Deep Equilibrium Object Detection

Shuai Wang1  Yao Teng1  Limin Wang1,2,*
1State Key Laboratory for Novel Software Technology, Nanjing University  2Shanghai 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 into the refinement layer. 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.

\[
\begin{align*}
\hat{y}_n &= f(x, y_{n-1} + \epsilon) \\
&\approx f(x, y_{n-1}) + \frac{\partial y_n}{\partial y_{n-1}} \cdot \epsilon \\
&= y_n + \frac{\partial y_n}{\partial y_{n-1}} \cdot \epsilon \sim \mathcal{N}(0, \sigma^2 I) 
\end{align*}
\]

C. Refinement-aware gradient derivation

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

\[
y^* = f(x, f(x, y^*|\theta)|\theta) 
\]

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

\[
h(x, y^*|\theta) = f(x, f(x, y^*|\theta)|\theta) 
\]

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

\[
\frac{\partial y^*}{\partial (\cdot)} = (I - \frac{\partial h(x, y^*|\theta)}{\partial y^*})^{-1} \frac{\partial h(x, y^*|\theta)}{\partial (\cdot)} 
\]

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

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

then we get our refinement aware gradient:

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

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:

\[
y = f(x, y), \\
y_n = f(x, y_{n-1}), \\
\text{while diffusion [3, 4] can be derived from ode form (ddim [4]) :} \\
\frac{dy}{dt} = g(x, y, t), \\
\text{when we make a finite integral for ode, we can get:} \\
y_n = y_{n-1} + \int_{t_{n-1}}^{t_n} g(x, y, t) dt, \\
y_n = y_{n-1} + g(x, y, t) \Delta t, 
\]
The content vector \( q \)
The positional vector \( p \)
The condition variable / multi-scale features \( x \)
The latent variable \( y = (p, q) \)
The parameters \( \theta, \eta \)
The refinement layer \( f(x, y) \)
The initialization layer \( g(x, y) \)
The deep supervision position set \( \Omega \)
The number fixpoint iteration steps for training \( T_{\text{train}} \)
The number fixpoint iteration steps for inference \( T_{\text{infer}} \)
The perturbation probability \( v \)
The perturbation size of content vector \( \sigma_q \)
The perturbation size of positional vector \( \sigma_p \)
The sampling points of initialization layer \( 64 \)
The sampling points of refinement layer \( 32 \)
The learning rate \( 0.000025 \)
The learning rate decay \( \times 0.1 \)
The learning rate decay epoch for \( 1 \times \text{training} \) \( 8, 11 \)
The learning rate decay epoch for \( 2 \times \text{training} \) \( 16, 22 \)
weight decay for backbone \( 0.01 \)
weight decay for decoder \( 0.1 \)
The loss weight for focal loss \( \lambda_{\text{focal}} \)
The loss weight for l1 loss \( \lambda_{\text{l1}} \)
The loss weight for giou loss \( \lambda_{\text{giou}} \)

<table>
<thead>
<tr>
<th>Hyperparameters</th>
<th>Notation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>The content vector</td>
<td>( q )</td>
<td></td>
</tr>
<tr>
<td>The positional vector</td>
<td>( p )</td>
<td></td>
</tr>
<tr>
<td>The condition variable</td>
<td>( x )</td>
<td></td>
</tr>
<tr>
<td>The latent variable</td>
<td>( y = (p, q) )</td>
<td></td>
</tr>
<tr>
<td>The parameters</td>
<td>( \theta, \eta )</td>
<td></td>
</tr>
<tr>
<td>The refinement layer</td>
<td>( f(x, y) )</td>
<td></td>
</tr>
<tr>
<td>The initialization layer</td>
<td>( g(x, y) )</td>
<td></td>
</tr>
<tr>
<td>The deep supervision position set</td>
<td>( \Omega )</td>
<td>( [1,3,6,9,12,20] )</td>
</tr>
<tr>
<td>The number fixpoint iteration steps for training</td>
<td>( T_{\text{train}} )</td>
<td>( 20 )</td>
</tr>
<tr>
<td>The number fixpoint iteration steps for inference</td>
<td>( T_{\text{infer}} )</td>
<td>( 25 )</td>
</tr>
<tr>
<td>The perturbation probability</td>
<td>( v )</td>
<td>( 0.2 )</td>
</tr>
<tr>
<td>The perturbation size of content vector</td>
<td>( \sigma_q )</td>
<td>( 0.1 )</td>
</tr>
<tr>
<td>The perturbation size of positional vector</td>
<td>( \sigma_p )</td>
<td>( 25 )</td>
</tr>
<tr>
<td>The sampling points of initialization layer</td>
<td></td>
<td>( 64 )</td>
</tr>
<tr>
<td>The sampling points of refinement layer</td>
<td></td>
<td>( 32 )</td>
</tr>
<tr>
<td>The learning rate</td>
<td></td>
<td>( 0.000025 )</td>
</tr>
<tr>
<td>The learning rate decay</td>
<td></td>
<td>( \times 0.1 )</td>
</tr>
<tr>
<td>The learning rate decay epoch for ( 1 \times \text{training} )</td>
<td></td>
<td>( 8, 11 )</td>
</tr>
<tr>
<td>The learning rate decay epoch for ( 2 \times \text{training} )</td>
<td></td>
<td>( 16, 22 )</td>
</tr>
<tr>
<td>weight decay for backbone</td>
<td></td>
<td>( 0.01 )</td>
</tr>
<tr>
<td>weight decay for decoder</td>
<td></td>
<td>( 0.1 )</td>
</tr>
<tr>
<td>The loss weight for focal loss</td>
<td></td>
<td>( \lambda_{\text{focal}} )</td>
</tr>
<tr>
<td>The loss weight for l1 loss</td>
<td></td>
<td>( \lambda_{\text{l1}} )</td>
</tr>
<tr>
<td>The loss weight for giou loss</td>
<td></td>
<td>( \lambda_{\text{giou}} )</td>
</tr>
</tbody>
</table>

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 APsmall.

<table>
<thead>
<tr>
<th>Detectors</th>
<th>Params</th>
<th>AP</th>
<th>AP$_{50}$</th>
<th>AP$_{75}$</th>
<th>AP$_s$</th>
<th>AP$_m$</th>
<th>AP$_l$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sparse R-CNN [5]</td>
<td>110M</td>
<td>45.0</td>
<td>63.4</td>
<td>48.2</td>
<td>26.9</td>
<td>47.2</td>
<td>59.5</td>
</tr>
<tr>
<td>+DEQDet (2x)</td>
<td>53M</td>
<td>47.0</td>
<td>67.7</td>
<td>51.6</td>
<td>30.3</td>
<td>49.8</td>
<td>61.0</td>
</tr>
<tr>
<td>+DEQDet (3x)</td>
<td>53M</td>
<td>47.5</td>
<td>66.5</td>
<td>52.4</td>
<td>30.6</td>
<td>50.1</td>
<td>61.5</td>
</tr>
</tbody>
</table>

Table 2: \( 3 \times \text{training scheme with 300 queries.} \) Extend DEQDet to other detectors e.g. sparse-rcnn [5].

G. Refinement convergence

We evaluate our DEQDet with different refinement steps in Tab. 3 and Tab. 4. We conduct experiments on DEQDet$^1$ trained under \( 1 \times \) scheme with 100 queries and \( 2 \times \) scheme with 300 queries. In Tab. 4, when DEQDet$^1$ 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.

<table>
<thead>
<tr>
<th>GFLOPS</th>
<th>steps</th>
<th>AP</th>
<th>AP50</th>
<th>AP75</th>
<th>APs</th>
<th>APm</th>
<th>APl</th>
</tr>
</thead>
<tbody>
<tr>
<td>107.50</td>
<td>4</td>
<td>44.9</td>
<td>63.5</td>
<td>48.4</td>
<td>26.1</td>
<td>48.0</td>
<td>60.9</td>
</tr>
<tr>
<td>109.73</td>
<td>5</td>
<td>45.4</td>
<td>64.0</td>
<td>49.0</td>
<td>26.6</td>
<td>48.4</td>
<td>61.2</td>
</tr>
<tr>
<td>111.97</td>
<td>6</td>
<td>45.7</td>
<td>64.1</td>
<td>49.4</td>
<td>26.8</td>
<td>48.7</td>
<td>61.4</td>
</tr>
<tr>
<td>116.43</td>
<td>8</td>
<td>45.8</td>
<td>64.5</td>
<td>49.6</td>
<td>27.0</td>
<td>48.9</td>
<td>61.3</td>
</tr>
<tr>
<td>120.90</td>
<td>10</td>
<td>45.9</td>
<td>64.6</td>
<td>49.7</td>
<td>27.2</td>
<td>48.9</td>
<td>61.2</td>
</tr>
<tr>
<td>132.07</td>
<td>15</td>
<td>45.9</td>
<td>64.7</td>
<td>49.5</td>
<td>27.4</td>
<td>49.0</td>
<td>61.2</td>
</tr>
<tr>
<td>143.24</td>
<td>20</td>
<td>46.0</td>
<td>64.7</td>
<td>49.6</td>
<td>27.4</td>
<td>49.0</td>
<td>61.4</td>
</tr>
<tr>
<td>154.41</td>
<td>25</td>
<td>46.0</td>
<td>64.8</td>
<td>49.6</td>
<td>27.5</td>
<td>49.0</td>
<td>61.2</td>
</tr>
<tr>
<td>210.25</td>
<td>50</td>
<td>46.0</td>
<td>64.7</td>
<td>49.6</td>
<td>27.5</td>
<td>49.0</td>
<td>61.5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>GFLOPS</th>
<th>steps</th>
<th>AP</th>
<th>AP50</th>
<th>AP75</th>
<th>APs</th>
<th>APm</th>
<th>APl</th>
</tr>
</thead>
<tbody>
<tr>
<td>129.87</td>
<td>4</td>
<td>48.7</td>
<td>67.3</td>
<td>52.8</td>
<td>32.0</td>
<td>51.7</td>
<td>63.1</td>
</tr>
<tr>
<td>136.57</td>
<td>5</td>
<td>49.0</td>
<td>67.7</td>
<td>53.3</td>
<td>32.3</td>
<td>51.8</td>
<td>63.3</td>
</tr>
<tr>
<td>143.27</td>
<td>6</td>
<td>49.1</td>
<td>67.9</td>
<td>53.5</td>
<td>32.5</td>
<td>52.0</td>
<td>63.2</td>
</tr>
<tr>
<td>156.67</td>
<td>8</td>
<td>49.2</td>
<td>68.1</td>
<td>53.7</td>
<td>32.9</td>
<td>52.0</td>
<td>63.4</td>
</tr>
<tr>
<td>170.07</td>
<td>10</td>
<td>49.4</td>
<td>68.3</td>
<td>54.0</td>
<td>33.2</td>
<td>52.1</td>
<td>63.3</td>
</tr>
<tr>
<td>203.58</td>
<td>15</td>
<td>49.5</td>
<td>68.3</td>
<td>54.0</td>
<td>33.2</td>
<td>52.1</td>
<td>63.4</td>
</tr>
<tr>
<td>237.08</td>
<td>20</td>
<td>49.5</td>
<td>68.3</td>
<td>54.0</td>
<td>33.2</td>
<td>52.1</td>
<td>63.4</td>
</tr>
<tr>
<td>270.59</td>
<td>25</td>
<td>49.5</td>
<td>68.3</td>
<td>54.0</td>
<td>33.2</td>
<td>52.1</td>
<td>63.3</td>
</tr>
<tr>
<td>438.11</td>
<td>50</td>
<td>49.6</td>
<td>68.3</td>
<td>54.0</td>
<td>33.3</td>
<td>52.2</td>
<td>63.1</td>
</tr>
<tr>
<td>-</td>
<td>200</td>
<td>46.0</td>
<td>64.8</td>
<td>49.7</td>
<td>27.6</td>
<td>49.1</td>
<td>61.5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Detectors Backbone queries</th>
<th>AP</th>
<th>AP50</th>
<th>AP75</th>
<th>APs</th>
<th>APm</th>
<th>APl</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEQDet (1x) R50 100</td>
<td>45.4</td>
<td>64.5</td>
<td>49.0</td>
<td>26.0</td>
<td>47.7</td>
<td>59.3</td>
</tr>
<tr>
<td>DEQDet† (1x) R50 100</td>
<td>46.5</td>
<td>65.5</td>
<td>50.4</td>
<td>27.2</td>
<td>48.9</td>
<td>60.2</td>
</tr>
<tr>
<td>DEQDet† (2x) R101 300</td>
<td>49.8</td>
<td>68.5</td>
<td>54.4</td>
<td>31.2</td>
<td>52.1</td>
<td>62.5</td>
</tr>
<tr>
<td>DEQDet† (2x) R101 300</td>
<td>50.6</td>
<td>69.4</td>
<td>55.1</td>
<td>31.3</td>
<td>53.3</td>
<td>64.2</td>
</tr>
</tbody>
</table>

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

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 available 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 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

```python
def noise_content(content, noise_size):
    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):
    bbox = decode(pos)
    noise = torch.randn_like(bbox)
    noise_bbox = bbox + noise_size * noise
    noise_pos = encode(noise_bbox)
    return noise_pos
```

References


Algorithm 2 Training code

```python
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, #init_content, init_pos
    init_pos, #init_pos, #init_pos
    feats, #feats, #feats
    annotations):
    ""
    # B: batch
    # N: number of proposal boxes
    ""
    solving_path = []
    all_loss = 0
    init_content, init_pos = initialization_layer(feats
    # supervision for initialization layer
    all_loss += loss(content, pos, annotations)
    # extra supervision for initialization layer
    # in order to stabilize 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.randn() < perturb_prob :
                content = noise_content(content, content_ps
            # 2. add noise to pos query
            if torch.randn() < 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 = \n                    refinement_layer(feats, content, pos)
            all_loss += loss(content, pos, annotations)
    return loss
```

