DOI: 10.19279/TVZ.PD.2020-8-1-12

ENHANCING THE PERFORMANCE OF IMAGE PREPROCESSING FOR CLASSIFICATION AND OBJECT DETECTION

OPTIMIZACIJA PREDPROCESIRANJA SLIKA ZA KLASIFIKACIJU I DETEKCIJU

Ivan Cesar[1], Valentin Solina[2], Renata Kramberger[1], Tin Kramberger[1]

[1]Tehničko veleučilište u Zagrebu, Vrbik 8, 10000 Zagreb, Hrvatska
[2]Aether-signum, Zleninska 29, 10340 Vrbovec, Croatia

ABSTRACT

The image preprocessing optimization is a challenging task with numerous applications including classification and object detection on which this paper is oriented the most. Enhancing the performance in terms of processing time for image preprocessing is crucial to every researcher engaged into deep learning. A few common mistakes and practices are presented in this paper which can greatly impact training time, alongside practices and tools used for diagnosing potential pitfalls. In this paper, we evaluate several common Python libraries which are used for image preprocessing and analyze the impact of different augmentation ordering with respect to central processing unit (CPU) usage.

Keywords: *computer vision, deep learning, training optimization*

SAŽETAK

Optimizacija predprocesiranja slike je izazovan zadatak koji zahvaća u niz područja, od kojih se ovaj rad fokusira primarno na klasifikaciju i prepoznavanje objekata na slici. Ubrzavanje performansi predprocesiranja slika je od iznimne važnosti istraživačima koji se bave područjem dubinskog učenja. Kroz ovaj rad prezentirano je nekoliko pogrešaka koje mogu utjecati na vrijeme treniranja modela, s naglaskom na metode dijagnosticiranja potencijalnih zamki. Evaluirano nekoliko poznatih Python biblioteka koje se koriste za predprocesiranje slika te je analiziran utjecaj pojedine popularne augmentacije i njihovog poretka u kontekstu maksimiziranja iskoristivosti resursa središnje procesorske jedinice (CPU).

Ključne riječi: *kompjuterski vid, dubinsko učenje, optimizacija treninga*

1. UVOD
1. INTRODUCTION

Deep learning has become one of the most significant research topics in the area of computer science. With availability of various datasets and affordable hardware, deep learning becomes feasible even on computers with one or two graphical processing units (GPU) which do not necessarily have to be very powerful. However, utilizing GPU processing power for training can be a challenge in scenarios where lots of data augmentation is needed which is especially pronounced in the area of image or video analysis, regardless if it is a classification or object detection problem. There are many pretrained generic models for classification [1]–[7] and object detection [8]–[12]. However, such pretrained models often need to be adjusted for specific purpose using transfer learning [13], or trained from scratch if a model itself needs to be adjusted for the specific purpose which could happen due to performance issues with bigger models. In the process of finding the model which has the best performance with respect to a given problem, many attempts and experiments need to be conducted, either by searching the neural architecture space and discovering connections between convolutional neural network building blocks, or finetuning model hyperparameters such as learning rate and optimizer parameters [14]–[16]. Even if an advanced algorithm for neural architecture search is applied, the number of training epochs is still required to find the most optimal architecture [17], [18]. Therefore, optimizing the training process can greatly impact the amount of time needed to find the optimal convolutional neural network architecture. In this paper, we analyze some of the common pitfalls in terms of GPU utilization and present methods which can help identify training time bottlenecks, alongside the extensive experiment on several Python libraries commonly used for image preprocessing, a necessary step for successful model training.

2. MATERIALS AND METHODS
2. MATERIJALI I METODE

Building a successful artificial neural network model consists of finding the appropriate network structure and hyperparameters which produce the best result on the test dataset. As it can be seen from previous works [2], [4], [11], [18], [19], searching for the best model often consists of experimenting with various combinations of parameters which is a process that is lengthy and mostly consists of initiating the training procedure and evaluating results after several epochs or several hundreds of iterations, modifying the hyperparameters accordingly, and initiating the training again. By optimizing the training time, a larger space of parameters can be searched which can increase the likelihood of finding a more performant model.

One training iteration roughly consists of loading a batch of images or frames, applying a series of data augmentations [20]–[22] which can be referred to as preprocessing, porting images to GPU, evaluating network output and finally updating weights based on the chosen optimizer for backpropagation. Batch size directly depends on the amount of available GPU units, GPU memory on each unit, network size and input resolution. Handling randomized image loading, parallelization among several GPU units as well as backpropagation is often a task for deep learning frameworks like *PyTorch*. Regardless of the chosen framework and architecture, it is important to note that not all operations will be done on the GPU. Sometimes, CPU processing can be the bottleneck, impeding the maximum GPU utilization.

The focus of this paper is on optimizing the entire process, thus improving the GPU utilization.

Low GPU utilization during the training process can be spotted easily by checking the GPU statistics through a tool like *nvidia-smi*, combined with the watch command and lowering the refresh frequency, assuming that the training is done on a CUDA capable device using a Linux operating system. The mentioned process would not give the exact measurement but could indicate if the training is not utilizing the GPU processing power to its maximum.

More detailed information can be obtained through more sophisticated tools available in the Nvidia developer tools suite. Figure 1 shows an example of how GPU utilization can appear over time during training with utilization peaks clearly indicating low GPU utilization, leaving a spot for possible improvement in the training process.

Without digging into the details of the code, one could assume that the problem lies within image preprocessing and augmentation. However, other steps are needed to pinpoint the bottlenecks causing the low GPU utilization. This applies to machines with high-end computing power, as well as inexpensive configurations, since a high-performance GPU requires more data to be supplied for full utilization.

The *PyTorch* framework defines *Dataset* and *DataLoader* classes which provide data to the training process. By extending the *Dataset* class with a specific implementation, a developer is obliged to perform all necessary data processing in terms of loading the images or augmentations and serve the resulting image to the pipeline. In general, augmentations are performed on-the-fly, with random probability whenever a training pipeline requests the next image. Common augmentations used in classification and object detection problems are horizontal flipping, jittering – adjusting the brightness and contrast, random cropping, and rotation [1], [4], [8]. In some cases, padding the image can be useful to enable detection of smaller objects in the scene [8]. Performing the above operations is not necessarily cheap in terms of resource utilization, especially when the input image is large.

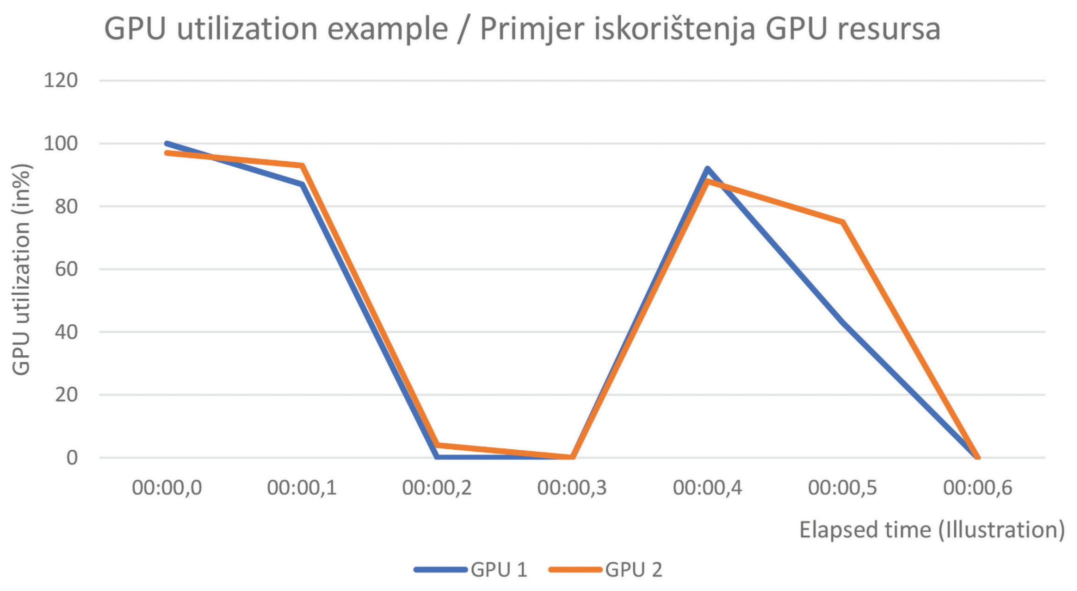

Figure 1 *Shows example GPU utilization parsed from the GPU monitoring tool. The y-axis denotes GPU utilization in percentage, while the x-axis represents time. In this example, GPU utilization is low between frames 00:00.2 and 00:00.3.*

Slika 1 *Pokazuje primjer iskorištenja GPU resursa dobivenih iz alata za promatranje GPU performansi. Vertikalna os (y) označava iskoristivost GPU resursa u postotku, dok x-os predstavlja vremensku komponentu. U gornjem primjeru, iskorištenje GPU resursa je niska između vremenskih točaka 00:00.2 i 00:00.3.*

Therefore, it is important to recognize the structure of the dataset, as well as the final augmentation size (e.g. what image size is required as a neural network input). Most of the mentioned augmentations are performed on a CPU and can slow down the entire training process if not optimized correctly. Training process optimizations, as well as any performance optimization, cannot be done thoughtlessly. It is important to correctly evaluate the performance and pinpoint the bottlenecks considering the entire process. For instance, the Tesla V100 GPU is not going to benefit the training performance if the processing power does not match it and cannot supply enough preprocessed images in time. Our experiment showed that training an SSD-300 detector with a mobilenet [6] backbone required 12 processor cores to be able to fully utilize both GPU and CPU and achieving top training performance of 160 epochs on the VOC dataset in total of 4h, measured on Google GPU cloud, with V100, 12 core processor and SSD hard disk.

Several options exist in Python in terms of common image augmentation tasks. However, the common ones are using the *Pillow* library, *OpenCV* or simply applying some of the available transformations through the commonly used *numpy* library.

We have evaluated *Pillow*, *OpenCV* and *numpy* libraries for the following augmentation tasks: resizing, cropping, rotation, jittering, and horizontal flip. Since *numpy* does not support rotation, resize, and jittering, we used the *OpenCV* library for those tasks specifically. For resize, we measured both resizing to dimensions of 300x300 and 1000x1000, therefore combined time for both resize actions were measured and presented as a separate augmentation part of the benchmark. Also, we consider ordering of the augmentations and benchmark the entire augmentation performance based on different ordering, as well as each augmentation by itself. We have used 3 images for our benchmark, one image from COCO dataset [23] (640x427px) , and two from the Open Images dataset [24] (2000x1500px and 3000x5000px). We have run the experiment for 10 iterations and noted the total execution time for each augmentation separately, and the total execution time for all augmentations applied consecutively. Images were stored on a SSD hard disk with 500MB/s reading speed and loading of images was included into the results since loading the image into memory is also a part of the model training procedure. The processor model was i7-6700HQ, 3.5Ghz, on a Windows 10 operating system. Code used for benchmarking is available on https://github.com/yohney/augs-benchmark.

3. RESULTS
3. REZULTATI

In general, the *Pillow* image library proved to be the best in terms of performance and usability, with a considerable performance improvement in terms of execution time on jittering (brightness/contrast) augmentation, and minor improvement on cropping augmentation.

Additionally, we were able to speed up preprocessing used in training Single Shot Detector (SSD) by analyzing preprocessing code with the kernprof line analyzer for Python. The experiment is focused on evaluating different augmentation implementations.

Figure 2 shows results for each augmentation performed separately, with jittering augmentation being most expensive, and cropping, flipping and rotation being least expensive in terms of CPU utilization.

Figure 3 shows the entire augmentation chain applied in order: horizontal flip, rotation, jitter, crop, and resize. Each line is increasing since at each augmentation we measured total time expired up to that augmentation, keeping the order as specified.

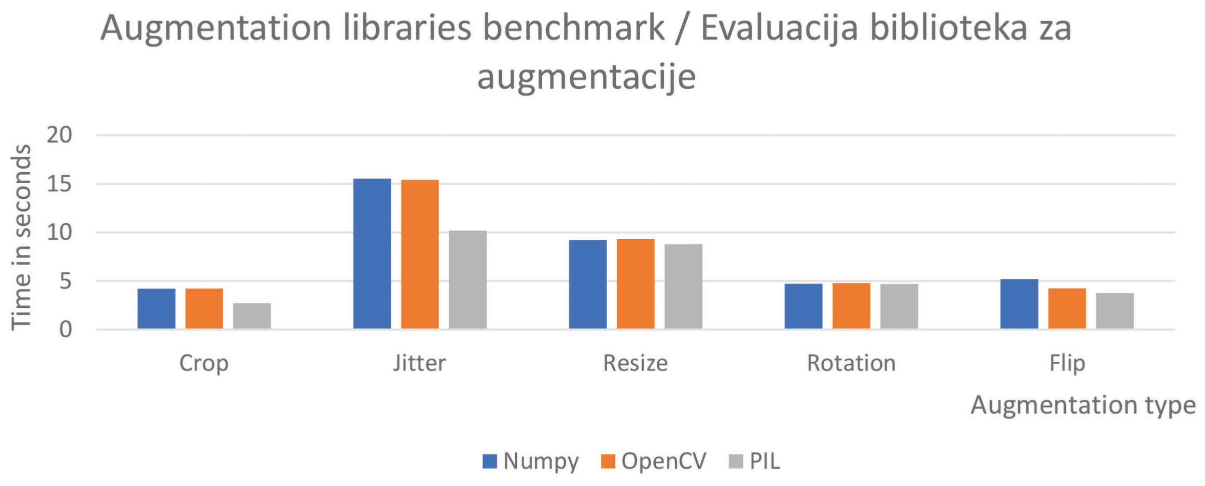

Figure 2 *Augmentation libraries benchmark results for 10 iterations (in seconds*

Slika 2 *Rezultati mjerenja augmentacijski biblioteka kroz 10 iteracija (u sekundama)*

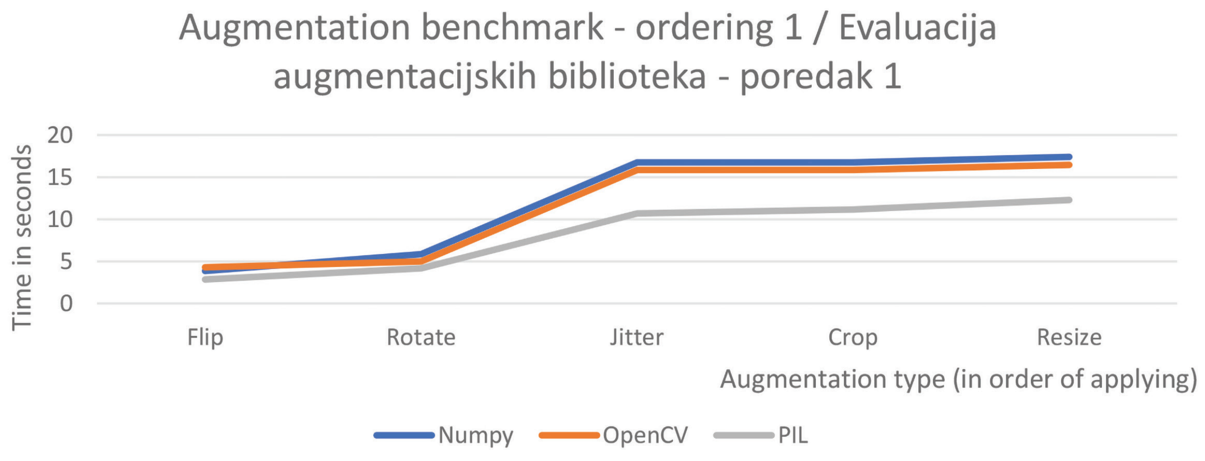

Figure 3 *Augmentation libraries benchmark results for 10 iterations (in seconds), each augmentation is applied in chain as indicated on the horizontal axis.*

Slika 3 *Rezultati evaluacije augmentacijskih biblioteka za 10 iteracija (u sekundama), gdje je svaka augmentacija lančano nadovezana na prošlu kako je prikazano na osi-x.*

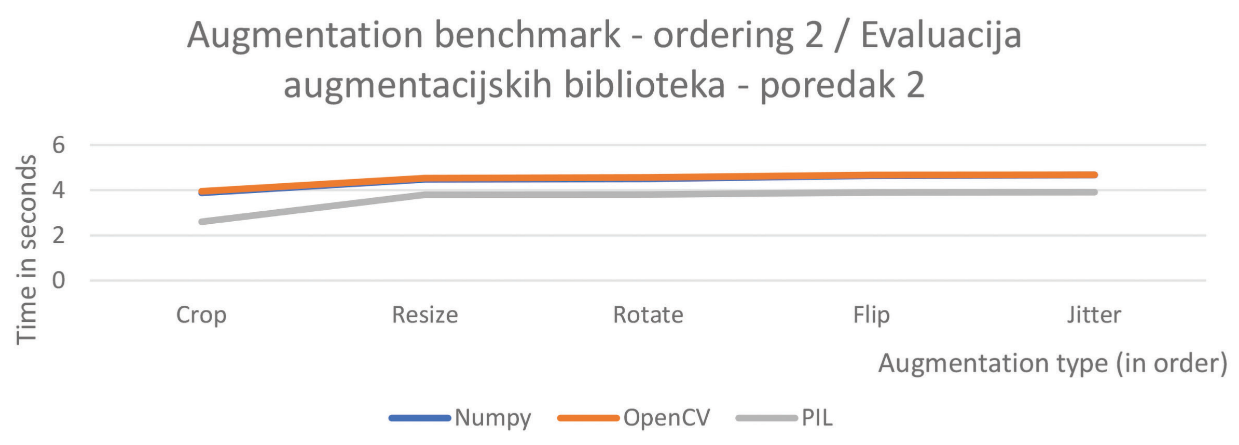

***Figure 4** Augmentation libraries benchmark results for 10 iterations (in seconds), each augmentation is applied in chain as indicated on the horizontal axis*

***Slika 4** Rezultati evaluacije augmentacijskih biblioteka za 10 iteracija (u sekundama), gdje je svaka augmentacija lančano nadovezana na prošlu kako je prikazano na osi-x.*

Figure 4 shows different augmentation ordering which yields better overall results when the dataset contains larger images, since cropping and resizing is done first and rest of the augmentations work with much smaller images.

Based on results shown in figure 4 we find that such approach can be up to 4x faster than the initial one, indicating that ordering of the augmentations can be crucial in optimizing the image preprocessing.

4. DISCUSSION
4. DISKUSIJA

The experiment and result analysis revealed domain-specific bottlenecks which improved the training speed. In some cases, it came down to using a better optimized function from another library, in some cases it was a matter of series of small optimizations which lead to better performance. We strongly recommend running such analysis on the entire preprocessing code since potentially minor fixes can greatly impact the training time and redeem already after few model trainings. Resize appears somewhat more expensive, nevertheless the cost is high due the fact that execution times of resizing to 300x300 and 1000x1000 were measured and combined since we were primarily interested in relation between benchmarked libraries and not exact times of execution. Based on the results shown in figure 3 we can note that the *Pillow* library is the fastest of the three, being up to 25% faster than *numpy* and *OpenCV*, which ranked similarly. The results for *OpenCV* are somewhat expected since *OpenCV* uses *numpy* for most of its algorithm implementations.

5. CONCLUSION
5. ZAKLJUČAK

Optimizing the preprocessing procedure seems like a marginal task compared to training and model architecture, but in the long run it ensures full utilization of resources which in turn enables experimenting with models more efficiently and therefore, increasing the probability of finding the optimal artificial neural network model. In this paper we presented some insights regarding spotting the possible bottlenecks in the training process as well as guidelines for data augmentation optimizations which proved to be essential for any high-performant deep neural network, especially in the computer vision area. Within this paper, a simple experiment was conducted comparing the Numpy, OpenCV and PIL libraries which are used in image preprocessing. The obtained research results show that different libraries for image preprocessing provide different results in terms of processing times.

6. REFERENCES

6. REFERENCE

[1.] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks." pp. 1097–1105, 2012, DOI: 10.1145/3065386.

[2.] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition." pp. 770–778, 2016, DOI: 10.1109/CVPR.2016.90.

[3.] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," Thirty-First AAAI Conf. Artif. Intell., Feb. 2017.

[4.] C. Szegedy et al., "Going Deeper With Convolutions," in CVPR, 2015, pp. 1–9, DOI: 10.1109/CVPR.2015.7298594.

[5.] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," CORR, vol. abs/1409.1, Sep. 2014.

[6.] A. G. Howard et al., "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," ArXiv, vol. abs/1704.0, Apr. 2017.

[7.] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," in 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2018, DOI: 10.1109/CVPR.2018.00474.

[8.] W. Liu et al., "SSD: Single Shot MultiBox Detector," 2016, pp. 21–37, DOI: 10.1007/978-3-319-46448-0_2.

[9.] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation," in 2014 IEEE Conference on Computer Vision and Pattern Recognition, 2014, pp. 580–587, DOI: 10.1109/CVPR.2014.81.

[10.] R. Girshick, "Fast R-CNN," in 2015 IEEE International Conference on Computer Vision (ICCV), 2015, pp. 1440–1448, DOI: 10.1109/ICCV.2015.169.

[11.] G. Ghiasi, T.-Y. Lin, R. Pang, and Q. V. Le, "NAS-FPN: Learning Scalable Feature Pyramid Architecture for Object Detection," Apr. 2019.

[12.] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollar, "Focal Loss for Dense Object Detection," in 2017 IEEE International Conference on Computer Vision (ICCV), 2017, pp. 2999–3007, DOI: 10.1109/ICCV.2017.324.

[13.] L. Torrey and J. Shavlik, "Transfer Learning," in Handbook of Research on Machine Learning Applications and Trends, IGI Global, 2010, pp. 242–264, DOI: 10.4018/978-1-60566-766-9.ch011.

[14.] J. Bergstra, D. Yamins, and D. D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in ICML'13 Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28, 2013, no. PART 1, pp. 115–123.

[15.] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for Hyper-Parameter Optimization," in 25th Annual Conference on Neural Information Processing Systems (NIPS 2011), 2011, pp. 2546–2554.

[16.] J. Bergstra and Y. Bengio, "Random Search for Hyper-Parameter Optimization," J. Mach. Learn. Res., vol. 13, no. Feb, pp. 281–305, 2012.

[17.] H. Zhu, Z. An, C. Yang, K. Xu, and Y. Xu, "EENA: Efficient Evolution of Neural Architecture," ArXiv, vol. abs/1905.0, May 2019.

[18.] T. Elsken, J. H. Metzen, and F. Hutter, "Neural Architecture Search: A Survey," J. Mach. Learn. Res., vol. 20, pp. 55:1-55:21, Aug. 2018.

[19.] N. Wang, Y. Gao, H. Chen, P. Wang, Z. Tian, and C. Shen, "NAS-FCOS: Fast Neural Architecture Search for Object Detection," ArXiv, vol. abs/1906.0, Jun. 2019.

[20.] P. Y. Simard, D. Steinkraus, and J. C. Platt, "Best practices for convolutional neural networks applied to visual document analysis," in Seventh International

Conference on Document Analysis and Recognition, 2003. Proceedings., 2003, vol. 1, pp. 958–963, DOI: 10.1109/ICDAR.2003.1227801.

[21.] D. C. Cireşan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, "High-Performance Neural Networks for Visual Object Classification," ArXiv, vol. abs/1102.0, Feb. 2011.

[22.] D. Ciresan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in 2012 IEEE Conference on Computer Vision and Pattern Recognition, 2012, pp. 3642–3649, DOI: 10.1109/CVPR.2012.6248110.

[23.] T.-Y. Lin et al., "Microsoft COCO: Common Objects in Context," Springer, Cham, 2014, pp. 740–755, DOI: 10.1007/978-3-319-10602-1_48.

[24.] A. Kuznetsova et al., "The Open Images Dataset V4: Unified image classification

AUTHORS · *AUTORI*

● **Ivan Cesar -** biograpgy can be found in the Polytechnic & Design Vol. 7, No. 2, 2019.

Correspondence · *Korespondencija*
icesar@tvz.hr

● **Valentin Solina**
Valentin earned his BCS in 2013. Since then he has been working on GPU accelerated Computer Vision applications focused on high-performance video stream processing, built an Augmented Reality framework for mobile devices, started a company and earned a sport pilot's license.

Korespondencija · *Correspondence*
valentin.solina@aether-signum.hr

● **Renata Kramberger -** biograpgy can be found in the Polytechnic & Design Vol. 5, No. 1, 2017.

Correspondence · *Korespondencija*
rkramberg@tvz.hr

● **Tin Kramberger -** biograpgy can be found in the Polytechnic & Design Vol. 5, No. 1, 2017.

Correspondence · *Korespondencija*
tkramberg@tvz.hr