

Kornia: an Open Source Differentiable Computer Vision Library for PyTorch

Edgar Riba
Computer Vision Center
OSVF-OpenCV.org
edgar.riba@gmail.com

Dmytro Mishkin
VRG, Faculty of Electrical Engineering
Czech Technical University in Prague
ducha.aiki@gmail.com

Daniel Ponsa
Computer Vision Center
daniel@cvc.uab.es

Ethan Rublee
Arraiy, Inc.
ethan.rublee@gmail.com

Gary Bradski
OSVF-OpenCV.org
garybradski@gmail.com

Abstract

This work presents Kornia – an open source computer vision library which consists of a set of differentiable routines and modules to solve generic computer vision problems. The package uses PyTorch as its main backend both for efficiency and to take advantage of the reverse-mode auto-differentiation to define and compute the gradient of complex functions. Inspired by OpenCV, Kornia is composed of a set of modules containing operators that can be inserted inside neural networks to train models to perform image transformations, camera calibration, epipolar geometry, and low level image processing techniques, such as filtering and edge detection that operate directly on high dimensional tensor representations. Examples of classical vision problems implemented using our framework are provided including a benchmark comparing to existing vision libraries.

1. Introduction

Computer vision has driven a lot of advances in modern society for different industries such as self driving cars, industrial robotics, visual effects, image search, etc resulting in a wide field of applications. One of the key components of this achievement has been due to the open-source software and the community that helped to make all this possible by providing open-source implementations of the main computer vision algorithms.

There exist several open-source libraries widely used by the computer vision community designed and optimized to process images using Central Processing Units (CPUs). However, many of the best performing computer vision algorithms are now based on deep learning, processing images in parallel using Graphical Processing Units (GPUs). Within that context, a framework that is gaining popularity is Pytorch [41] due to its reverse-mode automatic differen-

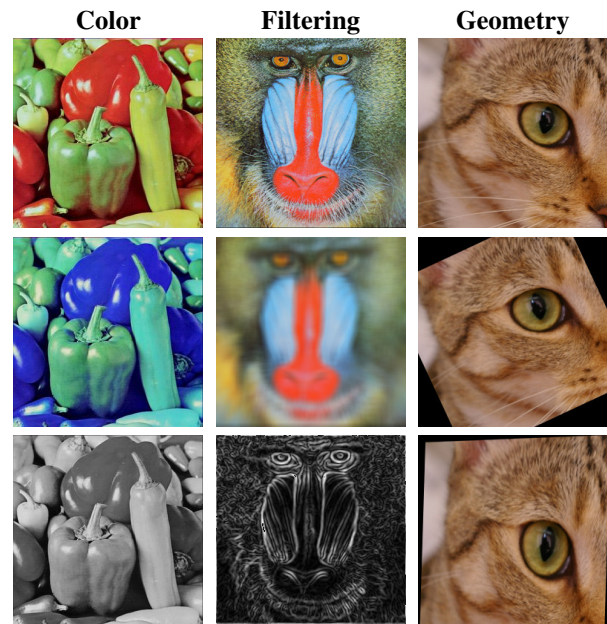


Figure 1: The library implements routines for low level image processing tasks using native *PyTorch* operators and their custom optimization. The purpose of the library is to be used for large-scale vision projects, data augmentation, or for creating computer vision layers inside of neural network layers that allow for backprograting error through them. The above results are obtained from a given batch of images using data parallelism in the GPU.

tiation mechanism, dynamic computation graph, distributed learning and eager/script execution modes. PyTorch and its ecosystem provide a few packages to work with images such as it's most popular toolkit, *torchvision*, which is mainly designed to perform data augmentation, read popular datasets and implementations of state-of-the-art models

for tasks such as detection, segmentation, image generation, and landmark detection. Yet, it lacks implementations for standard vision algorithms that can be run directly on GPUs using their native tensor data structures.

This paper introduces *Kornia*, an open source computer vision library built on top of PyTorch that will help students, researchers, companies and entrepreneurs to implement computer vision applications oriented towards deep learning. Our library, in contrast to traditional CPU-based vision frameworks, provides standard image processing functions implemented on GPUs that can also be embedded inside deep networks.

Kornia is designed to fill the gap between PyTorch and computer vision communities and it is based on some of the pre-existing open source solutions for computer vision (PIL, skimage, torchvision, tf.image), but with a strong inspiration on OpenCV [9]. *Kornia* combines the simplicity of both frameworks in order to leverage differentiable programming for computer vision taking properties from PyTorch such as differentiability, GPU acceleration, or distributed data-flows.

In addition to introducing *Kornia*, this paper contributes some demos showing how *Kornia* eases the implementation of several computer vision tasks like image registration, depth estimation or local features detection which are common in many computer vision systems.

The rest of the paper is organized as follows: we review the state of the art in terms of open source software for computer vision and machine learning in Section 2; Section 3 describes the design principles of the proposed library and all its components, and Section 4 introduces use cases that can be implemented using the library’s main features.

2. Related work

We present in this section a review of the state of the art for computer vision software. Related works will be divided in two main categories: traditional computer vision and deep learning oriented computer vision frameworks. The first with a focus on the very first libraries that implement mostly algorithms optimized for the CPU, and the second targeting solutions for GPU.

2.1. Traditional computer vision libraries

Nowadays there are many different frameworks that implement computer vision algorithms. However, during the early days of computer vision, it was difficult to find any centralized software with image processing algorithms. All the existing software for computer vision was mostly developed within universities or at small teams in companies, not shipped in any form and neither released to the public domain.

It was not until Intel released the first version of the Open Source Computer Vision Library (OpenCV). OpenCV [9]

which initially implemented computer vision algorithms for real-time ray tracing, visual interfaces and 3D display walls. All the algorithms were made available with a permissive library not only research, but also production. OpenCV changed the paradigm within the computer vision community given the fact that most of the state of art algorithms in computer vision were now put in a common framework written very efficient in C, becoming in that way a reference within the community.

The computer vision community shifted to improving or besting existing algorithms and started sharing their code with the community. This resulted in new code optimized mostly for CPU. Vedaldi et al. introduced *VLFeat* [57], an open source library that implements popular computer vision algorithms specializing in image understanding and local features extraction and matching. *VLFeat* was written in C for efficiency and compatibility, with interfaces in MATLAB. For ease of use, it supported Windows, Mac OS X, and Linux, and has been a reference e.g for efficient implementations of algorithms such as Fisher Vector [47], VLAD [24], SIFT [33], and MSER [35].

MathWorks released a proprietary Computer Vision Toolbox inside one of its famous product MATLAB [36] that covered many of the main computer vision, 3D vision, and video processing algorithms which has been used by many computer vision students and researchers becoming quite standard within the researcher community. The computer vision community have been using to MATLAB for some decades, and many still use it.

Existing frameworks like Scikit-learn [42] partially implement machine learning algorithms used by the computer vision community for classification, regression and clustering including support vector machines, random forests, gradient boosting and k-means. Similar project as Scikit-image [56] implement open source collections of algorithms for image processing.

2.2. Deep learning and computer vision

Computer vision frameworks have been optimized for CPU to fulfill realtime applications, but the recent success of deep learning in the field object classification changed the way of addressing many traditional computer vision tasks. A. Krizhevsky et al [29] took the old ideas from Yann LeCun’s Convolutional Neural Networks (CNNs) [31] paper with an architecture similar to LeNet-5 and achieved the best results by far in the ILSVRC [46] image classification task. This was a breakthrough moment for the computer vision community, and changed the way computer vision was understood. In terms of software, new frameworks such *Caffe* [25], Torch [13], MXNet [12], Chainer [55], Theano [7], MatConvNet [58], PyTorch [41], and TensorFlow [3] appeared on the scene implementing many old ideas in the GPU using parallel programming [14] as an

	CPU	GPU	Batch Processing	Differentiable	Distributed	Multi-dimensional array
torchvision[41]	✓	×	×	×	×	×
scikit-image [56]	✓	×	×	×	×	×
opencv [9]	✓	✓	×	×	×	×
tensorflow.image [3]	✓	✓	✓	✓	✓	✓
<i>Kornia</i>	✓	✓	✓	✓	✓	✓

Table 1: Comparison of different computer vision libraries by their main features. *Kornia* and tensorflow.image are the only frameworks that mostly implement their functionalities in the GPU, using batched data, differentiable and have the ability to be distributed.

approach to handle the need for large amounts of data processing in order to train deep learning models.

With the rise of deep learning, most of the standard computer vision frameworks have moved to being used more for certain geometric vision functions, data pre-processing, data augmentation on the CPU in order to be transferred later to the GPU as well as post processing to refine results. Examples of libraries that are currently used to perform pre and post-processing on the CPU within the deep learning frameworks are OpenCV or PIL.

Given that most of the deep learning frameworks still use standard vision libraries to perform the pre and post processing on CPU and similar to Tensorflow.image, as Table 1 shows, we fill the gap within the PyTorch ecosystem introducing a computer vision library that implements standard vision algorithms taking advantage of the different properties that modern frameworks for deep learning like PyTorch can provide: 1) *differentiability* for commodity avoiding to write derivative functions for complex loss functions; 2) *transparency* to perform parallel or serial computing either in CPU or GPU devices using batches in a common API; 3) *distributed* for computing large-scale applications; 4) code ready for *production*. For this reason, we present *Kornia*, a modern computer vision framework oriented for deep learning.

3. *Kornia*: Computer Vision for PyTorch.

*Kornia*¹ can be defined as a computer vision library for PyTorch, inspired by OpenCV and with strong GPU support. *Kornia* allows users to write code as they were using plain PyTorch providing high level interfaces to vision algorithms computed directly on tensors. In addition, some of the main PyTorch features are inherited by *Kornia* such as a high performance environment with easy access to automatic differentiation, executing models on different devices (CPU and GPU), parallel programming by default, communication primitives for multiprocess parallelism across several computation nodes and code ready for production. In the following, we remark these properties.

Differentiable. An image processing algorithm that can be defined as a Direct Acyclic Graph (DAG) structure can,

thanks to the reverse-mode [53] auto-differentiation [19], compute gradients via backpropagation [27]. In practice, this means that such computer vision functions are operators that can be placed as layers within the neural networks for training via backpropagating through them.

Transparent API. A key component in the library design is its easy way to seamlessly add hardware acceleration to your program with a minimum of effort. The library API is agnostic to the input source device, meaning that the algorithms can either be run in CPU or GPU.

Parallel programming. Batch processing is another important feature that enables running vision operators using data parallelism by default. The assumption for the operators is to receive as input batches of N-channel image tensors, contrary to standard vision libraries with single 1-3 channel images. Hence, for *Kornia* working with multispectral or hyperspectral images would be direct.

Distributed. Support for communication primitives for multi-process parallelism across several computation nodes running on one or more machines. The library design allows users to run their applications in different distributed systems, or even able to process large vision pipelines in an efficient way.

Production. Since its latest versions, PyTorch is able to serialize and optimize models for production purposes. Based on its just-in-time (JIT) compiler, PyTorch traces the models creating *TorchScript* programs at runtime in order to be run in a standalone C++ program using kernel fusion to do faster inference making out library a perfect fit also for built-in vision products.

3.1. Library structure

Similar to other frameworks, the library is composed of several submodules grouped by generic computer vision topics:

—**kornia.color:**— provides operators for color space conversions. The functionality found in this module covers conversions such as Grayscale, RGB, BGR, HSV, YCbCr. In addition, operators to adjust color properties such as brightness, contrast hue or saturation are also provided.

—**kornia.features:**— provides operators to detect local features, compute descriptors, and perform feature matching. The module provides differentiable versions of the

¹<https://kornia.org>

```

# load data: Bx3xHxW
img_batch = load_data_batch(...)

# send data to CUDA
if torch.cuda.is_available():
    img_batch = img_batch.cuda()

# define vision pipeline
sobel_fcn = torch.nn.Sequential(
    kornia.color.RgbToGrayscale(),
    kornia.filters.Sobel(),
)

# distribute data
sobel_fcn = torch.nn.DataParallel(
    sobel_fcn, [device_ids_list]
)

# run the pipeline: Bx1xHxW
img_sobel = sobel_fcn(img_batch)

```

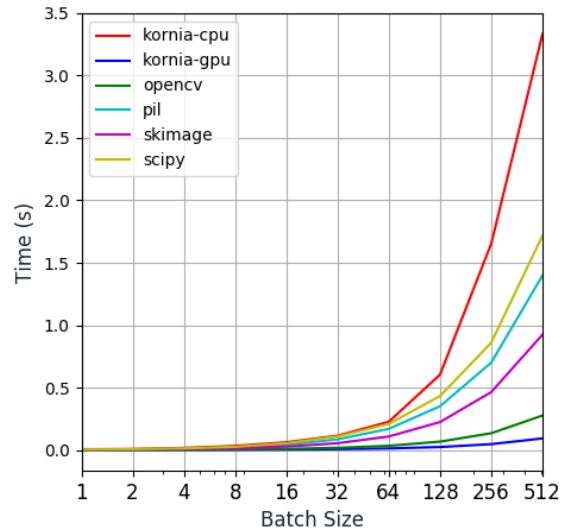


Figure 2: **Left:** Python script showing our image processing API. Notice that the API is transparent to the device, and can be easily combined with other PyTorch components. **Right:** Results of the benchmark comparing *Kornia* to other state-of-the-art vision libraries. We measure the elapsed time for computing Sobel edges (lower is better).

Harris corner detector[20], Hessian detector [6], their scale and affine covariant versions [38], DoG [33], patch dominant gradient orientation [33] and the SIFT descriptor [33]. —*kornia.features*— provides a high level API to perform detections in scale-space, where classical hard non-maxima suppression is replaced with its soft version similar to the recently proposed Multiscale Index Proposal layer (MSIP) [30]. One can seamlessly replace any or all modules with deep learned counterparts. A set of operators for work with local features geometry is also provided.

—*kornia.filters*:— provides operators to perform linear or non-linear filtering operations on tensor images. Functions to convolve tensors with kernels, for computing first and second order image derivatives, or high level differentiable implementations for blurring algorithms such as Gaussian and Box blurs, Laplace, and Sobel[26] edges detector.

—*kornia.geometry*:— module devoted to perform 2D and 3D geometry structured as follows:

- —**transforms**:— A set of operators to perform geometrical image transformations such rotation, translation, scaling, shearing, and primitives for more complex affine and homography based transforms.
- —**camera**:— A set of routines specific to different types of camera representation such as Pinhole or Orthographic models containing functionality such as projecting and unprojecting points from the camera to the world frame.

- —**conversions**:— A set of routines to perform conversions between angle representation such as radians to degrees, coordinates normalization, and homogeneous to euclidean. Moreover, we include advanced conversions for 3D geometry representations such as Quaternion, Axis-Angle, RotationMatrix, or Rodrigues formula.
- —**linalg**:— A set of routines to perform general rigid-body homogeneous transformations. We include implementations to transform points between frames and for homogeneous transformations, manipulation such as composition, inverse and to compute relative poses.
- —**warp**:— A set of primitives to sample image tensors from a reference to a non-reference frame by its related homography or the depth.

—*kornia.losses*:— A stack of loss functions to be used to solve specific vision tasks such as semantic segmentation, and image reconstruction such as the Structural Similar Index Loss (SSIM) [59].

—*kornia.contrib*:— A set of experimental operators and user contributions containing routines for splitting tensors in blocks, or to perform subpixel accuracy like the softargmax2d operator.

4. Use cases

This section presents practical examples of the library use for well known classical vision problems demonstrating

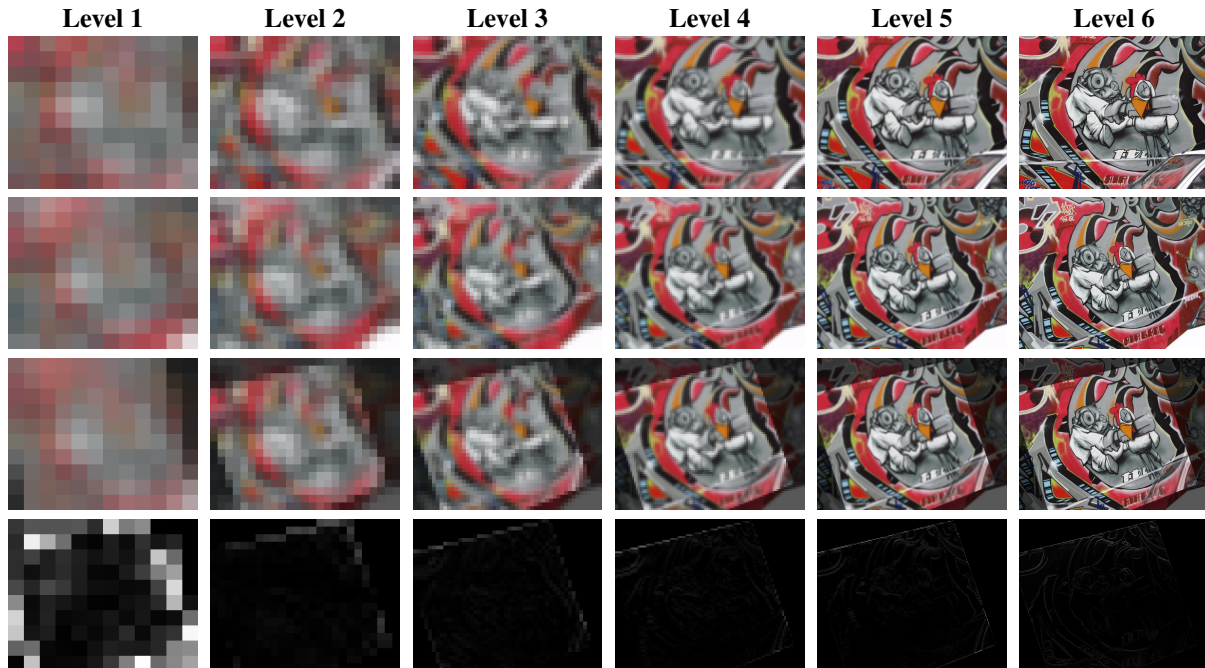


Figure 3: Results of the image registration by gradient descent. Each of the columns represent a different level of the image pyramid used to optimize the loss function. *Row 1*: the original source image; *Row 2*: the original destination image; *Row 3*: the source image warped to destination at the end of the optimization loop at that specific scale level. *Row 4*: the photometric error between the warped image using the estimated homography and the warped image using the ground truth homography. The algorithm starts to converge in the lower scales refining the solution as it goes to the upper levels of the pyramid.

its easiness for computing the derivatives of complex loss functions and releasing the user of that part. We first show quantitative and qualitative results on experiments comparing our image processing API compared to existing image processing libraries. Next, an example of image registration by its homography and a depth estimation problem showing the use of our differentiable warpers in a multi-scale fashion. Finally, we show an example making use of our differentiable local features implementations to solve a classical wide baseline stereo matching problem.

4.1. Batch image processing

In Section 2 we reviewed existing libraries implementing classical image processing algorithms optimized for practical applications such noise reduction, image enhancement, and restoration. In this example we want show the utility of our framework for similar purposes. In addition, we include a benchmark comparing our framework to other existing vision libraries showing that even though *Kornia* is not explicitly optimized for computer vision, similar results can be obtained in terms of performance.

As stated in section 3.1, *Kornia* provides implementations for low level processing e.g. color conversions, filtering and geometric image transformations that implicitly use

native PyTorch operators such as 2D convolutions and simple matrix multiplications all optimized for CPU and GPU usage. Qualitative results of our image processing API are illustrated in figure 1. Our API can be combined with other PyTorch components allowing to run vision algorithms via parallel programming, or even sending composed functions to distributed environments. In Figure 2, we provide Python code highlighting the simplicity of our API and how, with very few lines of code, we can create a composed function to compute the Sobel edges [26] of a given batch of images transparent to the device or even send the composed function to a distributed set of devices in order to build applications at large-scale, or for just simply do the data augmentation in the GPU.

Benchmark. The scope of this library is to not provide explicitly optimized code for vision, but we want to show an experiment comparing the performance of our library with respect to other existing vision libraries e.g. OpenCV [9], PIL, skimage [56] and scipy [42], see figure 2. The purpose of this experiment is to not give a detailed benchmark between frameworks, but just to have an idea of how our implementations compares to libraries that are very well optimized for computer vision. The setup of the experiment assumes as input an RGB tensor of images with a fixed res-

olution of (256x256) varying the size of the batch. In this experiment, we compute Sobel edges 500 times measuring the median elapsed time between samples. The results show that for small batches, *Kornia*'s performance is similar to those obtained using other libraries. It is worth noting that when we use a large batch size, the performance for our CPU implementation is the lowest, but when using the GPU we get the best timing performance. The machine used for this experiment was an Intel(R) Xeon(R) CPU E5-1620 v3 @ 3.50GHz and a Nvidia Geforce GTX 1080 Ti.

4.2. Image registration by Gradient Descent

In the following, we show the potential of the library for tasks requiring 2D planar geometry (for instance, marker-based camera pose estimation, spatial transformer networks, etc.). *Kornia* provides a set of differentiable operators to perform geometric image transformations such as rotations, translations, scalings, shearings, as well as affine and homography transformation. At the core of the geometry module, we have implemented an operator —*kornia.HomographyWarper*—, which warps by the homography a tensor in the reference frame A to a reference frame B that can be used to put in correspondence a set of images in a very efficient way.

Implementation. The task to solve is image registration using a multi-scale version of the Lucas-Kanade [5] strategy. Given a pair of images I_a and I_b , it optimizes the parameters of the homography H_a^b that minimizes the photometric error between I_b and the transformation of \hat{I}_b denoted as $\omega(I_a, H_a^b)$. Thanks to the Pytorch *Autograd* engine this can be implemented without explicitly computing the derivatives of the loss function from equation 1, resulting in a very compact and intuitive code.

$$\text{Loss} = \sum_{u,v}^N \|I_b - \omega(I_a, H_a^b)\|_1 \quad (1)$$

The loss function is optimized at each level of a multi-resolution pyramid, from the lower to the upper resolution levels. Figure 3 shows the original images, warped images and the error per pixel with respect to the ground truth warp at each of the scale levels. We use the Adam [28] optimizer with a learning rate of $1e - 3$, iterating 200 times at each scale level. As a side note, pixel coordinates are normalized in the range of $[-1, 1]$, meaning that there is no need to re-scale the optimized parameters between pyramid levels.

4.3. Multi-View Depth Estimation by Gradient Descent

In this example we have implemented a fully differential generic multi-view pipeline, using our framework, to allow for systematically using multi-view video data for machine

learning research and applications. For this purpose we provide the —*kornia.DepthWarper*— operator that takes an arbitrary number of calibrated camera views and warps them to a reference camera frame given the depth in the reference frame.

Multi-view reconstruction is a well understood problem with a good geometric model [21], and many approaches for matching and optimization [39, 8, 51], and some recent promising deep learning approaches [34]. We have found current machine learning approaches [16, 22] to be limiting, in that they have not been generalized to arbitrary numbers of views (spatial or temporal); and available datasets [10, 17] are only stereo and low resolution. Many of the machine learning approaches assume that there is high quality ground truth depth provided as commonly available datasets, which limits their application to new datasets or new camera configurations. Classical approaches such as planesweep, patch match or DTAM [40] have not been implemented with deep learning in mind, and do not fit easily into existing deep learning frameworks.

Implementation. We start with a simple formulation that allows us to solve for depth images using gradient descent with a variety of losses based on state of the art classical approaches (photometric, depth consistency, depth piece-wise smoothness, and multi-scale pyramids).

The multi-view reconstruction pipeline receives as input a set of views, with RGB images and calibrated intrinsic camera models, K_i , and pose estimates T_{ref}^i , and then solves for the depth image, d_{ref} , for a reference view. Since we assume a calibrated setup, the depth value of a given pixel $\mathbf{u}_{\text{ref}} = [u_{\text{ref}}, v_{\text{ref}}]$ in the reference view, d_{ref} , can be used to compute the corresponding pixel location, $\mathbf{u}_i = [u_i, v_i]$ in any of the other views through simple projective geometry $H_{\text{ref}}^i = K_i \cdot T_{\text{ref}}^i \cdot K_{\text{ref}}^{-1}$. Given this, we can warp views onto each other parameterized by depth and camera calibration using a differentiable bilinear sampling as proposed in [23], $I_{\text{ref}} = \omega(I_i, H_{\text{ref}}^i, d_{\text{ref}})$.

Similar to [39, 18, 43], depth is solved for by minimizing a photometric error between the views warped to the reference view, (equation 2 and 3). We compute an additional loss to encourage disparities to be locally smooth with a penalty on the disparity gradients weighted by image gradients as seen in equation 4. Finally, losses are combined with a weighted sum (see in equation 5). These losses are easily modified or extended, depending on how well the assumptions about these losses fit the data, e.g. it is naive and assumes photometric consistency which is only true for

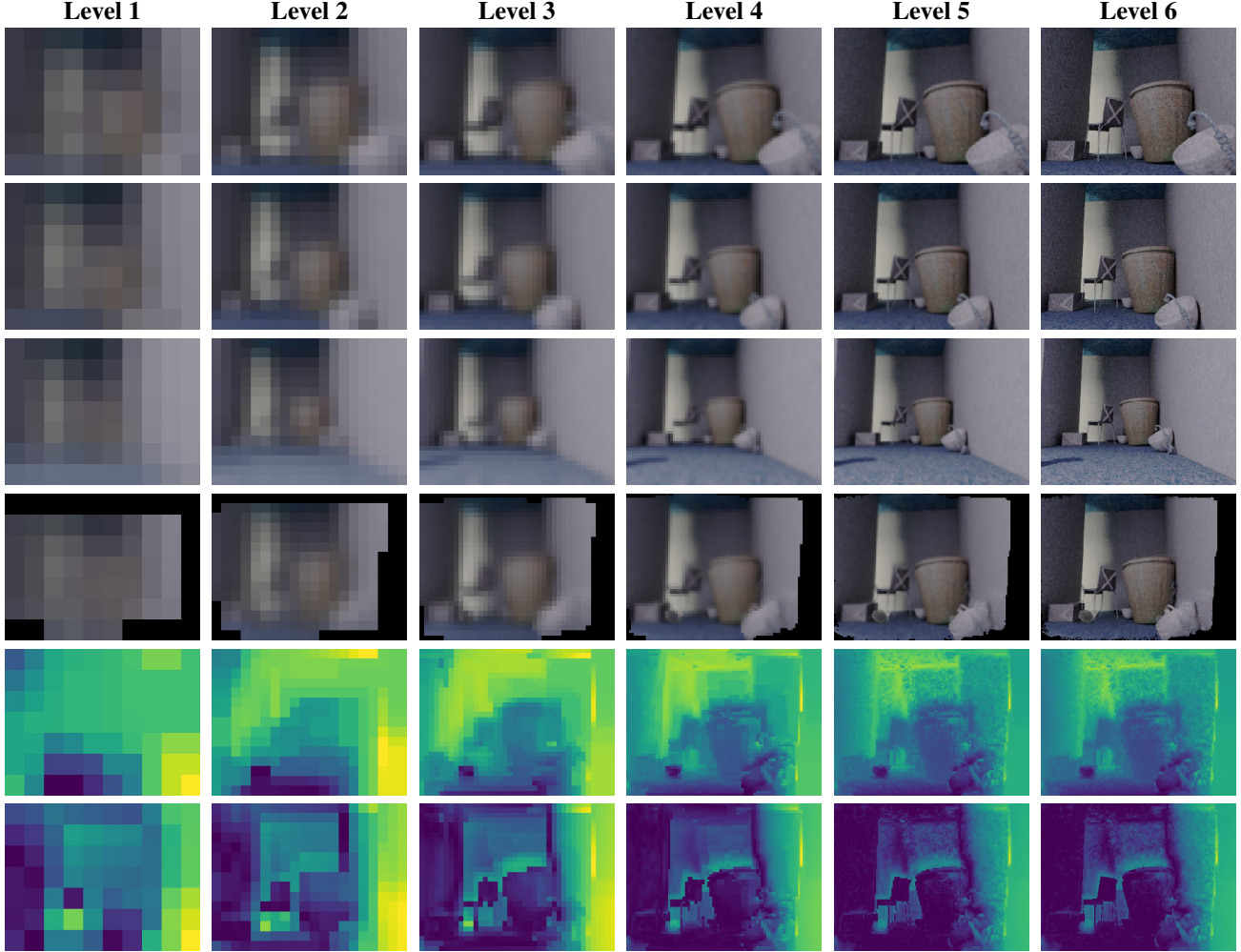


Figure 4: Results of the depth estimation by gradient descent showing the depth map produced by the given set of calibrated camera images over different scales. Each column represents a level of a multi-resolution image pyramid. *Row 1 to 3*: the source images, where the *2nd* row is the reference view; *Row 3*: the images from row 1 and 3 warped to the reference camera given the depth at that particular scale level. *Row 4 & 5*: the estimated depth map and the error per pixel compared to the ground truth depth map in the reference camera. The data used for these experiments was extracted from SceneNet RGB-D dataset [37], containing photorealistic indoor image trajectories.

small view displacements.

$$L_{\text{photo1}} = \frac{1}{n} \sum^n \frac{1 - \text{SSIM}(I_{\text{ref}}, \tilde{I}_{\text{ref}})}{2} \quad (2)$$

$$L_{\text{photo2}} = \frac{1}{n} \sum^n |I_{\text{ref}} - \tilde{I}_{\text{ref}}| \quad (3)$$

$$L_{\text{smooth}} = \frac{1}{n} \sum^n |\partial_x d| e^{-\|\partial_x I_i\|} + |\partial_y d| e^{-\|\partial_y I_i\|} \quad (4)$$

$$L_{\text{total}} = \alpha L_{\text{photo1}} + (1 - \alpha) L_{\text{photo2}} + \lambda L_{\text{smooth}} \quad (5)$$

Figure 4 shows partial results obtained by the depth algorithm implemented using *Kornia*. The algorithm receives

as input 3 calibrated views with RGB images (320x240). We used Stochastic Gradient Descent (SGD) with momentum and compute the depth at 7 different scales by blurring the image and down-sampling the resolution by a factor of 2 from the previous size. To compute the loss, we up-sample again to the original size using bilinear interpolation. The refinement at each level was done for 500 iterations starting from the lowest resolution and going up. The initial values for depth were obtained by a random uniform sampling in a range between 0 and 1.

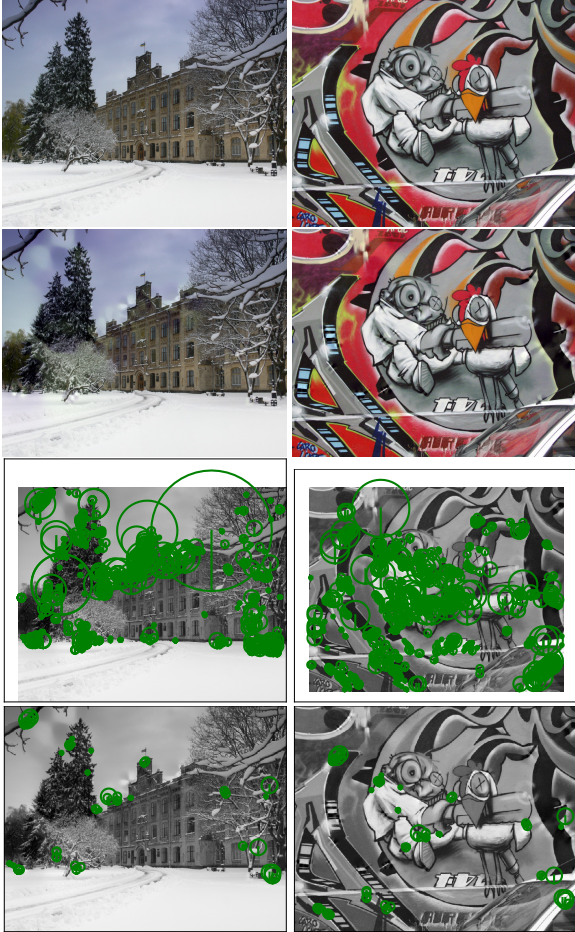


Figure 5: Targeted adversarial attack on image matching. From top to bottom: original images, which do not match; images, optimized by gradient descent to have local features that match; the result of the attack: matching features (Hessian detector + SIFT descriptor). Matching features, which survived RANSAC geometric verification

4.4. Targeted adversarial attack on two view matching with SIFT

In the following example we show how to implement fully differential wide baseline stereo matching with local feature detectors and descriptors using `—kornia.features—`. We demonstrate the differentiability by making a targeted adversarial attack on the wide baseline matching pipeline.

Local feature detectors and descriptors Local features are the workhorses of 3d reconstruction [49, 45], visual localization [48] and image retrieval [52]. Although learning-based methods now seems to dominate [15], recent benchmark top-performers still use Difference-of-Gaussians aka SIFT detector [2]. SIFT descriptor is still one of the best for 3d reconstruction [50] tasks. Thus, we believe that commu-

nity would benefit from having GPU-accelerated and differentiable version of the classical tools.

Adversarial attacks. Adversarial attacks is an area of research which recently gained popularity after the seminal work of Szegedy et al. [54] showing that small perturbations in the input image can switch the neural network prediction outcome. There are series of works showing that CNN-based solution of classification [1], segmentation [4], object detection [11], and image retrieval [32] tasks are all prone to such attacks. Yet, the authors do not know of any paper devoted to adversarial attacks on local features-based image matching. Most of attack methods are “white-box” [1], which means they require access to the model gradients w.r.t the input. This makes them an excellent choice for a `—kornia.features—` differentiability demonstration.

Implementation. The two view matching task is posed in a following way [44]: given two images I_a and I_b depicting the same scene, find the correspondences between pixels in images. If I_a and I_b do not depict the same scene, no correspondences should be returned. This is typically solved by detecting local features, describing the local patches with descriptor and then matching by minimum descriptor distance with some filtering. *Kornia* has all these parts implemented.

We consider the following adversarial attack: given the non-matching image pair I_a, I_b , and the desired homography H_a^b , modify images so that the correspondence finding algorithm will output a non-negligible number of matches consistent with the homography H_a^b . This means that both local detectors should fire in specific locations and the local patches around that location should be matchable by given function:

$$L_{\text{total}} = L_{\text{loc}} + \alpha L_{\text{desc}} + \beta L_{\text{reg}} \quad (6)$$

$$L_{\text{loc}} = \frac{1}{n} \sum^n (p_1 - Hp_2)^2 \quad (7)$$

$$L_{\text{desc}} = \frac{1}{n} \sum^n (1 + d(D_1, D_2) - d(D_1, D_{2neg})) \quad (8)$$

$$L_{\text{reg}} = \frac{1}{n} \sum^n (I - I_{\text{init}})^2 \quad (9)$$

where p_1 is keypoint detected in I_a , p_2 is closest reprojected by the H_a^b keypoint detected in image I_b , σ_1 and σ_2 are their scales, D_1 and D_2 – their descriptors, D_{2neg} – hard negative in batch, $d(\cdot, \cdot)$ – L2 distance, and I_{init} is original unmodified version of I_a and I_b .

The detector used in the example is the Hessian blob detector [6]; the descriptor is the SIFT [33]. We keep the top-2500 keypoints and use the Adam [28] optimizer with a learning rate of 0.003. Figure 5 shows the original images, optimized images and optimized images with matching features visualized. The perturbations are not quite impercep-

tible, but that it is not the goal of the current example.

5. Conclusions

We have introduced *Kornia*, a library for computer vision in PyTorch that implements traditional vision algorithms in a differentiable fashion making use of the hardware acceleration to improve the performance. We demonstrated how by using our library, classical vision problems such as image registration by homography, depth estimation, or local features matching can be very easily solved with a high performance similar to existing libraries. By leveraging this project, we believe that classical computer vision libraries can take a different role within the deep learning environments as components of layers of the networks as well as pre- and post-processing of the results. In the future, we expect researchers and companies increase the number of such contributions. At the time of submission, *Kornia* has 660 github stars and 60 forks

6. Acknowledgement

We would like to acknowledge Arraiy, Inc. to sponsor the initial stage of the project. The folks from the OSVF/OpenCV.org and the PyTorch open-source community for helpful contributions and feedback. The work of Edgar Riba and Daniel Ponsa has been partially supported by the Spanish Government under Project TIN2017-89723-P. Dmytro Mishkin is supported by CTU student grant SGS17/185/OHK3/3T/13 and by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH.

References

- [1] NeurIPS2018: Adversarial Vision Challenge, 2018.
- [2] CVPR 2019 Workshop. Image Matching: Local Features & Beyond., 2019.
- [3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.
- [4] A. Arnab, O. Miksik, and P. H. S. Torr. On the robustness of semantic segmentation models to adversarial attacks. In *CVPR*, 2018.
- [5] S. Baker and I. Matthews. Lucas-kanade 20 years on: A unifying framework. *Int. J. Comput. Vision*, 56(3):221–255, Feb. 2004.
- [6] P. R. Beaudet. Rotationally invariant image operators. In *IJCP*, pages 579–583, Kyoto, Japan, Nov. 1978.
- [7] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-farley, and Y. Bengio. Theano: A cpu and gpu math compiler in python. In *Proceedings of the 9th Python in Science Conference*, pages 3–10, 2010.
- [8] M. Bleyer, C. Rhemann, and C. Rother. Patchmatch stereo - stereo matching with slanted support windows. In *BMVC*, January 2011.
- [9] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.
- [10] D. J. Butler, J. Wulff, G. B. Stanley, and M. J. Black. A naturalistic open source movie for optical flow evaluation. In A. Fitzgibbon et al. (Eds.), editor, *ECCV*, Part IV, LNCS 7577, pages 611–625. Springer-Verlag, Oct. 2012.
- [11] S. Chen, C. Cornelius, J. Martin, and D. H. P. Chau. Shapeshifter: Robust physical adversarial attack on faster R-CNN object detector. In *ECML-PKDD*, pages 52–68, 2018.
- [12] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.
- [13] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.
- [14] S. Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.
- [15] G. Csurka and M. Humenberger. From handcrafted to deep local invariant features. *CoRR*, abs/1807.10254, 2018.
- [16] P. Fischer, A. Dosovitskiy, E. Ilg, P. Häusser, C. Hazirbas, V. Golkov, P. van der Smagt, D. Cremers, and T. Brox. Flownet: Learning optical flow with convolutional networks. *CoRR*, abs/1504.06852, 2015.
- [17] A. Geiger, P. Lenz, and R. Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *CVPR*, 2012.
- [18] C. Godard, O. Mac Aodha, and G. J. Brostow. Unsupervised monocular depth estimation with left-right consistency. In *CVPR*, 2017.
- [19] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2008.
- [20] C. Harris and M. Stephens. A combined corner and edge detector. In *In Proc. of Fourth Alvey Vision Conference*, pages 147–151, 1988.
- [21] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, New York, NY, USA, 2 edition, 2003.
- [22] E. Ilg, N. Mayer, T. Saikia, M. Keuper, A. Dosovitskiy, and T. Brox. Flownet 2.0: Evolution of optical flow estimation with deep networks. *CoRR*, abs/1612.01925, 2016.
- [23] M. Jaderberg, K. Simonyan, A. Zisserman, and k. kavukcuoglu. Spatial transformer networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *NIPS*, pages 2017–2025. 2015.

- [24] H. Jegou, M. Douze, C. Schmid, and P. Prez. Aggregating local descriptors into a compact image representation. In *CVPR*, 2010.
- [25] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia, MM '14*, pages 675–678, New York, NY, USA, 2014. ACM.
- [26] N. Kanopoulos, N. Vasanthavada, and R. L. Baker. Design of an image edge detection filter using the sobel operator. *IEEE Journal of solid-state circuits*, 23(2):358–367, 1988.
- [27] H. J. Kelley. Gradient theory of optimal flight paths. *Ars Journal*, 30(10):947–954, 1960.
- [28] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. In *ICLR*, Dec 2015.
- [29] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *NIPS*, pages 1097–1105, 2012.
- [30] A. B. Laguna, E. Riba, D. Ponsa, and K. Mikolajczyk. Key.net: Keypoint detection by handcrafted and learned CNN filters. In *ICCV*, 2019.
- [31] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.
- [32] J. Li, R. Ji, H. Liu, X. Hong, Y. Gao, and Q. Tian. Universal perturbation attack against image retrieval. In *ICCV*, 2019.
- [33] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *IJCV 2004*, 60(2):91–110, 2004.
- [34] W. Luo, A. Schwing, and R. Urtasun. Efficient deep learning for stereo matching. In *CVPR*, pages 5695–5703, 2016.
- [35] J. Matas, O. Chum, M. Urban, and T. Pajdla. Robust wide baseline stereo from maximally stable extremal regions. In *BMVC*, 2002.
- [36] MATLAB. *version 7.10.0 (R2010a)*. The MathWorks Inc., Natick, Massachusetts, 2010.
- [37] J. McCormac, A. Handa, S. Leutenegger, and A. J. Davison. Scenenet rgb-d: Can 5m synthetic images beat generic imagenet pre-training on indoor segmentation. 2017.
- [38] K. Mikolajczyk and C. Schmid. Scale & affine invariant interest point detectors. *IJCV 2004*, 60(1):63–86, 2004.
- [39] R. A. Newcombe, S. J. Lovegrove, and A. J. Davison. Dtam: Dense tracking and mapping in real-time. In *ICCV*, pages 2320–2327, 2011.
- [40] R. A. Newcombe, S. J. Lovegrove, and A. J. Davison. Dtam: Dense tracking and mapping in real-time. In *Proceedings of the 2011 International Conference on Computer Vision, ICCV '11*, pages 2320–2327, Washington, DC, USA, 2011. IEEE Computer Society.
- [41] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.
- [42] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [43] S. Pillai, R. Ambrus, and A. Gaidon. Superdepth: Self-supervised, super-resolved monocular depth estimation. *CoRR*, abs/1810.01849, 2018.
- [44] P. Pritchett and A. Zisserman. Wide baseline stereo matching. In *ICCV*, pages 754–, 1998.
- [45] A. Resindra, A. Torii, and M. Okutomi. Structure from motion using dense cnn features with keypoint relocalization. *IPSP Transactions on Computer Vision and Applications*, 10, Dec 2018.
- [46] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *IJCV*, pages 1–42, April 2015.
- [47] J. Sánchez, F. Perronnin, T. Mensink, and J. Verbeek. Image classification with the fisher vector: Theory and practice. *IJCV*, 105(3):222–245, Dec. 2013.
- [48] P.-E. Sarlin, C. Cadena, R. Siegwart, and M. Dymczyk. From coarse to fine: Robust hierarchical localization at large scale. 2019.
- [49] J. L. Schonberger and J.-M. Frahm. Structure-from-motion revisited. In *CVPR*, pages 4104–4113, 2016.
- [50] J. L. Schönberger, H. Hardmeier, T. Sattler, and M. Pollefeys. Comparative Evaluation of Hand-Crafted and Learned Local Features. In *CVPR*, 2017.
- [51] L. Sevilla-Lara, D. Sun, V. Jampani, and M. J. Black. Optical flow with semantic segmentation and localized layers. *CoRR*, abs/1603.03911, 2016.
- [52] T. Shen, Z. Luo, L. Zhou, R. Zhang, S. Zhu, T. Fang, and L. Quan. Matchable image retrieval by learning from surface reconstruction. *arXiv preprint arXiv:1811.10343*, 2018.
- [53] B. Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, January 1980.
- [54] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. In *ICLR*, 2014.
- [55] S. Tokui, R. Okuta, T. Akiba, Y. Niitani, T. Ogawa, S. Saito, S. Suzuki, K. Uenishi, B. Vogel, and H. Yamazaki Vincent. Chainer: A deep learning framework for accelerating the research cycle. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & #38; Data Mining, KDD '19*, pages 2002–2011, New York, NY, USA, 2019. ACM.
- [56] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu, and the scikit-image contributors. scikit-image: image processing in Python. *PeerJ*, 2:e453, 6 2014.
- [57] A. Vedaldi and B. Fulkerson. VLFeat: An open and portable library of computer vision algorithms. <http://www.vlfeat.org/>, 2008.
- [58] A. Vedaldi and K. Lenc. Matconvnet: Convolutional neural networks for matlab. In *ACM International Conference on Multimedia*, 2015.

- [59] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: From error visibility to structural similarity. *TIP*, 13(4):600–612, Apr. 2004.