Pose Recognition with Cascade Transformers笔记

- Paper: Pose Recognition with Cascade Transformers
- Code: mlpc-ucsd/PRTR

0. Summary Keywords

- end-to-end
- unified model for multi-person keypoint
- interpretable

1. Introduction

1.1 Why

1.1.1 Motivation

- 基于热图的关键点检测方法需要各种启发式设计，大多数情况下**不能做到端到端**；而基于回归的方法具有较少的不可微分的中间过程，它去除了一些复杂的前处理/后处理步骤、需要的启发式设计更少。(In general, heatmap-based methods achieve higher accuracy but are subject to various **heuristic designs** (not end-to-end mostly), whereas regression-based approaches attain relatively lower accuracy but they have less intermediate non-differentiable steps. It removes complex pre/post-processing procedures and requires fewer heuristic designs compared with existing heatmap-based approaches.)

1.1.2 Challenges

- 姿态估计(关键点检测)存在这些难点：姿态/形状变化大、人体自遮挡、外观变化大、背景杂乱等方面(The difficulty lies in various aspects such as large pose/shape variation, inter-person and self occlusion, large appearance variation, and background clutter.)（本论文并没有针对这些难点提出特别的处理方案。）

1.2 What

- **cascade Transformers**:
    - two-stage process: （我更倾向于称为two-model process；Faster-RCNN是一个二阶段的目标检测模型，它是1个模型；PRTR论文中的“two-stage process”是2个模型：人体检测和人体关键点检测）
    - sequential end-to-end process:（1个模型2个任务一起训练：人体检测和人体关键点检测）

1.3 How

- 本文使用DETR的基本框架，采用逐步回归的方式生成关键点。

1.4 Contributions

- 提出了一种基于回归的关键点检测方法**Pose Regression TRansformer(PRTR)**，通过搭建级联Transformers网络结构，这种方法基于一种通用的端到端目标检测方法**DETR**。这种方法能同时捕捉关键点的空间和外观特征。(We propose a regression-based human pose recognition method by building cascade Transformers, based on a general-purpose object detector, end-to-end object detection Transformer (DETR) [3]. Our method, named pose recognition Transformer (PRTR), enjoys the tokenized representation in Transformers with layers of selfattention to capture the joint spatial and appearance modeling for the keypoints.)
- 两种方案：two-stage、端到端。(Two types of cascade Transformers have been developed: 1). a two-stage one with the second Transformer taking image patches detected from the first Transformer, as shown in Figure 2; and 2). a sequential one using spatial Transformer network (STN) [16] to create an end-to-end framework, shown in Figure 3.)
- 可视化关键点查询的分布、揭开Transformer内部过程中检测逐渐细化的过程，为揭示Transformer 解码器内部机制开辟了道路。(We visualize the distribution of keypoint queries in various aspects to unfold the internal process of the Transformer for the gradual refinement of the detection.)

1.5 Future works

- 更强的骨干网络(more powerful backbone networks)
- 将基于回归的人体检测和关键点检测以更灵活的方式结合(combine regression-based human detection and pose recognition in a more flexible manner)

2. Method

- two stage

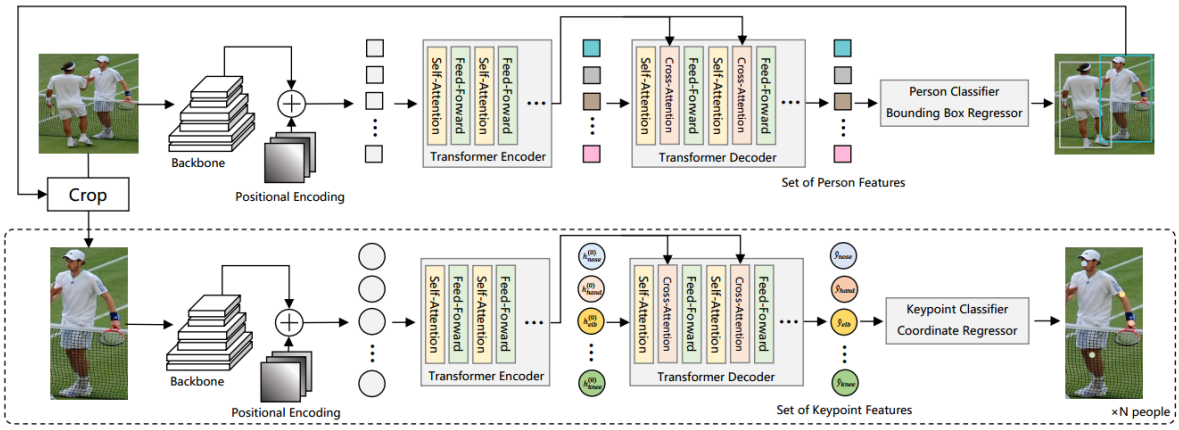

Figure 2: The architecture of Pose Recognition with TRansformer (**PRTR**), **two-stage variant**. First, using whole-picture image feature and absolute positional encoding, a person-detection Transformer detects people in the image with a set of learned person queries. After filtering background queries, we crop the original image with predicted boxes. Cropped images are fed into a keypoint-detection Transformer, together with positional encoding relative to corresponding bounding boxes. Finally, we read out J keypoints from a larger set of keypoint queries by Hungarian algorithm. The keypoint-detection Transformer processes all the non-background keypoint proposals in a vectorized way. $h_{\_}^{(0)}$ denotes hypotheses (queries), the feature vectors to be refined to final predictions, $\hat{y}_{\_}$, through Transformer decoder.

- end-to-end

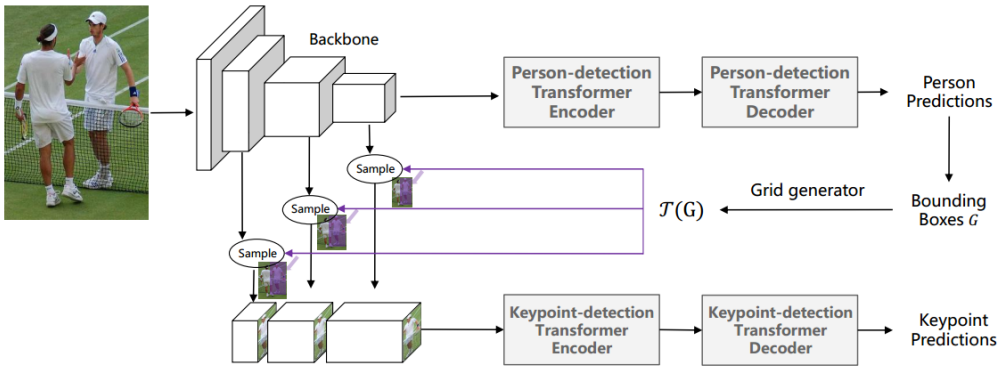

Figure 3: The architecture of Pose Recognition with TRansformer (**PRTR**), **end-to-end variant**. For end-to-end learning, instead of cropping at RGB image level, we apply differentiable bilinear sampling on multiple layers of backbone-generated features to provide *zoomed-in* and *multi-level* feature for keypoint-detection Transformer.

可微双线性抽样

3. Code=f(method) Two-stage

- 以COCO train2017 dataset为例:

```
python tools/train.py \
    --cfg experiments/coco/transformer/w32_384x288_adamw_lr1e-4.yaml
```

- 以two_stage/experiments/coco/transformer/w32_384x288_adamw_lr1e-4.yaml为例:

```
AUTO_RESUME: true
CUDNN:
  BENCHMARK: true
  DETERMINISTIC: false
  ENABLED: true
DATA_DIR: ''
GPUS: (0,1,2,3)
OUTPUT_DIR: 'output'
LOG_DIR: 'log'
WORKERS: 8
PRINT_FREQ: 50

DATASET:
  COLOR_RGB: true
  DATA_FORMAT: jpg
  DATASET: 'coco'
  ROOT: 'data/coco/'
  TEST_SET: 'val2017'
  TRAIN_SET: 'train2017'
  FLIP: true
  ROT_FACTOR: 40
  SCALE_FACTOR: 0.3
MODEL:
  INIT_WEIGHTS: true
  NAME: 'pose_transformer'
  PRETRAINED: 'models/pytorch/imagenet/hrnetv2_w32_imagenet_pretrained.pth'
  TARGET_TYPE: 'coord'  ##应用坐标回归
  HEATMAP_SIZE:
  - 72
  - 96
  IMAGE_SIZE:
  - 288
```

```
  - 384
NUM_JOINTS: 17
EXTRA:
  STAGE1:
    NUM_MODULES: 1
    NUM_RANCHES: 1
    BLOCK: BOTTLENECK
    NUM_BLOCKS:
    - 4
    NUM_CHANNELS:
    - 64
    FUSE_METHOD: SUM
  STAGE2:
    NUM_MODULES: 1
    NUM_BRANCHES: 2
    BLOCK: BASIC
    NUM_BLOCKS:
    - 4
    - 4
    NUM_CHANNELS:
    - 32
    - 64
    FUSE_METHOD: SUM
  STAGE3:
    NUM_MODULES: 4
    NUM_BRANCHES: 3
    BLOCK: BASIC
    NUM_BLOCKS:
    - 4
    - 4
    - 4
    NUM_CHANNELS:
    - 32
    - 64
    - 128
    FUSE_METHOD: SUM
  STAGE4:
    NUM_MODULES: 3
    NUM_BRANCHES: 4
    BLOCK: BASIC
    NUM_BLOCKS:
    - 4
    - 4
    - 4
    - 4
    NUM_CHANNELS:
    - 32
    - 64
    - 128
    - 256
    FUSE_METHOD: SUM
  ENC_LAYERS: 6  ##编码网络层数
  DEC_LAYERS: 6  ##解码网络层数
  DIM_FEEDFORWARD: 2048  ##FFN网络
  DROPOUT: 0.0
  NHEADS: 8  ##Multi-Head Attention的头数
  NUM_QUERIES: 100  ##keypoint queries
  HIDDEN_DIM: 256
```

```
    PRE_NORM: false
    AUX_LOSS: true
    NUM_LAYERS: hrnet
    DILATION: false
    POS_EMBED_METHOD: 'sine'
    EOS_COEF: 0.1
    KPT_LOSS_COEF: 5.0
LOSS:
  USE_TARGET_WEIGHT: true
TRAIN:
  BATCH_SIZE_PER_GPU: 16
  SHUFFLE: true
  BEGIN_EPOCH: 0
  END_EPOCH: 200
  OPTIMIZER: 'adamw'
  LR: 1.0e-4
  LR_BACKBONE: 1.0e-4
  CLIP_MAX_NORM: 0.1
  LR_FACTOR: 0.5
  LR_STEP:
  - 70
  - 95
  WD: 0.0001
  GAMMA1: 0.99
  GAMMA2: 0.0
  MOMENTUM: 0.9
  NESTEROV: false
TEST:
  BATCH_SIZE_PER_GPU: 32
  COCO_BBOX_FILE:
'data/coco/person_detection_results/COCO_val2017_detr_detections.json'
  BBOX_THRE: 1.0
  IMAGE_THRE: 0.0
  IN_VIS_THRE: 0.2
  MODEL_FILE: ''
  NMS_THRE: 1.0
  OKS_THRE: 0.9
  FLIP_TEST: true
  POST_PROCESS: true
  SHIFT_HEATMAP: true
  USE_GT_BBOX: true
DEBUG:
  DEBUG: true
  SAVE_BATCH_IMAGES_GT: true
  SAVE_BATCH_IMAGES_PRED: true
  SAVE_HEATMAPS_GT: false
  SAVE_HEATMAPS_PRED: false
```

two_stage/lib/core/function.py

```
for i, (input, target, target_weight, meta) in enumerate(train_loader):
    # measure data loading time
    data_time.update(time.time() - end)

    # compute output
    outputs = model(input)
```

3.1 Input

two_stage/lib/dataset/JointsDataset.py

```
##使用坐标，而不是高斯热图
else:
    target = np.array(joints[:, 0:2] / self.image_size, dtype=np.float32)
```

3.2 Process

Build model

two_stage/lib/models/pose_transformer.py

```
def get_pose_net(cfg, is_train, **kwargs):
    extra = cfg.MODEL.EXTRA

    transformer = build_transformer(hidden_dim=extra.HIDDEN_DIM,
dropout=extra.DROPOUT, nheads=extra.NHEADS,
dim_feedforward=extra.DIM_FEEDFORWARD,
                                    enc_layers=extra.ENC_LAYERS,
dec_layers=extra.DEC_LAYERS, pre_norm=extra.PRE_NORM)
    pretrained = is_train and cfg.MODEL.INIT_WEIGHTS
    backbone = build_backbone(cfg, pretrained)
    model = PoseTransformer(cfg, backbone, transformer, **kwargs)

    return model
```

two_stage/lib/models/pose_transformer.py

```
class PoseTransformer(nn.Module):

    def __init__(self, cfg, backbone, transformer, **kwargs):
        super(PoseTransformer, self).__init__()
        extra = cfg.MODEL.EXTRA
        self.num_queries = extra.NUM_QUERIES
        self.num_classes = cfg.MODEL.NUM_JOINTS
        self.transformer = transformer
        self.backbone = backbone
        hidden_dim = transformer.d_model
        self.class_embed = nn.Linear(hidden_dim, self.num_classes + 1)
        self.kpt_embed = MLP(hidden_dim, hidden_dim, 2, 3)
        self.query_embed = nn.Embedding(self.num_queries, hidden_dim)
        self.input_proj = nn.Conv2d(self.backbone.num_channels[0], hidden_dim,
1)

        self.aux_loss = extra.AUX_LOSS

    def forward(self, x):
        ##Backbone
        src, pos = self.backbone(x)
        ##Transformer Encoder-Decoder
        hs = self.transformer(self.input_proj(src[-1]), None,
                              self.query_embed.weight, pos[-1])[0]
```

```
        ##Keypoint Classifier
        outputs_class = self.class_embed(hs)
        ##Keypoint Coordinate Regressor
        outputs_coord = self.kpt_embed(hs).sigmoid()

        out = {'pred_logits': outputs_class[-1],
               'pred_coords': outputs_coord[-1]}
        if self.aux_loss:
            out['aux_outputs'] = self._set_aux_loss(
                outputs_class, outputs_coord)
        return out
```

Backbone

two_stage/lib/models/backbone.py

```
def build_backbone(cfg, pretrained):
    extra = cfg.MODEL.EXTRA
    num_layers = extra.NUM_LAYERS
    if type(num_layers) == str:
        name = num_layers
    else:
        name = f'resnet{num_layers}'
    position_embedding = build_position_encoding(
        extra.HIDDEN_DIM, extra.POS_EMBED_METHOD)
    return_interm_layers = hasattr(
        extra, 'NUM_FEATURE_LEVELS') and extra.NUM_FEATURE_LEVELS > 1
    if name.startswith('resnet'):
        backbone = ResNetBackbone(
            name, train_backbone=True,
return_interm_layers=return_interm_layers, pretrained=pretrained,
dilation=extra.DILATION)
    elif name == 'hrnet':
        backbone = HRNetBackbone(
            cfg, pretrained=pretrained,
return_interm_layers=return_interm_layers)
    else:
        raise NotImplementedError(f'Unsupported backbone type: {name}')
    model = Joiner(backbone, position_embedding)
    return model
```

Transformer

two_stage/lib/models/transformer.py

```
def build_transformer(**kwargs):
    return Transformer(
        d_model=kwargs['hidden_dim'],
        dropout=kwargs['dropout'],
        nhead=kwargs['nheads'],
        dim_feedforward=kwargs['dim_feedforward'],
        num_encoder_layers=kwargs['enc_layers'],
        num_decoder_layers=kwargs['dec_layers'],
        normalize_before=kwargs['pre_norm'],
        return_intermediate_dec=True,
    )
```

two_stage/lib/models/transformer.py

```
class Transformer(nn.Module):

    def __init__(self, d_model=512, nhead=8, num_encoder_layers=6,
                 num_decoder_layers=6, dim_feedforward=2048, dropout=0.1,
                 activation="relu", normalize_before=False,
                 return_intermediate_dec=False):
        super().__init__()

        encoder_layer = TransformerEncoderLayer(d_model, nhead, dim_feedforward,
                                                dropout, activation,
normalize_before)
        encoder_norm = nn.LayerNorm(d_model) if normalize_before else None
        self.encoder = TransformerEncoder(encoder_layer, num_encoder_layers,
encoder_norm)

        decoder_layer = TransformerDecoderLayer(d_model, nhead, dim_feedforward,
                                                dropout, activation,
normalize_before)
        decoder_norm = nn.LayerNorm(d_model)
        self.decoder = TransformerDecoder(decoder_layer, num_decoder_layers,
decoder_norm,

 return_intermediate=return_intermediate_dec)

        self._reset_parameters()

        self.d_model = d_model
        self.nhead = nhead

    def _reset_parameters(self):
        for p in self.parameters():
            if p.dim() > 1:
                nn.init.xavier_uniform_(p)

    def forward(self, src, mask, query_embed, pos_embed):
        # flatten NxCxHxW to HWxNxC
        bs, c, h, w = src.shape
        ##collapse the spatial dimensions of z0 into one dimension(DETR)
        src = src.flatten(2).permute(2, 0, 1)
        pos_embed = pos_embed.flatten(2).permute(2, 0, 1)
        query_embed = query_embed.unsqueeze(1).repeat(1, bs, 1)
        if mask is not None:
            mask = mask.flatten(1)

        tgt = torch.zeros_like(query_embed)
        memory = self.encoder(src, src_key_padding_mask=mask, pos=pos_embed)
        hs = self.decoder(tgt, memory, memory_key_padding_mask=mask,
                          pos=pos_embed, query_pos=query_embed)
        return hs.transpose(1, 2), memory.permute(1, 2, 0).view(bs, c, h, w)
```

Transformer Encoder

two_stage/lib/models/transformer.py

```
class TransformerEncoder(nn.Module):

    def __init__(self, encoder_layer, num_layers, norm=None):
        super().__init__()
        self.layers = _get_clones(encoder_layer, num_layers)
        self.num_layers = num_layers
        self.norm = norm

    def forward(self, src,
                mask: Optional[Tensor] = None,
                src_key_padding_mask: Optional[Tensor] = None,
                pos: Optional[Tensor] = None):
        output = src

        for layer in self.layers:
            output = layer(output, src_mask=mask,
                           src_key_padding_mask=src_key_padding_mask, pos=pos)

        if self.norm is not None:
            output = self.norm(output)

        return output
```

two_stage/lib/models/transformer.py

```
class TransformerEncoderLayer(nn.Module):

    def __init__(self, d_model, nhead, dim_feedforward=2048, dropout=0.1,
                 activation="relu", normalize_before=False):
        super().__init__()
        self.self_attn = nn.MultiheadAttention(d_model, nhead, dropout=dropout)
        # Implementation of Feedforward model
        self.linear1 = nn.Linear(d_model, dim_feedforward)
        self.dropout = nn.Dropout(dropout)
        self.linear2 = nn.Linear(dim_feedforward, d_model)

        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)

        self.activation = _get_activation_fn(activation)
        self.normalize_before = normalize_before

    def with_pos_embed(self, tensor, pos: Optional[Tensor]):
        return tensor if pos is None else tensor + pos

    def forward_post(self,
                     src,
                     src_mask: Optional[Tensor] = None,
                     src_key_padding_mask: Optional[Tensor] = None,
                     pos: Optional[Tensor] = None):
        q = k = self.with_pos_embed(src, pos)
```

```
        src2 = self.self_attn(q, k, value=src, attn_mask=src_mask,
                              key_padding_mask=src_key_padding_mask)[0]
        src = src + self.dropout1(src2)
        src = self.norm1(src)
        src2 = self.linear2(self.dropout(self.activation(self.linear1(src))))
        src = src + self.dropout2(src2)
        src = self.norm2(src)
        return src

    def forward_pre(self, src,
                    src_mask: Optional[Tensor] = None,
                    src_key_padding_mask: Optional[Tensor] = None,
                    pos: Optional[Tensor] = None):
        src2 = self.norm1(src)
        q = k = self.with_pos_embed(src2, pos)
        src2 = self.self_attn(q, k, value=src2, attn_mask=src_mask,
                              key_padding_mask=src_key_padding_mask)[0]
        src = src + self.dropout1(src2)
        src2 = self.norm2(src)
        src2 = self.linear2(self.dropout(self.activation(self.linear1(src2))))
        src = src + self.dropout2(src2)
        return src

    def forward(self, src,
                src_mask: Optional[Tensor] = None,
                src_key_padding_mask: Optional[Tensor] = None,
                pos: Optional[Tensor] = None):
        if self.normalize_before:
            return self.forward_pre(src, src_mask, src_key_padding_mask, pos)
        return self.forward_post(src, src_mask, src_key_padding_mask, pos)
```

Transformer Decoder

two_stage/lib/models/transformer.py

```
class TransformerDecoder(nn.Module):

    def __init__(self, decoder_layer, num_layers, norm=None,
return_intermediate=False):
        super().__init__()
        self.layers = _get_clones(decoder_layer, num_layers)
        self.num_layers = num_layers
        self.norm = norm
        self.return_intermediate = return_intermediate

    def forward(self, tgt, memory,
                tgt_mask: Optional[Tensor] = None,
                memory_mask: Optional[Tensor] = None,
                tgt_key_padding_mask: Optional[Tensor] = None,
                memory_key_padding_mask: Optional[Tensor] = None,
                pos: Optional[Tensor] = None,
                query_pos: Optional[Tensor] = None):
        output = tgt

        intermediate = []

        for layer in self.layers:
```

```
            output = layer(output, memory, tgt_mask=tgt_mask,
                           memory_mask=memory_mask,
                           tgt_key_padding_mask=tgt_key_padding_mask,
                           memory_key_padding_mask=memory_key_padding_mask,
                           pos=pos, query_pos=query_pos)
            if self.return_intermediate:
                intermediate.append(self.norm(output))

        if self.norm is not None:
            output = self.norm(output)
            if self.return_intermediate:
                intermediate.pop()
                intermediate.append(output)

        if self.return_intermediate:
            return torch.stack(intermediate)

        return output.unsqueeze(0)
```

two_stage/lib/models/transformer.py

```
class TransformerDecoderLayer(nn.Module):

    def __init__(self, d_model, nhead, dim_feedforward=2048, dropout=0.1,
                 activation="relu", normalize_before=False):
        super().__init__()
        self.self_attn = nn.MultiheadAttention(d_model, nhead, dropout=dropout)
        self.multihead_attn = nn.MultiheadAttention(d_model, nhead,
dropout=dropout)
        # Implementation of Feedforward model
        self.linear1 = nn.Linear(d_model, dim_feedforward)
        self.dropout = nn.Dropout(dropout)
        self.linear2 = nn.Linear(dim_feedforward, d_model)

        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)
        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)
        self.dropout3 = nn.Dropout(dropout)

        self.activation = _get_activation_fn(activation)
        self.normalize_before = normalize_before

    def with_pos_embed(self, tensor, pos: Optional[Tensor]):
        return tensor if pos is None else tensor + pos

    def forward_post(self, tgt, memory,
                     tgt_mask: Optional[Tensor] = None,
                     memory_mask: Optional[Tensor] = None,
                     tgt_key_padding_mask: Optional[Tensor] = None,
                     memory_key_padding_mask: Optional[Tensor] = None,
                     pos: Optional[Tensor] = None,
                     query_pos: Optional[Tensor] = None):
        q = k = self.with_pos_embed(tgt, query_pos)
        tgt2 = self.self_attn(q, k, value=tgt, attn_mask=tgt_mask,
                              key_padding_mask=tgt_key_padding_mask)[0]
```

```
        tgt = tgt + self.dropout1(tgt2)
        tgt = self.norm1(tgt)
        tgt2 = self.multihead_attn(query=self.with_pos_embed(tgt, query_pos),
                                   key=self.with_pos_embed(memory, pos),
                                   value=memory, attn_mask=memory_mask,
                                   key_padding_mask=memory_key_padding_mask)[0]
        tgt = tgt + self.dropout2(tgt2)
        tgt = self.norm2(tgt)
        tgt2 = self.linear2(self.dropout(self.activation(self.linear1(tgt))))
        tgt = tgt + self.dropout3(tgt2)
        tgt = self.norm3(tgt)
        return tgt

    def forward_pre(self, tgt, memory,
                    tgt_mask: Optional[Tensor] = None,
                    memory_mask: Optional[Tensor] = None,
                    tgt_key_padding_mask: Optional[Tensor] = None,
                    memory_key_padding_mask: Optional[Tensor] = None,
                    pos: Optional[Tensor] = None,
                    query_pos: Optional[Tensor] = None):
        tgt2 = self.norm1(tgt)
        q = k = self.with_pos_embed(tgt2, query_pos)
        tgt2 = self.self_attn(q, k, value=tgt2, attn_mask=tgt_mask,
                              key_padding_mask=tgt_key_padding_mask)[0]
        tgt = tgt + self.dropout1(tgt2)
        tgt2 = self.norm2(tgt)
        tgt2 = self.multihead_attn(query=self.with_pos_embed(tgt2, query_pos),
                                   key=self.with_pos_embed(memory, pos),
                                   value=memory, attn_mask=memory_mask,
                                   key_padding_mask=memory_key_padding_mask)[0]
        tgt = tgt + self.dropout2(tgt2)
        tgt2 = self.norm3(tgt)
        tgt2 = self.linear2(self.dropout(self.activation(self.linear1(tgt2))))
        tgt = tgt + self.dropout3(tgt2)
        return tgt

    def forward(self, tgt, memory,
                tgt_mask: Optional[Tensor] = None,
                memory_mask: Optional[Tensor] = None,
                tgt_key_padding_mask: Optional[Tensor] = None,
                memory_key_padding_mask: Optional[Tensor] = None,
                pos: Optional[Tensor] = None,
                query_pos: Optional[Tensor] = None):
        if self.normalize_before:
            return self.forward_pre(tgt, memory, tgt_mask, memory_mask,
                                    tgt_key_padding_mask,
memory_key_padding_mask, pos, query_pos)
        return self.forward_post(tgt, memory, tgt_mask, memory_mask,
                                 tgt_key_padding_mask, memory_key_padding_mask,
pos, query_pos)
```

3.3 Output

- 神经网络的输出是output

two_stage/lib/core/function.py

```
outputs = model(input)
output = outputs
```

two_stage/lib/models/pose_transformer.py

```
out = {'pred_logits': outputs_class[-1],  ##out['pred_logits']的维度是[BS, NUM_QUERIES, NUM_JOINTS+1]，表示query和关键点的匹配程度吗？
       'pred_coords': outputs_coord[-1]}  ##out['pred_coords']的维度是[BS, NUM_QUERIES, 2]
```

疑问：output与output query是什么关系呢？output query维度是多少？作用是什么？

- output query

two_stage/lib/models/pose_transformer.py

```
self.query_embed = nn.Embedding(self.num_queries, hidden_dim)
 ##cfg.MODEL.EXTRA.NUM_QUERIES为100，cfg.MODEL.EXTRA.HIDDEN_DIM为256
```

疑问：NUM_QUERIES是100，NUM_JOINTS是17，如何匹配的呢？匈牙利算法？

two_stage/lib/core/function.py

```
def validate(config, val_loader, val_dataset, model, criterion, output_dir,
             tb_log_dir, writer_dict=None):
    batch_time = AverageMeter()
    metrics_dict = {}

    def add_to_metrics(name, val, cnt):
        if name not in metrics_dict.keys():
            metrics_dict[name] = AverageMeter()
        metrics_dict[name].update(val, cnt)

    # switch to evaluate mode
    model.eval()
    criterion.eval()

    num_samples = len(val_dataset)
    all_preds = np.zeros(
        (num_samples, config.MODEL.NUM_JOINTS, 3),
        dtype=np.float32
    )
    all_boxes = np.zeros((num_samples, 6))
    image_path = []
    filenames = []
    imgnums = []
    idx = 0
    image_size = np.array(config.MODEL.IMAGE_SIZE)
    with torch.no_grad():
```

```
        end = time.time()

        for i, (input, target, target_weight, meta) in enumerate(val_loader):
            num_images = input.size(0)
            # compute output
            outputs = model(input)
            target = target.cuda(non_blocking=True)
            target_weight = target_weight.cuda(non_blocking=True)

            output = outputs

            loss_dict, _ = criterion(outputs, target, target_weight)
            weight_dict = criterion.weight_dict
            loss = sum(loss_dict[k] * weight_dict[k]
                    for k in loss_dict.keys() if k in weight_dict)

            bs = input.size(0)
            for k, v in loss_dict.items():
                add_to_metrics(f'{k}_unscaled', v.item(), bs)
                if k in weight_dict:
                    add_to_metrics(k, (v * weight_dict[k]).item(), bs)

            # measure elapsed time
            batch_time.update(time.time() - end)
            end = time.time()

            c = meta['center'].numpy()
            s = meta['scale'].numpy()
            score = meta['score'].numpy()

            ##preds的维度是[BS, NUM_JOINTS, 2]
            ##maxvals的维度是[BS, NUM_JOINTS, 1]
            ##pred的维度是[BS, NUM_JOINTS, 2]
            preds, maxvals, pred = get_final_preds_match(
                config, output, c, s
            )
```

two_stage/lib/core/inference.py

```
def get_final_preds_match(config, outputs, center, scale, flip_pairs=None):
    pred_logits = outputs['pred_logits'].detach().cpu()  ##out['pred_logits']的维度是[BS, NUM_QUERIES, NUM_JOINTS+1]
    pred_coords = outputs['pred_coords'].detach().cpu()  ##out['pred_coords']的维度是[BS, NUM_QUERIES, 2]

    num_joints = pred_logits.shape[-1] - 1

    if config.TEST.INCLUDE_BG_LOGIT:
        prob = F.softmax(pred_logits, dim=-1)[..., :-1]
    else:
        prob = F.softmax(pred_logits[..., :-1], dim=-1)  ##prob的维度是[BS, NUM_QUERIES, NUM_JOINTS]

    score_holder = []
    coord_holder = []
    orig_coord = []
    for b, C in enumerate(prob):
```

```
        _, query_ind = linear_sum_assignment(-C.transpose(0, 1)) # Cost Matrix:
[17, N]
        score = prob[b, query_ind, list(np.arange(num_joints))][...,
None].numpy()
        pred_raw = pred_coords[b, query_ind].numpy()
        if flip_pairs is not None:
            pred_raw, score = fliplr_joints(pred_raw, score, 1, flip_pairs,
pixel_align=False, is_vis_logit=True)
        # scale to the whole patch
        pred_raw *= np.array(config.MODEL.IMAGE_SIZE)
        # transform back w.r.t. the entire img
        pred = transform_preds(pred_raw, center[b], scale[b],
config.MODEL.IMAGE_SIZE)
        orig_coord.append(pred_raw)
        score_holder.append(score)
        coord_holder.append(pred)

    matched_score = np.stack(score_holder)
    matched_coord = np.stack(coord_holder)

    return matched_coord, matched_score, np.stack(orig_coord)
```

疑问：为什么pred_logits的每一行都基本一样呢？且前4列比第5列小这么多呢？

4. Code=f(method) Sequential

- 待深入研究后再补充

参考资料

1. 搞懂 Vision Transformer 原理和代码，看这篇技术综述就够了
2. Pose Recognition with Cascade Transformers 论文笔记

更新于2021-05-28