Differentiable Data Augmentation with Kornia

[1]**Jian Shi**, [2,3]**Edgar Riba**, [4]**Dmytro Mishkin**, [3]**Francesc Moreno-Noguer**, [5]**Anguelos Nicolaou**
[1]The Chinese University of Hong Kong, [2]Universitat Autònoma de Barcelona,
[3]Institut de Robòtica i Informàtica Industrial, [4]Czech Technical University in Prague,
[5]Friedrich-Alexander University Erlangen-Nuremberg

Abstract

In this paper we present a review of the Kornia [1, 2] differentiable data augmentation (DDA) module for both for spatial (2D) and volumetric (3D) tensors. This module leverages differentiable computer vision solutions from Kornia, with an aim of integrating data augmentation (DA) pipelines and strategies to existing PyTorch components (e.g. autograd for differentiability, optim for optimization). In addition, we provide a benchmark comparing different DA frameworks and a short review for a number of approaches that make use of Kornia DDA.

1 Introduction

Data augmentation (DA) is a widely used technique to increase the variance of a dataset by applying random transformations to data examples during the training stage of a learning system. Generally, image augmentations can be divided in two groups: color space transformations that modify pixel intensity values (e.g. brightness, contrast adjustment) and geometric transformations that change the spatial locations of pixels (e.g. rotation, flipping, affine transformations). Whilst training a neural network, DA is an important ingredient for regularization that alleviates overfitting problems [3]. An inherent limitation of most current augmentation frameworks is that they mostly rely on non-differentiable functions executed outside the computation graphs.

In order to optimize the augmentation parameters (e.g. degree of rotation) by a specific objective function, differentiable data augmentation (DDA) is used. Earlier works like spatial Transformers [4] formulated spatial image transformations in a differentiable manner, allowing backpropagation through pixel coordinates by using weighted average of the pixel intensities.Recent works proposed to use DDA to improve GAN's training [5], and to optimize augmentation policies [6, 7].

In this work, we present Kornia DDA to help resarchers and professionals to quickly integrate efficient differentiable augmentation pipelines into their works. Our framework is based on Kornia [1, 2], which is an open-sourced computer vision library inspired by OpenCV [8] and designed to solve generic computer vision problems. Additionally, Kornia re-implemented classical Computer Vision algorithms from scratch in a differentiable manner and built on top of PyTorch [9] to make use of the auto-differentiation engine to compute the gradients for complex operations. This paper shows: 1) the usability-centric API design, 2) a benchmark with a couple of state of the art libraries, and 3) practical usage examples.

2 PyTorch-Oriented Design

Kornia DDA APIs are designed following PyTorch's philosophy [9], using the same logic to define neural networks as modules in order to construct complex algorithms based on differentiable building blocks. In most existing frameworks, the DA process is a non-differentiable pre-processing step outside the computation graphs. While in `kornia.augmentation`, differentiable in-graph random transformations are are incorporated into the computation graph.

Workshop on Differentiable Vision, Graphics, and Physics in Machine Learning at NeurIPS 2020.

Color Space Augmentations			
Normalize	Denormalize	ColorJitter	Grayscale
Solarize	Equalize	Sharpness	MotionBlur
MixUp	CutMix		
2D Spatial Augmentations (on 4d tensor)			
CenterCrop	Affine	ResizedCrop	Rotation
Perspective	HorizontalFlip	VerticalFlip	Crop
Erasing			
3D Volumetric Augmentations (on 5d tensor)			
CenterCrop3D	Crop3D	Perspective3D	Affine3D
HorizontalFlip3D	VerticalFlip3D	DepthicalFlip3D	

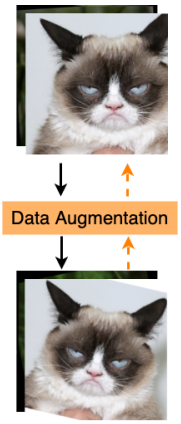

Figure 1: **Left**: Subset of supported differentiable augmentations under Kornia 0.4.1. **Right**: Our proposed scheme to represent differentiable data augmentation showing the gradients flow of the different transformations go forth and back throw the augmentations pipeline. Black arrow represents the forward pass while orange arrow represents backpropagation.

2.1 Augmentation as a Layer

Our framework introduces augmentation layers which can be seamlessly integrated with neural network layers (e.g. Conv2D, MaxPool2D). This approach offers the following advantages:

- **Automatic differentiation**. Gradients of augmentation layers could be computed whilst forward pass by taking the advantage of PyTorch autograd engine.
- **On-device computations**. DA can be moved to any computational resource available, namely CPUs, GPUs or even TPUs. Moreover, Kornia DDA is optimized for batched data processing which can be highly accelerated by GPU and TPU.
- **Higher reproducibilities**. Augmentation randomness is controlled by PyTorch random state for the reproducible DA under the same random seed. In addition, DA pipeline can be serialized along with any neural networks by simply `torch.save` and `torch.load`.

Example 1: DA Pipeline

```
import kornia.augmentation as K
class MyAugmentationPipeline(nn.Module):
    def __init__(self):
        super(MyAugmentationPipeline, self).__init__()
        self.mixup = K.RandomMixUp(p=1.)
        self.aff = K.RandomAffine(360, p=0.5)
        self.jitter = K.ColorJitter(0.2, 0.3, 0.2, 0.3, p=0.5)
        self.crp = K.RandomCrop((200, 200))
    def forward(self, input, label):
        input, label = self.mixup(input, label)
        input = self.crp(self.jitter(self.aff(input)))
        return input, label
aug = MyAugmentationPipeline()
```

On-device Computations

```
augmented = aug(images.to('cuda:0'))  # will happen in device cuda:0
augmented = aug(images.to('cuda:1'))  # will happen in device cuda:1
```

Save and Load

```
torch.save(aug, "./saved_da.pt")
aug_restored = torch.load("./saved_da.pt")
```

2.2 PyTorch-Backended Optimization

Our framework provides an easy and intuitive solution to backpropagate the gradients through augmentation layers using the native PyTorch workflow. In any augmentations, `kornia.augmentation` takes `nn.Parameter` as differentiable parameters while `torch.tensor` as static parameters. The following example shows how to optimize the differentiable parameters (including brightness, contrast, saturation) of `kornia.augmentation.ColorJitter` and backpropagate the gradients based on the computed error from a loss function.

Example 2: Optimizable DA

```
import kornia.augmentation as K; import torch; import torch.nn as nn;
t = lambda x: torch.tensor(x); p = lambda x: nn.Parameter(t(x))
torch.manual_seed(42);

images = torch.tensor(img, requires_grad=True)
jitter = K.ColorJitter(
    p([0.8, 0.8]), p([0.7, 0.7]), p([0.6, 0.6]), t([0.1, 0.1]))
out = jitter(images)

loss = nn.MSELoss()(out, images)
optimizer_img = torch.optim.SGD([images], lr=1e+5) # Large lr for demo
optimizer_param = torch.optim.SGD(jitter.parameters(), lr=0.1)

loss.backward()
optimizer_img.step()
optimizer_param.step()
```

Updated Image

From left to right: the original input, augmented image and gradient-updated image.

Updated Parameters

```
brightness -> [0.8048, 0.8363]      contrast -> [0.7030, 0.7323]
saturation -> [0.5999, 0.5976]      hue      -> [0.1000, 0.1000]
```

3 Benchmarks

Existing libraries such as TorchVision (based on PIL) and Albumentations [10] (based on OpenCV) are optimized for CPU processing taking the advantage of multi-threading. However, our framework is optimized for GPU batch processing that runs in a synchronous manner as a precedent module for neural networks. The different design choices among those libraries determined the difference in the performance under different circumstances (e.g. hardware, batch sizes, image sizes). As we state in Table 1, TorchVision/Albumentations show a better performance when lower computational resources are required (e.g. small image size, less images), while Kornia DDA gives a better performance when there is a CPU overhead. For the performance experiments, we used Intel Xeon E5-2698 v4 2.2 GHz (20-Core) and 4 Nvidia Tesla V100 GPUs. The code for the experiments will be publicly provided to compare against other hardware.

4 Use-case examples

In this section, we describe two state of the art computer vision approaches that use Kornia DDA APIs as the main backend: local feature orientation estimator and data augmentation optimization.

Num. GPUs for Data Parallelism	Comparison Among Different Image Sizes (Kornia / Albumentations / TorchVision) 32x32	224x224	512x512
1	14.28 / **12.07** / 12.33	14.23 / **12.10** / 12.48	14.22 / **12.08** / 12.77
2	15.99 / 16.47 / **14.06**	**12.85** / 12.93 / 13.91	**12.93** / 13.34 / 14.03
3	16.61 / 17.88 / **15.21**	**12.97** / 14.46 / 15.00	**13.08** / 13.96 / 15.36
4	16.87 / 18.99 / **15.66**	**13.32** / 15.38 / 15.94	**13.44** / 15.84 / 16.12

Table 1: **Speed benchmark among DA libraries.** The results are computed as the time cost (seconds) of training 1 epoch of ResNet18 using 2560 random generated faked data. Specifically, DA methods compared are RandomAffine, ColorJitter and Normalize. Batch size is 512 in all the experiments. The add-on GPU memory cost from `kornia.augmentation` is negligible.

4.1 Learning local feature orientation estimator with Kornia

An approach to learn a local feature detector is by using differentiable random spatial transformations [11], that is, spatial augmentation. The learned model has to predict the local patch geometry, which is then described by a local descriptor and the matching-related loss is minimized, as shown in Figure 2. Authors' implementation of all related function based on PyTorch [9] *grid_sample* function takes around 600 lines of code. The same functionality can be implemented with Kornia and `kornia.augmentation` using 30 lines of code including all necessary imports.

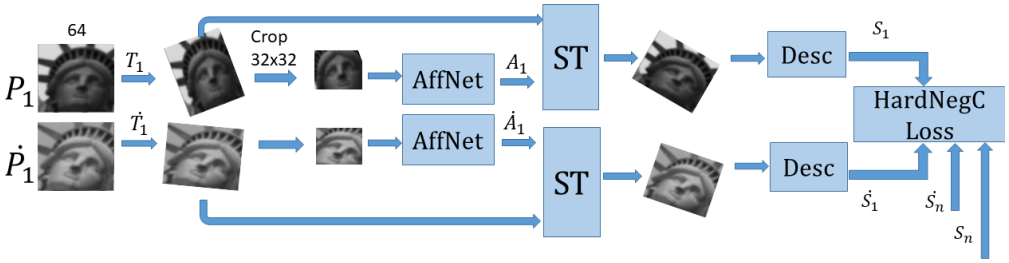

Figure 2: AffNet local feature shape and orientation estimation training training, image courtesy [11]. We re-implemented differentiable image transformations, $T_i, \dot{T}_i$, and $A_i, \dot{A}_i$ using `kornia.augmentation` functions, reducing relevant code line counts by order of magnitude.

4.2 Optimizable Data Augmentation with Kornia

Designing a proper combination of DA operations is a complicated task, which often requires the specific domain knowledge. On top of the Kornia DDA module, Faster AutoAugment [7] is implemented with an aim of learning the best augmentation policies by gradient optimization methods. Moreover, MADAO [7] (Meta Approach to Data Augmentation Optimization) is also implemented on top of Kornia which could optimize both deep learning models and DA policies simultaneously, which effectively improved the classification performance using gradient descent.

5 Discussion

We presented Kornia DDA that aligned with PyTorch API design principles with a focus on usability, to perform efficient differentiable augmentation pipelines for both production and research. In addition, we inherit the differentiability property that will help researchers to explore new DDA strategies with which we believe that can change the paradigm for designing handcrafted augmentation policies. Our future directions, will be to increase the efficiency of the different operators through the PyTorch JIT compiler and creating a generic API to perform meta-augmentation-learning.

References

[1] Edgar Riba, Dmytro Mishkin, Dani Ponsa, Rublee Ethan, and Gary Bradski. Kornia: an open source differentiable computer vision library for pytorch. In *Winter Conference on Applications of Computer Vision*, 2020.

[2] Edgar Riba, Dmytro Mishkin, Jian Shi, Dani Ponsa, Francesc Moreno-Noguer, and Gary Bradski. A survey on kornia: an open source differentiable computer vision library for pytorch. 2020.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[4] Max Jaderberg, Karen Simonyan, Andrew Zisserman, and Koray Kavukcuoglu. Spatial transformer networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2017–2025. Curran Associates, Inc., 2015.

[5] Shengyu Zhao, Zhijian Liu, Ji Lin, Jun-Yan Zhu, and Song Han. Differentiable augmentation for data-efficient gan training. *arXiv preprint arXiv:2006.10738*, 2020.

[6] Ryuichiro Hataya, Jan Zdenek, Kazuki Yoshizoe, and Hideki Nakayama. Meta approach to data augmentation optimization. *arXiv preprint arXiv:2006.07965*, 2020.

[7] Ryuichiro Hataya, Jan Zdenek, Kazuki Yoshizoe, and Hideki Nakayama. Faster autoaugment: Learning augmentation strategies using backpropagation. *arXiv preprint arXiv:1911.06987*, 2019.

[8] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[9] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[10] Alexander Buslaev, Vladimir I. Iglovikov, Eugene Khvedchenya, Alex Parinov, Mikhail Druzhinin, and Alexandr A. Kalinin. Albumentations: Fast and flexible image augmentations. *Information*, 11(2), 2020.

[11] D. Mishkin, F. Radenovic, and J. Matas. Repeatability is Not Enough: Learning Affine Regions via Discriminability. In *ECCV*, 2018.