Config instructions for the PSENet text detection module in MMOCR.

1. Training

```
tools/train.py implements basic training services.
# example:
CUDA_VISIBLE_DEVICES=-1 python tools/train.py configs/textdet/
psenet/_base_psenet_resnet50_fpnf.py
```

```
model = dict(
    type='PSENet',
    backbone=dict(
        type='mmdet.ResNet',
        depth=50,
        num_stages=4,
        out_indices=(0, 1, 2, 3),
        frozen_stages=-1,
        norm_cfg=dict(type='SyncBN', requires_grad=True),
        init_cfg=dict(type='Pretrained', checkpoint='torchvision://resnet50'),
        norm_eval=True,
        style='caffe'),
    neck=dict(
        type='FPNF',
        in_channels=[256, 512, 1024, 2048],
        out_channels=256,
        fusion_type='concat'),
    det_head=dict(
        type='PSEHead',
        in_channels=[256],
        hidden_dim=256,
        out_channel=7,
        module_loss=dict(type='PSEModuleLoss'),
        postprocessor=dict(type='PSEPostprocessor', text_repr_type='poly')),
    data_preprocessor=dict(
        type='TextDetDataPreprocessor',
        mean=[123.675, 116.28, 103.53],
        std=[58.395, 57.12, 57.375],
        bgr_to_rgb=True,
        pad_size_divisor=32))

train_pipeline = [
    dict(type='LoadImageFromFile', color_type='color_ignore_orientation'),
    dict(
        type='LoadOCRAnnotations',
        with_polygon=True,
        with_bbox=True,
        with_label=True),
    dict(
        type='TorchVisionWrapper',
        op='ColorJitter',
        brightness=32.0 / 255,
        saturation=0.5),
    dict(type='FixInvalidPolygon'),
    dict(type='ShortScaleAspectJitter', short_size=736, scale_divisor=32),
    dict(type='RandomFlip', prob=0.5, direction='horizontal'),
    dict(type='RandomRotate', max_angle=10),
    dict(type='TextDetRandomCrop', target_size=(736, 736)),
    dict(type='Pad', size=(736, 736)),
    dict(
        type='PackTextDetInputs',
        meta_keys=('img_path', 'ori_shape', 'img_shape', 'scale_factor'))
]

test_pipeline = [
    dict(type='LoadImageFromFile', color_type='color_ignore_orientation'),
    dict(type='Resize', scale=(2240, 2240), keep_ratio=True),
    dict(
        type='LoadOCRAnnotations',
        with_polygon=True,
        with_bbox=True,
        with_label=True),
    dict(
        type='PackTextDetInputs',
        meta_keys=('img_path', 'ori_shape', 'img_shape', 'scale_factor'))
]
```

This code is a configuration file for the Text Detection task, using the mmdetection library. Here is some interpretation of the code:

1. The `model` dictionary defines the overall structure of the model, including components such as backbone, neck (may be used for feature fusion), detection head (det_head), and data preprocessor (data_preprocessor). The model type is 'PSENet' and the backbone network used is ResNet-50.

2. `backbone` defines the configuration of the backbone network, including network type, depth, number of stages, output channel index, freezing stage, normalization configuration, initialization configuration, etc.

3. `neck` defines the configuration of the neck, including neck type, input channel, output channel, fusion type, etc.

4. `det_head` defines the configuration of the detection head, including head type, input channel, hidden dimension, output channel, module loss type, post-processor type, etc.

5. `data_preprocessor` defines the configuration of the data preprocessor, including type, mean, standard deviation, color channel conversion, etc.

6. `train_pipeline` defines the data processing process during training, including loading images from files, loading OCR annotations, color enhancement, repairing invalid polygons, short scale and aspect ratio jitter, random horizontal flipping, random rotation, random cropping, and filling. , package text detection input and other steps.

7. `test_pipeline` defines the data processing process during testing, including steps such as loading images from files, resizing, loading OCR annotations, and packaging text detection input.

2. Model structure

This section will expand on the structure of the model in the type='PSENet' field and how to modify it. The neck, det_head, and data_preprocessor in the text detection model structure in mmocr are all integrated in mmocr/models/textdet.

2.1 backbone

- `type='mmdet.ResNet'`: Use the ResNet implementation in the mmdetection library as the backbone network.

- `depth=50`: Use ResNet with a depth of 50.

- `num_stages=4`: The number of stages in ResNet is 4, which means it includes four stages of ResNet.

- `out_indices=(0, 1, 2, 3)`: Specifies the stage indices for extracting features from the backbone network, here are all four stages.

- `frozen_stages=-1`: The number of frozen stages, -1 means that all stages are not frozen and training can be carried out.

- `norm_cfg=dict(type='SyncBN', requires_grad=True)`: Configuration of normalization layer, using Sync Batch Normalization and allowing gradient calculation.

- `init_cfg=dict(type='Pretrained', checkpoint='torchvision://resnet50')`: Parameter initialization configuration, initialized using pre-trained ResNet-50 model parameters.

- `norm_eval=True`: Use normalization layer also in evaluation mode.

- `style='caffe'`: Use 'caffe' style weight initialization, which is related to the initialization style of model training.

This code configures a ResNet-50 backbone network, normalized using Sync Batch Normalization, and initialized with pre-trained ResNet-50 weights.

2.2 neck

```python
# Copyright (c) OpenMMLab. All rights reserved.
from typing import Dict, List, Optional, Union

import torch
import torch.nn.functional as F
from mmcv.cnn import ConvModule
from mmengine.model import BaseModule, ModuleList
from torch import Tensor

from mmocr.registry import MODELS


@MODELS.register_module()
class FPNF(BaseModule):
    """FPN-like fusion module in Shape Robust Text Detection with Progressive
    Scale Expansion Network.

    Args:
        in_channels (list[int]): A list of number of input channels.
            Defaults to [256, 512, 1024, 2048].
        out_channels (int): The number of output channels.
            Defaults to 256.
        fusion_type (str): Type of the final feature fusion layer. Available
            options are "concat" and "add". Defaults to "concat".
        init_cfg (dict or list[dict], optional): Initialization configs.
            Defaults to
            dict(type='Xavier', layer='Conv2d', distribution='uniform')
    """

    def __init__(
        self,
        in_channels: List[int] = [256, 512, 1024, 2048],
        out_channels: int = 256,
        fusion_type: str = 'concat',
        init_cfg: Optional[Union[Dict, List[Dict]]] = dict(
            type='Xavier', layer='Conv2d', distribution='uniform')
    ) -> None:
        super().__init__(init_cfg=init_cfg)
        conv_cfg = None
        norm_cfg = dict(type='BN')
        act_cfg = dict(type='ReLU')

        self.in_channels = in_channels
        self.out_channels = out_channels
        self.lateral_convs = ModuleList()
        self.fpn_convs = ModuleList()
        self.backbone_end_level = len(in_channels)
        for i in range(self.backbone_end_level):
            l_conv = ConvModule(
                in_channels[i],
                out_channels,
                1,
                conv_cfg=conv_cfg,
                norm_cfg=norm_cfg,
                act_cfg=act_cfg,
                inplace=False)
            self.lateral_convs.append(l_conv)

            if i < self.backbone_end_level - 1:
                fpn_conv = ConvModule(
                    out_channels,
                    out_channels,
                    3,
                    padding=1,
                    conv_cfg=conv_cfg,
                    norm_cfg=norm_cfg,
                    act_cfg=act_cfg,
                    inplace=False)
                self.fpn_convs.append(fpn_conv)

        self.fusion_type = fusion_type

        if self.fusion_type == 'concat':
            feature_channels = 1024
        elif self.fusion_type == 'add':
            feature_channels = 256
        else:
            raise NotImplementedError

        self.output_convs = ConvModule(
            feature_channels,
            out_channels,
            3,
            padding=1,
            conv_cfg=None,
            norm_cfg=norm_cfg,
            act_cfg=act_cfg,
            inplace=False)
```

```python
def forward(self, inputs: List[Tensor]) -> Tensor:
    """
    Args:
        inputs (list[Tensor]): Each tensor has the shape of
            :math:`(N, C_i, H_i, W_i)`. It usually expects 4 tensors
            (C2-C5 features) from ResNet.

    Returns:
        Tensor: A tensor of shape :math:`(N, C_{out}, H_0, W_0)` where
        :math:`C_{out}` is ``out_channels``.
    """
    assert len(inputs) == len(self.in_channels)

    # build laterals
    laterals = [
        lateral_conv(inputs[i])
        for i, lateral_conv in enumerate(self.lateral_convs)
    ]

    # build top-down path
    used_backbone_levels = len(laterals)
    for i in range(used_backbone_levels - 1, 0, -1):
        # step 1: upsample to level i-1 size and add level i-1
        prev_shape = laterals[i - 1].shape[2:]
        laterals[i - 1] = laterals[i - 1] + F.interpolate(
            laterals[i], size=prev_shape, mode='nearest')
        # step 2: smooth level i-1
        laterals[i - 1] = self.fpn_convs[i - 1](laterals[i - 1])

    # upsample and cat
    bottom_shape = laterals[0].shape[2:]
    for i in range(1, used_backbone_levels):
        laterals[i] = F.interpolate(
            laterals[i], size=bottom_shape, mode='nearest')

    if self.fusion_type == 'concat':
        out = torch.cat(laterals, 1)
    elif self.fusion_type == 'add':
        out = laterals[0]
        for i in range(1, used_backbone_levels):
            out += laterals[i]
    else:
        raise NotImplementedError
    out = self.output_convs(out)
```

```
        return out
```

Call the ‘FPNF’ module to implement feature fusion.

1. Initialization method (`__init__`):

   - `in_channels`: A list of channel numbers as input, the default is `[256, 512, 1024, 2048]`, corresponding to the input features of different stages.

   - `out_channels`: Number of output channels, default is `256`.

   - `fusion_type`: The type of feature fusion, optional 'concat' or 'add', the default is 'concat'.

   - `init_cfg`: Configuration of model initialization, used to initialize parameters such as weights. By default, 'Xavier' is used for initialization.

2. Forward propagation method (`forward`):

   - Receives a list `inputs` containing feature tensors from different stages and performs the following steps:

     - Apply a convolutional layer to each input stage, resulting in a list of `laterals`, where each element represents a lateral connection of the corresponding stage.

     - Through a top-down path, high-order feature maps are upsampled and added to low-order feature maps, and then convolution is applied for smoothing.

     - Finally, the processed feature maps are upsampled and fused according to the specified method, and the final fused features are output.

3. Structural description:

   - Use `ModuleList` to store lateral connected convolutional layers `lateral_convs` and upsampled convolutional layers `fpn_convs`.

   - The number of output channels is determined by `out_channels`, and the final output convolution operation is performed through `output_convs`.

4. Fusion method (`fusion_type`):

   - 'concat': Use `torch.cat` to concatenate in the channel dimension.

   - 'add': Add feature maps directly.

2.3 det_head

```
# Copyright (c) OpenMMLab. All rights reserved.
from typing import Dict, List, Optional, Union

from mmocr.registry import MODELS
```

```python
from . import PANHead


@MODELS.register_module()
class PSEHead(PANHead):
    """The class for PSENet head.

    Args:
        in_channels (list[int]): A list of numbers of input channels.
        hidden_dim (int): The hidden dimension of the first convolutional
            layer.
        out_channel (int): Number of output channels.
        module_loss (dict): Configuration dictionary for loss type. Supported
            loss types are "PANModuleLoss" and "PSEModuleLoss". Defaults to
            PSEModuleLoss.
        postprocessor (dict): Config of postprocessor for PSENet.
        init_cfg (dict or list[dict], optional): Initialization configs.
    """

    def __init__(self,
                 in_channels: List[int],
                 hidden_dim: int,
                 out_channel: int,
                 module_loss: Dict = dict(type='PSEModuleLoss'),
                 postprocessor: Dict = dict(
                     type='PSEPostprocessor', text_repr_type='poly'),
                 init_cfg: Optional[Union[Dict, List[Dict]]] = None) -> None:

        super().__init__(
            in_channels=in_channels,
            hidden_dim=hidden_dim,
            out_channel=out_channel,
            module_loss=module_loss,
            postprocessor=postprocessor,
            init_cfg=init_cfg)
```

This code defines a class named `PSEHead`, which inherits from `PANKead` and is used to implement the head structure of PSENet.

1. Initialization method (`__init__`):

   - `in_channels`: a list of integers containing the number of input channels.

   - `hidden_dim`: the hidden dimension of the first convolutional layer.

- `out_channel`: Number of output channels.

- `module_loss`: Dictionary used to configure loss type, supports "PANModuleLoss" and "PSEModuleLoss" two loss types, the default is "PSEModuleLoss".

- `postprocessor`: Dictionary used to configure PSENet's post-processor.

- `init_cfg`: Configuration for initialization, which can be a dictionary or a list of dictionaries.

2. Initialize the parent class (`super().__init__`):

   - Call the initialization method of the parent class `PANKead` and pass the above parameters to the parent class for initialization.

This code mainly defines the header structure for the PSENet model, which is implemented by inheriting `PANKead`. The specific implementation of the `PSEHead` class involves calling the parent class `BaseTextDetHead'. The final overall setup will be traced back to model/base_model.py in the mmengine dependency.

In actual use, only the parameters of the initialization parameters need to be defined.

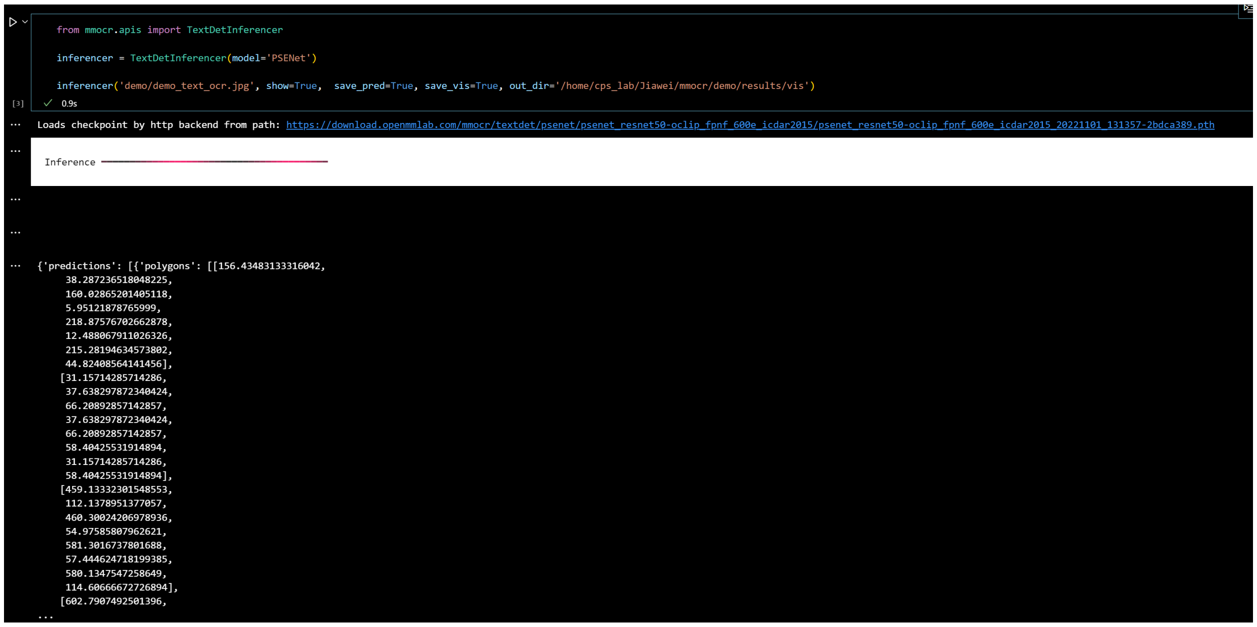
3. Inference output format

MMOCR defines the script used for inference in mmocr/tools/infer.py.

```
from mmocr.apis import TextDetInferencer

inferencer = TextDetInferencer(model='PSENet')

inferencer('demo/demo_text_ocr.jpg', show=True,  save_pred=True, save_vis=True, 
out_dir='/home/cps_lab/Jiawei/mmocr/demo/results/vis')
```

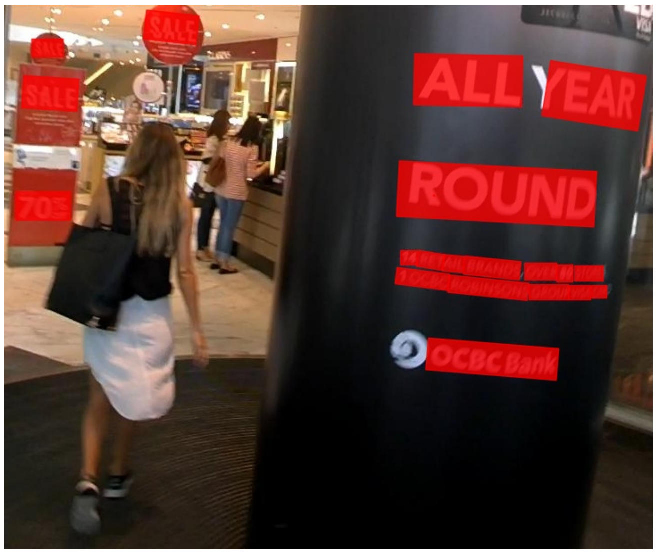
The following parameters can be defined in inferencer:

```
results = inferencer(
    inputs=your_inputs,
    batch_size=your_batch_size,
    det_batch_size=your_det_batch_size, # If you need to override the default batch_size
    out_dir='your_output_directory',
    return_vis=True, # Whether to return visual results
    save_vis=True, # Whether to save the visualization results
    save_pred=True, #Whether to save the inference results
   print_result=True,
)
```

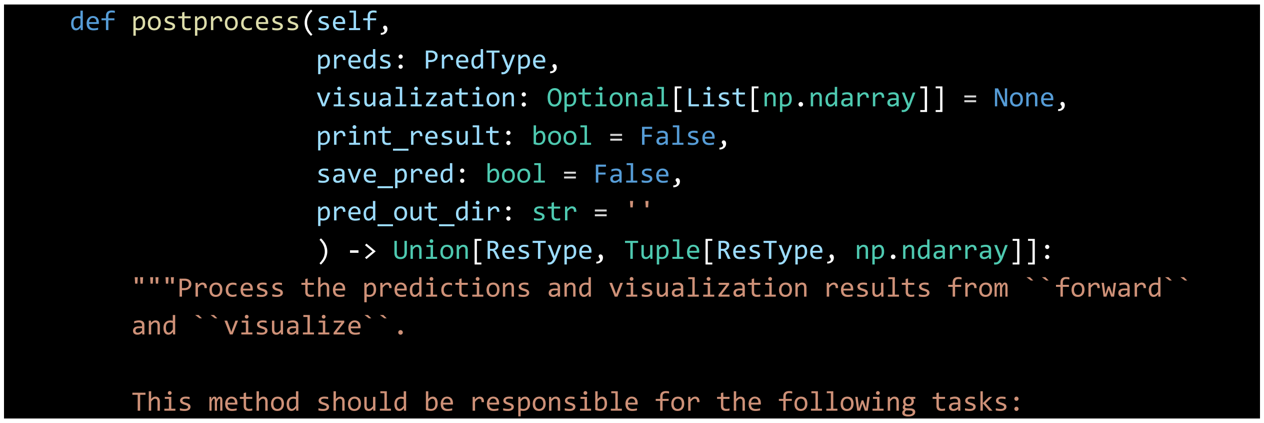


```
from mmocr.apis import TextDetInferencer

inferencer = TextDetInferencer(model='PSENet')

inferencer('demo/demo_text_ocr.jpg', show=True,  save_pred=True, save_vis=True, out_dir='/home/cps_lab/Jiawei/mmocr/demo/results/vis')
[3]  ✓ 0.9s

… Loads checkpoint by http backend from path: https://download.openmmlab.com/mmocr/textdet/psenet/psenet_resnet50-oclip_fpnf_600e_icdar2015/psenet_resnet50-oclip_fpnf_600e_icdar2015_20221101_131357-2bdca389.pth
… 
  Inference ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
…
…
… {'predictions': [{'polygons': [[156.43483133316042,
      38.287236518048225,
      160.02865201405118,
      5.95121878765999,
      218.87576702662878,
      12.488067911026326,
      215.28194634573802,
      44.82408564141456],
     [31.15714285714286,
      37.638297872340424,
      66.20892857142857,
      37.638297872340424,
      66.20892857142857,
      58.40425531914894,
      31.15714285714286,
      58.40425531914894],
     [459.13332301548553,
      112.1378951377057,
      460.30024206978936,
      54.97585807962621,
      581.3016737801688,
      57.444624718199385,
      580.1347547258649,
      114.60666672726894],
     [602.7907492501396,
   ...
```

After executing the script, the json files of the image visualization results and text box coordinates are obtained respectively.



demo_text_ocr.json



```
    def postprocess(self,
                    preds: PredType,
                    visualization: Optional[List[np.ndarray]] = None,
                    print_result: bool = False,
                    save_pred: bool = False,
                    pred_out_dir: str = ''
                    ) -> Union[ResType, Tuple[ResType, np.ndarray]]:
        """Process the predictions and visualization results from ``forward``
        and ``visualize``.

        This method should be responsible for the following tasks:
```

```python
    1. Convert datasamples into a json-serializable dict if needed.
    2. Pack the predictions and visualization results and return them.
    3. Dump or log the predictions.

    Args:
        preds (PredType): Predictions of the model.
        visualization (Optional[np.ndarray]): Visualized predictions.
        print_result (bool): Whether to print the result.
            Defaults to False.
        save_pred (bool): Whether to save the inference result. Defaults to
            False.
        pred_out_dir: File to save the inference results w/o
            visualization. If left as empty, no file will be saved.
            Defaults to ''.

    Returns:
        Dict: Inference and visualization results, mapped from
            "predictions" and "visualization".
    """

    result_dict = {}
    pred_results = [{} for _ in range(len(next(iter(preds.values()))))]
    if 'rec' in self.mode:
        for i, rec_pred in enumerate(preds['rec']):
            result = dict(rec_texts=[], rec_scores=[])
            for rec_pred_instance in rec_pred:
                rec_dict_res = self.textrec_inferencer.pred2dict(
                    rec_pred_instance)
                result['rec_texts'].append(rec_dict_res['text'])
                result['rec_scores'].append(rec_dict_res['scores'])
            pred_results[i].update(result)
    if 'det' in self.mode:
        for i, det_pred in enumerate(preds['det']):
            det_dict_res = self.textdet_inferencer.pred2dict(det_pred)
            pred_results[i].update(
                dict(
                    det_polygons=det_dict_res['polygons'],
                    det_scores=det_dict_res['scores']))
    if 'kie' in self.mode:
        for i, kie_pred in enumerate(preds['kie']):
            kie_dict_res = self.kie_inferencer.pred2dict(kie_pred)
            pred_results[i].update(
                dict(
                    kie_labels=kie_dict_res['labels'],
                        kie_scores=kie_dict_res['scores']),
                    kie_edge_scores=kie_dict_res['edge_scores'],
                    kie_edge_labels=kie_dict_res['edge_labels'])

        if save_pred and pred_out_dir:
            pred_key = 'det' if 'det' in self.mode else 'rec'
            for pred, pred_result in zip(preds[pred_key], pred_results):
                img_path = (
                    pred.img_path if pred_key == 'det' else pred[0].img_path)
                pred_name = osp.splitext(osp.basename(img_path))[0]
                pred_name = f'{pred_name}.json'
                pred_out_file = osp.join(pred_out_dir, pred_name)
                mmengine.dump(pred_result, pred_out_file, ensure_ascii=False)

        result_dict['predictions'] = pred_results
        if print_result:
            print(result_dict)
        result_dict['visualization'] = visualization
        return result_dict
```

The above code snippet defines the format of the output. In this code, pred_results is a list containing the results of each prediction sample. The specific result content depends on the model's task type (detection, recognition, information extraction) and the output of each task. In this example, the results of detection, recognition, and information extraction are added to pred_results one by one through the update method. Finally, the result_dict contains all predicted results, with the 'predictions' key corresponding to pred_results and the 'visualization' key corresponding to the visualization results.