Deep Equilibrium Object Detection

Shuai Wang[1] Yao Teng[1] Limin Wang[1,2,✉]
[1]State Key Laboratory for Novel Software Technology, Nanjing University [2]Shanghai AI Lab
https://github.com/MCG-NJU/DEQDet

A. Notions and hyperparameters in DEQDet

To avoid confusion by the notions in DEQDet, we summarize all symbols in Tab. 1. For the convenience of others to reproduce the experiment in DEQDet, we also provides the training hyper-parameters.

B. Noise projection for *fixed-point*

To impose the refinement jacobian matrix on the noise term, in practice, we directly feed the noisy latent variables into the refinement layer. Then, the gradients provided by the noise term is equivalent to have the refinement jacobian matrix as the multiplier. We conduct directly feeding noise to refinement layer instead of computing jacobian, due to

- jacobian based projection is equivalent to the one step Taylor expansion of refinement layer.
- computing jacobian matrix spends more time
- actually, as detaching position vector in object detectors is a common practice, employing automatic differentiation library to solve jacobian matrix will delivery wrong results.

$$\begin{aligned}\hat{\mathbf{y}}_n &= f(x, \mathbf{y}_{n-1} + \epsilon) &(1)\\ &\approx f(x, \mathbf{y}_{n-1}) + \frac{\partial \mathbf{y}_n}{\partial \mathbf{y}_{n-1}} \cdot \epsilon &(2)\\ &= \mathbf{y}_n + \frac{\partial \mathbf{y}_n}{\partial \mathbf{y}_{n-1}} \cdot \epsilon \;\; \epsilon \sim \mathcal{N}(\mathbf{0}, \sigma^2 I) &(3)\end{aligned}$$

C. Refinement-aware gradient derivation

To handle Refinement awareness, we reformulate *fixed-point* formula to a two-step unrolled *fix-point* to take the query term into account:

$$\mathbf{y}^* = f(\mathbf{x}, f(\mathbf{x}, \mathbf{y}^*|\theta)|\theta) \tag{4}$$

For simplicity, we define a new function h, which is the two-step unrolled refinement layer:

$$h(\mathbf{x}, \mathbf{y}^*|\theta) = f(\mathbf{x}, f(\mathbf{x}, \mathbf{y}^*|\theta)|\theta) \tag{5}$$

Then, the IFT gradient of Eq. (4) becomes:

$$\frac{\partial \mathbf{y}^*}{\partial(\cdot)} = (I - \frac{\partial h(\mathbf{x}, \mathbf{y}^*|\theta)}{\partial \mathbf{y}^*})^{-1} \frac{\partial h(\mathbf{x}, \mathbf{y}^*|\theta)}{\partial(\cdot)} \tag{6}$$

We first replace the inverse jacobian term $(I - \frac{\partial h(\mathbf{x}, \mathbf{y}^*|\theta)}{\partial \mathbf{y}^*})^{-1}$ with identity matrix I as JFB, and then we implicitly differentiate two sides of Eq. (5):

$$\frac{\partial \mathbf{y}^*}{\partial(\cdot)} \approx \frac{\partial h(\mathbf{x}, \mathbf{y}^*|\theta)}{\partial(\cdot)} = [I + \frac{\partial f(\mathbf{x}, \mathbf{y}^*|\theta)}{\partial \mathbf{y}^*}] \frac{\partial f(\mathbf{x}, \mathbf{y}^*|\theta)}{\partial(\cdot)} \tag{7}$$

then we get our refinement aware gradient:

$$\frac{\partial \mathbf{y}^*}{\partial(\cdot)} \approx [I + \frac{\partial f(\mathbf{x}, \mathbf{y}^*|\theta)}{\partial \mathbf{y}^*}] \frac{\partial f(\mathbf{x}, \mathbf{y}^*|\theta)}{\partial(\cdot)} \tag{8}$$

D. Connection between DEQ model and diffusion model

There are some differences and connections between diffusion model and *fixed-point* iterations based DEQ model, a basic *fixed -point* form likes :

$$y = f(x, y), \tag{9}$$

$$y_n = f(x, y_{n-1}), \tag{10}$$

while diffusion [3, 4] can be derived from ode form (ddim [4]) :

$$\frac{dy}{dt} = g(x, y, t), \tag{11}$$

when we make a finite integral for ode, we can get:

$$y_n = y_{n-1} + \int_{t_{n-1}}^{t_n} g(x, y, t)dt, \tag{12}$$

$$y_n = y_{n-1} + g(x, y, t)\Delta t, \tag{13}$$

✉: Corresponding author (lmwang@nju.edu.cn).

Hyperparameters	Notation	Value
The content vector	$\mathbf{q}$	
The positional vector	$\mathbf{p}$	
The condition variable / multi-scale features	$\mathbf{x}$	
The latent variable	$\mathbf{y} = (\mathbf{p}, \mathbf{q})$	
The parameters	θ, η	
The refinement layer	$f(\mathbf{x}, \mathbf{y})$	
The initialization layer	$g(\mathbf{x}, \mathbf{y})$	
The deep supervision position set	Ω	[1,3,6,9,12,20]
The number fixpoint iteration steps for training	T_{train}	20
The number fixpoint iteration steps for inference	T_{infer}	25
The perturbation probability	v	0.2
The perturbation size of content vector	σ_q	0.1
The perturbation size of positional vector	σ_p	25
The sampling points of initialization layer		64
The sampling points of refinement layer layer		32
The learning rate		0.000025
The learning rate decay		*0.1
The learning rate decay epoch for $1\times$ training		8, 11
The learning rate decay epoch for $2\times$ training		16, 22
weight decay for backbone		0.01
weight decay for decoder		0.1
The loss weight for focal loss	λ_{focal}	2
The loss weight for l1 loss	λ_{l1}	5
The loss weight for giou loss	λ_{giou}	2

Table 1: The hyper-parameters of DEQDet.

so, it is clear that the function $f(x, y)$ in *fixed-point* iteration can be any arbitrary form, instead **ode always keeps an identity branch or residual connection**. But actually we also use identity branch in our DEQDet decoder layer. The second difference is **ode is step aware** as $g(x, y, t)$ takes step t as input, while *fixed-point* iteration not.

E. Comparison with similar works

Except our method mainly focuses on *fixed-point* iteration, others devote to migrate diffusion diagram to object detection [1]. The remaining major difference between our detector and DiffusionDet [1] is that our decoder consists of only two layers, the first layers aims to get a good initial guess while the second layer progressively refines this initial result. Then the definition of a refinement step is also distinct. we regard running refinement layer once as a refinement step while their refinement step runs the entire decoder, which consists of 6 layers.

F. Extend DEQDet to other detector

We also extend our DEQDet to sparse-RCNN [5]. we keep the training settings *e.g.* training epochs, optimizer, learning rate scheduler consistent with the original sparse-rcnn, Our DEQDet improves the sparse-rcnn by 2.5 on mAP and 3.7 on AP_{small}.

Detectors	Params	AP	AP_{50}	AP_{75}	AP_s	AP_m	AP_l
Sparse R-CNN [5]	110M	45.0	63.4	48.2	26.9	47.2	59.5
+DEQDet (2x)	53M	47.0	65.7	51.6	30.3	49.8	61.0
+DEQDet (3x)	53M	47.5	66.5	52.4	30.6	50.1	61.5

Table 2: **$3\times$ training scheme with 300 queries.** Extend DEQDet to other detectors *e.g.* sparse-rcnn [5].

G. Refinement convergence

We evaluate our DEQDet with different refinment steps in Tab. 3 and Tab. 4. We conduct experiments on DEQDet† trained under $1\times$ scheme with 100 queries and $2\times$ scheme with 300 queries. In Tab. 4, when DEQDet† with 300 queries refines 5 steps, the number of valid decoder layers is as same as AdaMixer [2], but DEQDet achieves 49.0 mAP, exceeds AdaMixer by 2.0 mAP. As the refinement step increases, the performance will be further improved.

GFLOPS	steps	AP	AP_{50}	AP_{75}	AP_s	AP_m	AP_l
107.50	4	44.9	63.5	48.4	26.1	48.0	60.9
109.73	5	45.4	64.0	49.0	26.6	48.4	61.2
111.97	6	45.7	64.3	49.4	26.8	48.7	61.4
116.43	8	45.8	64.5	49.6	27.0	48.9	61.3
120.90	10	45.9	64.6	49.7	27.2	48.9	61.2
132.07	15	45.9	64.7	49.6	27.4	49.0	61.2
143.24	20	46.0	64.7	49.6	27.4	49.0	61.4
154.41	25	46.0	64.8	49.6	27.5	49.0	61.2
210.25	50	46.0	64.7	49.6	27.5	49.0	61.5
-	200	46.0	64.8	49.7	27.6	49.1	61.5

Table 3: Refinement steps for DEQDet† trained under $1\times$ scheme with ResNet50 backbone and 100 queries.

GFLOPS	steps	AP	AP_{50}	AP_{75}	AP_s	AP_m	AP_l
129.87	4	48.7	67.3	52.8	32.0	51.7	63.0
136.57	5	49.0	67.7	53.3	32.3	51.9	63.0
143.27	6	49.1	67.8	53.5	32.5	52.0	63.2
156.67	8	49.3	68.0	53.7	32.9	52.0	63.2
170.07	10	49.5	68.2	53.9	33.1	52.1	63.4
203.58	15	49.5	68.3	53.9	33.2	52.1	63.3
237.08	20	49.5	68.3	54.0	33.2	52.1	63.4
270.59	25	49.5	68.3	54.0	33.2	52.1	63.3
438.11	50	49.6	68.3	54.0	33.3	52.2	63.1
-	200	49.5	68.3	54.0	33.2	52.2	63.1

Table 4: Refinement steps for DEQDet† trained under $2\times$ scheme with ResNet50 backbone and 300 queries.

We try to employ off-the-shelf *fixed-point* solver *e.g.* anderson solver to accelerate the *fixed-point* solving, but the result is not what we expected. We think this is mainly due to the highly nonlinear property of the refinement layer and we should couple the solver with training instead of decoupling. We left this for our future work.

H. Detection Performance on COCO test set

We also provide the detection performance of DEQDet models on COCO *test-dev* set in Tab. 6. Different from the COCO *minival* set, there is no publicly avaliable labels of *test-dev*.

I. Limitations

Although our DEQDet achieves comparable results with acceptable resource consumption, the training time consumption is still very large compared to other methods. As for inference time, it is acceptable to choose refinement steps adaptively according to resource constraints. There are also

m	steps	AP	AP_{50}	AP_{75}	AP_s	AP_m	AP_l
2	5	48.9	67.5	53.1	32.2	51.8	63.1
2	10	49.2	68.0	53.6	32.9	52.0	63.1
2	20	49.4	68.2	53.8	32.9	52.0	63.4
2	50	49.5	68.2	53.9	33.1	52.2	63.3
4	5	48.8	67.5	53.1	32.2	51.8	63.3
4	10	49.2	67.9	53.5	32.7	52.0	63.1
4	20	49.3	68.1	53.7	33.1	52.0	63.4
4	50	49.4	68.1	53.8	33.1	52.1	63.3

Table 5: Refinement steps of Anderson Solver for EQDet† with ResNet50 backbone and 300 queries. m is a hyperparameter in Anderson Solver.

Detectors	Backbone	queries	AP	AP_{50}	AP_{75}	AP_s	AP_m	AP_l
DEQDet (1x)	R50	100	45.4	64.5	49.0	26.0	47.7	59.3
DEQDet† (1x)	R50	100	46.5	65.5	50.4	27.2	48.9	60.2
DEQDet† (2x)	R50	300	49.8	68.5	54.4	31.2	52.1	62.5
DEQDet† (2x)	R101	300	50.6	69.4	55.1	31.3	53.3	64.2

Table 6: Detection Performance of DEQDet on COCO *test-dev* set, $1\times$ means $1\times$ training scheme, including 12 epochs, while $2\times$ contains 24 epochs.

a lot of improvement space for training strategy. Also note that the refinement layer is not light and each refinement iteration is not really cheap. Future improvements can be made from light weight refinement layer design and reduction of refinement steps

J. Training algorithm

Algorithm 1 Noise Perturbation Code

```
def noise_content(content, noise_size):
    """ add noise to content query """
    noise = torch.randn_like(content)*
          torch.norm(content, dim=-1)
    noise_content = (1-noise_size)*content +
        noise_size*noise
    return noise_content
def noise_pos(pos, noise_size):
    """ add noise to position query """
    bbox = decode(pos)
    noise = torch.randn_like(bbox)
    noise_bbox = bbox + noise_size*noise
    noise_pos = encode(noise_bbox)
    return noise_pos
```

References

[1] Shoufa Chen, Peize Sun, Yibing Song, and Ping Luo. Diffusiondet: Diffusion model for object detection. *arXiv preprint arXiv:2211.09788*, 2022. 2

[2] Ziteng Gao, Limin Wang, Bing Han, and Sheng Guo. Adamixer: A fast-converging query-based object detector. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5364–5373, 2022. 3

Algorithm 2 Training code

```
def train(
        T , #iteration forward times
        perturb_prob, # perturbation probability
        content_ps, #content query perturb size
        pos_ps, #position query perturb size
        supervision_pos, #deep supervision positions
        init_content, #[B, N, C]
        init_pos, #[B, N, 4]
        feats, #[B, L, C, H, W]
        annotations):
  """
  # B: batch
  # N: number of proposal boxes
  """
  solving_path = []
  all_loss = 0
  init_content, init_pos = initialization_layer(feats
       , init_content, init_pos)
  # supervision for initialization layer
  all_loss += loss(content, pos, annotations)
  # extra supervision for initialization layer
  # in order to stablize the gradient connection
  # between refinement layer and initialization layer
  content, pos = init_content,init_pos
  for i in range(2):
     content, pos = refinement_layer(
              feats,content, pos)
     all_loss += loss(content, pos, annotations)

  # naive fix-point solving...
  with torch.no_grad():
    content, pos = init_content,init_pos
    solving_path = []
    for i in range(T):
       # refinement aware perturbation
       # 1. add noise to content query
       if torch.rand(1) < perturb_prob :
          content = noise_content(
                   content, content_ps
                   )
       # 2. add noise to pos query
       if torch.rand(1) < perturb_prob :
          pos = noise_pos(pos, pos_ps)
       # 3. project noise
       content, pos = refinement_layer(
                   feats, content, pos
                   )
       if i in supervision_pos:
          solving_path.append((content, pos))

  # deep supervision and gradient construction
  for content, pos in solving_path:
    # refinement aware gradient
    for i in range(2):
        content, pos = \
        refinement_layer(feats,content, pos)
    all_loss += loss(content, pos, annotations)
  return loss
```

[3] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in Neural Information Processing Systems*, 33:6840–6851, 2020. 1

[4] Jiaming Song, Chenlin Meng, and Stefano Ermon. Denoising diffusion implicit models. *arXiv preprint arXiv:2010.02502*, 2020. 1

[5] Peize Sun, Rufeng Zhang, Yi Jiang, Tao Kong, Chenfeng Xu, Wei Zhan, Masayoshi Tomizuka, Lei Li, Zehuan Yuan, Changhu Wang, et al. Sparse r-cnn: End-to-end object detection with learnable proposals. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 14454–14463, 2021. 2