Pluggable Drone Imaging Analysis Framework for Mob Detection during Open-air Events

Jerico Moeyersons, Brecht Verhoeve, Pieter-Jan Maenhaut, Bruno Volckaert and Filip De Turck
IDLab, Department of Information Technology at Ghent University - imec
Zwijnaarde-Technologiepark 15 B-9052 Ghent, Belgium

Keywords: Drone Thermal Imaging, Video Streaming, Framework, Microservices, Object Detection, Plugin.

Abstract: Drones and thermal cameras are often combined within applications such as search and rescue, and fire fighting. Due to vendor specific hardware and software, applications for these drones are hard to develop and maintain. As a result, a pluggable drone imaging analysis architecture is proposed that facilitates the development of custom image processing applications. This architecture is prototyped as a microservice-based plugin framework and allows users to build image processing applications by connecting media streams using microservices that connect inputs (e.g. regular or thermal camera image streams) to image analysis services. The prototype framework is evaluated in terms of modifiability, interoperability and performance. This evaluation has been carried out on the use case of detecting large crowds of people (mobs) during open-air events. The framework achieves modifiability and performance by being able to work in soft real-time and it achieves the interoperability by having an average successful exchange ratio of 99.998%. A new dataset containing thermal images of such mobs is presented, on which a YOLOv3 neural network is trained. The trained model is able to detect mobs on new thermal images in real-time achieving frame rates of 55 frames per second when deployed on a modern GPU.

1 INTRODUCTION

Throughout history, having an overview of the environment from high viewpoints held many benefits. Nowadays, drones are relatively cheap and provide a rapid manner to get an overview of a specific area. The advent of drones and advanced cameras provides low-cost aerial imaging that creates numerous novel application opportunities. Traditional cameras for the visible light spectrum offer high quality images, but are limited to daytime or artificially lighted scenes. Thermal cameras measure thermal radiation of objects in a scene and can therefore operate in utter darkness, revealing information not visible to the normal eye (Gade and Moeslund, 2014). The combination of drones and thermal cameras is used in many different applications such as geography (Harvey et al., 2016), agriculture (Bendig et al., 2012), search and rescue (Rivera et al., 2017), wildlife monitoring (Christiansen et al., 2014), disaster response (Gonnissen, 2016), maintenance (Workswell, 2016), etc.

Several vendors offer thermal camera products, some specifically designed for drone platforms. These cameras often use different image formats, color schemes and interfaces (Gade and Moeslund, 2014). This leads to issues if applications want to change the camera that is used, or when the camera is no longer supported by the vendor, because different software needs to be built to interact with the new camera, which often is a non-negligible cost. This leads to a problem called vendor lock-in that makes customers dependent on a certain vendor as they cannot switch product without making substantial costs, a problem already very tangible for cloud-based applications (Satzger et al., 2013). Applications across various fields often have slightly different functional and non-functional requirements. As a precursor, several Belgian fire fighting departments were asked for requirements for a thermal drone platform application. It quickly became clear that they had various problems that needed to be solved, such as finding zones with potential danger of explosions (due to the combination of high temperature and explosive goods stored in the vicinity), measuring temperatures in containers, identifying hot entrances (to counter backdraft e.g. when opening a door), detecting invisible methane fires, finding missing persons, etc. Some use cases need to be evaluated in real-time (during fires), while

Moeyersons, J., Verhoeve, B., Maenhaut, P., Volckaert, B. and De Turck, F.
Pluggable Drone Imaging Analysis Framework for Mob Detection during Open-air Events.
DOI: 10.5220/0007260400640072
In *Proceedings of the 8th International Conference on Pattern Recognition Applications and Methods (ICPRAM 2019)*, pages 64-72
ISBN: 978-989-758-351-3

others focus on accuracy (e.g. inspections). An application should therefore be able to quickly implement new detection and analysis features to meet these requirements. Due to the lack of modifiability present in current software solutions, applications built on top of aerial thermal imaging remain largely niche and vendor/equipment-locked (Divya, 2017). Applications would benefit from a software framework focused on modifiability and interoperability, to aid in developing technology-agnostic end-to-end solutions connecting thermal cameras to different image analysis / detection modules.

This paper presents the requirements for such a framework and proposes a suitable software architecture. To test the viability of the architecture, a proof-of-concept prototype is implemented and evaluated against the initial requirements. As a use case, detection of large crowds (so-called mobs) during open-air events is investigated. Monitoring crowds during open-air events is important, as mobs can create potentially dangerous situations through bottlenecks, blocking escape routes, etc. Detection of mobs can also be used for rapid identification of fights and brawls. Through monitoring and detecting these mobs, these situations can be identified before they become problematic (Steffen and Seyfried, 2010).

The remainder of this paper is organized as follows. Section 2 presents similar projects on the topic of modifiable image analysis frameworks and thermal object detection. Section 3 presents the requirements of the framework and the software architecture designed from these requirements. Section 4 presents the implementation of the framework prototype. The mob detection experiment is described in Section 5. Evaluation of the framework and results of the mob detection experiment are presented in Section 6. Finally, Section 7 draws conclusions from this research and indicates where future efforts in this field should go to.

2 RELATED WORK

The Irish start-up DroneSAR (Slattery, 2017) developed a search-and-rescue (SAR) drone platform allowing users to stream live images and video from a drone as it conducts a search for missing persons. The platform works with any camera, visual and thermal but focuses on drones from vendor DJI, DroneSAR's industry partner. The solution proposed in this paper is not restricted to only DJI drones and can be employed for different use-cases.

Amazon introduced the Amazon Kinesis VideoStreams platform in January 2018 as a new service for the Amazon Web Services (AWS) cloud platform. It allows users to stream live video from devices to the AWS cloud and build applications for real-time video processing (Inc., 2018). While the Amazon Kinesis VideoStreams platform certainly has interesting features, it imposes a vendor lock-in as it relies solely on Amazon Cloud services, whereas the open-source solution proposed here can be deployed on a variety of (cloud) resources and it can even be deployed locally when there is no up-link to the cloud available.

The VIPER project by EAVISE and KU Leuven investigated how thermal and video images could be used for real-time detection of persons using object detection algorithms based on deep learning (Goedeme, 2017). Our work evaluated a mob detection experiment based on the YOLOv3 (Redmon and Farhadi, 2018) object detection algorithm, using a deep neural network. In previous work, a drone was used to detect objects in a predefined incident area such as barrels containing explosive fluid, fires, firefighters, etc. (Tijtgat et al., 2017a; Tijtgat et al., 2017b). The framework proposed in this paper offers an abstraction allowing developers to connect microservice based flows in terms of input (e.g. thermal video, regular video, images), processing (e.g. object detection, classification, localization) and output (e.g. REST API, custom applications).

3 REQUIREMENTS AND SOFTWARE ARCHITECTURE

3.1 Function Requirements

Three general actors are identified for the framework: an end-user that wants to build an image processing application for a specific use case, input sensor developers integrating new cameras / streams into the framework, and stream processing module developers integrating new analysis/detection algorithms (e.g. analysis, detection, localization, decision support) into the framework so that end-users can use them to build their applications. An end-user should be able to construct an image processing application by interconnecting predefined camera and analysis modules. Hence-built applications should be adaptable to new use cases by adding or replacing select modules, retaining the overall functionality of the application. Camera and analysis module developers should be able to add and distribute new camera integration modules (e.g. new type of thermal camera with support for per-pixel labeling of temperatures)

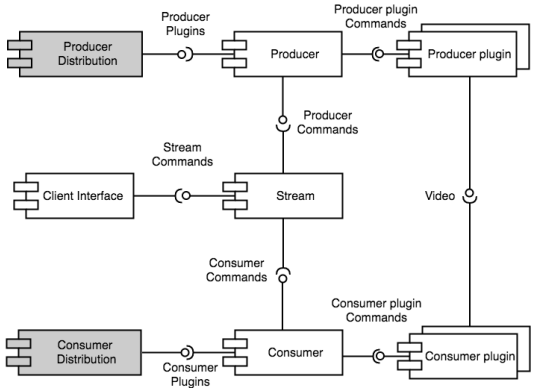


Figure 1: Component-connector overview of the framework. The clear components are the core components of the framework that each user needs to install to use the framework. The colored components are used for the distribution of plugins.

and analysis modules (e.g. detection of people at risk of being crushed during mass-events) to the framework. This allows end-users to focus on the use case, instead of the technical details of the hardware platforms or algorithms, and allows them to have a wider selection of hardware and algorithms.

3.2 Non-functional Requirements

Interoperability, modifiability and performance are identified as the architecturally significant requirements. Interoperability specifies that the framework should be able to interact with various cameras and analysis software modules via interfaces. The amount of systems the framework can successfully interact will increase the business value of the framework, as end-users can use more devices via the framework to build applications. The framework needs to be extendable with new thermal cameras and analysis modules. Applications built with the framework should be modifiable to integrate new hardware and software. The available hardware on-site for use cases such as forest fire monitoring is not always powerful enough to support heavy image analysis software. The framework should therefore be able to deploy in a distributed fashion, to allow more computationally expensive operations to be executed on more powerful (potentially cloud-based) remote devices. Some use cases such as live incident response require soft real-time streaming of video and manipulation of these video streams, which should be supported for the framework to be relevant.

3.3 Software Architecture

An architectural pattern analysis based on the requirements presented in Section 3.2 was conducted, from which a combination of the microservices and microkernel pattern was selected as the base pattern for the software architecture. The microservice architecture is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often a HTTP resource API (Lewis and Fowler, 2014). The microkernel pattern enables the framework to be extended via a plugin system and allows the framework to be deployed in a distributed and scalable manner.

Figure 1 presents a high-level overview of the architecture. End-users interact with the framework via the Command Line Interface (CLI), a textual interface. Commands are forwarded to the Stream module that manages the layout and state of the components in the media pipeline (stream) that the end-user builds for his use case. To activate and place the components in a certain layout, the Stream module connects with the Producer and Consumer components that control the plugins that process the media. Producer plugins are devices that produce media, such as thermal cameras. Consumer plugins process and consume media, such as analysis software and displays. Once a stream is established, the plugins forward media to each other in the layout specified by the Stream module. Support for new cameras and analysis modules can be added as plugins to the Producer/Consumer Distribution components that distribute this software so that end-users can download and install the plugins. Each module in the architecture is a microservice, allowing for distributed deployment and enabling the modifiability requirements through interchangeable modules. Cameras and analysis modules are realized as plugins for the Producer/Consumer modules implemented as a microkernel. This allows the framework to easily build new streams for different use cases and interchange components when needed.

3.3.1 Plugin Model

Figure 2 depicts the model of a general framework plugin. A plugin is part of the microkernel pattern and defines three interfaces: a source media endpoint to receive media from different sources, a listener endpoint to forward the processed media to other listening plugins and an API for framework control. The framework uses the API to change which sources and listener a plugin has and to manage its state. By linking plugins together by configuring the source and listener resources, the framework can build a media

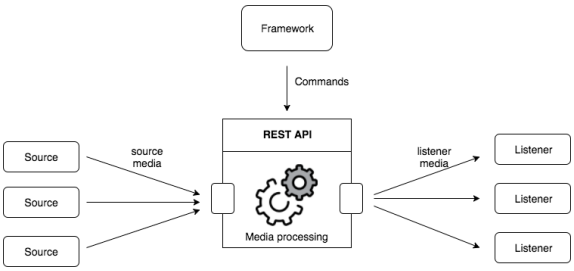


Figure 2: Schematic overview of a plugin.

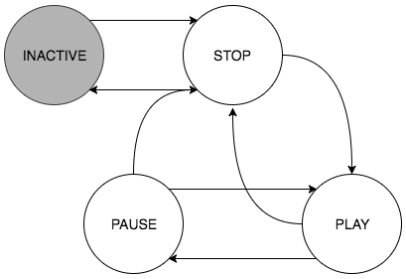


Figure 3: State transition diagram of a plugin.

processing stream pipeline. Producer plugins have no sources since they produce media. States are used to stop and start the media processing of the plugins in the stream. A plugin has the following states: INACTIVE, PLAY, STOP and PAUSE. Figure 3 depicts the state transition diagram for a plugin. A plugin implements the visible states STOP, PAUSE and PLAY describing if the media processing of the plugin is stopped, paused or processing respectively. The INACTIVE state is only visible to the framework as it indicates that there is no active process that runs the plugin. This is the initial state of a plugin in the framework. When a plugin is added to a stream, the plugin microservice is started, transitions to the STOP state and waits for commands. The REST paradigm is selected to build this API, with /state, /sources and /listeners resources that need to be minimally implemented.

3.3.2 Network Topology and Communication Protocol

The microservices of the framework (Producer, Steam, Consumer and CLI) and the plugins need a communication protocol to exchange commands and data. For sending the commands the HTTP/TCP communication is used. This ensures that commands are always received and acknowledged increasing the reliability of the communication. The asynchronous RTP/UDP protocol is selected to transfer media between the plugins to ensure low latency video transfer between plugins to enable soft real-time video streams. An example of a network topology for a

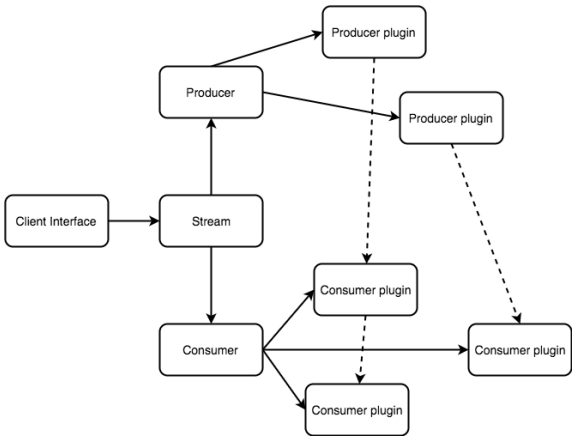


Figure 4: Network topology. The full lines represent HTTP/TCP communications, the dashed line RTP/UDP communications.

stream with 2 Producer plugins and 3 Consumer plugins is depicted in Figure 4.

4 PROTOTYPE IMPLEMENTATION

The goal of the prototype implementation is to provide a proof-of-concept framework allowing evaluation with regards to adherence to the requirements imposed in Section 3. The core framework components are implemented, the distribution components are left out of scope as they focus primarily on the distribution of supported plugins and as such any service repository technology can be used (e.g. Docker Hub (Docker, 2016)). The core microservices as well as the plugins are implemented using the Docker software containerization framework (Docker Inc., 2018). Containers virtualize on the operating system and allow for portable, lightweight software environments for processes with minor performance overhead. Using Docker, the core modules and plugins can be deployed in a local and distributed fashion, scaled up and down easily as per the needs of the application and be setup and shutdown easily. The microservice containers communicate via the protocols presented in Section 3.3.2. The REST APIs are built with the Flask framework (Ronacher, 2017), a lightweight Python web development framework well-suited for rapid prototyping. The Producer/Consumer components need access to the Docker daemon running on the Docker host in order to spin up and shutdown Docker containers containing the plugin processes. This is achieved by mounting the Docker client socket in the Producer/Consumer containers. As a downside, this gives the container a way to achieve access to the

Figure 5: Filecam plugin - GStreamer pipeline.

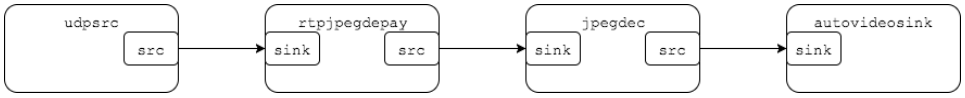


Figure 6: Local plugin - GStreamer pipeline.

host, and as such only trustworthy plugins should be used (Lvh, 2015; Yasrab, 2018). Two sample plugins were implemented: Filecam, a plugin that produces video read from a file as shown in Figure 5, and Display, a plugin that forwards media to the display of the local device as shown in Figure 6. The plugins transmit media using the GStreamer video streaming framework (GStreamer, 2018).

The prototype itself can be managed by the created CLI. Through the CLI, the framework can be started, stopped and new streams, connecting a producer plugin with a consumer plugin, can be created, linked, started and managed.

5 MOB DETECTION

5.1 Dataset

Several publicly available datasets for thermal images exist (Davis and Keck, 2005; Hwang et al., 2015; Wu et al., 2014; Li et al., 2007). None of these include large crowds of people, so a new dataset called the Last Post dataset was created.[1]. It consists of thermal video captured at the Last Post ceremony in Ypres, Belgium. The videos were captured using the Flir One Pro thermal camera for Android using the Iron colorscheme. Two main scenes are present in the dataset, depicted in Figure 7. Mobs are present in the thermal images, not in the visual images due to the images being made on separate days. The images captured for the experiment were manually annotated, outliers were removed and the dataset was randomly split in a training and validation set.

(a) Thermal view of the square

(b) Visual view of the square

(c) Thermal view of the bridge

(d) Visual view of the bridge

Figure 7: The Last Post dataset main scenes.

5.2 Model

Detecting and classifying objects of interest in images is known as the object detection problem in machine learning (Alpaydin, 2014). Several object detection algorithms and frameworks have been implemented in the past years. A distinction is made between traditional models, deep learning two-stage networks and deep learning dense networks. The traditional and two-stage methods make predictions relatively slow (in the order of seconds on GPU) when compared to the dense networks (order of milliseconds on GPU) (Redmon et al., 2015). Since the goal is to use the framework in soft real-time use cases the latter is preferred. The YOLOv3 model is selected as it achieves state of the art prediction performances, can make soft real-time predictions and is available via the open source neural network framework darknet (Redmon and Farhadi, 2018; Redmon, 2016). The model is pre-trained on the ImageNet dataset (Deng et al., 2009), trained on a NVIDIA Geforce 980 TX GPU and the SSE loss is optimized using batch gradient descent (Redmon and Farhadi, 2018). To select the best weights, the average Intersection of Union (IoU) and mean Average Precision (mAP) are calculated on predictions on the validation set. The weights that achieve the highest mAP are selected as the final weights.

[1] https://github.com/IBCNServices/Last-Post-Dataset

Table 1: Performance test statistics summary, measured in seconds. All the possible commands from the CLI are evaluated.

Statistic	Play	Stop	Pause	Add	Delete	Elements
Mean	0.690	0.804	0.634	1.363	8.402	0.562
Std deviation	0.050	0.059	0.088	1.037	4.669	0.070
Minimum	0.629	0.708	0.549	0.516	0.505	0.517
25 Percentile	0.665	0.775	0.594	1.049	1.154	0.534
Median	0.678	0.800	0.623	1.11	11.132	0.550
75 Percentile	0.700	0.820	0.653	1.233	11.189	0.562
Maximum	1.016	1.279	1.631	6.25	11.846	1.227
Statistic	**Print**	**View**	**On**	**Off**	**Link**	
Mean	0.564	1.22	3.58	24.023	0.849	
Std deviation	0.0747	0.260	0.498	0.481	0.170	
Minimum	0.517	0.757	3.015	23.707	0.637	
25 Percentile	0.536	0.998	3.143	23.750	0.798	
Median	0.552	1.214	3.500	23.886	0.853	
75 Percentile	0.560	1.433	3.850	24.034	0.877	
Maximum	1.149	1.691	4.562	25.326	1.261	

6 RESULTS

6.1 Framework

First, acceptance tests for the requirements from Section 3 were conducted. Common framework operations such as manipulating and building a stream have an average execution time of 0.84 seconds with a standard deviation of 0.37 seconds. Less common operations such as deactivating a plugin, starting up the framework and shutting down the framework have an average execution time of 3.58, 8.40 and 24.02 seconds with standard deviations 4.67, 0.50 and 0.48 respectively. Deactivating plugins (STOP to INACTIVE transitions) takes a long time, as the container running the process needs to be removed. Real-time streaming could not be tested due to the GStreamer framework having no readily available end-to-end tests. However when streaming and displaying a video with the framework a panel of 5 human users could not differentiate between a streaming video and a video played using a native media player. Great care must be taken when building plugins, as their processing speed has a direct impact on the real-time streaming performance. The results of the acceptance tests are summarized in Table 1.

Interoperability is achieved with the REST APIs and plugin model presented in Section 3.3.2. The interoperability is tested by having the framework exchange information with a mock plugin implementing the specified interface and counting the number of correct exchanges. The average successful exchange ratio is 99.998%, caused by the Flask framework not supporting special characters in an url. The framework can install and detect new plugins at runtime, achieving runtime modifiability at plugin level. Full results are summarized in Table 2. Different deployment schemes were not tested for the prototype.

6.2 Mob Detection

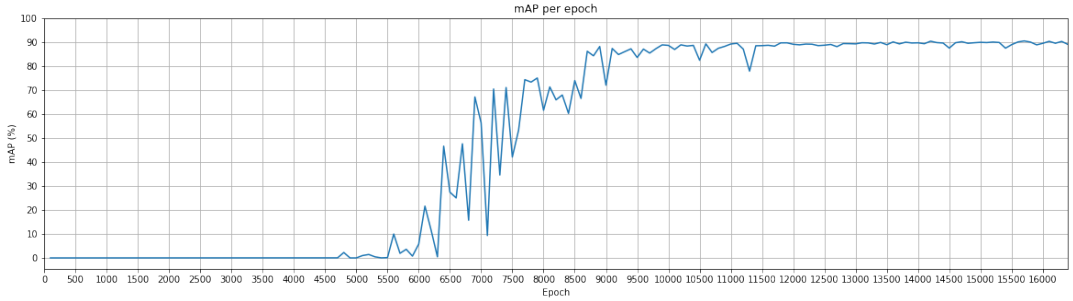
A YOLOv3 object detection consumer plugin is created and added to a full stream for the evaluation. YOLO creates a snapshot from the weights of the model every 100 epochs. (An epoch is when all the training data are used once to update the weights in the neural network). This allows to validate each set of weights on the validation set and show the evolution of the validation performance. Figure 8 shows these evolutions for the average IoU and mAP metrics, based on a pre-defined model provided by YOLO. This pre-defined model is trained on several colored images from the COCO-dataset (Lin et al., 2014) whereby multiple classes such as a dog, a bike, a car, etc. can be detected. As shown in Figure 8, the mAP gradually grows from epoch 4500 onwards and stagnates around epoch 11500. This shows that the model is not learning anymore and is at risk of overfitting. The mAP stagnates in the interval of [88%, 91%]. The average IoU shows a similar trend, but varies more, because predictions on the same images rarely are exactly the same.

If we restart the training process without the predefined model, the evolutions for the average IoU and mAP metrics show a remarkable difference compared to the training process with the pre-defined model, as shown in Figure 9.

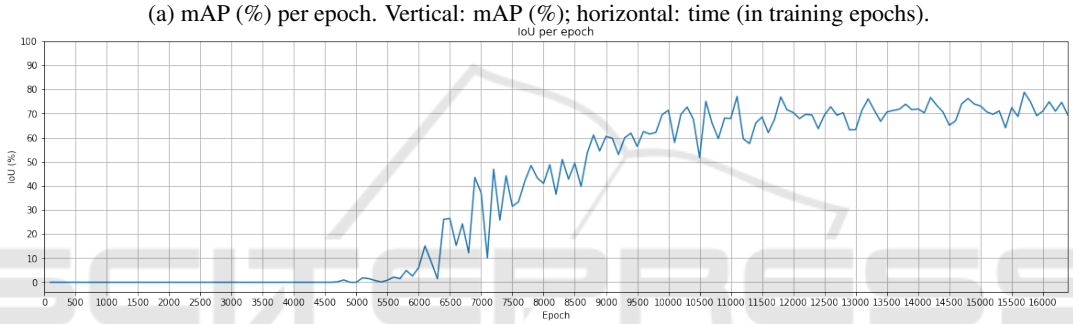
The best mAP value is achieved at epoch 15700

Table 2: Interoperability tests results (S.: Source, L.: Listener).

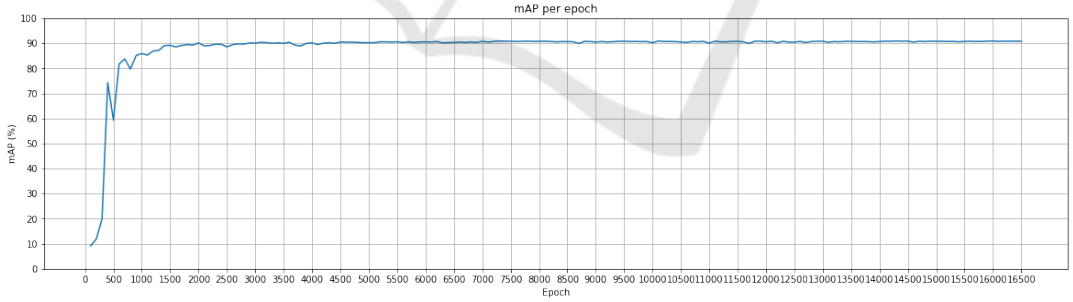
Value	Play	Pause	Stop	Add S.	Update S.	Delete S.	Add L.	Update L.	Delete L.
Correct	50000	50000	50000	50000	50000	49999	50000	50000	49999
Incorrect	0	0	0	0	0	1	0	0	1
%	100	100	100	100	100	99.998	100	100	99.998



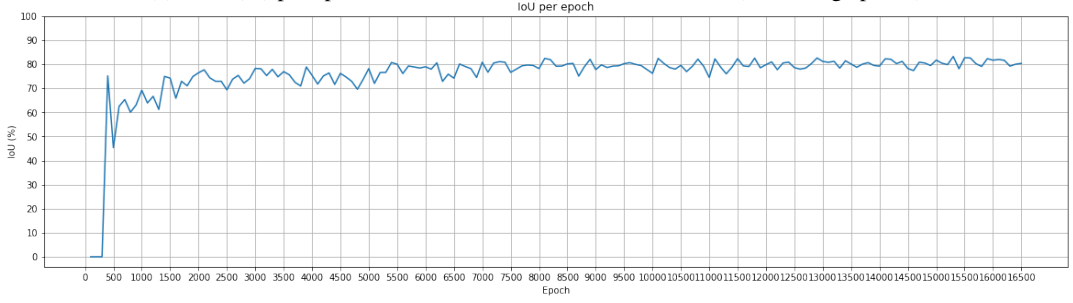
(a) mAP (%) per epoch. Vertical: mAP (%); horizontal: time (in training epochs).



(b) IoU (%) per epoch. Vertical: IoU (%); horizontal: time (in training epochs).
Figure 8: Validation metrics per epoch based on a pre-defined YOLO model.



(a) mAP (%) per epoch. Vertical: mAP (%); horizontal: time (in training epochs).



(b) IoU (%) per epoch. Vertical: IoU (%); horizontal: time (in training epochs).
Figure 9: Validation metrics per epoch without a pre-defined YOLO model.

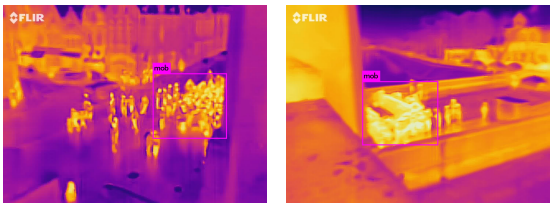


Figure 10: Model predictions on validation set.

being 90.52%. The weights from this epoch are used for further testing and validation. The mAP for the 0.5 IoU threshold of YOLOv3 on the COCO benchmark dataset is 74.8 %, comparing this to the achieved mAP for the Last Post dataset, the Last Post mAP is very high. The reason for this difference is that the validation set has a high correlation with the training set. Due to the training set and validation set being extracted from videos, all images from one video are correlated in time to each other. Images from the validation set are thus correlated to images in the training set, and the model is optimized on these types of images, explaining the high mAP. This indicates that the model is overfitting on the training data. This was confirmed when testing the model on unseen videos. Although the model could detect a mob, it produced more visual errors. Because this data was not annotated, no metrics could be extracted. Figure 10 depicts some predictions of the model on images from the validation set. The predicted bounding boxes resemble the ground truth bounding boxes quite accurately visually.

To test the speed of the predictions of the model, the total time to predict images in the validation set was measured. For the NVIDIA Geforce GTX 980 GPU the average prediction time for one image is 14.673 milliseconds, with a standard deviation of 0.517 milliseconds. This indicates that the upper limit of the frame rate when making predictions on a video is approximately 68 frames per second on the GPU. For comparison, predictions with the model were also made on a CPU, a 2.6 GHz Intel Core i5-2540 processor with AVX instructions speedup. The average prediction time on the CPU is 5.849 seconds with a standard deviation of 0.438 seconds, resulting in an upper limit for the frame rate on the CPU of 0.171 frames per second. Clearly real time object detection with this model is only possible on a GPU. When generating predictions on a test video the average frame rate of the video was 55 frames per second.

7 CONCLUSION AND FUTURE WORK

In this paper a modifiable drone thermal imaging analysis framework is proposed to allow end-users to build flexible video processing pipelines using different sources and analysis modules. The framework implements a microservice container plugin architecture. Users can rapidly build image processing applications by interconnecting input, processing and output plugins. The framework is evaluated by means of a proof-of-concept implementation, which is tested on the initial requirements. The proposed framework achieves the modifiability and interoperability requirements at the cost of performance (long shutdown time of a plugin). The framework is applied for detecting large crowds of people (mobs) during open-air events. A new dataset containing thermal images of such mobs is presented, on which a YOLOv3 neural network is trained. The trained model is able to detect mobs on new thermal images in real-time achieving frame rates of 55 frames per second when deployed on a modern GPU. Cloud auto-scaling is an extension to the framework that will be investigated in future work.

REFERENCES

Alpaydin, E. (2014). *Introduction to machine learning*. MIT Press, 3 edition.

Bendig, J., Bolten, A., and Bareth, G. (2012). INTRODUCING A LOW-COST MINI-UAV FOR THERMAL- AND MULTISPECTRAL-IMAGING.

Christiansen, P., Steen, K. A., Jørgensen, R. N., and Karstoft, H. (2014). Automated detection and recognition of wildlife using thermal cameras. *S*ensors (Basel, Switzerland), 14(8):13778–93.

Davis, J. W. and Keck, M. A. (2005). A Two-Stage Template Approach to Person Detection in Thermal Imagery. *P*roc. Workshop on Applications of Computer Vision.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. In *C*VPR09.

Divya, J. (2017). Drone Technology and Usage: Current Uses and Future Drone Technology.

Docker (2016). Docker Hub.

Docker Inc. (2018). Docker - Build, Ship, and Run Any App, Anywhere.

Gade, R. and Moeslund, T. B. (2014). Thermal cameras and applications: a survey. *M*achine Vision and Applications, 25:245–262.

Goedeme, T. (2017). Projectresultaten VLAIO TETRA-project. Technical report, KU Leuven, Louvain.

Gonnissen, R. (2016). 3DSafeGuard-VL.

GStreamer (2018). GStreamer: open source multimedia framework.

Harvey, M. C., Rowland, J. V., and Luketina, K. M. (2016). Drone with thermal infrared camera provides high resolution georeferenced imagery of the Waikite geothermal area, New Zealand.

Hwang, S., Park, J., Kim, N., Choi, Y., and Kweon, I. S. (2015). Multispectral Pedestrian Detection: Benchmark Dataset and Baseline. *C*VPR.

Inc., A. W. S. (2018). What Is Amazon Kinesis Video Streams?

Lewis, J. and Fowler, M. (2014). Microservices Resource Guide.

Li, S. Z., Chu, R., Liao, S., and Zhang, L. (2007). Illumination Invariant Face Recognition Using Near-Infrared Images. *I*EEE Transactions on Pattern Analysis and Machine Intelligence, 29(4):627–639.

Lin, T. Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. (2014). Microsoft COCO: Common objects in context. *L*ecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 8693 LNCS(PART 5):740–755.

Lvh (2015). Don't expose the Docker socket (not even to a container).

Redmon, J. (2013–2016). Darknet: Open source neural networks in c. http://pjreddie.com/darknet/.

Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. (2015). You Only Look Once: Unified, Real-Time Object Detection.

Redmon, J. and Farhadi, A. (2018). YOLOv3: An Incremental Improvement. *a*xXiv.

Rivera, A. J., Villalobos, A. D., Monje, J. C., Mariñas, J. A., and Oppus, C. M. (2017). Post-disaster rescue facility: Human detection and geolocation using aerial drones. *I*EEE Region 10 Annual International Conference, Proceedings/TENCON, pages 384–386.

Ronacher, A. (2017). Welcome to Flask Flask Documentation (0.12).

Satzger, B., Hummer, W., Inzinger, C., Leitner, P., and Dustdar, S. (2013). Winds of change: From vendor lock-in to the meta cloud. *I*EEE Internet Computing, 17(1):69–73.

Slattery, L.-L. (2017). DroneSAR wants to turn drones into search-and-rescue heroes.

Steffen, B. and Seyfried, A. (2010). Methods for measuring pedestrian density, flow, speed and direction with minimal scatter. *P*hysica A: Statistical Mechanics and its Applications, 389(9):1902–1910.

Tijtgat, N., Ranst, W. V., Volckaert, B., Goedemé, T., and De Turck, F. (2017a). Embedded Real-Time Object Detection for a UAV Warning System. *2*017 IEEE International Conference on Computer Vision Workshops (ICCVW), pages 2110–2118.

Tijtgat, N., Volckaert, B., and De Turck, F. (2017b). Real-Time Hazard Symbol Detection and Localization Using UAV Imagery. *2*017 IEEE 86th Vehicular Technology Conference (VTC-Fall), pages 1–5.

Workswell (2016). Pipeline inspection with thermal diagnostics.

Wu, Z., Fuller, N., Theriault, D., and Betke, M. (2014). A Thermal Infrared Video Benchmark for Visual Analysis. *I*EEE Conference on Computer Vision and Pattern Recognition Workshops.

Yasrab, R. (2018). Mitigating Docker Security Issues. Technical report, University of Science and Technology of China, Hefei.