An Empirical Study on Software Aging of Long-Running Object Detection Algorithms

Roberto Pietrantuono[1,*], Domenico Cotroneo[1], Ermeson Andrade[2], Fumio Machida[3]
[1]University of Naples Federico II, Naples, Italy
[2]Federal Rural University of Pernambuco, Recife, Brazil
[3]University of Tsukuba, Tsukuba, Japan
{roberto.pietrantuono, cotroneo}@unina.it, ermeson.andrade@ufrpe.br, machida@cs.tsukuba.ac.jp
*corresponding author

***Abstract*—Efficient and effective object detection is a key problem in Computer Vision. Numerous object detection algorithms have been developed, whose aim is to achieve two conflicting goals, namely accuracy and efficiency, while being executed in real-time with high robustness. Many of these algorithms must run for an extended period of time, i.e., in video surveillance or in self-driving cars – a working condition that make them subject to the risk of *software aging*.**

In this work, we focus on evaluating several object detection algorithms to understand if and to what extent they are affected by software aging. A measurement-based aging approach was adopted, with a series of long-running tests and subsequent data analysis. The results report significant trends of performance degradation, sometimes leading to aging-related failures, as well as memory consumption trends, which turned out to be the main issue across all the experiments.

***Keywords*—Object detection, software aging, performance, computer vision**

I. Introduction

Computer vision (CV) aims to replicate the human eye in order to collect, process and analyze all media-related data, including photos and videos. It has a wide range of real-world applications, such as security and surveillance, home automation, biometric recognition, motion capture, optical character recognition, facial recognition, navigation, and etc. Some examples of typical computer vision tasks are [1]: (i) image classification is to identify what class an object belongs to; (ii) object detection is to identify and locate an object in a given image; (iii) instance segmentation is the same as object detection but at the pixel level; (iv) image captioning: to describe the image; and (v) Simultaneous Localization and Mapping (SLAM) is to construct and update an environment map while a robot/agent moves in it.

Among these popular computer vision tasks, one of the most fundamental and critical problems in CV is object detection [2]. As stated by [3], the goal of object detection is "to determine whether there are any instances of objects from given categories (such as humans, cars, bicycles, dogs or cats) in an image and, if present, to return the spatial location and extent of each object instance (e.g., via a bounding box)". In some domains, such as video surveillance, image retrieval, or Advanced Driver Assistance Systems (ADAS) [4], object detection algorithms, which are crucial technology, may be required to run for an extended period of time. One of the concerns in the use of computer vision for these domains is the software reliability issue during a longtime operation. Although object detection algorithms represent a widespread and mature technology, to the best of our knowledge, there is no work focused on long-running scenarios, despite them being adopted in many systems ranging from business-oriented to safety-critical.

In this work, we analyze the phenomenon of software aging in state-of-the-art object detection algorithms. More specifically, we performed long-running experiments to analyze how software aging manifests under different algorithms, libraries/implementations and datasets. We collected both resource consumption indicators (e.g., free/buffer/cache memory and resident memory size) and performance-related indicators (e.g., frames per second) and statistically analyze the presence or absence of aging phenomena, quantify their extent and assess the difference between various settings (i.e., algorithms, libraries, datasets).

Results highlighted that every aging indicator used in our experiments shows resource consumption or performance degradation, regardless of the algorithm, implementation or dataset. It is also showed that four out of six aging indicators, more than 50% of the experiments, manifest aging effects with a peak of over 95%. Additionally, the results revealed that the algorithm and dataset factors seem to be less important than the library one (i.e., the specific implementation).

This paper is organized as follows. Section II briefly describes the background and related work. Section III details our experiments. Section IV shows the results of the experiments and statistical analysis. Finally, Section V presents the conclusions and briefly describes future work.

II. Background and related work

A. Software aging

Software aging is a runtime phenomenon affecting a numerous number of software systems. It consists in the progressive increase of the failure rate as the system executes [5], [6], which can lead to a progressive performance degradation and eventually to crash. Software aging can be due to the accumulation of errors in the system state and/or to the progressive consumption of resources such as physical memory [7].

Studies in this area are broadly divided into model-based and measurement-based approaches. With respect to model-based approaches, the systems are commonly described through stochastic models such as stochastic Petri nets (SPN) and stochastic reward nets (SRN) [8], continuous-time Markov chains (CTMC) [9], semi-Markov processes (SMP) [10], as well as combinatorial models such as reliability block diagrams (RBD) [11] and dynamic fault trees (DFT) [12]. With respect to measurement-based approaches, non-parametric statistical methods are used for empirically characterizing software aging phenomena in software systems. In this work, more specifically, we adopt a measurement-based approach.

Over the past year, many measurement approaches have been used to identify suspicious software aging. Garg *et al.* [13] were among the first researchers who adopted the Mann-Kendall test and Sen's slope estimator to identify suspicious software aging. They monitored for more than 50 days the health of Unix workstations at the OS level using a distributed SNMP (simple network management protocol) framework, so as to identify aging trends. As a matter of fact, a visual inspection is often not enough to detect these trends. This was one of the first works that tackled the quantification of aging and time-to-failure estimates. Cotroneo *et al.* [14], [15] found software aging both in the Linux operating system and in the Java VM. Statistically significant trends in memory consumption either-or in throughput were discovered. Other techniques such as *Principal Component Analysis* (PCA) were applied to remove linear correlation between parameters and *Clustering* to isolate distinct workload states that could present different aging trends. More recenlty, Andrade *et al.* [16] investigated software aging in an image classification system that continuously runs on cloud and edge computing environments. They statistically confirmed degradation trends for cloud and edge. In addition, they revealed that performance degradation trends in cloud computing were significantly more severe than in the edge environment.

B. *Object detection algorithms*

Object detection algorithms are either machine learning-based (it is preliminary required to define features to then classify images using algorithms such as *support vector machines*, SVM), or deep learning-based (typically CNN-based, end-to-end approaches). Deep-learning ones are predominant since they provide top-class accuracy and performance when combined with powerful GPUs and a vast training dataset. Generic object detection frameworks fall into two categories [17]:

1) region proposal-based or two-stage detection which generates region proposals at first and then classifying each proposal into different object categories;
2) regression/classification-based or one-stage detection adopts a unified framework to achieve final results (categories and locations) directly.

Region-based CNN (R-CNN), *spatial pyramid pooling* (SPP-net), *Fast R-CNN*, *Faster R-CNN* [18], *region-based fully convolutional networks* (R-FCN) [19], *feature pyramid network* (FPN), and *Mask R-CNN* [20] are examples of the former group. Unified or one-stage detection frameworks include *Overfeat*, *Multibox*, *You Only Look Once* (YOLO) [21], and *Single Shot MultiBox Detector* (SSD) [22]. These algorithms are state-of-the-art in this context.

In general, region-based approaches are more computationally expensive (due to the region proposal search), especially for mobile devices, and they provide higher detection accuracy than unified ones. When considerable computational power is available and there are less strict time requirements Faster R-CNN, R-FCN, and Mask R-CNN are commonly used. However, the one-stage frameworks like SSD and YOLO are much more efficient (their detection speed is close to real-time) and less sensitive to the backbone network's quality but have lower-end performance when the objects are small. Actually, deeper backbone networks perform better but require better, hence more expensive, hardware and significantly more training data. The most known algorithms are implemtend by common libraries such as TensorFlow [23], *GluonCV* [24], and *ONNX* [25].

There are several standard benchmark datasets to compare different object detection algorithms like *PASCAL VOC 2007* or *PASCAL VOC 2012*, *Microsoft COCO*, *ImageNet Large Scale Visual Recognition* (ILSVRC), and *Open Images*. In specific contexts, such as ADAS and autonomous driving, other datasets like *KITTI* should be considered. The most popular datasets are PASCAL VOC 2007/2012, which consists of 20 categories, and MS COCO with 80 categories (20 of which are the PASCAL VOC ones). For further reading about object detection, see the comprehensive survey work in [3], [17].

Researchers in the Computer Vision area have analyzed several DNNs, including algorithms for object detection, from the performance perspective [26], [27], even on Raspberry Pi devices [28], and on mobile robotics devices [29]. These studies consider metrics that are of interest for software aging too, including power consumption, inference time, memory. However, none of them is focused on software aging, namely on what happens to performance and resource consumption metrics in the long running, which is the main goal of this work.

III. Empirical evaluation

A. *Reserach questions*

The objective of the empirical evaluation is to figure out if, and to what extent, object detection algorithms suffer from aging phenomena, and under what conditions one would observe more or less aging. We formulate the following two questions:

- **RQ1:** *Are object detection algorithms affected by software aging? In particular, does software aging affect responsiveness and resource utilization, and to what extent?*
- **RQ2:** *How does software aging vary under different algorithms, libraries/implementations, and workloads (e.g., datasets)?*

B. Factors

The algorithms considered for our analysis are: SSD, YOLO, Faster R-CNN, Mask R-CNN[1], and R-FCN. These are arguably the widely adopted algorithms both from industry and from academics [3], [17].

Numerous libraries provide the principal and already trained object detection algorithms' implementations. *Tensorflow* [23], *GluonCV* [24], and *ONNX* [25] were chosen, since they include the above-mentioned algorithms. In addition, a *Keras* [30] implementation by P. Ferrari [31] of the original SSD model was considered (hereafter named Keras_F).

As for the data to be used as test set, we rely on a widely-used one for comparing object detection algorithms, which is PASCAL VOC [32] [2]. Specifically We use two sets, PASCAL VOC 2007 test and VOC 2007 trainval.

C. Experimental Plan

The experiment consists of a series of long-running *load testing*[2], planned according to a Design of Experiment (DoE) strategy. In each experimental run, the system has been exercised with a long-running workload to increase the likelihood that software aging effects accumulate over time [34]. The duration of each test meets both these stopping criteria: run the test for at least 12 hours, and until at least 100 samples are collected. The total experimental time was about 300 hours.

We investigate the impact on software aging of three (controllable) factors, which are:

1) **Algorithm**, with five possible values (a.k.a. *levels*): SSD, YOLO, Faster R-CNN, Mask R-CNN, and R-FCN;
2) **Implementation**, with four levels: Tensorflow, GluonCV, ONNX, and Keras_F;
3) **Test dataset**, with two levels: PASCAL VOC 2007 test set (2007 test) and PASCAL VOC 2007 trainval set (2007 trainval), hereafter referred to as ”test set”.

It is worth noting that not all levels can be combined together (e.g., the YOLO algorithm is not implemented in the Tensorflow library), hence we have an unbalanced design. The factorial design resulting from all the allowed levels' combinations is shown in the Table I.

D. Response variables

The response variables to be observed in each long-running test are the aging indicators. We gather both resource consumption indicators (specifically, memory consumption indicators, as most of the aging literature [7]) and performance-related indicators, which, in the case of object detection, are the *Frame per Second* (FPS). Specifically, the collected indicators are as follows.

- Frames Per Second (FPS) count the number of images classified by the algorithms in a second. This denotes the performance in terms of inference rate.
- *Resident memory size* (RES) is the non-swapped physical memory for a given task.
- *Swap* is the non-resident portion of a task's address space, used when the amount of physical memory (RAM) is full. If the system needs more memory and the RAM is full, inactive pages in memory are moved to the swap space.
- *RealFree* memory is the sum of *free*, *buffer*, and *cache memory*. They represent, respectively, the amount of idle memory, memory used as buffers, and memory used as cache. Their sum is the memory that can be actually freed by the OS in case it is needed [15].

TABLE I: Experiment factorial design

ID	Algorithm	Library/Implem.	Test set
EXP1	SSD	Keras_F	2007 test
EXP2	SSD	Keras_F	2007 trainval
EXP3	SSD	GluonCV	2007 test
EXP4	SSD	GluonCV	2007 trainval
EXP5	SSD	Tensorflow	2007 test
EXP6	SSD	Tensorflow	2007 trainval
EXP7	SSD	ONNX	2007 test
EXP8	SSD	ONNX	2007 trainval
EXP9	YOLO	GluonCV	2007 test
EXP10	YOLO	GluonCV	2007 trainval
EXP11	YOLO	ONNX	2007 test
EXP12	YOLO	ONNX	2007 trainval
EXP13	Faster R-CNN	GluonCV	2007 test
EXP14	Faster R-CNN	GluonCV	2007 trainval
EXP15	Faster R-CNN	Tensorflow	2007 test
EXP16	Faster R-CNN	Tensorflow	2007 trainval
EXP17	Faster R-CNN	ONNX	2007 test
EXP18	Faster R-CNN	ONNX	2007 trainval
EXP19	Mask R-CNN	Tensorflow	2007 test
EXP20	Mask R-CNN	Tensorflow	2007 trainval
EXP21	Mask R-CNN	ONNX	2007 test
EXP22	Mask R-CNN	ONNX	2007 trainval
EXP23	R-FCN	Tensorflow	2007 test
EXP24	R-FCN	Tensorflow	2007 trainval

[1]To be rigorous, Mask R-CNN is an instance segmentation algorithm, but it is implemented in almost all the object detection libraries, so it was considered anyhow.

[2]Performance- , stress- and load- testing terms typically overlap each other and there is no clear taxonomy [33]. Usually, when the workload used for performance testing is the nominal or operational one, the practice is referred as load-testing, whereas, when the workload is particularly intensive or devoted to assess robustness under unspecified conditions, performance testing is also named stress-testing. Our case fits the load-testing definition better.

- *Inactive* refers to memory pages that have not been accessed "recently", and is a further indication of memory that can be used: in fact, inactive memory pages are candidates for being swapped out, either pre-emptively, before free memory pages are required, or when free pages are (expected to be imminently) needed, and is therefore related to swap usage.

The chosen aging indicators (FPS, RES, Swap, RealFree, Inactive) are the response variables to observe. We consider the slope values of the response variables over the experiment period to analyze potential aging phenomena. It should be noted that these response variables are not a single-point measurement of the chosen metric, but they are a slope estimate obtained from the observed samples along the experiment, and are therefore associated with a confidence interval [34].

E. Data collection and testbed

Experiments were run on a machine equipped with 2 3GHz CPUs, 8GB Ram, 512 SSD storage, running Ubuntu 18.04 OS. The response variable measurements were collected through the well-known `vm-stat` and `top` utilities, which provide system indicators, such as *virtual memory* (VIRT), *shared memory* (SHR), *memory used* (mem_used), among others, with a sampling period of 1 second.

The FPS was calculated over sets of 100 randomly-chosen images. Specifically, a workload generator iteratively samples 100 random images (out of thousands of images from the test datasets), and computes the FPS (i.e., how many images are processed per second) – thus the sampling period is not fixed for this indicator.

F. Statistical analysis

The statistical techniques to analyze the time series are:

- The Mann-Kendall (MK) test [35] to determine the presence of trends in the data. The Mann-Kendall hypothesis is that there is no monotonic trend in data. For this reason, if the *p-value* is lower than α, the hypothesis is rejected with a confidence greater than $(1-\alpha)$ ($\alpha = .05$ in our case);
- The Sen's procedure to compute the slope of the trend in the data [36]. It is a robust non-parametric procedure insensitive to outliers;
- A pairwise comparison between the slopes to determine which pairs of factor levels are significantly different from each other, by means of the Student's t-test for regression lines comparison [37]. We report the number of "wins" of one algorithm/library/dataset over another – a "win" means a significantly less severe slope, namely less degradation, at $\alpha = .05$. Note that less severe means bigger for FPS, *RealFree* and *Inactive* and smaller for RES and Swap.

IV. Experimental results

A. RQ1: Aging trend analysis

The first question concerns whether object detection algorithms suffer from software aging. Table II reports the

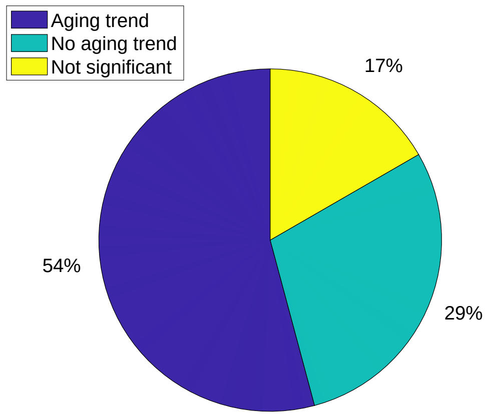


Figure 1: FPS: Proportion of experiments with aging/non-aging/non-significant trend

value of the slope as estimated by the Sen's procedure for each response variable, with the boldface text highlighting the statistically significant trends (at significance level: $\alpha = .05$). Moreover, the estimation of the depletion of each resource (or degradation of the FPS value) after 12 hours of execution is reported. For instance, in EXP10 the FPS after 12 hours is estimated to be 1.16E-01 slower than at the beginning of the experiment, and the RES memory to be depleted by 6.33E+05 KB (i.e., 633 MB). A first remark is that aging effects manifest themselves in the majority but not all the experiments (details are discussed in the next section), both as performance degradation (i.e., FPS) and as memory consumption. Looking at the mean over all the experiments, aging effects seem to be negligible in terms of FPS (with a slow down of -1.49E-02 in 12 hours) while very severe in terms of memory depletion (all the indicators tell that depletion is hundreds of MB as order of magnitude, and even more for Swap space).

1) Performance degradation: Performance is assessed by the FPS indicator. Figure 1 summarizes the number of times an aging effect is present. Overall, in 13 out of 24 experiments the slope are negative, in 4 out of 24 are non-significant, while in the remaining 7 out of 24 the slope are positive.

Although there is a trend in many cases, the extent of the degradation (i.e., the slope) is not too remarkable. However, a progressive trend can be harmful in long-running applications even if the trend is slight. For example, in EXP2 (Keras$_F$ SSD on 2007 trainval), after the first classifications, the FPS value is around 2.1396. Instead, the last observation's value is 2.1187 (after more than 900 observations in 12 hours). With the slope being equal to -2.05E-05, the final estimated FPS is 2.1211, close to the actual value. If we ideally consider the estimate after 54,000 observations (corresponding to about one month), the FPS would be 1.1067 lower, which is more than a half of the original value (clearly, the much more pronounced trends in memory consumption will likely cause aging failures very earlier). Long-running applications would thus experience problems even with small but constant trends.

TABLE II: Slope value of the aging trend for the response variables. Boldface text indicates the trend is significant (p-value <0.05). The expected depletion of the indicator after 12 hours is also reported (e.g., in EXP1, the FPS slows down by 3.13E-02 after 12h). The '/' symbol indicates the slope is estimated to be 0. For RES and SWAP, the bigger the more sever aging is; the opposite for FPS, *RealFree* and Inactive

ID	**Frame per Second (FPS)**		**RES (Kb)**		**SWAP**		**RealFree**		**Inactive**	
	Slope (FPS/sample*)	12h depl. (FPS)	Slope (KB/s)	12h depl. (KB)	Slope (KB/s)	12h depl. (KB)	Slope (KB/s)	12h depl. (KB)	Slope (KB/s)	12h depl. (KB)
EXP1	**-7.25E-05**	-6.53E-02	/	/	**1.67E+01**	7.23E+05	**7.32E-01**	3.16E+04	/	/
EXP2	**-2.05E-05**	-1.85E-02	/	/	**4.35E+00**	1.88E+05	**-1.28E+01**	-5.55E+05	/	/
EXP3	**-7.44E-05**	-6.70E-02	**-3.03E+00**	-1.31E+05	**2.20E+02**	9.48E+06	**-2.06E+00**	-8.92E+04	**-2.53E+00**	-1.09E+05
EXP4	**-6.33E-05**	-5.70E-02	**-3.57E+00**	-1.54E+05	**2.28E+02**	9.84E+06	**1.12E+00**	4.82E+04	**-2.60E+00**	-1.12E+05
EXP5	**-4.99E-05**	-4.49E-02	**-7.92E+00**	-3.42E+05	**5.88E+00**	2.54E+05	**1.09E+00**	4.70E+04	**-1.79E+01**	-7.74E+05
EXP6	**5.57E-06**	5.01E-03	**-2.58E+01**	-1.11E+06	**1.52E+01**	6.57E+05	**4.92E+00**	2.13E+05	**1.13E+00**	4.88E+04
EXP7	**8.11E-06**	7.30E-03	**1.38E+00**	5.96E+04	**3.97E-01**	1.71E+04	**-2.38E-01**	-1.03E+04	**4.10E+00**	1.77E+05
EXP8	**2.42E-05**	2.18E-02	**-2.11E+00**	-9.11E+04	**4.12E+00**	1.78E+05	**-2.20E+01**	-9.49E+05	**-2.12E+01**	-9.18E+05
EXP9	**-1.09E-04**	-9.81E-02	**-1.85E+00**	-7.97E+04	**2.84E+02**	1.23E+07	**-8.61E-01**	-3.72E+04	**-9.61E-01**	-4.15E+04
EXP10	**-1.29E-04**	-1.16E-01	**-1.46E+01**	-6.33E+05	**3.80E+02**	1.64E+07	**-4.60E-01**	-1.99E+04	**-4.23E-01**	-1.83E+04
EXP11	-2.93E-06	-2.64E-03	**1.63E-01**	7.04E+03	**2.47E+00**	1.07E+05	**1.96E+01**	8.46E+05	**-3.12E+01**	-1.35E+06
EXP12	**5.90E-05**	5.31E-02	**2.31E+00**	9.98E+04	**0.00E+00**	0.00E+00	**-4.55E+00**	-1.96E+05	**-3.59E+00**	-1.55E+05
EXP13	**-2.59E-06**	-2.33E-03	**4.23E+00**	1.83E+05	**1.12E+02**	4.83E+06	**6.39E+00**	2.76E+05	**1.49E+00**	6.43E+04
EXP14	**-2.70E-06**	-2.43E-03	**8.03E+00**	3.47E+05	**1.34E+02**	5.77E+06	**-2.84E+00**	-1.23E+05	**2.29E+00**	9.88E+04
EXP15	5.61E-06	5.05E-03	**-2.87E+00**	-1.24E+05	**4.71E+00**	2.03E+05	**2.01E-01**	8.68E+03	**9.94E+00**	4.29E+05
EXP16	**-1.63E-04**	-1.47E-01	**-6.28E+00**	-2.71E+05	**8.97E+00**	3.87E+05	**9.61E+00**	4.15E+05	**-4.10E+01**	-1.77E+06
EXP17	-3.73E-06	-3.36E-03	**9.57E+00**	4.14E+05	**5.41E+01**	2.34E+06	**-4.56E+00**	-1.97E+05	**6.72E+00**	2.90E+05
EXP18	**1.59E-04**	1.43E-01	**4.42E+02**	1.91E+07	**2.53E+02**	1.09E+07	**-1.53E+01**	-6.59E+05	**7.58E+01**	3.27E+06
EXP19	**1.27E-04**	1.14E-01	**-4.94E-01**	-2.13E+04	**7.17E+00**	3.10E+05	**6.33E+00**	2.73E+05	**-4.23E+00**	-1.83E+05
EXP20	**-4.51E-05**	-4.06E-02	**-2.67E+01**	-1.15E+06	**9.97E+00**	4.31E+05	**1.33E+01**	5.75E+05	**6.22E+01**	2.69E+06
EXP21	**2.09E-05**	1.88E-02	**1.09E+01**	4.71E+05	**4.60E+01**	1.99E+06	**1.62E+01**	6.98E+05	**5.94E+00**	2.57E+05
EXP22	**-3.54E-05**	-3.19E-02	**4.66E+01**	2.01E+06	**6.50E+01**	2.81E+06	**-6.62E+01**	-2.86E+06	**-1.21E+01**	-5.25E+05
EXP23	1.96E-06	1.76E-03	**-1.60E+01**	-6.90E+05	**3.48E-02**	1.50E+03	**-3.19E+00**	-1.38E+05	**-1.54E+00**	-6.64E+04
EXP24	**-3.58E-05**	-3.22E-02	**-1.00E+02**	-4.32E+06	**5.50E+01**	2.38E+06	**8.25E+01**	3.57E+06	**3.35E+01**	1.45E+06
Mean	*-1.66E-05*	*-1.49E-02*	*1.43E+01*	*6.16E+05*	*7.96E+01*	*3.44E+06*	*1.12E+00*	*4.85E+04*	*2.90E+00*	*1.25E+05*
St.Dev	*7.13E-05*	*6.41E-02*	*9.88E+01*	*4.27E+06*	*1.11E+02*	*4.77E+06*	*2.39E+01*	*1.03E+06*	*2.62E+01*	*1.13E+06*

**The sampling periof for FPS is not constant, as it is calculated over sets of 100 randomly-chosen images (cf. with Section III-E)*

2) *Resource consumption:* In terms of resource consumption indicators, similar considerations apply in terms of proportion of experiments with aging trends, but with considerably more pronounced slopes:

- RES: eight experiments show an increase in resident set size;
- SWAP: all the experiments but EXP12 (95.83%) exhibit a positive trend: this means that the swap space increase is a common effect;
- *RealFree* memory (i.e., Free plus Buffer plus Cache): half of the experiments (12) exhibit a reduction of the real free memory, since the slopes are negative;
- *Inactive*: half of the experiments (12) have a trend in inactive memory.

Figure 2 summarizes the percentage of experiments having a significant trend on the considered indicators, while the average slope per indicator are at the bottom of Table II.

While the swap space aging trend is consistently present across all the experiments and with remarkable trend, the memory indicators are present in about half of the experiments, with the most severe average trend being manifested as RES depletion.

3) *Trends by factor:* The contribution of each factor is presented in Table III. It reports the average slopes per factor, for all the indicators, and considering only statistically significant experiments. Boldface text highlights the most severe trends. A general consideration is that every algorithm, library, dataset is somehow affected by some aging phenomena, namely, *there is no configuration that is aging-free for all the indicators.* This can be inferred both in terms of avearge slopes (Table

TABLE III: Average slope value for each factor and response variable. Boldface text highlights the most severe trends

(a) Average trend by algorithm

	Average slope				
	FPS	RES	Swap	RealFree	Inactive
SSD	-3.03E-05	-6.84E+00	6.17E+01	-4.28E+00	-6.51E+00
YOLO	**-5.96E-05**	-4.73E+00	**2.22E+02**	3.43E+00	**-9.04E+00**
Faster R-CNN	-2.33E-06	**7.57E+01**	9.44E+01	-1.08E+00	9.20E+00
Mask R-CNN	1.69E-05	7.59E+00	3.20E+01	**-7.61E+00**	1.29E+01
R-FCN	-3.58E-05	-5.80E+01	2.75E+01	3.97E+01	1.60E+01

(b) Average trend by library

	Average slope				
	FPS	RES	Swap	RealFree	Inactive
Keras$_F$	-4.65E-05	-	1.05E+01	**-1.28E+01**	-
GluonCV	**-6.35E-05**	-1.80E+00	**2.22E+02**	2.10E-01	4.60E-01
ONNX	3.93E-05	**7.29E+01**	6.07E+01	-9.63E+00	3.04E+00
T.Flow	-2.68E-05	-2.32E+01	1.37E+01	1.64E+01	5.27E+00

(c) Average trend by dataset

	Average slope				
	FPS	RES	Swap	RealFree	Inactive
2007 test	-1.9E-05	-6.10E-01	6.28E+01	3.86E+00	**-2.74E+00**
2007 trainval	**-2.06E-05**	**2.00E+01**	**1.05E+02**	**-1.05E+00**	8.54E+00

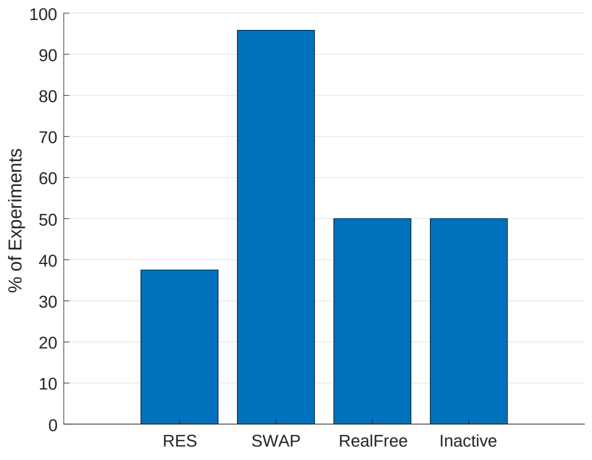


Figure 2: Percentage of aging-affected experiments

III) and as number of aging-suffering experiments (looking at the rows of Table II, there is no aging-free experiment). In the next Section, with RQ2 we detail the within-factor differences.

B. *RQ2: Aging analysis by factor*

In the previous section, software aging has been proven to exist across several algorithms, implementations, and datasets. Nevertheless, the by-factor aging trends show that there are indeed differences between the considered factors. For example, some libraries exhibit more aging effects than others. Thus, to answer RQ2, we investigate whether a specific factor has a greater impact on performance degradation or resource consumption.

1) Algorithms: Looking at the algorithms, Table III(a) highlights that, averaging over all the experiments wherein the algorithm is used, YOLO has the worst average trend of FPS, *Swap* space and *Inactive* memory, while Faster R-CNN has the worst RES average trend, and Mask R-CNN the worst free memory average trend.

Averages however can be affected by some experiments with a particularly pronounced trend (i.e., outliers). Therefore, we compare each pairs of algorithms (implemented on a same library and experimented on the same dataset) statistically by counting how many times the slope value is significantly better than another (by t-test for slope values comparison of regression lines [37]) – i.e., what we called a a *win*. When the difference is not significant, we state there is a *tie* between the two. Table IV reports the number of wins/ties/losses for each pair of algorithms.[3] Note that the total number of comparisons for a given algorithm differs from each other, as the design is unbalanced (e.g., there are 8 experiments with SSD and 4 with YOLO) and, as said in footnote 3, some pairwise comparisons cannot be done. Figure 3 summarizes the total result per algorithm, in terms of number of wins/losses/ties, and commented hereafter.

a) FPS: Given the same library and dataset, performance of YOLO turns out to be worse in 70% (7 out of 10) of the pairwise comparisons: 3 against SSD, 3 against Faster R-CNN, 1 against Mask R-CNN (Table IV). The others have comparable number of wins/losses, in approximately half of the comparisons.

b) RES: Given the same library and dataset, YOLO wins 80% of comparisons. Despite its worst performance in terms of FPS, YOLO is the algorithm that makes less use of resident set size. SSD is good also in this indicator, winning most of comparisons (62.5%), while R-FCN is again in the middle (50% of wins). The worst one in terms of RES is Faster R-CNN, with only one win and one tie out of sixteen.

c) SWAP: In terms of SWAP consumption, there is a substantial balance, with R-FCN being slighlty superior with 4 wins out of 6 at the expense of Mask-R-CNN (4 out of 12). SSD, Faster R-CNN, and YOLO have similar performances (50%, 56.25%, and 50%, respectively). While YOLO had exhibited the greatest average swap consumption, Mask-R-CNN has the worst swap consumption more often.

d) RealFree: : SSD and Faster R-CNN consume more memory more often, as they win only 37.5% of comparisons. While this is coherent with RES for Faster R-CCNN, it is not for SSD (for which RES had slightly better values). This

[3]The "*" symbol means the comparison is not possible (there is no comparison between YOLO and R-FCN algorithms with the same implementation and dataset, since R-FCN is only implemented in Tensorflow, but Tensorflow does not implement YOLO)

TABLE IV: Pairwise algorithms comparison. *#wins of row algorithm over column algorithm / total #comparisons (#number of ties)*

		YOLO	Faster R-CNN	Mask R-CNN	R-FCN
SSD	FPS	3/4	2/6	2/4 (1)	1/2
	RES	2/4 (1)	0/6	1/4	2/2
	Swap	1/4	4/6	1/4 (1)	2/2
	RealFree	1/4	3/6	1/4	1/2
	Inactive	1/4	1/6	0/4	0/2
YOLO	FPS	–	0/4 (1)	1/2	*
	RES		0/4	0/2	*
	Swap		2/4	0/2	*
	RealFree		2/4 (1)	2/2	*
	Inactive		0/4	1/2	*
Faster R-CNN	FPS	–	–	1/4	0/2 (1)
	RES			2/4 (1)	2/2
	Swap			2/4	1/2
	RealFree			1/4	1/2
	Inactive			4/4	2/2
Mask R-CNN	FPS	–	–	–	1/2 (1)
	RES				2/2
	Swap				1/2
	RealFree				1/2
	Inactive				1/2

implies that when SSD runs, the usage of cache and buffers impact negatively on the real memory free (e.g., they are less used). Mask R-CNN and R-FCN are in the middle, while YOLO, coherently with RES, wins 70% of the comparisons.

e) Inactive: Faster R-CNN wins almost every time (93.75%), coherently with the limited trend in the Swap space (in fact, inactive pages are the candidates pages to be swapped out). Mask R-CNN and R-FCN are, again, in the middle (winning half the comparisons), followed by YOLO. SSD, although the average trend is the second best one, loses most of comparisons; together with results on *RealFree*, it is likely that paging is managed differently when SSD is executed.

2) Libraries: For libraries, Table III(b) highlights that, averaging over all the experiments wherein the libraries used, Keras$_F$ has the worst *RealFree* average trend, while GluonCV has the worst average trend of FPS and Swap, and ONNX has the worst RES average trend.

Table V reports the number of wins/ties/losses for each pair of libraries. Figure 4 summarizes the total results per library.

a) FPS: Given the same algorithm and the same dataset, GluonCV and Keras$_F$ are the implementations that cause more performance degradation, respectively with only 8.33% and 16.67% of wins. Tensorflow is in the middle, while ONNX implementations give the best FPS more often (9 out of 14). This is also confirmed by the average trend in Table III.

TABLE V: Pairwise libraries comparison. *#wins of row library over column library / total #comparisons (#number of ties)*

		GluonCV	TensorFlow	ONX
Keras$_F$	FPS	1/2 (1)	0/2	0/2
	RES	1/2	0/2	2/2
	Swap	0/2	1/2	2/2
	RealFree	1/2	0/2	2/2
	Inactive	0/2	1/2	2/2
GluonCV	FPS	–	1/4	0/6 (1)
	RES		4/4	0/6
	Swap		2/4	5/6
	RealFree		2/4	4/6
	Inactive		2/4	3/6
TensowFlow	FPS	–	–	1/6 (2)
	RES			0/6
	Swap			2/6
	RealFree			5/6
	Inactive			3/6

TABLE VI: Pairwise dataset comparison. *#wins of row dataset over column dataset / total #comparisons (#number of ties)*

		2007 trainval
2007 testval	FPS	6/12 (3)
	RES	3/11 (2)
	Swap	11/13
	RealFree	6/13 (1)
	Inactive	3/11 (1)

b) RES: Tensorflow has the least RES-consuming implementations, since it wins every comparison; on the opposite side, ONNX, which had the best FPS, systematically give the worst RES consumption.

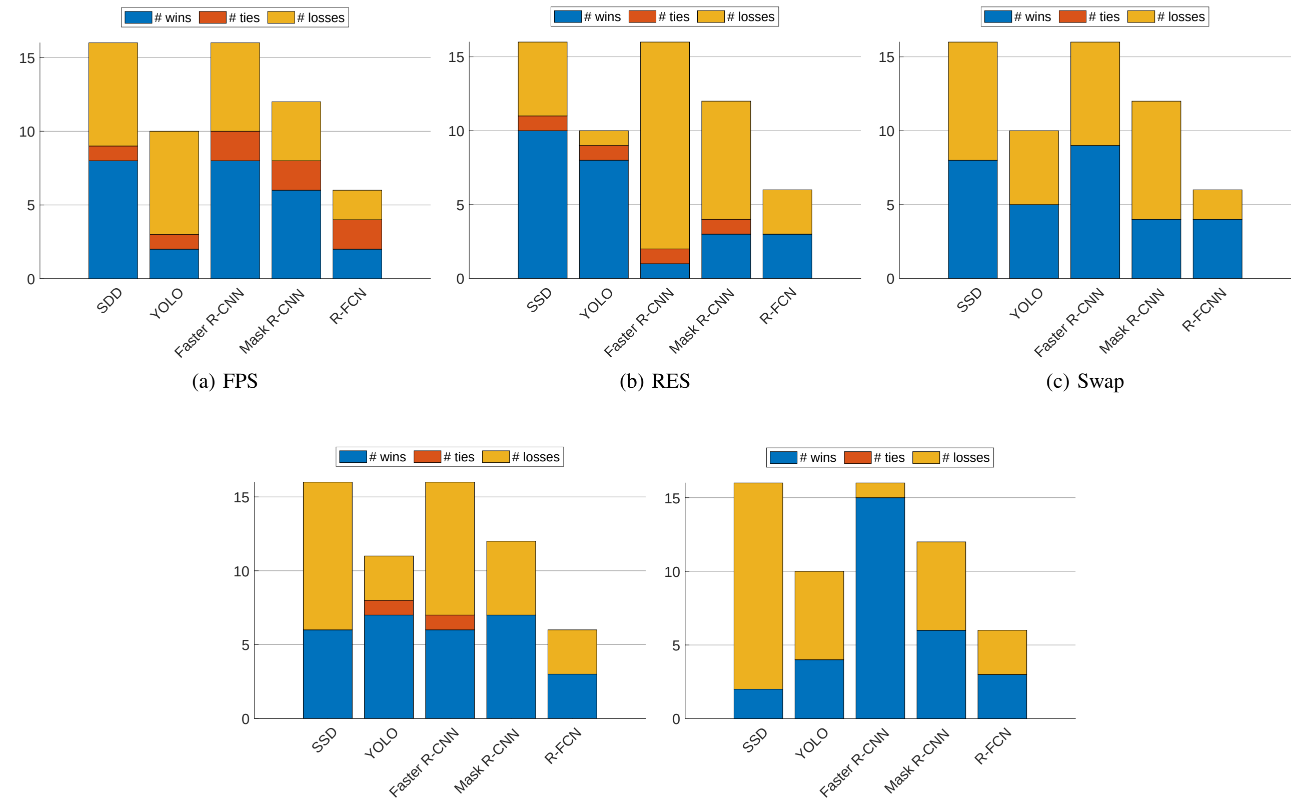
c) SWAP: GluonCV is the implementation that causes more SWAP consumption (25% of wins), followed by Keras$_F$ (50%) and Tensorflow (58.33%). The best one in terms of swap consumption is ONNX.

d) RealFree: Results confirm the RES consumption, with Tensorflow being the best one (75% of wins), followed by GluonCV (58.33%), and Keras$_F$ (50%), and ONX (21.42%).

e) Inactive: There is no difference among the libraries in the Inactive pages trends, all the other factors being equal.

3) Datasets: Table VI reports the number of wins/ties/losses, given the same algorithm and library, between two datasets. Figure 5 summarizes the total result per dataset.

The use of Pascal VOC 2007 test entails better performance for the algorithms in terms of FPS and, especially, of swap space consumption, while a slightly worse performance in terms of memory consumption compared to using Pascal VOC 2007 training.



(a) FPS (b) RES (c) Swap

(d) RealFree (e) Inactive

Figure 3: Summary of wins, ties and losses, for the compared algorithms, per indicator

4) Discussion: Tables VII to IX summarize the results of by-factor aging analysis. For each aging indicator and factor, a rank is reported. The first item in the ranking is the least consuming/degrading one.

A first remark is that in no case there is one level (algorithm/library/dataset) systematically better than others for all the indicators. On the other hand, clear patterns allow distinguishing in what aspects a level is better than another. For instance, in the case of algorithms (Table VII) Faster R-CNN has top-rated FPS trends, a limited SWAP usage increase (corroborated by a smaller inactive pages trend) but a more severe memory-related trends (*RES* and *RealFree*), while YOLO behaves oppositely. SSD has good FPS, SWAP and RES consumption, but it likely causes a worse usage of caching/buffering (as witnessed by Inactive and *RealFree*, which depend on caching/buffering). One more consideration is about the importance of the indicator. Clearly, FPS is the indicator directly perceived by end users, and is very important; on the other hand, FPS average trends are less pronounced than memory-related trends and, in a very long running application, the latter could cause more severe problems even before an FPS degradation is noticed by end users. The choice of the algorithm would then depend on how often the system is rejuvenated, on the FPS requirements and on the available hardware resources.

As for the implementations, the libraries differ in the various indicators but with less heterogeneously: ONNX and Tensorflow exhibit slower aging effects in all the indicators (except ONX for *RealFree*), while GluonCV and Keras$_F$ have more severe trends. GluonCV has good RES and *RealFree* trends, but it experienced very severe trends in the swap usage (the average if 2.22E+02, Table III).

As for the dataset, it even produces a difference, contrarily to the expectation, as there is a smaller RES and Inactive aging trends when using 2007 trainval and a bigger trend for FPS, SWAP and *RealFree*.

As different images were not expected to produce a different ranking in the indicators, we ran a further experiment (EXP25) to investigate the impact of the classified images. In this experiment, we replicate the configuration of EXP1, but use a new dataset built as follows: we first merged the two datasets, then we classified all the images four times (to account for randomness) and measure the classification (i.e., inference) time. For each repetition, we get the 20 images that required longer classification time. Finally, we build a unique list including the images, if any, present in the top-20 list of every repetition, then the images present in all but one repetition, and so forth. The output is a list of 60 images, ordered by how many times they were present in the repetitions' lists.

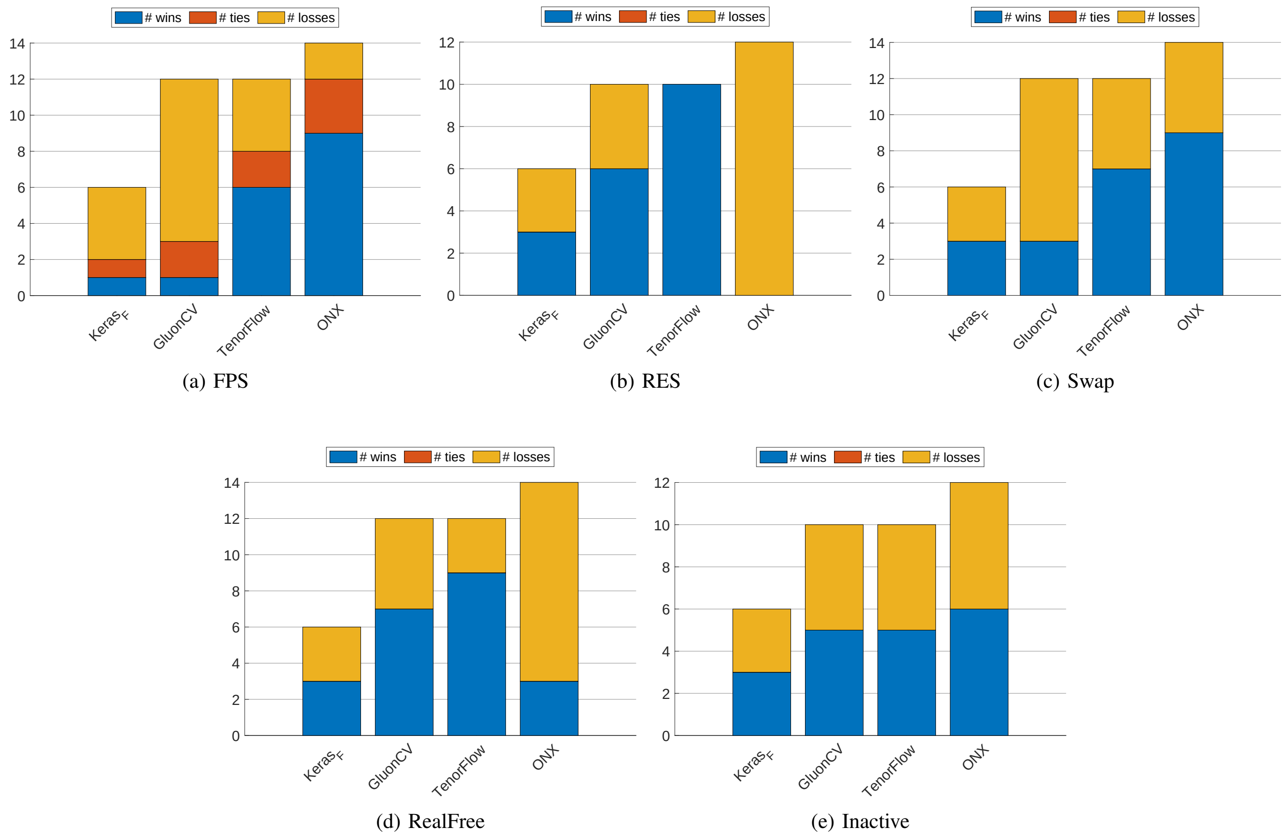


Figure 4: Summary of wins, ties and losses, for the compared libraries, per indicator

TABLE VII: Summary of comparison between algorithms (ranking from the least to most aging-suffering)

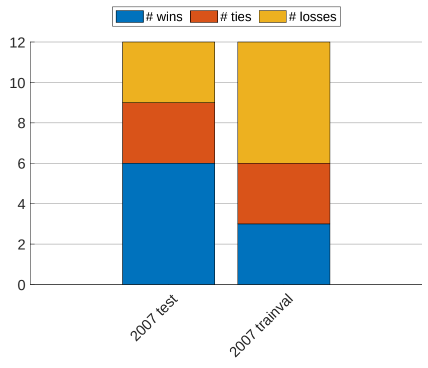
FPS	SWAP	RES	RealFree	Inactive
1. Faster R-CNN	1. R-FCN	1. YOLO	1. YOLO	1. Faster R-CNN
2. SSD	2. Faster R-CNN	2. SSD	2. Mask R-CNN	2. Mask R-CNN
3. Mask R-CNN	3. SSD	3. R-FCN	3. R-FCN	3. R-FCN
4. R-FCN	4. YOLO	4. Mask R-CNN	4. Faster R-CNN	4. YOLO
5. YOLO	5. Mask R-CNN	5. Faster R-CNN	5. SSD	5. SSD

TABLE VIII: Summary of comparison between libraries (ranking from the least to most aging-suffering)

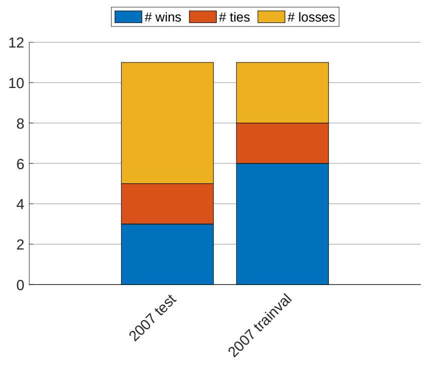
FPS	SWAP	RES	RealFree	Inactive
1. ONNX	1. ONNX	1. Tensorflow	1. Tensorflow	1. ONNX
2. Tensorflow	2. Tensorflow	2. GluonCV	2. GluonCV	2. Tensorflow
3. Keras$_F$	3. Keras$_F$	3. ONNX	3. Keras$_F$	3. GluonCV
4. GluonCV	4. GluonCV	4. Keras$_F$	4. ONNX	4. Keras$_F$

With this dataset, we ran the new long-running experiment (EXP25), whose results are shown below.

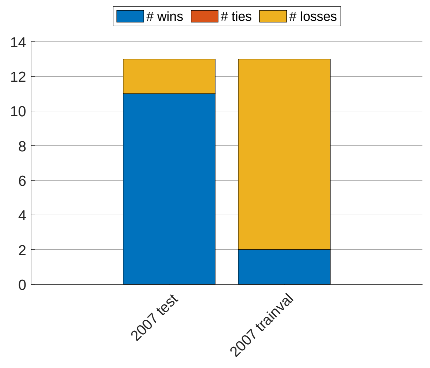
Table X shows a comparison between EXP1 and EXP25. EXP25 is confirmed to exhibit a greater aging trend in terms of FPS; it also has a more severe trend in terms of memory consumption (as per the *RealFree* indicator), while this does not reflect on a more severe swap consumption. The type of



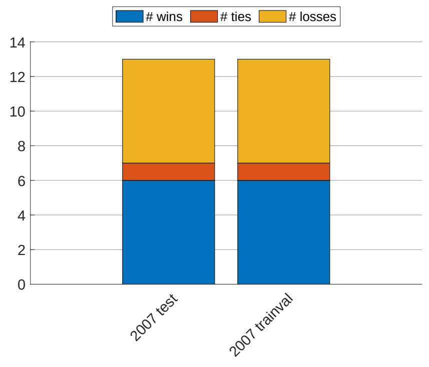
(a) FPS



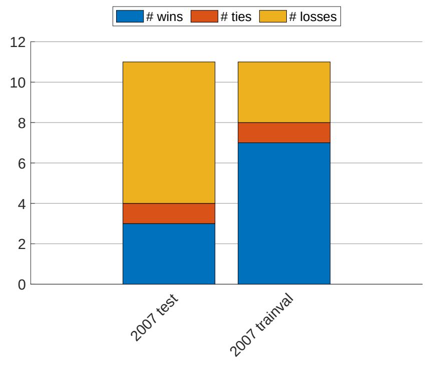
(b) RES



(c) Swap



(d) RealFree



(e) Inactive

Figure 5: Summary of wins, ties and losses, for the compared datasets, per indicator

TABLE IX: Summary of comparison between datasets (ranking from the least to most aging-suffering)

FPS	SWAP	RES	RealFree	Inactive
1. 2007 test	1. 2007 test	1. 2007 trainval	1. 2007 test	1. 2007 trainval
2. 2007 trainval	2. 2007 trainval	2. 2007 test	1. 2007 trainval	2. 2007 test

images is likely to have a systematic impact on aging trends. Some of the top-20 most time-consuming images of 2007 test are displayed in Figure 6.

The more apparent commonalities in this set of images are:

- they are less illuminated;
- there are subjects rotated or in the background;
- some objects are covering the main subjects;
- there are smaller subjects;
- in general, the subjects are in less natural positions.

A deeper investigation on the features of the images impacting more on the inference time is left to future work.

C. Threats to validity

In this work, the experimental plan covered only a small, yet representative, subset of each factor's possible levels (i.e., algorithms, libraries, dataset). The selection relied firstly on the cited surveys in the object detection area [2], [3], [17] and, secondly, on the ease of setting up the experiments and the time required to complete all of them in a factorial design. Besides the three main libraries (Tensorflow, ONNX, GluonCV, which provide several algorithms), we also added a customized implementation of the SSD algorithm in Keras. This is not at the same level of the other three libraries, as Keras depends on TensorFlow. The Keras results, which refer just to SSD in only EXP1 and EXP2, should be interpreted to judge the particular implementation of the SSD algorithm rather than aging due to the underlying library.

Results are based on a single, although long, time series for each experimental combination. Although the slope value of each of the 24 experiments were accompanied by the non-parametric 95% confidence interval, which accounts for noise in the data, the replication of each experiment would have provided indeed more accurate results. In other words, repeating a run could lead to different results, but it should be considered that a great part of the variance is already accounted for by the long duration of the experiment, and trend

TABLE X: A comparison between EXP1 and EXP25 ('/' means the slope is 0)

EXP1	CI Lower	CI Upper	Sen's slope
FPS	-8,11E-05	-6,39E-05	-7,25E-05
RES	/	/	/
SWAP	16,5293	16,9631	16,7476
RealFree	-2,42132	4,39689	0,73197
Inactive	/	/	/
EXP25	CI Lower	CI Upper	Sen's slope
FPS	-2,70 E-05	-1,91 E-05	-2,30 E-05
RES	-22,3179	-17,2162	-19,7703
SWAP	7,2477	7,9292	7,5640
RealFree	-13,1973	-9,0185	-11,0726
Inactive	15,4199	17,2834	16,5032

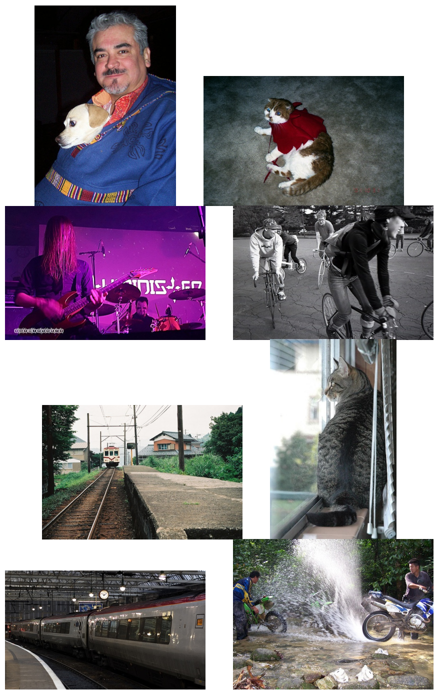


Figure 6: EXP25: The most time-consuming images

values within the slope's confidence intervals are expected with probability 0.95 when repeating a run.

Besides, the experiments' duration is inevitably limited. Although it was far enough to observe aging phenomena, it goes without saying that a longer duration would further improve the accuracy of aging phenomena assessment (e.g., a crash could have been observed).

Finally, changing the hardware configuration, the relative performance of the algorithms or libraries could change, a hypothesis that needs to be investigated with further experiments.

V. Conclusions

This paper experimentally investigated the phenomenon of software aging in deep learning object detection algorithms. It also showed how software aging manifests under different algorithms, libraries/implementations and datasets. The results clearly showed that object detection applications exhibit software aging. There is no single aging-free algorithm, implementation, or dataset. Statistical analysis confirmed that performance deteriorates in more than half of the experiments, and memory consumption increases in all aging indicators collected. It also revealed that some libraries exhibited more aging effects than others.

As future works, we plan to extend the experimental plan with new applications, algorithms, libraries, and datasets. We also plan to identify the causes of this phenomenon to better understand the cause-effect relationships (e.g., investigating the bug repositories for the implementations looking for aging-related bugs), as well as propose software rejuvenation solutions tailored for this domain.

Acknowledgment

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 871342.

We would also like to thank Dr. Fabio Lo Brutto for his contribution to this work.

References

[1] Joseph Walsh, Niall O' Mahony, Sean Campbell, Anderson Carvalho, Lenka Krpalkova, Gustavo Velasco-Hernandez, Suman Harapanahalli, and Daniel Riordan. Deep learning vs. traditional computer vision. 04 2019.

[2] Muhammad Ahmed, Khurram Azeem Hashmi, Alain Pagani, Marcus Liwicki, Didier Stricker, and Muhammad Zeshan Afzal. Survey and performance analysis of deep learning based object detection in challenging environments. *Sensors*, 21(15):5116, 2021.

[3] Li Liu, Wanli Ouyang, Xiaogang Wang, Paul Fieguth, Jie Chen, Xinwang Liu, and Matti Pietikäinen. Deep learning for generic object detection: A survey, 2018.

[4] What is object detection? 3 things you need to know. https://it.mathworks.com/discovery/object-detection.html.

[5] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: analysis, module and applications. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 381–390, 1995.

[6] Michael Grottke, Rivalino Matias Jr, and Kishor Trivedi. The fundamentals of software aging. pages 1 – 6, 12 2008.

[7] Domenico Cotroneo, Roberto Natella, Roberto Pietrantuono, and Stefano Russo. A survey of software aging and rejuvenation studies. *ACM Journal on Emerging Technologies in Computing Systems*, 10, 01 2014.

[8] Fumio Machida, Dong Seong Kim, and Kishor S. Trivedi. Modeling and analysis of software rejuvenation in a server virtualized system with live VM migration. *Performance Evaluation*, 70(3):212–230, 2013.

[9] M.T.H. Myint and T. Thein. Availability Improvement in Virtualized Multiple Servers with Software Rejuvenation and Virtualization. In *Fourth International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*, pages 156–162. IEEE, 2010.

[10] Fumio Machida, Victor F. Nicola, and Kishor S. Trivedi. Job Completion Time on a Virtualized Server with Software Rejuvenation. *ACM Journal on Emerging Technologies in Computing Systems*, 10(1):10:1–10:26, 2014.

[11] Matheus Melo, Paulo Maciel, Jean Araujo, Rubens Matos, and Carlos Araujo. Availability Study on Cloud Computing Environments: Live Migration As a Rejuvenation Mechanism. In *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2013.

[12] Jean Rahme and Haiping Xu. A Software Reliability Model for Cloud-Based Software Rejuvenation Using Dynamic Fault Trees. *International Journal of Software Engineering and Knowledge Engineering*, 25(09n10):1491–1513, 2015.

[13] S. Garg, A. van Moorsel, K. Vaidyanathan, and K. S. Trivedi. A methodology for detection and estimation of software aging. In *Proceedings Ninth International Symposium on Software Reliability Engineering (Cat. No.98TB100257)*, pages 283–292, 1998.

[14] Domenico Cotroneo, Salvatore Orlando, Roberto Pietrantuono, and Stefano Russo. A measurement-based ageing analysis of the jvm. *Software Testing Verification and Reliability*, 23, 05 2013.

[15] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo. Software aging analysis of the linux operating system. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 71–80, 2010.

[16] Ermeson Andrade, Roberto Pietrantuono, Fumio Machida, and Domenico Cotroneo. A comparative analysis of software aging in image classifiers on cloud and edge. *IEEE Transactions on Dependable and Secure Computing*, 2021.

[17] Zhong-Qiu Zhao, Peng Zheng, Shou tao Xu, and Xindong Wu. Object detection with deep learning: A review, 2018.

[18] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks, 2015.

[19] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-fcn: Object detection via region-based fully convolutional networks, 2016.

[20] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn, 2017.

[21] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2015.

[22] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. Ssd: Single shot multibox detector. *Lecture Notes in Computer Science*, page 21–37, 2016.

[23] Tensorflow: An end-to-end open source machine learning platform. https://www.tensorflow.org/.

[24] Gluoncv. https://gluon-cv.mxnet.io/.

[25] Open neural network exchange: The open standard for machine learning interoperability. https://onnx.ai/.

[26] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *CoRR*, abs/1605.07678, 2016.

[27] Simone Bianco, Rémi Cadène, Luigi Celona, and Paolo Napoletano. Benchmark analysis of representative deep neural network architectures. *CoRR*, abs/1810.00736, 2018.

[28] Delia Velasco-Montero, J. Fernández-Berni, R. Carmona-Galán, and Ángel Rodríguez-Vázquez. Performance analysis of real-time dnn inference on raspberry pi, 2018.

[29] Ricardo Pereira, Tiago Barros, Luís Garrote, Ana Lopes, and Urbano J. Nunes. An experimental study of the accuracy vs inference speed of rgb-d object recognition in mobile robotics. In *2020 29th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*, pages 588–595, 2020.

[30] Keras. simple. flexible. powerful. https://keras.io/.

[31] Pierluigi Ferrari. Ssd: Single-shot multibox detector implementation in keras. https://github.com/pierluigiferrari/ssd$_keras$.

[32] The pascal visual object classes homepage. http://host.robots.ox.ac.uk/pascal/VOC/.

[33] Zhen Ming Jiang. Automated analysis of load testing results. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, page 143–146, New York, NY, USA, 2010. Association for Computing Machinery.

[34] Domenico Cotroneo, Antonio Ken Iannillo, Roberto Natella, and Roberto Pietrantuono. A comprehensive study on software aging across android versions and vendors. *Empirical Software Engineering*, 06 2020.

[35] Henry B Mann. Nonparametric tests against trend. *Econometrica: Journal of the econometric society*, pages 245–259, 1945.

[36] Pranab Kumar Sen. Estimates of the regression coefficient based on kendall's tau. *Journal of the American statistical association*, 63(324):1379–1389, 1968.

[37] Jose Andrade and Graciela Estévez-Pérez. Statistical comparison of the slopes of two regression lines: A tutorial. *Analytica Chimica Acta*, 838, 08 2014.