ideas was to implement an additional damage detection model that is applied to the identified bounding boxes to factor out the damage detection as a sequential step. In other words, as soon as a package would be identified by our "base model", the focal pixel coordinates would be rescaled and applied to the original image out so that the more complex damage analysis would be based only on the package itself. This way, for inference the model would be able to feed in the maximum of relevant pixels , hopefully increasing precision. Also, this would implicitly encode that damages can only occur on packages. While we still perceive this as a promising approach, we had to leave it unattended due to its architectural complexity and expenditure of time. Also, the effects might be weakend by the low Jetson Nano resolution in the first place.

2.7.3 Implementation of Polygon Labels & Instance Segmentation

Our model was initially planned to read in and process polygon annotations for the damages. However, due to time and computing power restrictions, we decided to go for the simple way and transformed them into bounding boxes. We assume that accounting for the more precise polygons would significantly increase mAP for the damages. Also, it would be valuable for instance segmentation which is a possible future enhancement but out of scope for this project. At the same time it poses way more labeling effort and consumes more of the GPU's computing power. From a coding perspective, the data augmentation and the training pipeline would only need some minor adaptations. However, we would need to switch to Poly-YOLO or some segmentation model.

2.7.4 Deployment: Integration into a Warehouse Management System (WMS)

Finally, with regards to deployment, our service is perfectly suited to be integrated within a WMS and thus be embedded within a service system. By sharing a simple interface, the model output and the warning message could be transmitted to a control center functionality of a WMS. The control center receives the labeled image as well as an evaluation of the severeness which add up to a recommended action. The control center then decides whether the package has to be filtered out or even if the conveyor has to be stopped. Moreover, for an even more complex application, our model might be integrated into the warehouse data stream. In combination with an object tracking functionality, our system would know at any time which package is being scanned for damages. Thus, the warning message could additionally contain information about the package (ID, origin, destination, client etc.) to facilitate decisions and enable data-driven subsequent analyses.

2.8 Learnings

Written by Luca Deck and Manuel Sauter

While working on this project, we faced several challenges and struggled with a lot of unforeseen issues. Additionally, in hindsight we encountered a multitude of things we would approach differently in future endeavors regarding data science in general or specifically for computer vision. Apart from our handed-in repository, we see these findings and learnings as the major takeaway of this course and want to summarize them shortly.

2.8.1 Training Data Collection

With little to no experience in computer vision projects, especially in the early stages of training data collection we had to make some decisions that were fundamental for the further development of the project. Some of these turned out to be suboptimal in hindsight. With more experience, some deeper preliminary considerations would have been helpful. However, we of course acknowledge that some of these issues might never be resolved perfectly due to the inherently iterative nature of data science projects.

- **Define use case narrowly precisely** The more specific the problem at hand the lower the complexity and effort. By narrowing down the use case upfront e.g. only for a specific kind of package (brown with white label) and only one or two damages, we could have concentrated our efforts more effectively and thus possibly reach a higher precision.
- **Take the productive setting into consideration** By strictly defining external factors for the deployment stage we could have created a more realistic or authentic training dataset. Determining image quality, angle, background etc. promises a more effective data collection process as well as higher precision for the productive setting. In our case it would have been advisable to collect the dataset on-site instead of our rooms.
- **Label once to save effort** Knowing the exact demand of training data helps to perform this step in one go providing more efficiency. Instead, our training data collection was more of an iterative process based on noticed shortcomings. We underestimated the huge amount of data for our model to perform sufficiently.
- **Hinge the decision on number and type of classes on the amount of training data** The effort for training data collection increases in the number of label classes. Agreeing on less label classes thus can reduce the effort significantly.
- **Include "negative" examples in the training set** Depending on the architecture (especially YOLO), adding negative instances increases model performance significantly. These should show the surroundings without the focal object, for example in our case our room, the floor and the background without the package. Apparently, this helps the model to adapt to the environment and to be more confident in the detections.
- **Set up strict guidelines for labeling and communicate them appropriately** A consistent dataset requires all people working at the labeling tasks to agree on guidelines (regarding label type, narrowness, annotation format etc.). These have to be communicated explicitly in a manner that no misunderstandings can happen whatsoever. Even minor misunderstandings can be very costly if not noticed in time.
- **Choose the model framework upfront to know the data requirements** Knowing the model framework in advance helps to define the data requirements. For example, this implies certain annotation formats or that polygons are not compatible with the model anyway. Similarly, early downscaling saves a lot of computing effort and time if you know that the model can only handle a maximum of pixels.

2.8.2 Choice of Models and Training

Written by Johannes Jestram

- **Do not only consider performance benchmarks** When having different models at hand, the choice for a model should also depend on softer factors such as hardware support, effort to train, effort to deploy, and in our case FPS in the video inference. During the project, we found that Darknet and Tensorflow differ strongly in those points.
- **Build the end-to-end pipeline early** When trying out a model, before focusing for many hours on preparing data and training, make sure that you can already deploy it for inference. Further, make sure to get to the point where you successfully start the training once, even with dummy data, on the target system where you want to train. We found that out the hard way when trying to train YOLO on Azure, which required a lot of effort in the last weeks of the project. If we had considered this earlier, we could have saved time and nerves.

2.8.3 Co-Working & Communication

Written by Luca Deck and Johannes Jestram

While we were a completely motivated team and enjoyed working together, we stumbled across some hurdles in our workflow.

- **Read instructions, guidelines and Readmes thoroughly** Although we documented most of our internal work and processes carefully and generally maintained a healthy communication, one is always prone to overlooking essential notes. This refers, for example, to the labeling guidelines. This also makes the entire work reproducible and comprehensible for external people.
- **Git** To avoid tedious merge conflicts, it is essential to always pull all recent changes before working on anything and to always push afterwards. Git is a powerful tool for workflow coordination, collaboration and versioning but requires a certain level of experience and knowhow.
- **Check each other's code** In order to prevent bad coding habits, missing tests for important code, and sloppy docstrings, check each other's code. Either via automatic pipelines, or by hand. We did this and thereby were able to greatly improve the overall code quality.

CHAPTER

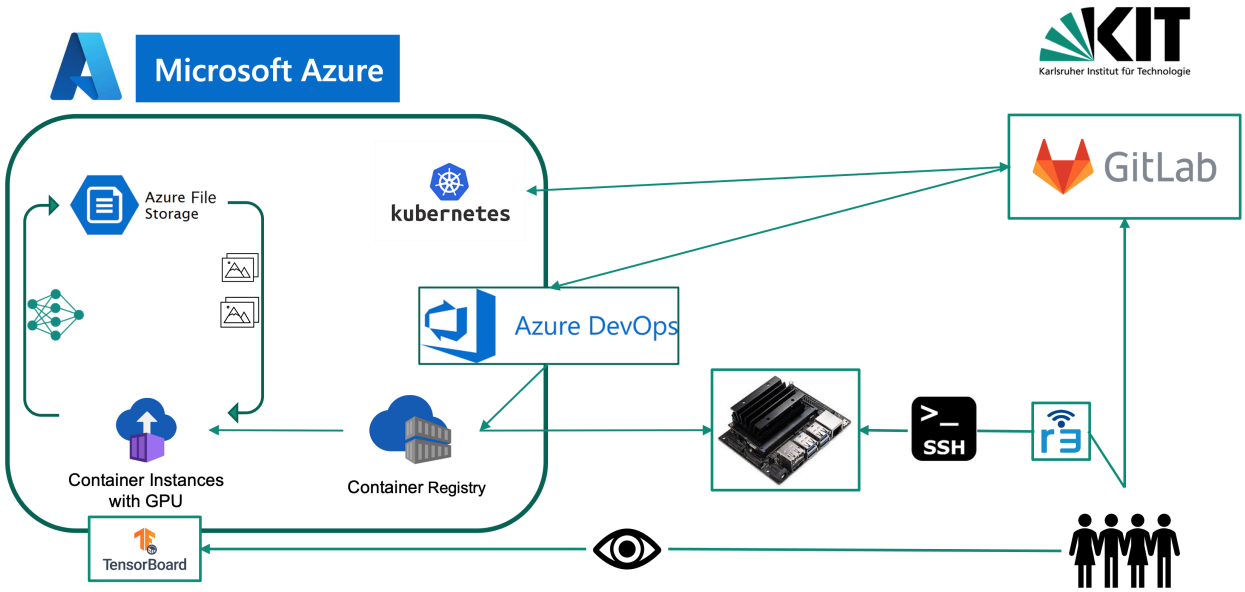
THREE

SYSTEM DESCRIPTION

3.1 Architecture Overview

Created by Joel Oswald

In order to build a robust system, we used state-of-the-art DevOps to build a scalable system that can easily be adapted to changing demands. This section covers the technology infrastructure we used throughout the project. We first describe our CI/CD workflow, followed by the setup of the Jetson and the cloud-based services.



3.1.1 CI/CD Pipelines

Written by Joel Oswald

To support our developers, we built multiple CI pipelines that automatically take care of important checks and prepare the cloud training environment.

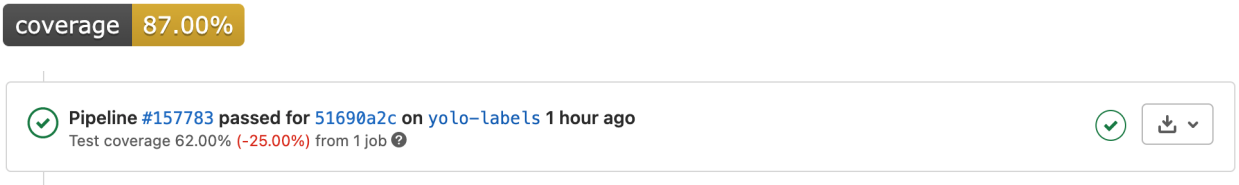
GitLab Pipelines

GitLab offers many possibilities to easily integrate basic CI pipelines with little effort. In order to run those pipelines we setup a Kubernetes cluster in Azure that hosts the GitLab workers (*Runners* in the GitLabs namespace). The pipelines can be conveniently specified in a yaml file.

Code Quality

The "Code Quality Pipeline" checks the readability and best practices (based on PEP) for each merge request and informs the developer whether the code quality degraded on some points, e.g., by too many nested loops or function arguments. By means of this pipeline each developer gains immediate insight into the overall quality of the submitted code.

Coverage



The "Coverage Pipeline" performs our *unit tests* and calculates the percentage of code that is covered by the tests. For each merge request it informs about the change in coverage, so that tests can be added if the coverage decreases.

Overall, those pipelines help the developer as well as the asignee of the merge request decide on the quality of the submitted code. Hence, they pose a valuable support in maintaining such a large code repository.

Azure Pipeline

In addition to GitLab's own pipelines we utilized Azure Pipelines, which allow executing the tests script. If the test succeed, the Azure pipeline rebuilds the Docker images and pushes them to a container registry. These Docker images are then used for training the models with sophisticated GPUs. Azure offers accounts for students allowing usage free of charge.

The GitLab repository is not public. Therefore, in order to allow the Azure pipelines to access the repository without the need to hard-core login credentials, we created an API access token for GitLab, which can be revoked at any time. This token is securely stored in a vault as a key-value pair and can be accessed by the Azure pipelines. Each pipeline is automatically triggered on new pushes to the master branch of the code repository.

3.1.2 Jetson

Written by Joel Oswald

[2]

In this section we describe how the Jetson Nano is used, accessed and set up for optimal performance during the project. The NVIDIA Jetson Nano[1] is a powerful embedded device for running neural network inference. Since it runs an Ubuntu-derivative on the arm64 architecture, it is able to execute all cloud-trained models directly on the jetson itself.

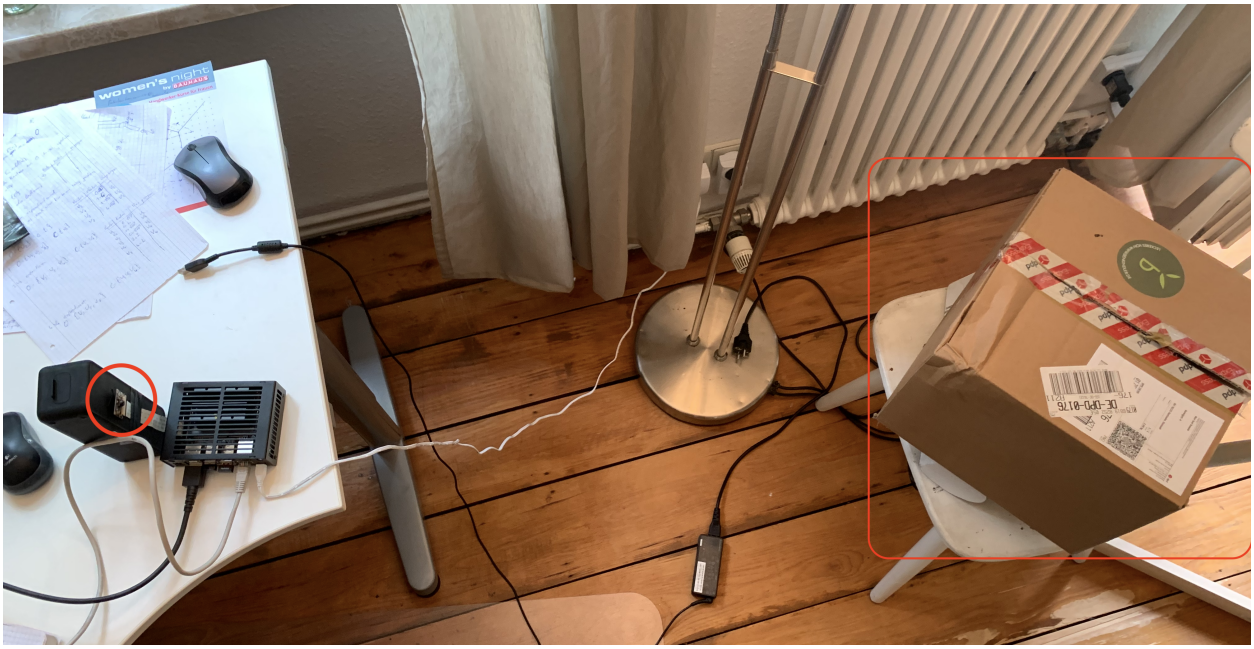
However, in order to completely leverage the Jetson's capabilities it is necessary to adjust and convert the trained models into an optimized format and execute that on the Jetson instead of the model's original framework. In this regard, the highest performance can be expected when using NVIDIA's proprietary *TensorRT* library.

The Hardware

We got the Jetson shipped with all necessary hardware components (i.e., case, charger) and an additional camera module that can be attached to the board. During the project, the Jetson was placed with the camera facing packages, so that every developer had the chance to test models and program logic whenever necessary.

[2] https://static.generation-robots.com/15689-product_cover/nvidia-nano-development-kit.jpg

[1] For brevity, we will omit the *Nano* when referring to the Jetson Nano.



Access for the Team

In order to allow every team member to access the Jetson remotely, we set it up as an IoT-Device at remote.it. With this method, each team member was able to access the device via a secure ssh connection. We used X11-forwarding to pass windows containing live-inference results to the local machine of the developer.

Software Deployment

Since regular updates at the edge-systems are expected, we required a solution that allows updating code without any end user interaction. A *systemd* service on the Jetson can continuously check for new docker images in our Azure container registry and pulls as well as starts the image when a new image is found.

Model Deployment

[34]

As mentioned above, some work is necessary to optimize the machine learning models for the Jetson. We decided to use NVIDIA TensorRT, because we expected highest performance and good usability from it. We strongly relied on the code of jkjungs Github Repo that already provides much of the necessary code.

We used TensorRT for the YOLO models (see *YOLOv4*) first. In order to convert the Darknet models to TensorRT, we first converted them to the *Open Neural Network Exchange* (ONNX) format. In a second step, we used the ONNX parser of the TensorRT library to convert the model into the TensorRT format. Using this procedure we are able to run a YOLO model at 27 fps on the Jetson. Without the optimization, i.e., using the Darknet YOLO model, the same model runs with 5 fps.

[3] https://developer-blogs.nvidia.com/wp-content/uploads/2018/11/NV_TensorRT_Visual_2C_RGB-625x625-1.png
[4] https://tse3.mm.bing.net/th?id=OIP.7q4Fbq0xfq7TuPy-NZERbQHaB5&pid=Api

For the deployment of the Tensorflow model there were different options. We used the TensorRT binding in Tenosrflow (*TF-TRT*) to optimize subgraphs of the model. With this method, the actual inference still uses the Tensorflow framework with GPU support.

The main inference file is the control_loop, which continuously grabs images from the camera and runs the detection model on top of it. The control loop is model-agnostic, so there is no need to change any code to switch between the YOLO-TensorRT model and the Tensorflow detection models. The model that is to be used can be passed as a command line argument. This is possible due to our polymorphic approach, which implements a parent model-class from which all our models must inherit.

As a prototype we have implemented a simple user warning message via an app-notification. We used Pushbullet, because it has free applications for iOS and android. Once a damage is noticed, we send the current image with a warning message via Pushbullet's API. Afterwards, a cool-down phase starts, so that the user will not get multiple warnings for the same damage.

Image Sources

3.1.3 Azure

Written by Joel Oswald


[1]

Microsoft Azure is a public cloud provider and together with AWS, GCP, Alibaba and IBM among the largest of its kind.[2] It offers a variety of services for IaaS, PaaS, SaaS, and MLaaS. We used multiple Azure services for different parts of our project. An overview of the services and how they are connected can be seen in the *overview*. Like most big players in the public cloud business, Azure provides free credit for students without the need to provide a credit card or other payment options. Being six students gave us enough credit to execute all required tasks (Kubernetes cluster, pipelines, docker instances, etc.) on Azure. In this Section we explain some services and how they are used that are not covered by the *CI/CD Pipelines*. The most essential part is the possibility to train in the cloud. To be as platform independent as possible we did not use cloud-specific services like Azure Machine Learning, which would make switching to a different cloud provider harder. Instead we just used IaaS and PaaS resources, such as container instances, that could be easily transferred to another provider.

Container Registry

We have our own container registry in place, which hosts all docker images that are used for the training of our networks. This allows autonomous updating of the images when a new version of the corresponding dockerfile is pushed to our repository.

[1] https://swimburger.net/media/0zcpmk1b/azure.jpg
[2] https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/

File Share

With the amount of data and models we trained, it was necessary to have a central place where all training data are stored. In addition, we wanted a central place for all trained models, so that they could be easily compared with a tool like Tensorboard. Using *Azure File Shares* solved this for us in a cost-efficient way. Using access tokens there was no need to share private login credentials. Furthermore the file share can be mounted as a volume to Docker container running in a Container instance, so that the models can be saved right there without any manual copy and move commands. Lastly, the file share is easily accessible by GUI and hence allows drag-and-drop uploading, either via its web interface, or via the Azure Storage Explorer.

Container Instances

Container instances are the central part for our cloud training pipeline. A container instance is a running container with a certain amount of resources that it can access. While it is possible to create such a resource via the Azure GUI, we aimed for scalability. So we defined a template via a yaml-file which allowed the repeatable and automatic deployment of container instances with our own images from the container registry. Besides that, we specified our file share to be mounted as a volume and the amount of resources that are available. If needed, it now takes only seconds to adjust the necessary amount of memory or GPU power. Deployment can either be triggered locally via the Azure CLI or web-based using the cloud shell. As the main process we are usually running a permanent service in the container, such as Tensorboard. The yaml-file also specifies exposed ports, so that the Tensorboard could always be accessed by the entire team via a web browser.

We then connected to the container to start the chosen training, where all necessary configuration files reside in the file share, thus persisting the state over the lifetime of a single container. In the same way, all checkpoints during training are written to that file share so that it is possible to continue an aborting training from another person with a new container instance.

This allowed training multiple models in parallel from all members of the team, even if they had little experience in setting up a training environment.

Kubernetes

Out of curiosity we also setup a Kubernetes cluster (*Azure Kubernetes Service*) and installed the GitLab runners to execute our GitLab specific pipelines, which worked well throughout the whole project.

Image Sources

3.2 Object Detection

Written by Johannes Jestram

This section covers the object detection frameworks and models that we use, namely an SDD with a MobileNet backbone implemented in the Tensorflow Object Detection API, and YOLOv4 from the Darknet library.

3.2.1 YOLO and Darknet

Written by Johannes Jestram

As YOLOv4 reaches state-of-the-art performance in object detection tasks, we decided on trying it out for our use case. The original YOLO uses its own library, Darknet. Implementations in other frameworks, such as Tensorflow and TensorRT (for inference only), exist. Due to the constrained hardware of the Jetson Nano, we chose to use the optimized TensorRT implementation from jkjung for inference on the Jetson. However, this version is only for inference, so in addition we set up a Docker container with the Darknet-YOLOv4 for training[1].

Installation

Due to its C heritage, Darknet is not as convenient to install as other frameworks such as PyTorch and Tensorflow. Having installed the requirements, the workflow on Ubuntu 18.04 with a NVIDIA GPU, cuDNN 8.1, and CUDA 11.3 is the following:

```
$git clone https://github.com/AlexeyAB/darknet.git
$cd darknet
$sed -i "s|GPU=0|GPU=1|g" Makefile
$sed -i "s|CUDNN=0|CUDNN=1|g" Makefile
```

If the GPU supports half-precision floats, also

```
$sed -i "s|CUDNN_HALF=0|CUDNN_HALF=1|g" Makefile
```

In order to do video inference and MJPEG streaming of the training progress (accessible via browser)

```
$sed -i "s|OPENCV=0|OPENCV=1|g" Makefile
```

Finally, compile Darknet with

```
$make
```

Training Configuration

In order to train YOLO for a custom object detection task, several configuration files are required. Setting up those files requires changes depending on the number of classes and the setup of the training-machine. The configuration files consist of:

- `model.cfg` stores the model architecture, which consists of the specification of the layers, input-image size, and hyper-parameters for training. In order to work for a custom detection task, the number of filters in several convolution layers and the number of classes the yolo-layers must be changed.
- `train.data` contains the paths to `class.names`, `train.txt`, `validation.txt`, the number of classes, and the path for storing intermediate weights during training.
  - `class.names` stores all class names for the object detection task.
  - `train.txt` contains the path to each training image.
  - `validation.txt` contains the path to all validation images. According to the author, you can place all training images here as well without negatively affecting the training.

[1] For improved readability, we will refer to YOLOv4 as YOLO from here on.

- `yolov4.conv.137` and `yolov4-tiny.conv.29` contain pre-trained weights, so that we did not have to start training from scratch. This means, we only needed data to fine-tune the model, and not train it from scratch. The models are pre-trained on the MS COCO dataset.

Updating Labels

YOLO uses a custom format for its image annotations. Therefore, we created a script `utils.pascal_voc_to_yolo` that converts Pascal VOC labels to YOLO labels. Further, because the paths to the images in `train.txt` change every time the images are moved, we added a function to the script that can change the paths given in `train.txt`.

Local Training

With the prepared configuration files and updated labels, the command for training YOLO is

```
$cd darknet
$./darknet detector train path/to/train.data path/to/model.cfg path/to/yolov4.conv.137 -
↪map
```

The flag `-map` is used for plotting the mean average precision (mAP). Because the training command includes the absolute paths to the required files, and `train.data` contains the absolute paths to all other files, it seems natural to place them all in the same folder outside of Darknet. This also helps with mounting the complete training configuration as a volume to the Docker container.

The following image shows the output during training. The chart opens as a separate window and continuously shows the training progress.

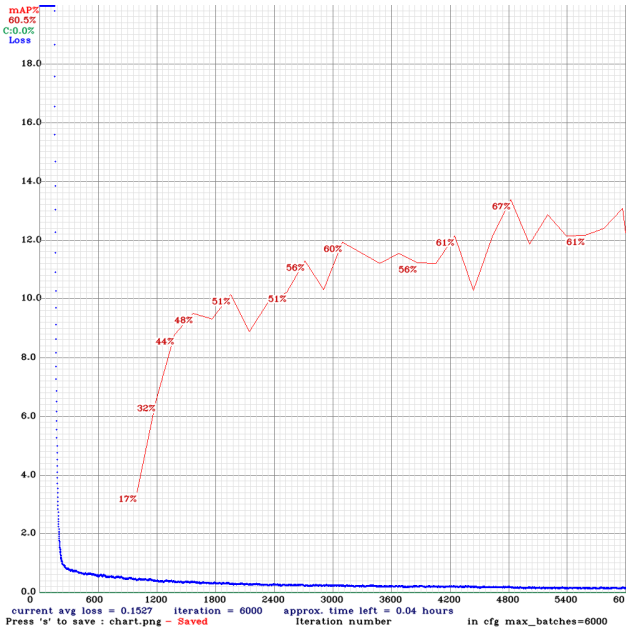


Fig. 3.1: The red line indicates the mAP. It requires the `-map` flag to be set, as well as validation data to be provided.

For the training we used the following hyperparameters:

- Default learning rate = 0.00261
- Default momentum = 0.9

- Default weight decay = 0.0005
- Random=1 changes the image resolution randomly during training in order to improve model quality
- Subdivisions = 64 is higher than the default and therefore creates more, smaller batches. This is mainly determined by the GPU performance.
- There are other hyperparameters that we did not change.

Training with Docker

The non-trivial installation of Darknet was an issue in the beginning, because not everyone could run and train YOLO. Therefore, for increased portability, we set up Docker containers for both local training and training on Azure (see *Container Instances*). Using the container for local training only requires a NVIDIA GPU and corresponding driver, as well as nvidia-docker installed. The training data and configuration of YOLO is loaded via a Docker volume. With this setup, the user can start the container on the local computer with

```
nvidia-docker run -it -p 8888:8888 -v ~/yolo-data:/training-data 74797469/yolo-amd64-
↪gpu:no-entrypoint
```

The configuration for training as well as the training data must reside in the mounted folder, in this case `~/yolo-data`. In `train.txt`, the image paths must be set to the corresponding path within the mounted volume, e.g., `/training-data/dataset_final_04_yolo/train`, and the paths for the configuration files in `train.data` must point to the correct path in the volume as well. The training can now be started with

```
/darknet/darknet detector train /training/aisscv-yolo-4class.data /training/aisscv-
↪yolov4-tiny-4class.cfg /training/yolov4-tiny.conv.29 -map -dont_show -mjpeg_port 8888
```

With the flag `-mjpeg_port 8888`, Darknet streams the training graph to a given port that can be accessed via web browser, in this case via `localhost:8888`. The flag `-dont_show` suppresses the creation of a dedicated window containing the training progress, which is not wanted when using Docker.

Training in Azure

Based on the Azure configuration described in *Container Instances*, training can also be executed in the cloud. After loading the `azure_yolo_train.yaml` to the user's Azure storage, the container instance can be started with

```
az container create -g aiss-cv -f azure_yolo_train.yaml
```

The version of the Docker container that we use here has an entrypoint and immediately starts training.

Used Models and Performance

YOLOv4

We started training with YOLOv4, because it was the configuration used in the tutorials. Further, we started with only detecting one class - the boxes - for which we have enough training data to train such a large neural network.

Performance with detecting the boxes was good, but not perfect, with a mAP of *84%*. Deployed on the Jetson with TensorRT, this model reaches 5 frames per second. However, detecting only one class is not sufficient for detecting damaged packages. In order to increase the number of frames per second and be more efficient with the training data, we switched to YOLOv4-Tiny.

YOLOv4-Tiny

YOLOv4-Tiny is a smaller version of YOLO, with two yolo layers, 29 convolutional layers, and fewer anchor boxes, leading to a decrease in parameters compared to YOLO. Therefore, YOLOv4-Tiny seems a good fit for the restricted hardware of the Jetson Nano, which is underlined by the five-fold increase in frames per second on live inference (see *Model Deployment*) when switching from YOLO to YOLOv4-Tiny.

We first trained the smaller model for all five classes with the dataset described in *Data Augmentation*. The training did not converge and the overall results for detection were unsatisfying. Further research suggested that the roughly 2600 training images are too few for training a detector for five classes.

In order to overcome those issues, we took two measures. First, we reduced the number of classes to two, only keeping *box* and *open*. Second, we extended the dataset with negative training images, i.e., pictures of surfaces without boxes standing on them. Those measures led to the training converging. Detection performance was more satisfying, but sometimes struggled with to detection, especially when multiple boxes are in the frame and at least one is open.

After collecting more training data (see *Additional Phase of Training Data Collection in July*), we trained the final model on four classes, reaching a mAP of ‘54%`. In practice, this model performs satisfactory for all four classes.

3.2.2 Tensorflow Object Detection API

Written by Bolatito Zäch

In this chapter, the steps necessary to create a model with the TensorFlow 2 Object Detection API are explained. For the next steps to work as intended, make sure that the base folder of this repository is your current working directory. To download the Tensorflow Object Detection API, use the command

```
$git submodule update --init --recursive
```

The API can now be found in the folder `./tf-models`.

The first step is to create the container image from the provided dockerfile. This is done by running

```
$docker build -f Dockerfile.tftrain -t tg-train-image
```

After the successful creation of the container, it needs to be started.

```
$docker run -it -v $(pwd):/home/app -p 6010:6010 --name tf-training tf-train-image
```

This container runs as CPU-only. To enable GPU-support, follow the instructions and modify the commands accordingly. For Windows users `$(pwd)` needs to be changed to `${pwd}`. If the path contains any whitespaces, the respective sections need to be enclosed in quotation marks.

Preperation

Before the training can start Tensorflow requires a label map. This is a file that maps each of the used labels to an integer value. The file should have the extension `.pbtxt`. For our example with a total of five classes the label map looks like the following:

```
item {
    id: 1
    name: 'box'
}

item {
```

(continues on next page)

(continued from previous page)

```
    id: 2
    name: 'wet'
}

item {
    id: 3
    name: 'dent'
}

item {
    id: 4
    name: 'hole'
}

item {
    id: 5
    name: 'open'
}
```

The label map should be saved in the annotations folder with the annotations of the images.

The last preparation step is to convert the annotations and images into the TFRecord format. If a training/test split is used, a TFRecord file needs to be created for each. Run the following command to create the file:

```
$python aisscv/utils/pascal_voc_to_tfrecord.py -x data/annotations/ -i data/images -l␣
↪data/annotations/label_map.pbtxt -o data/all_data.record
```

Training

We mainly used an SSD with a MobileNetv2 as backbone from the TensorFlow model zoo, but the following steps should also work with all models of the zoo. The Tensorflow Object Detection API completely abstracts from the actual python/C++ code and allows to specify the entire model and its hyperparameters via a pipeline configuration. The configuration file is usually called "pipeline.config" and comes with the pretrained models of Tensorflow's model zoo. We saved the pipelines that were of interest for us at "./models/CHOOSEN_MODEL/pipeline.config". Key parameters that should be adjusted are:

- paths to the label_map, training set and evaluation set
- `num_classes`: Set to the number of different label classes
- `batch_size`: Increase or decrease depending on available memory
- `lable_map_path`: Path to label map
- `num_steps`: Number of steps used to train the model
- `num_steps`: Number of steps used to train the model

To start an exemplary training run

```
$python tf-models/research/object_detection/model_main_tf2.py --model_dir=./models/
↪finetuned --pipeline_config_path=./models/ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-
↪8/pipeline.config --alsologtostderr
```

If there are still memory problems, the parameters `queue_capacity` and `min_after_dequeue` can be decreased to reduce memory consumption.

We worked with different (SSD-based) architectures and experimented with different hyperparameters. Our final model training was conducted using the following hyperparameters:

- Batch size = 32 was chosen to balance memory consumption and training time.
- Number of steps = 100000
- Base learning rate = 0.08
- We used a momentum optimizer, with a warm up phase and a decaying learning rate. The warm up phase starts with a learning rate of 0.03 and lasts 1000 steps. The rate of decay depends on the number of totals steps.
- Other hyperparameters were not changed.

Evaluation

If the data is split into a train and a test set, the training process can be evaluated in real time with Tensorboard.

Run the following command in a new terminal:

```
$python tf-models/research/object_detection/model_main_tf2.py --model_dir=./models/PATH_
↪TO_MODEL --pipeline_config_path=./models/ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8/
↪pipeline.config --alsologtostderr --checkpoint_dir=models/PATH_TO_CHECKPOINT
```

The SSD with a MobileNetv2 backbone reached a mAP of *79%*. The Model with a ResNet backbone performed slightly worse in all metrics.

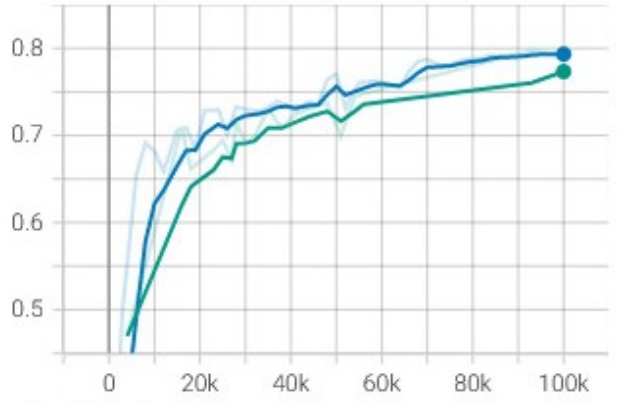
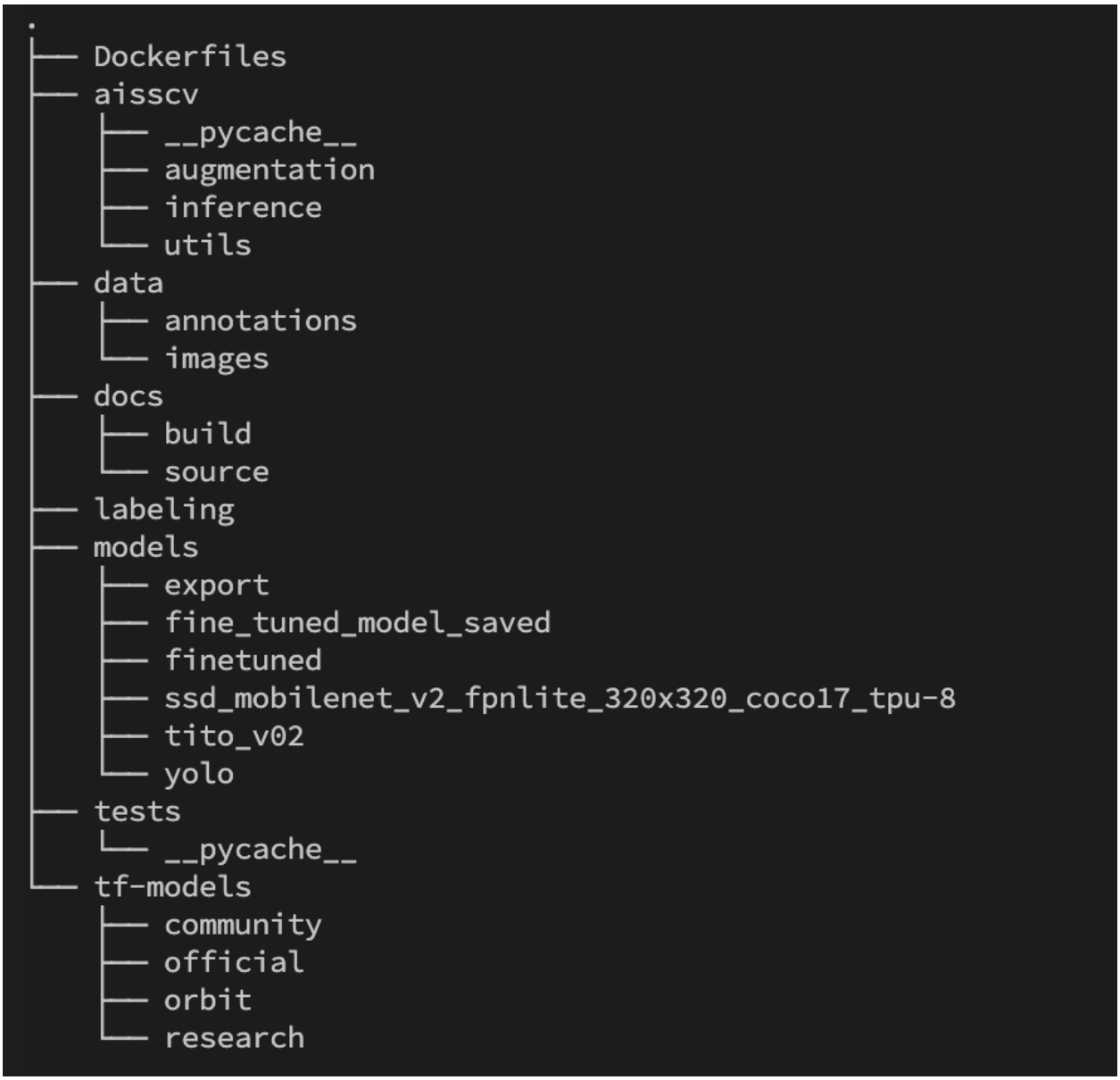


Fig. 3.2: The blue line indicates the mAP of the Model with a MobileNetv2 backbone and the green line of the Model with a ResNet backbone.

The simultaneous evaluation requires additional memory. This should be considered when determining the memory requirements during the creation of the container. Alternatively, the evaluation can be started from a second container that has access to the volume where the checkpoints are saved.

3.3 Code Overview

Written by Joel Oswald Here we give a brief overview over our code. For detailed reference, please refer to the *Modules* section in the Table of Contents.



3.4 Subsequent Damage Handling

Written by Alexey Rosenberg *and* Joel Oswald

This section covers the business logic of the prototype, including the results created by software from the model's inference, the tools used and the output shown to the user.

3.4.1 Description of Business Logic in the Software

The goal of the software is to inform the user, when a package with damages is detected by the system. Therefore, a user should receive a notification only if a package is detected and it has at least one damage.

The main work is done in the control_loop.py. In this while-loop, the chosen model is called via it's unified interface and the returned boxes are threshholded. If one or more damage classes are present in the clss list, this is remembered to be confirmed in the next cycles. A counter and an if-clause check, if damges are present in at least three consequetive inferences.

```
if confirm_cycles >= 3:
```

If this is the case, the if-clause is entered and a new thread with the notify-method for user notification created with the following command. The labeled image with the damaged package is given as argument.

```
threading.Thread(target=notify, args=(img, cls_dict[damage])).start()
```

In the notify-method, the Pushover API is called to create a notification with a message including the identified damages and the labeled image. "token" and "user" serves as unique identification for correct user detection by Pushover. The "files=" creates the image attachment.

```
r = requests.post("https://api.pushover.net/1/messages.json",
              data={"token": "azfjvmj5p96sydm337nqsezsc3ts5h", "user":
↪"urjgtk3d1o6yp9sv5ngi2zg1ooo6pn", "message": "WARNING- Detected Package with {}".
↪format(damage) },
              files={ "attachment": ("image.jpg", cv2.imencode('.jpeg', img)[1].tobytes(),
↪ "image/jpeg")
```

Afterwards, a cool-down phase is activated, during which the system won't notify the user again, as it is likely still the same damage.

3.4.2 Examples of Results

The user receives a push notification on a phone or other mobile device with the message and attachment created earlier. In the figure 4.5 below an example user notification with a damged package is shown.

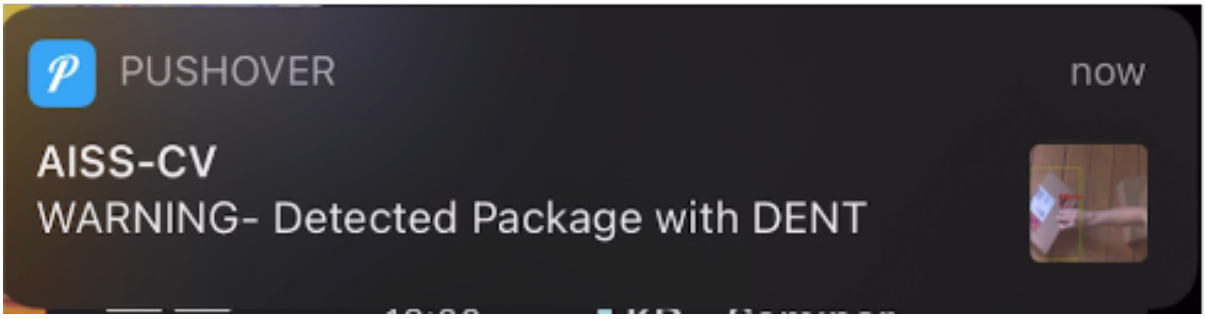


Fig. 3.3: Example of a preview notification

CHAPTER

FOUR

CODE DOCUMENTATION

4.1 Inference

4.1.1 Camera module

camera.py Based on jkjung: https://github.com/jkjung-avt/tensorrt_demos This code implements the Camera class, which encapsulates code to handle IP CAM, USB webcam or the Jetson onboard camera. In addition, this Camera class is further extended to take a video file or an image file as input.

class aisscv.inference.camera.**Camera**(*args*)

Bases: object

Camera class which supports reading images from theses video sources:

1. Image (jpg, png, etc.) file, repeating indefinitely
2. Video file
3. RTSP (IP CAM)
4. USB webcam
5. Jetson onboard camera

isOpened()

read()

Read a frame from the camera object.

Returns None if the camera runs out of image or error.

release()

aisscv.inference.camera.**add_camera_args**(*parser*)

Add parser augument for camera options.

aisscv.inference.camera.**grab_img**(*cam*)

This 'grab_img' function is designed to be run in the sub-thread. Once started, this thread continues to grab a new image and put it into the global 'img_handle', until 'thread_running' is set to False.

aisscv.inference.camera.**open_cam_gstr**(*gstr*, *width*, *height*)

Open camera using a GStreamer string.

Example: gstr = 'v4l2src device=/dev/video0 ! video/x-raw, width=(int){width}, height=(int){height} ! videoconvert ! appsink'

aisscv.inference.camera.**open_cam_onboard**(*width*, *height*)

Open the Jetson onboard camera.

aisscv.inference.camera.**open_cam_rtsp**(*uri*, *width*, *height*, *latency*)
Open an RTSP URI (IP CAM).

aisscv.inference.camera.**open_cam_usb**(*dev*, *width*, *height*)
Open a USB webcam.

4.1.2 Control Loop module

ControlLoop.py

Key Loop for our AISS_CV Projekt. Heavily based on the work of jkjung: https://github.com/jkjung-avt/tensorrt_demos

aisscv.inference.control_loop.**loop_and_detect**(*cam:* aisscv.inference.camera.Camera, *model:* aisscv.inference.model_interface.Model, *conf_th: float*, *vis:* aisscv.inference.visualization.BBoxVisualization, *cls_dict: dict*, *freq: int = 10*, *headless: bool = False*) → None
Continuously capture images from camera and detectt Boxes&Damages. If an image is detected, notify the connected phones via pushover API (in a thread, so that the loop may go on). If not in headless mode, this function will also display a window with current detections.

Parameters

- **cam** – the camera instance (video source).
- **model** – the object detector instance wich implements the model_interface (e.g. TRT-Yolo or Tf-SSD).
- **conf_th** – confidence/score threshold for object detection.
- **cls_dict** – Dictionary which assigns a class name to each index.
- **vis** – for visualization.
- **freq** (*int,*) – frequency to run the control loop
- **headless** (*bool*) – prevent the image and boxes to be visualized in an X-Window

Returns

Return type None

aisscv.inference.control_loop.**main**() → None
Main Function that reads arguments and sets up the control loop. End with 'ESC' or 'q' key to properly close camera stream

Returns

Return type None

aisscv.inference.control_loop.**notify**(*img: numpy.ndarray*, *damage: str = 'Damage'*)
Call Pushover-API to notify smartphone about warning. Please call in a thread, as this will block the loop otherwise.

Parameters

- **img** (*np.ndarray*) – Opencv image that should be transmitted
- **damage** (*str*) – Name of the damage that should be displayed

Returns

Return type None

aisscv.inference.control_loop.**parse_args**()
Parse command line arguments.

4.1.3 Display module

display.py Based on jkjung: https://github.com/jkjung-avt/tensorrt_demos

class aisscv.inference.display.**FpsCalculator**(*decay_factor=0.95*)
Bases: object

Helper class for calculating frames-per-second (FPS).

reset()

update()

class aisscv.inference.display.**ScreenToggler**
Bases: object

Helper class for toggling between non-fullscreen and fullscreen.

toggle()

aisscv.inference.display.**open_window**(*window_name*, *title*, *width=None*, *height=None*)
Open the display window.

aisscv.inference.display.**set_display**(*window_name*, *full_scrn*)
Set disply window to either full screen or normal.

aisscv.inference.display.**show_fps**(*img*, *fps*)
Draw fps number at top-left corner of the image.

aisscv.inference.display.**show_help_text**(*img*, *help_text*)
Draw help text on image.

4.1.4 Model Interface

class aisscv.inference.model_interface.**Model**(*model_path*, *input_shape*)
Bases: object

Interface for detection model. This allows a model-agnostic control loop

detect(*img*, *conf_th*)
Abstract function. To be overritten

4.1.5 SSD-TF module

ssd_tf.py

This module implements the TfSSD class.

class aisscv.inference.ssd_tf.**TfSSD**(*model_path*, *input_shape*)
Bases: *aisscv.inference.model_interface.Model*

TfSSD class encapsulates things needed to run TensorFlow SSD.

detect(*img*, *conf_th*)
Abstract function. To be overritten

4.1.6 Visualization module

visualization.py Based on jkjung: https://github.com/jkjung-avt/tensorrt_demos

The BBoxVisualization class implements drawing of nice looking bounding boxes based on object detection results.

class aisscv.inference.visualization.**BBoxVisualization**(*cls_dict*)
Bases: object

BBoxVisualization class implements nice drawing of boudning boxes.

 # Arguments cls_dict: a dictionary used to translate class id to its name.

draw_bboxes(*img*, *boxes*, *confs*, *clss*)
Draw detected bounding boxes on the original image.

aisscv.inference.visualization.**draw_boxed_text**(*img*, *text*, *topleft*, *color*)
Draw a transluent boxed text in white, overlayed on top of a colored patch surrounded by a black border. FONT, TEXT_SCALE, TEXT_THICKNESS and ALPHA values are constants (fixed) as defined on top.

 # Arguments img: the input image as a numpy array. text: the text to be drawn. topleft: XY coordinate of the topleft corner of the boxed text. color: color of the patch, i.e. background of the text.

 # Output img: note the original image is modified inplace.

aisscv.inference.visualization.**gen_colors**(*num_colors*)
Generate different colors.

 # Arguments num_colors: total number of colors/classes.

 # Output

bgrs: a list of (B, G, R) tuples which correspond to each of the colors/classes.

4.1.7 Yolo module

yolo_with_plugins.py Based on jkjung: https://github.com/jkjung-avt/tensorrt_demos Implementation of TrtYOLO class with the yolo_layer plugins.

class aisscv.inference.yolo_with_plugins.**HostDeviceMem**(*host_mem*, *device_mem*)
Bases: object

Simple helper data class that's a little nicer to use than a 2-tuple.

class aisscv.inference.yolo_with_plugins.**TrtYOLO**(*model_path*, *category_num=80*, *letter_box=False*, *cuda_ctx=None*)
Bases: *aisscv.inference.model_interface.Model*

TrtYOLO class encapsulates things needed to run TRT YOLO.

detect(*img*, *conf_th=0.3*, *letter_box=None*)
Detect objects in the input image.

aisscv.inference.yolo_with_plugins.**allocate_buffers**(*engine*)
Allocates all host/device in/out buffers required for an engine.

aisscv.inference.yolo_with_plugins.**do_inference**(*for TensorRT 6.x or lower*)
This function is generalized for multiple inputs/outputs. Inputs and outputs are expected to be lists of HostDeviceMem objects.

aisscv.inference.yolo_with_plugins.**do_inference_v2**(*for TensorRT 7.0+*)
This function is generalized for multiple inputs/outputs for full dimension networks. Inputs and outputs are expected to be lists of HostDeviceMem objects.

aisscv.inference.yolo_with_plugins.**get_input_shape**(*engine*)
Get input shape of the TensorRT YOLO engine.

aisscv.inference.yolo_with_plugins.**load_plugin_layer**()
Try to load the TensorRT plugin layer for yolo

4.2 Augmentation

4.2.1 Augment Dataset module

aisscv.augmentation.augment_dataset.**augment_one**(*image: imageio.core.util.Array*, *repetitions: int = 3*, *num_augmenters: int = 2*, *bboxes: Optional[numpy.array] = None*, *polygons: Optional[numpy.array] = None*, *show: bool = False*) → tuple
Augment one image multiple times. Boundingbboxes can be passed to be augmented in the same way the image is augmented. The augmentations is seperated into two steps:

- pixel augmentations that do not affect the bounding box
- affine augmentations that change the shape of the image, and hence the bounding boxes

For each iteration, some of the augmentations are chosen of each type. Each augmentation itself is random again in the intensity it is applied.

Parameters

- **image** (*imageio.core.util.Array or np.ndarray*) – image to be augmented, numpy-like
- **repetitions** (*int*) – How many pictures to create out of the input image.
- **num_augmenters** (*int*) – How many augmentations steps of each class (pixel and affine) are applied
- **bboxes** (*np.ndarray or list of list*) – List or array containing multiple bounding boxes in format [[xmin,xmax,ymin,ymax],[...],...]
- **polygons** (*np.array or list of lists*) – polygons to be augmented. NOT YET IMPLEMENTED
- **show** (*boolean, optional*) – wether to open a window wich shows the augmented images

Returns **list**

Return type [images_list, bbox_list]

aisscv.augmentation.augment_dataset.**create_augmented_dataset**(*source_dir: str*, *label_path: str*, *target_dir: str*, *repetitions_per_image: int = 5*, *stop_after: int = False*) → None
Function which creates an entire augmented Dataset. Reads in JSON/COCO and images. Write out the images + xml-labels to the specified directory. This takes only packages into account!

Parameters

- **source_dir** (*str*) – Source images that should be augmented
- **label_path** (*str*) – Path to the label file in json/coco format.

- **target_dir** (*str*) – Directory where the final, augmented dataset is saved, along with the labels.
- **repetitions_per_image** (*int, optional*) – How many augmented images to create per real image. Defaults to 5.
- **stop_after** (*int*) – Stop the loop after so many steps

Returns

Return type None

aisscv.augmentation.augment_dataset.**main**(*test=False*)

4.2.2 Augment Dataset with damages module

aisscv.augmentation.augment_dataset_with_damages.**augment_one**(*image: imageio.core.util.Array*, *repetitions: int = 3*, *num_augmenters: int = 2*, *objects: Optional[dict] = None*, *polygons: Optional[numpy.array] = None*, *show: bool = False*) → tuple

Augment one image multiple times. Bounding boxes can be passed to be augmented in the same way the image is augmented. The augmentations is seperated into two steps:

- pixel augmentations that do not affect the bounding box
- affine augmentations that change the shape of the image, and hence the bounding boxes

For each iteration, some of the augmentations are chosen of each type. Each augmentation itself is random again in the intensity it is applied.

Parameters

- **image** (*imageio.core.util.Array or np.ndarray*) – image to be augmented, numpy-like
- **repetitions** (*int*) – How many pictures to create out of the input image.
- **num_augmenters** (*int*) – How many augmentations steps of each class (pixel and affine) are applied
- **bboxes** (*np.ndarray or list of list*) – List or array containing multiple bounding boxes in format [[xmin,xmax,ymin,ymax],[...],...]
- **polygons** (*np.array or list of lists*) – polygons to be augmented. NOT YET IMPLEMENTED
- **show** (*boolean, optional*) – wether to open a window wich shows the augmented images

Returns **list**

Return type [images_list, bbox_list]

aisscv.augmentation.augment_dataset_with_damages.**create_augmented_dataset**(*source_dir: str*, *label_path: str*, *target_dir: str*, *repetitions_per_image: int = 5*, *stop_after: int = False*) → None

Function which creates an entire augmented Dataset. This includes all type of labels, not only boxes. Reads in

JSON/COCO and images. Write out the images + xml-labels to the specified directory.

Parameters

- **source_dir** (*str*) – Source images that should be augmented
- **label_path** (*str*) – Path to the label file in json/coco format.
- **target_dir** (*str*) – Directory where the final, augmented dataset is saved, along with the labels.
- **repetitions_per_image** (*int, optional*) – How many augmented images to create per real image. Defaults to 5.
- **stop_after** (*int*) – Stop the loop after so many steps

Returns

Return type None

4.3 Utils

4.3.1 Crop-Resize module

Resize the image to given size. Don't strech images, crop and center instead. Adapted from https://gist.github.com/zed/4221180

aisscv.utils.crop_resize.**crop_resize**(*image*, *size*, *ratio*)

aisscv.utils.crop_resize.**crop_resize_mp**(*input_dir*, *outputdir*, *size*, *ratio*)

aisscv.utils.crop_resize.**main**()

4.3.2 Label Helper module

aisscv.utils.label_helper.**add_bbox_to_image**(*image: numpy.ndarray*, *object: dict*) → numpy.ndarray
Add the boundingbox of the given object to the image. Since using CV, order is (B,G,R)

Parameters

- **img** (*np.ndarray*) – Opencv image
- **object** (*dict*) – Dictionary of the boundingbox with meta infos.

Returns image with the bbndbox in (B,G,R)

Return type np.ndarray

aisscv.utils.label_helper.**add_polygon_to_image**(*image: numpy.ndarray*, *object: dict*) → numpy.ndarray
Add the polynom of the given object to the image. Since using CV, order is (B,G,R)

Parameters

- **img** (*np.ndarray*) – Opencv image
- **object** (*dict*) – Dictionary of the polynom with meta infos.

Returns image with the polynom in (B,G,R)

Return type np.ndarray

aisscv.utils.label_helper.**demo**(*path: str = '/Users/joel/UNI_Offline/Group-Git/docs/data/annotations/labels_joel_json.json'*, *n: int = 15*, *headless: bool = False*) → None

Simple demo that shows some labels

Parameters

- **path** (*str*) – Directory where via-exported json is located
- **headless** (*bool*) – Prevent cv from showing (for automated tests)

Returns

Return type True, for testing

aisscv.utils.label_helper.**resize_images_df**(*df: pandas.core.frame.DataFrame*, *resize_width: int*, *resize_height: int*, *directory: str = False*, *img_to_df: bool = False*) → pandas.core.frame.DataFrame

Reading a DataFrame as returned by via_[coco|json]_to_df. Resizing the image itself and the objects. Resized objects get resized in place, resized images can be saved in extra Column in the Dataframe or in a directory. Attention, currently only support resizing of bounding boxes, so please use poly_to_box=True when creating the DataFrame.

Parameters

- **df** (*pd.DataFrame*) – DataFrame that contains all Information and labels
- **resize_width** (*int*) – Number of width-pixels after rescaling
- **resize_height** (*int*) – Number of height-pixels after rescaling
- **directory** (*str, optional*) – Wheter rescaled images should be written to directories
- **img_to_df** (*bool, optional*) – Wheter rescaled images should be added to returned df

Returns Dataframe containing all the information

Return type pd.DataFrame

aisscv.utils.label_helper.**visualize_labels**(*df: pandas.core.frame.DataFrame*, *n: int*, *headless: bool = False*) → None

Visualize the images of the given DataFrame. Since using CV, order is (B,G,R). n Images are shown, get to next image by pressing enter. Every other key will exit the loop. Loop will exit if no enter pressed for 10 secods.

Parameters

- **img** (*pd.DataFrame*) – dataframe with file_name, objects and all the label infos
- **n** (*int*) – How many images are visualized (sampled from dataframe)
- **headless** (*bool*) – Prevent cv from showing (for automated tests)

Returns

Return type None

aisscv.utils.label_helper.**write_to_pascal_voc**(*df: pandas.core.frame.DataFrame*, *directory: str*, *only_box_label: bool = False*) → None

Reading a DataFrame as returned by via_[coco|json]_to_df. Writing it to single xml files for each image, as specified by the Pascal VOC format. Attention, Pascal-Voc only supports bounding boxes, so please use poly_to_box=True when creating the DataFrame.

Parameters

- **df** (*pd.DataFrame*) – DataFrame that contains all Information and labels

- **directory** (*str*) – Directories where the xml files are written to.
- **only_box_labels** (*bool*) – only write box labels, skip all other

Returns

Return type None

Raises **Exception, if dataframe woth polygon was passed. Please convert polygons to boundingboxes before.** –

4.3.3 Label Reader module

aisscv.utils.label_reader.**via_coco_to_df**(*label_file_path: str*, *image_dir: str = '/Users/joel/UNI_Offline/Group-Git/docs/data/images'*, *poly_to_box=False*, *add_img_size=False*) → pandas.core.frame.DataFrame

Function which reads in a json in COCO-Format, that was exported by the VGG Image Annotator (via) and transorfms it to a dataframe. The dataframe is already equipped with pascal-voc-like keywords.

Parameters

- **label_file_path** (*str*) – Path to the COCO-styled json-labelfile
- **image_dir** (*str, optional*) – Directories where the images reside. Defaults to the path in git repository /data/images
- **poly_to_box** (*bool, optional*) – Wether polynoms should be converted to a bounding_box. Defaults to no
- **add_img_size** (*bool, optional*) – Load the image to extract size. Will massively slow down the conversion. Should be turned on, when meant to export to a different format (i.e. PASCAL VOC) but can be kept of when just visualizing

Returns Dataframe with one row per image.

Return type pd.DataFrame

aisscv.utils.label_reader.**via_json_to_df**(*label_file_path: str*, *image_dir: str = '/Users/joel/UNI_Offline/Group-Git/docs/data/images'*, *poly_to_box=False*, *add_img_size=False*, *skip_format_problems=False*) → pandas.core.frame.DataFrame

Function which reads in a json, that was exported by the VGG Image Annotator (via) and transorfms it to a dataframe. The dataframe is already equipped with pascal-voc-like keywords.

Parameters

- **label_file_path** (*str*) – Path to the json-labelfile
- **image_dir** (*str, optional*) – Directories where the images reside. Defaults to the path in git repository /data/images
- **poly_to_box** (*bool, optional*) – Wether polynoms should be converted to a bounding_box. Defaults to no
- **add_img_size** (*bool, optional*) – Load the image to extract size. Will massively slow down the conversion. Should be turned on, when meant to export to a different format (i.e. PASCAL VOC) but can be kept of when just visualizing
- **skip_format_problems** (*bool, optional*) – If a wrong label-type is detected skip that annotation. Defaults to False, which throws an error to fix the label

Returns Dataframe with one row per image.

Return type pd.DataFrame

4.3.4 Negative Examples module

Generates empty label files and appends the filenames to the train.txt for yolo. usage: negative_examples.py [-h] [–train_txt_path TRAIN_TXT_PATH] [–negative_image_dir NEGATIVE_IMAGE_DIR]

Generates empty label files and appends the filenames to the train.txt for yolo.

optional arguments:

-h, --help show this help message and exit

--train_txt_path TRAIN_TXT_PATH Path to the train.txt file that you want to append the files to.

--negative_image_dir NEGATIVE_IMAGE_DIR If non-empty, adds empty .txt files for the images in the given path.

aisscv.utils.negative_examples.**create_empty_labels**(*dir: str*, *train_txt_path: str*) → None
Writes empty label files for the images in out_dir and appends the train.txt. If the empty label files exists, does not overwrite it.

aisscv.utils.negative_examples.**main**()

4.3.5 Pascal-VOC to tf-record module

Sample TensorFlow XML-to-TFRecord converter

usage: generate_tfrecord.py [-h] [-x XML_DIR] [-l LABELS_PATH] [-o OUTPUT_PATH] [-i IMAGE_DIR] [-c CSV_PATH]

optional arguments:

-h, --help show this help message and exit

-x XML_DIR, --xml_dir XML_DIR Path to the folder where the input .xml files are stored.

-l LABELS_PATH, --labels_path LABELS_PATH Path to the labels (.pbtxt) file.

-o OUTPUT_PATH, --output_path OUTPUT_PATH Path of output TFRecord (.record) file.

-i IMAGE_DIR, --image_dir IMAGE_DIR Path to the folder where the input image files are stored. Defaults to the same directory as XML_DIR.

-c CSV_PATH, --csv_path CSV_PATH Path of output .csv file. If none provided, then no file will be written.

aisscv.utils.pascal_voc_to_tfrecord.**class_text_to_int**(*row_label*)

aisscv.utils.pascal_voc_to_tfrecord.**create_tf_example**(*group*, *path*)

aisscv.utils.pascal_voc_to_tfrecord.**main**(_)

aisscv.utils.pascal_voc_to_tfrecord.**parse_args**()

aisscv.utils.pascal_voc_to_tfrecord.**split**(*df*, *group*)

aisscv.utils.pascal_voc_to_tfrecord.**xml_to_csv**(*path*)
Iterates through all .xml files (generated by labelImg) in a given directory and combines them in a single Pandas dataframe.

Parameters `path` (*str*) – The path containing the .xml files

Returns The produced dataframe

Return type Pandas DataFrame

4.3.6 Pascal-VOC to yolo module

usage: pascal_voc_to_yolo.py [-h] [-x XML_DIR] [-o OUTPUT_PATH] [–relative_img_path RELATIVE_IMG_PATH] [–train_
[–negative_image_dir NEGATIVE_IMAGE_DIR]

Sample Pascal VOC XML-to-YOLO converter

optional arguments:

-h, --help show this help message and exit

-x XML_DIR, --xml_dir XML_DIR Path to the folder where the input .xml files are stored.

-o OUTPUT_PATH, --output_path OUTPUT_PATH Path of output (all yolo-training-related data).

--relative_img_path RELATIVE_IMG_PATH Path to prepend to the train.txt files. This path must determines where the training images will reside. Set train_txt_path to path/to/train.txt you want to change.

--train_txt_path TRAIN_TXT_PATH Path to the train.txt file that you want to change. Only necessary if you want to modify this file with –relative_img_path.

--negative_image_dir NEGATIVE_IMAGE_DIR If non-empty, adds empty .txt files for the images in the given path.

`aisscv.utils.pascal_voc_to_yolo.`**`change_train_txt_paths`**(*train_txt_path: str*, *path_to_add: str*) → None

Changes the path that is prepended to the image files in the train.txt file. This path must be the path where the files for YOLO training will reside.

`aisscv.utils.pascal_voc_to_yolo.`**`convert`**(*size: tuple*, *box: tuple*) → tuple

From https://github.com/AlexeyAB/darknet

`aisscv.utils.pascal_voc_to_yolo.`**`convert_annotations`**(*in_dir: str*, *classes: list*, *out_dir: str*, *relative_img_path: str*) → None

Converts Pascal VOC annotations to YOLO annotations and creates a train.txt required for training. Adapted from https://github.com/AlexeyAB/darknet

`aisscv.utils.pascal_voc_to_yolo.`**`create_obj_names`**(*classes: list*, *out_dir: str*) → None

Creates the obj.names file required for training.

`aisscv.utils.pascal_voc_to_yolo.`**`main`**(*args*)

`aisscv.utils.pascal_voc_to_yolo.`**`parse_args`**()

4.3.7 Proto to Tensorboard module

Imports a protobuf model as a graph in Tensorboard.

aisscv.utils.read_to_tensorboard.**import_to_tensorboard**(*model_dir*, *log_dir*, *tag_set*)
View an SavedModel as a graph in Tensorboard.

Parameters

- **model_dir** – The directory containing the SavedModel to import.
- **log_dir** – The location for the Tensorboard log to begin visualization from.
- **tag_set** – Group of tag(s) of the MetaGraphDef to load, in string format, separated by ‘,’. For tag-set contains multiple tags, all tags must be passed in.

Usage: Call this function with your SavedModel location and desired log directory. Launch Tensorboard by pointing it to the log directory. View your imported SavedModel as a graph.

aisscv.utils.read_to_tensorboard.**main**(_)

4.4 Unit Tests

4.4.1 Test module

tests.test_module.**test_augmentation**()

tests.test_module.**test_augmentation_with_damages**()

tests.test_module.**test_function**()

tests.test_module.**test_helper**()

tests.test_module.**test_pascal_conversion**()

tests.test_module.**test_rescale**()

tests.test_module.**test_yolo_conversion**()

tests.test_module.**test_yolo_create_empty_labels**()

PYTHON MODULE INDEX

a

t

INDEX

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X