Supplementary of "PackDet: Packed Long-Head Object Detector"

Kun Ding[1], Guojin He[1], Huxiang Gu[2,3], Zisha Zhong[2], Shiming Xiang[2], and Chunhong Pan[2]

[1] Aerospace Information Research Institute, Chinese Academy of Sciences
kding1225@gmail.com, hegj@radi.ac.cn
[2] National Laboratory of Pattern Recognition, Institute of Automation, Chinese Academy of Sciences, Beijing, China
{smxiang,chpan}@nlpr.ia.ac.cn, zisha.zhong@ia.ac.cn
[3] Beijing EvaVisdom Tech, guhuxiang@evavisdom.com

Abstract. This supplementary material provides more experimental results regarding to the proposed method, which includes: 1) experiments to benchmark PackOp under mainstream deep learning frameworks with different software versions; 2) experiments to evaluate the effectiveness of PackOp integrated into PackDet with different input sizes and feature channels; 3) comparison of the memory cost of PackDet and FCOS; 4) results of PackOp integrated with RetinaNet, an anchor-based method; 5) some visualization results. Codes will be released.[4]

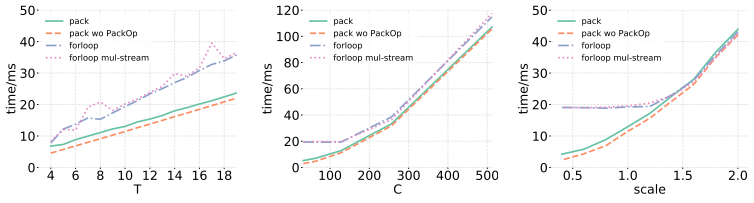
1 Benchmark PackOp

To demonstrate the effectiveness of PackOp in speeding up forwarding computation, regardless of different platforms and software versions, we additionally conduct the experiments on more platforms with different software versions. The experimental settings are identical to those in main paper.

1.1 PyTorch

We switch to the latest PyTorch version 1.5, with the CUDA 10.2 and cuDNN 7.6.5 environment. The results are shown in Fig. 1, where `pack wo PackOp` denotes the packing method excluding the PackOp time, `pack` is the same method counts this time, and `forloop` denotes the branch-by-branch method. As a quick reference, the results with PyTorch-1.1.0+CUDA-9.0+cuDNN-7.3.0 are also presented. From the figure, we could find that when using newer PyTorch version, the speedup by PackOp is even higher.

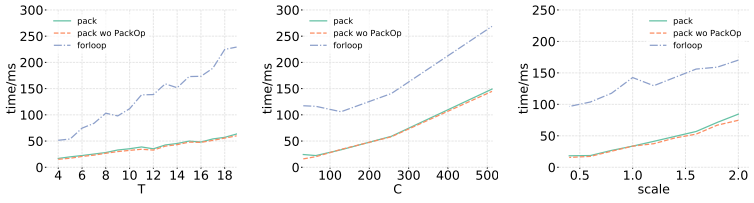
[4] https://github.com/kding1225/PackDet



(a) varying T (b) varying C (c) varying scale

(d) varying T (e) varying C (f) varying scale

Fig. 1: GPU time comparison with varying T, C and scale s on PyTorch. The scales $\{2.0, 1.8, 1.6, 1.4, 1.2, 1.0, 0.8, 0.6, 0.4\}$ are used to resize input tensors. The default value of T, C, s are $10, 128, 1.0$, respectively. Top row: PyTorch-1.1.0, bottom row: PyTorch-1.5.



(a) varying T (b) varying C (c) varying scale

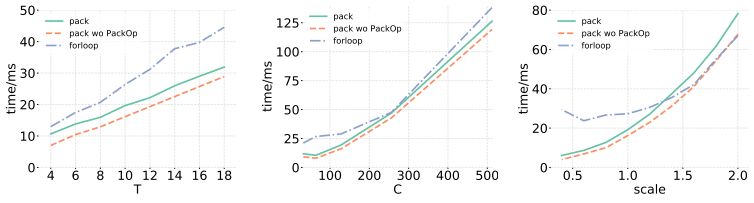
(d) varying T (e) varying C (f) varying scale

Fig. 2: GPU time comparison with varying T, C and scale s on MXNet. Top row: MXNet-1.5.1, bottom row: MXNet-1.6.0.

1.2 MXNet

We benchmark the time of PackOp under the MXNet framework[5]. Two environments are tested: MXNet-1.5.1+CUDA-9.0+cuDNN-7.3.0 and MXNet-1.6.0+

[5] Imperative mode is used here, the results in symbol mode may be different. But we conjecture the overall trend should be consistent.



(a) varying T (b) varying C (c) varying scale

(d) varying T (e) varying C (f) varying scale

Fig. 3: GPU time comparison with varying T, C and scale s on TensorFlow. Top row: TensorFlow-1.12, bottom row: TensorFlow-2.0.

CUDA-10.2+cuDNN-7.6.5. For credible timing, `mxnet.nd.waitall()` is adopted and the average time of 30 trials is reported.

The results are shown in Fig. 2. We can see that, under different cases, varying T (number of conv blocks), C (number of channels) and scale, using PackOp can speed up the computation significantly. Meanwhile, the latency caused by packing all feature maps together is negligible.

1.3 TensorFlow

We have also benchmarked the time under the TensorFlow framework. Two environments TensorFlow-1.12+CUDA-9.0+cuDNN-7.3.0 and TensorFlow-2.0+ CUDA-10.0+cuDNN7.6.5 are tested. Note that, the symbol mode is adopted. For accurate timing, CUDA event based timers are used. All other settings are consistent to PyTorch and MXNet experiments.

The results with varying T, C and input scale are shown in Fig. 3. Actually, we find that the trends are very similar to those in the PyTorch experiments: PackOp is more advantageous for larger T, smaller C and smaller input scale. Note that our current implementation of PackOp is based on `scatter_nd_update()` that is a little inefficient. Using a full CUDA implementation may further reduce the PackOp's latency.

1.4 Summary

Based on all the above experiments, under different deep learning frameworks and different software versions, we can conclude that although current mainstream

input size	channels	tr-pack	ts-pack	ch-gn	time	AP	AP_{50}	AP_{75}
1000×600	128				62.8	37.8	55.2	40.6
		✓	✓	✓	39.3	38.1	55.6	40.9
		✓		✓	65.3	30.1	46.3	32.6
1300×800	64				69.3	37.5	55.0	40.5
		✓	✓	✓	51.7	38.5	56.0	41.4
		✓		✓	65.0	24.4	38.6	26.1
1000×600	64				63.7	36.1	53.3	38.6
		✓	✓	✓	34.6	36.7	53.8	39.7
		✓		✓	64.7	24.6	39.2	26.5

Table 1: PackDet results using ResNet-50 with different inputs and channels.

frameworks are well optimized, there is still an appreciable space to further improve the computation efficiency, especially for multi-branch structures. PackOp that collects fragmented features together and performs computation in parallel, supplies a simple but useful solution for speeding up head forwarding. Meanwhile, the solution is platform-agnostic. Finally, the results also imply that regularity of structure is important for designing low-latency neural networks.

2 Effectiveness of PackOp Integrated into PackDet

The effectiveness of packed head in PackDet for speeding up inference and improving accuracy has already been demonstrated in the main paper with fixed input size and number of channels. Here, we provide additional results with varying input size and channel number. The results are given in Table 1. The backbone is ResNet-50, the 1x learning schedule is adopted, M_e is set as 3. The running environment is PyTorch-1.5+CUDA-10.2+cuDNN-7.6.5.

With fewer channels and/or smaller input sizes, the speedup is more evident, 13-30 ms absolutely or 20-46% relatively. For industry applications, roughly speaking, 46% speedup means 46% servers can be cut off. In addition, longer heads bring visible increase on accuracy especially for small backbones as shown in the main paper. In summary, PackOp is effective for both reducing latency and improving accuracy.

3 Memory Cost Comparison

In the main paper, we have demonstrated the memory efficiency of PackOp by theoretical derivation. Here, we demonstrate the memory efficiency of PackDet via experiments. Please note that they are different things, as PackOp is only one of many components in PackDet. The memory cost of FCOS [2] and PackDet are listed in Table 2. The relative increases of train memory w.r.t. FCOS are about 6%, 9%, 7%, 9%, respectively, which are higher than 5.7% (relative memory increase of PackOp mentioned in main paper) as there are more layers in head and the dense skip connections instead of PackOp itself costs more memory. At test stage, as only forward information is kept, the cost is nearly constant.

Method	Stage	MobileNet-v2	ResNet-50	ResNet-101	ResNeXt-101-DCN
FCOS	train	7739×4	5519×4	7489×4	6797×8
PackDet	train	8173×4	6021×4	8005×4	7430×8
FCOS	test	1323	1751	1825	2525
PackDet	test	1331	1751	1825	2525

Table 2: Memory cost (in MiB) comparison of PackDet and FCOS. In training stage, the memory cost is obtained by `torch.cuda.max_memory_allocated()`; in testing stage, the memory cost is obtained by `nvidia-smi`. $\times 4$ and $\times 8$ mean the model is trained with 4 and 8 GPUs, respectively.

Method	Input Size	Backbone	MS Train	FPS	AP	AP_{50}	AP_{75}	AP_S	AP_M	AP_L
RetinaNet800	$\sim 1300 \times 800$	R-101	✓	9.5	39.1	59.1	42.3	21.8	42.7	50.2
RetinaNet800+PackOp	$\sim 1300 \times 800$	R-50	✓	8.8	39.1	58.4	42.1	21.9	42.3	49.4

Table 3: Results on COCO `test-dev` set of RetinaNet using PackOp. `R`: ResNet.

4 Integrated with RetinaNet

PackOp is available for both anchor-free and anchor-based methods. The difference between them lies in the last prediction layer, which is independent from the lower layers in head. Moreover, anchor-free method can be viewed as a special case of anchor-based method with only one anchor [3].

We apply PackOp to RetinaNet [1], an anchor-based method, and obtain AP=39.1%, AP_{50}=58.4%, AP_{75}=42.1%, which are comparable to RetinaNet's results using ResNet-101 as backbone in Table 3. Please not that the FPS of `RetinaNet800+PackOp` is a little low as the post-processing is not optimized to fully use the advantage of packing all scales together. We rudely split the packed prediction to obtain five maps and call the original post-processing code of RetinaNet for each scale separately, which is obviously slow.

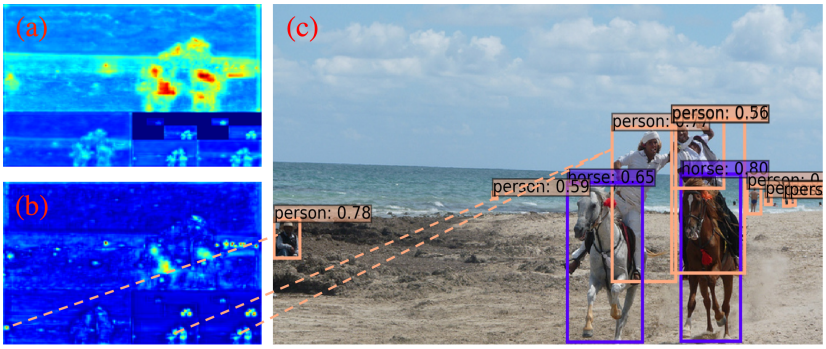


Fig. 4: Visualization of features. (a) Outputted feature map $F^{(p)}$ of PackOp. (b) Outputted feature map $F^{(s)}$ of shared head. (c) Image with predicted boxes.

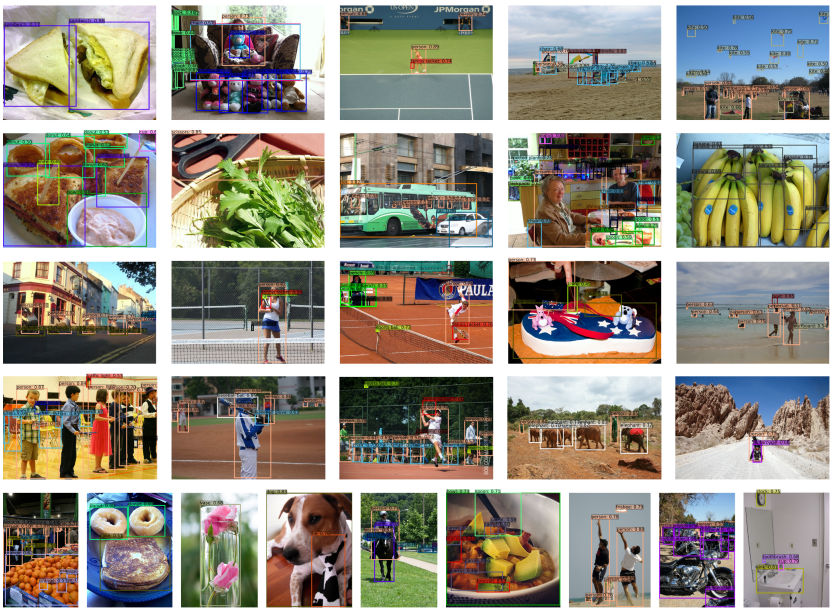


Fig. 5: Detection examples on COCO `test-dev` of PackDet with the ResNeXt-101-DCN backbone.

5 Visualization

A visual comparison of the feature map $F^{(p)}$ and $F^{(s)}$ is presented in Fig. 4. As can be seen from Fig. 4(a), the same position of different scales have similar activation values. In contrast, by optimizing loss function, objects of different sizes would be detected at different scales as indicated by Fig. 4(b)-(c).

The qualitative detection results of PackDet are shown in Fig. 5. The results are generated with ResNeXt-101-DCN as the backbone and M_e (number of extra feature maps) is 3.

References

1. Lin, T., Goyal, P., Girshick, R.B., et al.: Focal loss for dense object detection. In: ICCV (2017)
2. Tian, Z., Shen, C., Chen, H., et al.: FCOS: Fully convolutional one-stage object detection. arXiv: 1904.01355 (2019)
3. Zhang, S., Chi, C., Yao, Y., et al.: Bridging the gap between anchor-based and anchor-free detection via adaptive training sample selection. In: CVPR (2020)