End-to-End Object Detection with Transformers: Supplementary material

Table of Contents

1 Appendix

1.1 Preliminaries: Multi-head attention layers

Since our model is based on the Transformer architecture, we remind here the general form of attention mechanisms we use for exhaustivity. The attention mechanism follows [8], except for the details of positional encodings (see Equation 6) that follows [1].

Multi-head The general form of *multi-head attention* with M heads of dimension d is a function with the following signature (using $d' = \frac{d}{M}$, and giving matrix/tensors sizes in underbrace)

$$\text{mh-attn} : \underbrace{X_{\text{q}}}_{d\times N_{\text{q}}}, \underbrace{X_{\text{kv}}}_{d\times N_{\text{kv}}}, \underbrace{T}_{M\times 3\times d'\times d}, \underbrace{L}_{d\times d} \mapsto \underbrace{\tilde{X}_{\text{q}}}_{d\times N_{\text{q}}} \tag{1}$$

where X_{q} is the *query sequence* of length N_{q}, X_{kv} is the *key-value sequence* of length N_{kv} (with the same number of channels d for simplicity of exposition), T is the weight tensor to compute the so-called query, key and value embeddings, and L is a projection matrix. The output is the same size as the query sequence. To fix the vocabulary before giving details, multi-head *self*-attention (mh-s-attn) is the special case $X_{\text{q}} = X_{\text{kv}}$, i.e.

$$\text{mh-s-attn}(X, T, L) = \text{mh-attn}(X, X, T, L)\,. \tag{2}$$

The multi-head attention is simply the concatenation of M single attention heads followed by a projection with L. The common practice [8] is to use residual

connections, dropout and layer normalization. In other words, denoting $\tilde{X}_{\mathrm{q}} = \text{mh-attn}(X_{\mathrm{q}}, X_{\mathrm{kv}}, T, L)$ and $\bar{\bar{X}}^{(q)}$ the concatenation of attention heads, we have

$$X'_{\mathrm{q}} = [\mathrm{attn}(X_{\mathrm{q}}, X_{\mathrm{kv}}, T_1); ...; \mathrm{attn}(X_{\mathrm{q}}, X_{\mathrm{kv}}, T_M)] \tag{3}$$

$$\tilde{X}_{\mathrm{q}} = \mathrm{layernorm}\big(X_{\mathrm{q}} + \mathrm{dropout}(LX'_{\mathrm{q}})\big)\,, \tag{4}$$

where [;] denotes concatenation on the channel axis.

Single head An attention head with weight tensor $T' \in \mathbb{R}^{3\times d'\times d}$, denoted by $\mathrm{attn}(X_{\mathrm{q}}, X_{\mathrm{kv}}, T')$, depends on additional positional encoding $P_{\mathrm{q}} \in \mathbb{R}^{d\times N_{\mathrm{q}}}$ and $P_{\mathrm{kv}} \in \mathbb{R}^{d\times N_{\mathrm{kv}}}$. It starts by computing so-called query, key and value embeddings after adding the query and key positional encodings [1]:

$$[Q; K; V] = [T'_1(X_{\mathrm{q}} + P_{\mathrm{q}}); T'_2(X_{\mathrm{kv}} + P_{\mathrm{kv}}); T'_3 X_{\mathrm{kv}}] \tag{5}$$

where T' is the concatenation of T'_1, T'_2, T'_3. The *attention weights* α are then computed based on the softmax of dot products between queries and keys, so that each element of the query sequence attends to all elements of the key-value sequence (i is a query index and j a key-value index):

$$\alpha_{i,j} = \frac{e^{\frac{1}{\sqrt{d'}}Q_i^T K_j}}{Z_i} \quad \text{where } Z_i = \sum_{j=1}^{N_{\mathrm{kv}}} e^{\frac{1}{\sqrt{d'}}Q_i^T K_j}\,. \tag{6}$$

In our case, the positional encodings may be learnt or fixed, but are shared across all attention layers for a given query/key-value sequence, so we do not explicitly write them as parameters of the attention. We give more details on their exact value when describing the encoder and the decoder. The final output is the aggregation of values weighted by attention weights: The i-th row is given by $\mathrm{attn}_i(X_{\mathrm{q}}, X_{\mathrm{kv}}, T') = \sum_{j=1}^{N_{\mathrm{kv}}} \alpha_{i,j} V_j$.

Feed-forward network (FFN) layers The original transformer alternates multi-head attention and so-called FFN layers [8], which are effectively multi-layer 1x1 convolutions, which have Md input and output channels in our case. The FFN we consider is composed of two-layers of 1x1 convolutions with ReLU activations. There is also a residual connection/dropout/layernorm after the two layers, similarly to equation 4.

1.2 Losses

For completeness, we present in detail the losses used in our approach. All losses are normalized by the number of objects inside the batch. Extra care must be taken for distributed training: since each GPU receives a sub-batch, it is not sufficient to normalize by the number of objects in the local batch, since in general the sub-batches are not balanced across GPUs. Instead, it is important to normalize by the total number of objects in all sub-batches.

Box loss Similarly to [7, 5], we use a soft version of Intersection over Union in our loss, together with a ℓ_1 loss on $\hat{b}$:

$$\mathcal{L}_{\text{box}}(b_{\sigma(i)}, \hat{b}_i) = \lambda_{\text{iou}}\mathcal{L}_{\text{iou}}(b_{\sigma(i)}, \hat{b}_i) + \lambda_{\text{L1}}||b_{\sigma(i)} - \hat{b}_i||_1 \,, \tag{7}$$

where $\lambda_{\text{iou}}, \lambda_{\text{L1}} \in \mathbb{R}$ are hyperparameters and $\mathcal{L}_{\text{iou}}(\cdot)$ is the generalized IoU [6]:

$$\mathcal{L}_{\text{iou}}(b_{\sigma(i)}, \hat{b}_i) = 1 - \left(\frac{|b_{\sigma(i)} \cap \hat{b}_i|}{|b_{\sigma(i)} \cup \hat{b}_i|} - \frac{|B(b_{\sigma(i)}, \hat{b}_i) \setminus b_{\sigma(i)} \cup \hat{b}_i|}{|B(b_{\sigma(i)}, \hat{b}_i)|} \right). \tag{8}$$

$|.|$ means "area", and the union and intersection of box coordinates are used as shorthands for the boxes themselves. The areas of unions or intersections are computed by min / max of the linear functions of $b_{\sigma(i)}$ and $\hat{b}_i$, which makes the loss sufficiently well-behaved for stochastic gradients. $B(b_{\sigma(i)}, \hat{b}_i)$ means the largest box containing $b_{\sigma(i)}, \hat{b}_i$ (the areas involving B are also computed based on min / max of linear functions of the box coordinates).

DICE/F-1 loss [3] The DICE coefficient is closely related to the Intersection over Union. If we denote by $\hat{m}$ the raw mask logits prediction of the model, and m the binary target mask, the loss is defined as:

$$\mathcal{L}_{\text{DICE}}(m, \hat{m}) = 1 - \frac{2m\sigma(\hat{m}) + 1}{\sigma(\hat{m}) + m + 1} \tag{9}$$

where σ is the sigmoid function. This loss is normalized by the number of objects.

1.3 Detailed architecture

The detailed description of the transformer used in DETR, with positional encodings passed at every attention layer, is given in Fig. 1. Image features from the CNN backbone are passed through the transformer encoder, together with spatial positional encoding that are added to queries and keys at every multi-head self-attention layer. Then, the decoder receives queries (initially set to zero), output positional encoding (object queries), and encoder memory, and produces the final set of predicted class labels and bounding boxes through multiple multi-head self-attention and decoder-encoder attention. The first self-attention layer in the first decoder layer can be skipped.

Computational complexity Every self-attention in the encoder has complexity $\mathcal{O}(d^2HW + d(HW)^2)$: $\mathcal{O}(d'd)$ is the cost of computing a single query/key/value embeddings (and $Md' = d$), while $\mathcal{O}(d'(HW)^2)$ is the cost of computing the attention weights for one head. Other computations are negligible. In the decoder, each self-attention is in $\mathcal{O}(d^2N + dN^2)$, and cross-attention between encoder and decoder is in $\mathcal{O}(d^2(N + HW) + dNHW)$, which is much lower than the encoder since $N \ll HW$ in practice.

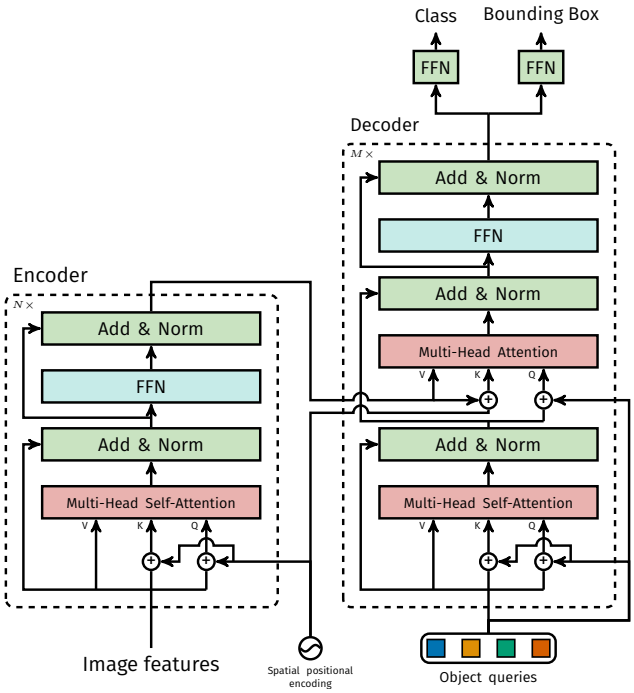


Fig. 1: Architecture of DETR's transformer. Please, see Section 1.3 for details.

FLOPS computation Given that the FLOPS for Faster R-CNN depends on the number of proposals in the image, we report the average number of FLOPS for the first 100 images in the COCO 2017 validation set. We compute the FLOPS with the tool `flop_count_operators` from Detectron2 [9]. We use it without modifications for Detectron2 models, and extend it to take batch matrix multiply (`bmm`) into account for DETR models.

1.4 Training hyperparameters

We train DETR using AdamW [2] with improved weight decay handling, set to 10^{-4}. We also apply gradient clipping, with a maximal gradient norm of 0.1. The backbone and the transformers are treated slightly differently, we now discuss the details for both.

Backbone ImageNet pretrained backbone ResNet-50 is imported from Torchvision, discarding the last classification layer. Backbone batch normalization weights and statistics are frozen during training, following widely adopted practice in object detection. We fine-tune the backbone using learning rate of 10^{-5}. We observe that having the backbone learning rate roughly an order of magnitude smaller than the rest of the network is important to stabilize training, especially in the first few epochs.

Table 1: Effect of encoder size. Each row corresponds to a model with varied number of encoder layers and fixed number of decoder layers. Performance gradually improves with more encoder layers.

#layers	GFLOPS/FPS	#params	AP	AP_{50}	AP_S	AP_M	AP_L
0	76/28	33.4M	36.7	57.4	16.8	39.6	54.2
3	81/25	37.4M	40.1	60.6	18.5	43.8	58.6
6	86/23	41.3M	40.6	61.6	19.9	44.3	60.2
12	95/20	49.2M	41.6	62.1	19.8	44.9	61.9

Transformer We train the transformer with a learning rate of 10^{-4}. Additive dropout of 0.1 is applied after every multi-head attention and FFN before layer normalization. The weights are randomly initialized with Xavier initialization.

Losses We use linear combination of ℓ_1 and GIoU losses for bounding box regression with $\lambda_{\text{L1}} = 5$ and $\lambda_{\text{iou}} = 2$ weights respectively. All models were trained with $N = 100$ decoder query slots.

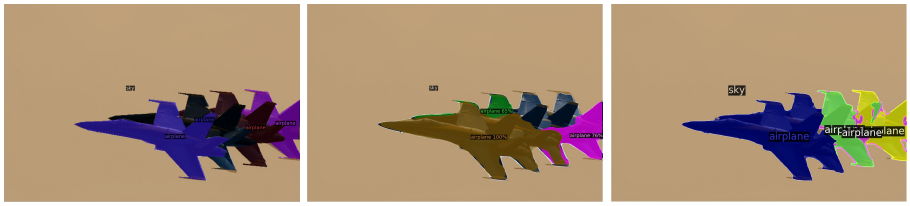
Baseline Our enhanced Faster-RCNN+ baselines use GIoU [6] loss along with the standard ℓ_1 loss for bounding box regression. We performed a grid search to find the best weights for the losses and the final models use only GIoU loss with weights 20 and 1 for box and proposal regression tasks respectively. For the baselines we adopt the same data augmentation as used in DETR and train it with 9× schedule (approximately 109 epochs). All other settings are identical to the same models in the Detectron2 model zoo [9].

Spatial positional encoding Encoder activations are associated with corresponding spatial positions of image features. In our model we use a fixed absolute encoding to represent these spatial positions. We adopt a generalization of the original Transformer [8] encoding to the 2D case [4]. Specifically, for both spatial coordinates of each embedding we independently use $\frac{d}{2}$ sine and cosine functions with different frequencies. We then concatenate them to get the final d channel positional encoding.

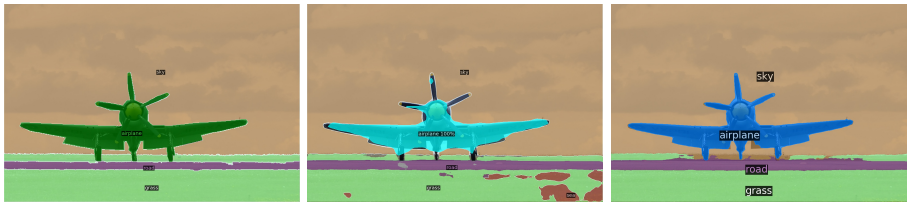
1.5 Additional results

Some extra qualitative results for the panoptic prediction of the DETR-R101 model are shown in Fig.2.

Loss ablations. To evaluate the importance of different components of the matching cost and the loss, we train several models turning them on and off. There are three components to the loss: classification loss, ℓ_1 bounding box distance loss, and GIoU [6] loss. The classification loss is essential for training and cannot be turned off, so we train a model without bounding box distance loss, and a model without the GIoU loss, and compare with baseline, trained with all three losses. Results are presented in table 3. GIoU loss on its own accounts for most of the model performance, losing only 0.7 AP to the baseline with



(a) Failure case with overlapping objects. PanopticFPN misses one plane entirely, while DETR fails to accurately segment 3 of them.



(b) `Things` masks are predicted at full resolution, which allows sharper boundaries than PanopticFPN

Fig. 2: Comparison of panoptic predictions. From left to right: Ground truth, PanopticFPN with ResNet 101, DETR with ResNet 101

Table 2: Results for different positional encodings compared to the baseline (last row), which has fixed sine pos. encodings passed at every attention layer in both the encoder and the decoder. Learned embeddings are shared between all layers. Not using spatial positional encodings leads to a significant drop in AP. Interestingly, passing them in decoder only leads to a minor AP drop. All these models use learned output positional encodings.

spatial pos. enc.		output pos. enc.				
encoder	decoder	decoder	AP	Δ	AP_{50}	Δ
none	none	learned at input	32.8	-7.8	55.2	-6.5
sine at input	sine at input	learned at input	39.2	-1.4	60.0	-1.6
learned at attn.	learned at attn.	learned at attn.	39.6	-1.0	60.7	-0.9
none	sine at attn.	learned at attn.	39.3	-1.3	60.3	-1.4
sine at attn.	sine at attn.	learned at attn.	**40.6**	-	**61.6**	-

combined losses. Using ℓ_1 without GIoU shows poor results. We only studied simple ablations of different losses (using the same weighting every time), but other means of combining them may achieve different results.

Decoder output slot analysis In Fig. 3 we visualize the boxes predicted by different slots for all images in COCO 2017 val set. DETR learns different specialization for each query slot. We observe that each slot has several modes of operation focusing on different areas and box sizes. In particular, all slots have the mode for predicting image-wide boxes (visible as the red dots aligned in the

Table 3: Effect of loss components on AP. We train two models turning off ℓ_1 loss, and GIoU loss, and observe that ℓ_1 gives poor results on its own, but when combined with GIoU improves AP_M and AP_L. Our baseline (last row) combines both losses.

class	ℓ_1	GIoU	AP	Δ	AP_{50}	Δ	AP_S	AP_M	AP_L
✓	✓		35.8	-4.8	57.3	-4.4	13.7	39.8	57.9
✓		✓	39.9	-0.7	**61.6**	0	**19.9**	43.2	57.9
✓	✓	✓	**40.6**	-	**61.6**	-	**19.9**	**44.3**	**60.2**

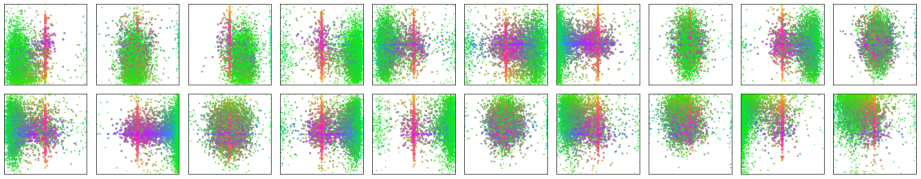


Fig. 3: Visualization of all box predictions on all images from COCO 2017 val set for 20 out of total $N = 100$ prediction slots in DETR decoder. Each box prediction is represented as a point with the coordinates of its center in the 1-by-1 square normalized by each image size. The points are color-coded so that green color corresponds to small boxes, red to large horizontal boxes and blue to large vertical boxes. We observe that each slot learns to specialize on certain areas and box sizes with several operating modes. We note that almost all slots have a mode of predicting large image-wide boxes that are common in COCO dataset.

middle of the plot). We hypothesize that this is related to the distribution of objects in COCO.

Increasing the number of instances By design, DETR cannot predict more objects than it has query slots, i.e. 100 in our experiments. In this section, we analyze the behavior of DETR when approaching this limit. We select a canonical square image of a given class, repeat it on a 10×10 grid, and compute the percentage of instances that are missed by the model. To test the model with less than 100 instances, we randomly mask some of the cells. This ensures that the absolute size of the objects is the same no matter how many are visible. To account for the randomness in the masking, we repeat the experiment 100 times with different masks. The results are shown in Fig.4. The behavior is similar across classes, and while the model detects all instances when up to 50 are visible, it then starts saturating and misses more and more instances. Notably, when the image contains all 100 instances, the model only detects 30 on average, which is less than if the image contains only 50 instances that are all detected. The counter-intuitive behavior of the model is likely because the images and the detections are far from the training distribution.

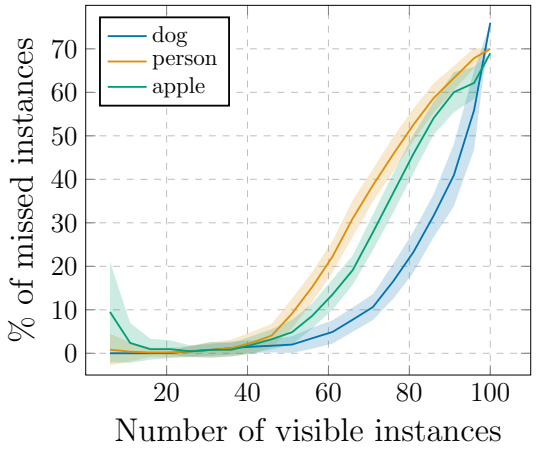


Fig. 4: Analysis of the number of instances of various classes missed by DETR depending on how many are present in the image. We report the mean and the standard deviation. As the number of instances gets close to 100, DETR starts saturating and misses more and more objects

Note that this test is a test of generalization out-of-distribution by design, since there are very few example images with a lot of instances of a single class. It is difficult to disentangle, from the experiment, two types of out-of-domain generalization: the image itself vs the number of object per class. But since few to no COCO images contain only a lot of objects of the same class, this type of experiment represents our best effort to understand whether query objects overfit the label and position distribution of the dataset. Overall, the experiments suggests that the model does not overfit on these distributions since it yields near-perfect detections up to 50 objects.

1.6 PyTorch inference code

To demonstrate the simplicity of the approach, we include inference code with PyTorch and Torchvision libraries in Listing 1. The code runs with Python 3.6+, PyTorch 1.4 and Torchvision 0.5. Note that it does not support batching, hence it is suitable only for inference or training with DistributedDataParallel with one image per GPU. Also note that for clarity, this code uses learnt positional encodings in the encoder instead of fixed, and positional encodings are added to the input only instead of at each transformer layer. Making these changes requires going beyond PyTorch implementation of transformers, which hampers readability. The entire code to reproduce the experiments will be made available before the conference.

```python
import torch
from torch import nn
from torchvision.models import resnet50

class DETR(nn.Module):

    def __init__(self, num_classes, hidden_dim, nheads,
                 num_encoder_layers, num_decoder_layers):
        super().__init__()
        # We take only convolutional layers from ResNet-50 model
        self.backbone = nn.Sequential(*list(resnet50(pretrained=True).children())[:-2])
        self.conv = nn.Conv2d(2048, hidden_dim, 1)
        self.transformer = nn.Transformer(hidden_dim, nheads,
                                          num_encoder_layers, num_decoder_layers)
        self.linear_class = nn.Linear(hidden_dim, num_classes + 1)
        self.linear_bbox = nn.Linear(hidden_dim, 4)
        self.query_pos = nn.Parameter(torch.rand(100, hidden_dim))
        self.row_embed = nn.Parameter(torch.rand(50, hidden_dim // 2))
        self.col_embed = nn.Parameter(torch.rand(50, hidden_dim // 2))

    def forward(self, inputs):
        x = self.backbone(inputs)
        h = self.conv(x)
        H, W = h.shape[-2:]
        pos = torch.cat([
            self.col_embed[:W].unsqueeze(0).repeat(H, 1, 1),
            self.row_embed[:H].unsqueeze(1).repeat(1, W, 1),
        ], dim=-1).flatten(0, 1).unsqueeze(1)
        h = self.transformer(pos + h.flatten(2).permute(2, 0, 1),
                             self.query_pos.unsqueeze(1))
        return self.linear_class(h), self.linear_bbox(h).sigmoid()

detr = DETR(num_classes=91, hidden_dim=256, nheads=8, num_encoder_layers=6, num_decoder_layers=6)
detr.eval()
inputs = torch.randn(1, 3, 800, 1200)
logits, bboxes = detr(inputs)
```

Listing 1: DETR PyTorch inference code. For clarity it uses learnt positional encodings in the encoder instead of fixed, and positional encodings are added to the input only instead of at each transformer layer. Making these changes requires going beyond PyTorch implementation of transformers, which hampers readability. The entire code to reproduce the experiments will be made available before the conference.

1.7 Acknowledgements

We thank Sainbayar Sukhbaatar, Piotr Bojanowski, Natalia Neverova, David Lopez-Paz, Guillaume Lample, Danielle Rothermel, Kaiming He, Ross Girshick, Xinlei Chen, Aishwarya Kamath and the whole Facebook AI Research Paris team for discussions and advices without which this work would not be possible.

References

1. Cordonnier, J.B., Loukas, A., Jaggi, M.: On the relationship between self-attention and convolutional layers. In: ICLR (2020)
2. Loshchilov, I., Hutter, F.: Decoupled weight decay regularization. In: ICLR (2017)
3. Milletari, F., Navab, N., Ahmadi, S.A.: V-net: Fully convolutional neural networks for volumetric medical image segmentation. In: 3DV (2016)
4. Parmar, N., Vaswani, A., Uszkoreit, J., Kaiser, L., Shazeer, N., Ku, A., Tran, D.: Image transformer. In: ICML (2018)
5. Ren, M., Zemel, R.S.: End-to-end instance segmentation with recurrent attention. In: CVPR (2017)
6. Rezatofighi, H., Tsoi, N., Gwak, J., Sadeghian, A., Reid, I., Savarese, S.: Generalized intersection over union. In: CVPR (2019)
7. Romera-Paredes, B., Torr, P.H.S.: Recurrent instance segmentation. In: ECCV (2015)
8. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. In: NeurIPS (2017)
9. Wu, Y., Kirillov, A., Massa, F., Lo, W.Y., Girshick, R.: Detectron2. https://github.com/facebookresearch/detectron2 (2019)