accurate-lane-detection-using-resa

June 4, 2024

0.1 Required Packages Import

0.1.1 Required Packages Import

[1]:
```
# import required packages
import json
import numpy as np
import cv2
import matplotlib.pyplot as plt

import torch

from PIL import Image

import json
import os

import cv2
import matplotlib.pyplot as plt
import numpy as np
from sklearn.cluster import DBSCAN
import torch
import torch.nn as nn
import torch.nn.functional as F

import torch.optim as optim
from torch.nn.modules.loss import _Loss
from torch.autograd import Variable
from tqdm import tqdm

from sklearn.linear_model import LinearRegression

import seaborn as sns
```

[2]:
```
json_gt = [json.loads(line) for line in open('/kaggle/input/tusimple/
 ↪test_label_new.json')]
```

[3]:
```
#json_gt[0]
```

1 1. Make Utils Process for Processing TUSimple Dataset

[4]:
```
def split_path(path):
    """split path tree into list"""
    folders = []
    while True:
        path, folder = os.path.split(path)
        if folder != "":
            folders.insert(0, folder)
        else:
            if path != "":
                folders.insert(0, path)
            break
    return folders

def getLane_tusimple(prob_map, y_px_gap, pts, thresh, resize_shape=None):
    """
    Arguments:
    ----------
    prob_map: prob map for single lane, np array size (h, w)
    resize_shape:  reshape size target, (H, W)

    Return:
    ----------
    coords: x coords bottom up every y_px_gap px, 0 for non-exist, in resized␣
 ↪shape
    """
    if resize_shape is None:
        resize_shape = prob_map.shape
    h, w = prob_map.shape
    H, W = resize_shape

    coords = np.zeros(pts)
    for i in range(pts):
        y = int((H - 10 - i * y_px_gap) * h / H)
        if y < 0:
            break
        line = prob_map[y, :]
        id = np.argmax(line)
        if line[id] > thresh:
            coords[i] = int(id / w * W)
    if (coords > 0).sum() < 2:
        coords = np.zeros(pts)
    return coords

def prob2lines_tusimple(seg_pred, exist, resize_shape=None, smooth=True,␣
 ↪y_px_gap=10, pts=None, thresh=0.3):
```

```
    """
    Arguments:
    ----------
    seg_pred:      np.array size (5, h, w)
    resize_shape:  reshape size target, (H, W)
    exist:       list of existence, e.g. [0, 1, 1, 0]
    smooth:      whether to smooth the probability or not
    y_px_gap:    y pixel gap for sampling
    pts:     how many points for one lane
    thresh:  probability threshold

    Return:
    ----------
    coordinates: [x, y] list of lanes, e.g.: [ [[9, 569], [50, 549]] ,[[630,␣
↪569], [647, 549]] ]
    """
    if resize_shape is None:
        resize_shape = seg_pred.shape[1:]  # seg_pred (5, h, w)
    _, h, w = seg_pred.shape
    H, W = resize_shape
    coordinates = []

    if pts is None:
        pts = round(H / 2 / y_px_gap)

    seg_pred = np.ascontiguousarray(np.transpose(seg_pred, (1, 2, 0)))
    for i in range(6):
        prob_map = seg_pred[..., i + 1]
        if smooth:
            prob_map = cv2.blur(prob_map, (9, 9), borderType=cv2.
↪BORDER_REPLICATE)
        if exist[i] > 0:
            coords = getLane_tusimple(prob_map, y_px_gap, pts, thresh,␣
↪resize_shape)
            coordinates.append(
                [[coords[j], H - 10 - j * y_px_gap] if coords[j] > 0 else [-1,␣
↪H - 10 - j * y_px_gap] for j in
                 range(pts)])

    return coordinates

class LaneEval(object):
    lr = LinearRegression()
    pixel_thresh = 20
    pt_thresh = 0.85

    @staticmethod
```

```
    def get_angle(xs, y_samples):
        xs, ys = xs[xs >= 0], y_samples[xs >= 0]
        if len(xs) > 1:
            LaneEval.lr.fit(ys[:, None], xs)
            k = LaneEval.lr.coef_[0]
            theta = np.arctan(k)
        else:
            theta = 0
        return theta

    @staticmethod
    def line_accuracy(pred, gt, thresh):
        pred = np.array([p if p >= 0 else -100 for p in pred])
        gt = np.array([g if g >= 0 else -100 for g in gt])
        return np.sum(np.where(np.abs(pred - gt) < thresh, 1., 0.)) / len(gt)

    @staticmethod
    def bench(pred, gt, y_samples, running_time):
        if any(len(p) != len(y_samples) for p in pred):
            raise Exception('Format of lanes error.')
        if running_time > 200 or len(gt) + 2 < len(pred):
            return 0., 0., 1.
        angles = [LaneEval.get_angle(np.array(x_gts), np.array(y_samples)) for␣
↪x_gts in gt]
        threshs = [LaneEval.pixel_thresh / np.cos(angle) for angle in angles]
        line_accs = []
        fp, fn = 0., 0.
        matched = 0.
        for x_gts, thresh in zip(gt, threshs):
            accs = [LaneEval.line_accuracy(np.array(x_preds), np.array(x_gts),␣
↪thresh) for x_preds in pred]
            max_acc = np.max(accs) if len(accs) > 0 else 0.
            if max_acc < LaneEval.pt_thresh:
                fn += 1
            else:
                matched += 1
            line_accs.append(max_acc)
        fp = len(pred) - matched
        if len(gt) > 4 and fn > 0:
            fn -= 1
        s = sum(line_accs)
        if len(gt) > 4:
            s -= min(line_accs)
        return s / max(min(4.0, len(gt)), 1.), fp / len(pred) if len(pred) > 0␣
↪else 0., fn / max(min(len(gt), 4.) , 1.)

    @staticmethod
```

```
    def bench_one_submit(pred_file, gt_file):
        try:
            json_pred = [json.loads(line) for line in open(pred_file).
↪readlines()]
        except BaseException as e:
            raise Exception('Fail to load json file of the prediction.')
        json_gt = [json.loads(line) for line in open(gt_file).readlines()]
        if len(json_gt) != len(json_pred):
            raise Exception('We do not get the predictions of all the test␣
↪tasks')
        gts = {l['raw_file']: l for l in json_gt}
        accuracy, fp, fn = 0., 0., 0.
        for pred in json_pred:
            if 'raw_file' not in pred or 'lanes' not in pred or 'run_time' not␣
↪in pred:
                raise Exception('raw_file or lanes or run_time not in some␣
↪predictions.')
            raw_file = pred['raw_file']
            pred_lanes = pred['lanes']
            run_time = pred['run_time']
            if raw_file not in gts:
                raise Exception('Some raw_file from your predictions do not␣
↪exist in the test tasks.')
            gt = gts[raw_file]
            gt_lanes = gt['lanes']
            y_samples = gt['h_samples']
            try:
                a, p, n = LaneEval.bench(pred_lanes, gt_lanes, y_samples,␣
↪run_time)
            except BaseException as e:
                raise Exception('Format of lanes error.')
            accuracy += a
            fp += p
            fn += n
        num = len(gts)
        # the first return parameter is the default ranking parameter
        return json.dumps([
            {'name': 'Accuracy', 'value': accuracy / num, 'order': 'desc'},
            {'name': 'FP', 'value': fp / num, 'order': 'asc'},
            {'name': 'FN', 'value': fn / num, 'order': 'asc'}
        ])


def bench_one(pred_file, gt_file):
    try:
        json_pred = [json.loads(line) for line in open(pred_file).readlines()]
```

```
    except BaseException as e:
        raise Exception('Fail to load json file of the prediction.')
    json_gt = [json.loads(line) for line in open(gt_file).readlines()]
    if len(json_gt) != len(json_pred):
        raise Exception('We do not get the predictions of all the test tasks')
    gts = {l['raw_file']: l for l in json_gt}
    accuracy, fp, fn = 0., 0., 0.
    accuracy_list = []
    for pred in json_pred:
        if 'raw_file' not in pred or 'lanes' not in pred or 'run_time' not in␣
 ↪pred:
            raise Exception('raw_file or lanes or run_time not in some␣
 ↪predictions.')
        raw_file = pred['raw_file']
        pred_lanes = pred['lanes']
        run_time = pred['run_time']
        if raw_file not in gts:
            raise Exception('Some raw_file from your predictions do not exist␣
 ↪in the test tasks.')
        gt = gts[raw_file]
        gt_lanes = gt['lanes']
        y_samples = gt['h_samples']
        try:
            a, p, n = LaneEval.bench(pred_lanes, gt_lanes, y_samples, run_time)
            accuracy_list.append((a, raw_file))
        except BaseException as e:
                raise Exception('Format of lanes error.')
    return accuracy_list
```

[5]:
```
def to_one_hot(tensor, nClasses):
    n, h, w = tensor.size()
    one_hot = torch.zeros(n, nClasses, h, w).to(tensor.device).scatter_(1,␣
 ↪tensor.view(n, 1, h, w), 1)
    return one_hot


class mIoULoss(nn.Module):
    def __init__(self, weight=None, size_average=True, n_classes=6):
        super(mIoULoss, self).__init__()
        self.classes = n_classes

    def forward(self, inputs, target_oneHot):
        """
        IoU Loss for individual examples
        inputs - N x {Classes or higher} x H x W
        target_oneHot - N x {Classes or higher} x H x W
        BG can be ignored
```

```
        """

        N = inputs.size()[0]
        C = inputs.size()[1]

        # predicted probabilities for each pixel along channel
        inputs = F.softmax(inputs, dim=1)

        # Numerator Product
        inter = inputs * target_oneHot
        # Sum over all pixels N x C x H x W => N x C
        inter = inter.view(N, C, -1).sum(2)

        # Denominator
        union = inputs + target_oneHot - (inputs * target_oneHot)
        # Sum over all pixels N x C x H x W => N x C
        union = union.view(N, C, -1).sum(2)

        loss = inter / union

        ## Return average loss over classes and batch
        return -(loss[:, -self.classes].mean() - 1.)
```

[6]:
```
PATH = '/kaggle/input/tusimple/TUSimple'

class LaneDataset(torch.utils.data.Dataset):
    def __init__(self, dataset_path= PATH, train=True, size=(800, 360)):
        self._dataset_path = dataset_path
        self._mode = "train" if train else "test"
        self._image_size = size # w, h
        self._data = []

        if self._mode == "train":
            file_path = "train_val_gt.txt"
        elif self._mode == "test":
            file_path = "test_gt.txt"
        self._process_list(os.path.join(self._dataset_path, "train_set/
 ↪seg_label/list", file_path))

    def __getitem__(self, idx):
        img_path = self._dataset_path + ("/train_set" if self._mode == "train"␣
 ↪else "/test_set") + self._data[idx][0]
        image = cv2.imread(img_path)
        h, w, c = image.shape
        raw_image = image
        image = cv2.resize(image, self._image_size, interpolation=cv2.
 ↪INTER_LINEAR)
```

```
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

        ins_segmentation_path = self._dataset_path + "/train_set" + self.
↪_data[idx][1]
        ins_segmentation_image = cv2.imread(ins_segmentation_path)
        ins_segmentation_image = ins_segmentation_image[:, :, 0]
        ins_segmentation_image = cv2.resize(ins_segmentation_image, self.
↪_image_size, interpolation=cv2.INTER_LINEAR)

        segmentation_image = ins_segmentation_image.copy()
        segmentation_image[segmentation_image > 0] = 1

        image = torch.from_numpy(image).float().permute((2, 0, 1))
        segmentation_image = torch.from_numpy(segmentation_image.copy()).
↪to(torch.int64)
        ins_segmentation_image =  torch.from_numpy(ins_segmentation_image.
↪copy())

        exists = [int(i) for i in self._data[idx][2]]
        exists = torch.as_tensor(exists)

        output = {
            'img_path' : img_path,
            'img' : image,
            "meta" : { "full_img_path" : img_path ,
                       "img_name" :  self._data[ idx ][ 0 ]},
            'segLabel' : segmentation_image,
            'IsegLabel' : ins_segmentation_image,
            'exist' : exists,
            "original_image" : raw_image,
            "label" : segmentation_image
        }

        return output

    def probmap2lane(self, seg_pred, exist, resize_shape=(720, 1280),␣
↪smooth=True, y_px_gap=10, pts=56, thresh=0.6):
        """
        Arguments:
        ----------
        seg_pred:      np.array size (5, h, w)
        resize_shape:  reshape size target, (H, W)
        exist:       list of existence, e.g. [0, 1, 1, 0]
        smooth:      whether to smooth the probability or not
        y_px_gap:    y pixel gap for sampling
        pts:     how many points for one lane
        thresh:  probability threshold
```

```
        Return:
        ----------
        coordinates: [x, y] list of lanes, e.g.: [ [[9, 569], [50, 549]]␣
↪,[[630, 569], [647, 549]] ]
        """
        if resize_shape is None:
            resize_shape = seg_pred.shape[1:]  # seg_pred (5, h, w)
        _, h, w = seg_pred.shape
        H, W = resize_shape
        coordinates = []

        for i in range(self.cfg.num_classes - 1):
            prob_map = seg_pred[i + 1]
            if smooth:
                prob_map = cv2.blur(prob_map, (9, 9), borderType=cv2.
↪BORDER_REPLICATE)
            coords = self.get_lane(prob_map, y_px_gap, pts, thresh,␣
↪resize_shape)
            if self.is_short(coords):
                continue
            coordinates.append(
                [[coords[j], H - 10 - j * y_px_gap] if coords[j] > 0 else [-1,␣
↪H - 10 - j * y_px_gap] for j in
                 range(pts)])


        if len(coordinates) == 0:
            coords = np.zeros(pts)
            coordinates.append(
                [[coords[j], H - 10 - j * y_px_gap] if coords[j] > 0 else [-1,␣
↪H - 10 - j * y_px_gap] for j in
                 range(pts)])
        #print(coordinates)

        return coordinates

    def _process_list(self, file_path):
        with open(file_path) as f:
            for line in f:
                words = line.split()
                image = words[0]
                segmentation = words[1]
                exists = words[2:]
                self._data.append((image, segmentation, exists))

    def _show_sample_dataset( self, number_samples = 10 ):
```

```
        # Visualizing some Lane Detection dataset

        sns.set_theme()

        f, axarr = plt.subplots( number_samples    ,3 , figsize = ( 20 , 30 ))

        plt.axis('off')

        for i in range( number_samples ):

            axarr[ i , 0].imshow(  self.__getitem__( idx = i )[␣
↪"original_image" ]  )
            axarr[ i , 0 ].set_title( "Lane Image Data No " + str( i + 1) )
            axarr[ i , 0 ].set_axis_off()

            axarr[ i , 1 ].imshow(  self.__getitem__( idx = i )[ "segLabel" ] )
            axarr[ i , 1 ].set_title( "Lane Image Segmentation Data No " + str(␣
↪i + 1) )
            axarr[ i , 1 ].set_axis_off()

            axarr[ i , 2 ].imshow(  self.__getitem__( idx = i )[ "IsegLabel" ] )
            axarr[ i , 2 ].set_title( "Lane Image Segmentation Data No " + str(␣
↪i + 1) )
            axarr[ i , 2 ].set_axis_off()

        f.tight_layout()
        plt.show()

    def __len__(self):
        return len(self._data)
```

2 2.Make Function for Reccurrent Feature Aggregator CNN

[7]:
```
import torch
from torch import nn
import torch.nn.functional as F
from torch.hub import load_state_dict_from_url


# This code is borrow from torchvision.

model_urls = {
    'resnet18': 'https://download.pytorch.org/models/resnet18-5c106cde.pth',
    'resnet34': 'https://download.pytorch.org/models/resnet34-333f7ec4.pth',
    'resnet50': 'https://download.pytorch.org/models/resnet50-19c8e357.pth',
```

```
    'resnet101': 'https://download.pytorch.org/models/resnet101-5d3b4d8f.pth',
    'resnet152': 'https://download.pytorch.org/models/resnet152-b121ed2d.pth',
    'resnext50_32x4d': 'https://download.pytorch.org/models/
 ↪resnext50_32x4d-7cdf4587.pth',
    'resnext101_32x8d': 'https://download.pytorch.org/models/
 ↪resnext101_32x8d-8ba56ff5.pth',
    'wide_resnet50_2': 'https://download.pytorch.org/models/
 ↪wide_resnet50_2-95faca4d.pth',
    'wide_resnet101_2': 'https://download.pytorch.org/models/
 ↪wide_resnet101_2-32ee1156.pth',
}


def conv3x3(in_planes, out_planes, stride=1, groups=1, dilation=1):
    """3x3 convolution with padding"""
    return nn.Conv2d(in_planes, out_planes, kernel_size=3, stride=stride,
                     padding=dilation, groups=groups, bias=False,␣
 ↪dilation=dilation)


def conv1x1(in_planes, out_planes, stride=1):
    """1x1 convolution"""
    return nn.Conv2d(in_planes, out_planes, kernel_size=1, stride=stride,␣
 ↪bias=False)


class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, inplanes, planes, stride=1, downsample=None, groups=1,
                 base_width=64, dilation=1, norm_layer=None):
        super(BasicBlock, self).__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        if groups != 1 or base_width != 64:
            raise ValueError(
                'BasicBlock only supports groups=1 and base_width=64')
        # if dilation > 1:
        #     raise NotImplementedError(
        #         "Dilation > 1 not supported in BasicBlock")
        # Both self.conv1 and self.downsample layers downsample the input when␣
 ↪stride != 1
        self.conv1 = conv3x3(inplanes, planes, stride, dilation=dilation)
        self.bn1 = norm_layer(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes, dilation=dilation)
```

```
        self.bn2 = norm_layer(planes)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out


class Bottleneck(nn.Module):
    expansion = 4

    def __init__(self, inplanes, planes, stride=1, downsample=None, groups=1,
                 base_width=64, dilation=1, norm_layer=None):
        super(Bottleneck, self).__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        width = int(planes * (base_width / 64.)) * groups
        # Both self.conv2 and self.downsample layers downsample the input when␣
↪stride != 1
        self.conv1 = conv1x1(inplanes, width)
        self.bn1 = norm_layer(width)
        self.conv2 = conv3x3(width, width, stride, groups, dilation)
        self.bn2 = norm_layer(width)
        self.conv3 = conv1x1(width, planes * self.expansion)
        self.bn3 = norm_layer(planes * self.expansion)
        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        identity = x
```

```
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)

        out = self.conv3(out)
        out = self.bn3(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out


class ResNetWrapper(nn.Module):

    def __init__(self, cfg):
        super(ResNetWrapper, self).__init__()
        self.cfg = cfg
        self.in_channels = [64, 128, 256, 512]
        if 'in_channels' in cfg.backbone:
            self.in_channels = cfg.backbone.in_channels
        self.model = eval(cfg.backbone.resnet)(
            pretrained=cfg.backbone.pretrained,
            replace_stride_with_dilation=cfg.backbone.
 ↪replace_stride_with_dilation, in_channels=self.in_channels)
        self.out = None
        if cfg.backbone.out_conv:
            out_channel = 512
            for chan in reversed(self.in_channels):
                if chan < 0: continue
                out_channel = chan
                break
            self.out = conv1x1(
                out_channel * self.model.expansion, 128)

    def forward(self, x):
        x = self.model(x)
        if self.out:
            x = self.out(x)
        return x
```

```
class ResNet(nn.Module):

    def __init__(self, block, layers, zero_init_residual=False,
                 groups=1, width_per_group=64,␣
 ↪replace_stride_with_dilation=None,
                 norm_layer=None, in_channels=None):
        super(ResNet, self).__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        self._norm_layer = norm_layer

        self.inplanes = 64
        self.dilation = 1
        if replace_stride_with_dilation is None:
            # each element in the tuple indicates if we should replace
            # the 2x2 stride with a dilated convolution instead
            replace_stride_with_dilation = [False, False, False]
        if len(replace_stride_with_dilation) != 3:
            raise ValueError("replace_stride_with_dilation should be None "
                             "or a 3-element tuple, got {}".
 ↪format(replace_stride_with_dilation))
        self.groups = groups
        self.base_width = width_per_group
        self.conv1 = nn.Conv2d(3, self.inplanes, kernel_size=7, stride=2,␣
 ↪padding=3,
                               bias=False)
        self.bn1 = norm_layer(self.inplanes)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.in_channels = in_channels
        self.layer1 = self._make_layer(block, in_channels[0], layers[0])
        self.layer2 = self._make_layer(block, in_channels[1], layers[1],␣
 ↪stride=2,
                                       dilate=replace_stride_with_dilation[0])
        self.layer3 = self._make_layer(block, in_channels[2], layers[2],␣
 ↪stride=2,
                                       dilate=replace_stride_with_dilation[1])
        if in_channels[3] > 0:
            self.layer4 = self._make_layer(block, in_channels[3], layers[3],␣
 ↪stride=2,
                                           ␣
 ↪dilate=replace_stride_with_dilation[2])
        self.expansion = block.expansion
```

```
        # self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        # self.fc = nn.Linear(512 * block.expansion, num_classes)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(
                    m.weight, mode='fan_out', nonlinearity='relu')
            elif isinstance(m, (nn.BatchNorm2d, nn.GroupNorm)):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

        # Zero-initialize the last BN in each residual branch,
        # so that the residual branch starts with zeros, and each residual␣
↪block behaves like an identity.
        # This improves the model by 0.2~0.3% according to https://arxiv.org/
↪abs/1706.02677
        if zero_init_residual:
            for m in self.modules():
                if isinstance(m, Bottleneck):
                    nn.init.constant_(m.bn3.weight, 0)
                elif isinstance(m, BasicBlock):
                    nn.init.constant_(m.bn2.weight, 0)

    def _make_layer(self, block, planes, blocks, stride=1, dilate=False):
        norm_layer = self._norm_layer
        downsample = None
        previous_dilation = self.dilation
        if dilate:
            self.dilation *= stride
            stride = 1
        if stride != 1 or self.inplanes != planes * block.expansion:
            downsample = nn.Sequential(
                conv1x1(self.inplanes, planes * block.expansion, stride),
                norm_layer(planes * block.expansion),
            )

        layers = []
        layers.append(block(self.inplanes, planes, stride, downsample, self.
↪groups,
                            self.base_width, previous_dilation, norm_layer))
        self.inplanes = planes * block.expansion
        for _ in range(1, blocks):
            layers.append(block(self.inplanes, planes, groups=self.groups,
                                base_width=self.base_width, dilation=self.
↪dilation,
                                norm_layer=norm_layer))
```

```
        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        if self.in_channels[3] > 0:
            x = self.layer4(x)

        # x = self.avgpool(x)
        # x = torch.flatten(x, 1)
        # x = self.fc(x)

        return x


def _resnet(arch, block, layers, pretrained, progress, **kwargs):
    model = ResNet(block, layers, **kwargs)
    if pretrained:
        state_dict = load_state_dict_from_url(model_urls[arch],
                                              progress=progress)
        model.load_state_dict(state_dict, strict=False)
    return model


def resnet18(pretrained=False, progress=True, **kwargs):
    r"""ResNet-18 model from
    `"Deep Residual Learning for Image Recognition" <https://arxiv.org/pdf/1512.
 ↪03385.pdf>`_

    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
        progress (bool): If True, displays a progress bar of the download to␣
 ↪stderr
    """
    return _resnet('resnet18', BasicBlock, [2, 2, 2, 2], pretrained, progress,
                   **kwargs)


def resnet34(pretrained=False, progress=True, **kwargs):
    r"""ResNet-34 model from
```

```
    `"Deep Residual Learning for Image Recognition" <https://arxiv.org/pdf/1512.
↪03385.pdf>`_

    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
        progress (bool): If True, displays a progress bar of the download to␣
↪stderr
    """
    return _resnet('resnet34', BasicBlock, [3, 4, 6, 3], pretrained, progress,
                   **kwargs)


def resnet50(pretrained=False, progress=True, **kwargs):
    r"""ResNet-50 model from
    `"Deep Residual Learning for Image Recognition" <https://arxiv.org/pdf/1512.
↪03385.pdf>`_

    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
        progress (bool): If True, displays a progress bar of the download to␣
↪stderr
    """
    return _resnet('resnet50', Bottleneck, [3, 4, 6, 3], pretrained, progress,
                   **kwargs)


def resnet101(pretrained=False, progress=True, **kwargs):
    r"""ResNet-101 model from
    `"Deep Residual Learning for Image Recognition" <https://arxiv.org/pdf/1512.
↪03385.pdf>`_

    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
        progress (bool): If True, displays a progress bar of the download to␣
↪stderr
    """
    return _resnet('resnet101', Bottleneck, [3, 4, 23, 3], pretrained, progress,
                   **kwargs)
```

[8]:
```
import inspect

import six

# borrow from mmdetection

def is_str(x):
```

```
    """Whether the input is an string instance."""
    return isinstance(x, six.string_types)

class Registry(object):

    def __init__(self, name):
        self._name = name
        self._module_dict = dict()

    def __repr__(self):
        format_str = self.__class__.__name__ + '(name={}, items={})'.format(
            self._name, list(self._module_dict.keys()))
        return format_str

    @property
    def name(self):
        return self._name

    @property
    def module_dict(self):
        return self._module_dict

    def get(self, key):
        return self._module_dict.get(key, None)

    def _register_module(self, module_class):
        """Register a module.

        Args:
            module (:obj:`nn.Module`): Module to be registered.
        """
        if not inspect.isclass(module_class):
            raise TypeError('module must be a class, but got {}'.format(
                type(module_class)))
        module_name = module_class.__name__
        if module_name in self._module_dict:
            raise KeyError('{} is already registered in {}'.format(
                module_name, self.name))
        self._module_dict[module_name] = module_class

    def register_module(self, cls):
        self._register_module(cls)
        return cls


def build_from_cfg(cfg, registry, default_args=None):
    """Build a module from config dict.
```

```
    Args:
        cfg (dict): Config dict. It should at least contain the key "type".
        registry (:obj:`Registry`): The registry to search the type from.
        default_args (dict, optional): Default initialization arguments.

    Returns:
        obj: The constructed object.
    """
    assert isinstance(cfg, dict) and 'type' in cfg
    assert isinstance(default_args, dict) or default_args is None
    args = {}
    obj_type = cfg.type
    if is_str(obj_type):
        obj_cls = registry.get(obj_type)
        if obj_cls is None:
            raise KeyError('{} is not in the {} registry'.format(
                obj_type, registry.name))
    elif inspect.isclass(obj_type):
        obj_cls = obj_type
    else:
        raise TypeError('type must be a str or valid type, but got {}'.format(
            type(obj_type)))
    if default_args is not None:
        for name, value in default_args.items():
            args.setdefault(name, value)
    return obj_cls(**args)
```

[9]:
```
TRAINER = Registry('trainer')
EVALUATOR = Registry('evaluator')

def build(cfg, registry, default_args=None):
    if isinstance(cfg, list):
        modules = [
            build_from_cfg(cfg_, registry, default_args) for cfg_ in cfg
        ]
        return nn.Sequential(*modules)
    else:
        return build_from_cfg(cfg, registry, default_args)

def build_trainer(cfg):
    return build(cfg.trainer, TRAINER, default_args=dict(cfg=cfg))

def build_evaluator(cfg):
    return build(cfg.evaluator, EVALUATOR, default_args=dict(cfg=cfg))
```

[10]:
```
from torch import nn
import torch.nn.functional as F

class PlainDecoder(nn.Module):
    def __init__(self, cfg):
        super(PlainDecoder, self).__init__()
        self.cfg = cfg

        self.dropout = nn.Dropout2d(0.1)
        self.conv8 = nn.Conv2d(128, cfg.num_classes, 1)

    def forward(self, x):
        x = self.dropout(x)
        x = self.conv8(x)
        x = F.interpolate(x, size=[self.cfg.img_height,  self.cfg.img_width],
                          mode='bilinear', align_corners=False)

        return x


def conv1x1(in_planes, out_planes, stride=1):
    """1x1 convolution"""
    return nn.Conv2d(in_planes, out_planes, kernel_size=1, stride=stride,␣
 ↪bias=False)


class non_bottleneck_1d(nn.Module):
    def __init__(self, chann, dropprob, dilated):
        super().__init__()

        self.conv3x1_1 = nn.Conv2d(
            chann, chann, (3, 1), stride=1, padding=(1, 0), bias=True)

        self.conv1x3_1 = nn.Conv2d(
            chann, chann, (1, 3), stride=1, padding=(0, 1), bias=True)

        self.bn1 = nn.BatchNorm2d(chann, eps=1e-03)

        self.conv3x1_2 = nn.Conv2d(chann, chann, (3, 1), stride=1, padding=(1 *␣
 ↪dilated, 0), bias=True,
                                   dilation=(dilated, 1))

        self.conv1x3_2 = nn.Conv2d(chann, chann, (1, 3), stride=1, padding=(0,␣
 ↪1 * dilated), bias=True,
                                   dilation=(1, dilated))

        self.bn2 = nn.BatchNorm2d(chann, eps=1e-03)
```

```
        self.dropout = nn.Dropout2d(dropprob)

    def forward(self, input):
        output = self.conv3x1_1(input)
        output = F.relu(output)
        output = self.conv1x3_1(output)
        output = self.bn1(output)
        output = F.relu(output)

        output = self.conv3x1_2(output)
        output = F.relu(output)
        output = self.conv1x3_2(output)
        output = self.bn2(output)

        if (self.dropout.p != 0):
            output = self.dropout(output)

        # +input = identity (residual connection)
        return F.relu(output + input)


class UpsamplerBlock(nn.Module):
    def __init__(self, ninput, noutput, up_width, up_height):
        super().__init__()

        self.conv = nn.ConvTranspose2d(
            ninput, noutput, 3, stride=2, padding=1, output_padding=1,␣
 ↪bias=True)

        self.bn = nn.BatchNorm2d(noutput, eps=1e-3, track_running_stats=True)

        self.follows = nn.ModuleList()
        self.follows.append(non_bottleneck_1d(noutput, 0, 1))
        self.follows.append(non_bottleneck_1d(noutput, 0, 1))

        # interpolate
        self.up_width = up_width
        self.up_height = up_height
        self.interpolate_conv = conv1x1(ninput, noutput)
        self.interpolate_bn = nn.BatchNorm2d(
            noutput, eps=1e-3, track_running_stats=True)

    def forward(self, input):
        output = self.conv(input)
        output = self.bn(output)
        out = F.relu(output)
```

```
        for follow in self.follows:
            out = follow(out)

        interpolate_output = self.interpolate_conv(input)
        interpolate_output = self.interpolate_bn(interpolate_output)
        interpolate_output = F.relu(interpolate_output)

        interpolate = F.interpolate(interpolate_output, size=[self.up_height, ␣
 ↪self.up_width],
                                    mode='bilinear', align_corners=False)

        return out + interpolate

class BUSD(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        img_height = cfg.img_height
        img_width = cfg.img_width
        num_classes = cfg.num_classes

        self.layers = nn.ModuleList()

        self.layers.append(UpsamplerBlock(ninput=128, noutput=64,
                                          up_height=int(img_height)//4,␣
 ↪up_width=int(img_width)//4))
        self.layers.append(UpsamplerBlock(ninput=64, noutput=32,
                                          up_height=int(img_height)//2,␣
 ↪up_width=int(img_width)//2))
        self.layers.append(UpsamplerBlock(ninput=32, noutput=16,
                                          up_height=int(img_height)//1,␣
 ↪up_width=int(img_width)//1))

        self.output_conv = conv1x1(16, num_classes)

    def forward(self, input):
        output = input

        for layer in self.layers:
            output = layer(output)

        output = self.output_conv(output)

        return output
```

[11]:
```
NET = Registry('net')

def build(cfg, registry, default_args=None):
```

```
    if isinstance(cfg, list):
        modules = [
            build_from_cfg(cfg_, registry, default_args) for cfg_ in cfg
        ]
        return nn.Sequential(*modules)
    else:
        return build_from_cfg(cfg, registry, default_args)


def build_net(cfg):
    return build(cfg.net, NET, default_args=dict(cfg=cfg))
```

[12]:
```
@NET.register_module
class RESA_module(nn.Module):
    def __init__(self, cfg):
        super(RESA_module, self).__init__()
        self.iter = cfg.resa.iter
        chan = cfg.resa.input_channel
        fea_stride = cfg.backbone.fea_stride
        self.height = cfg.img_height // fea_stride
        self.width = cfg.img_width // fea_stride
        self.alpha = cfg.resa.alpha
        conv_stride = cfg.resa.conv_stride

        for i in range(self.iter):
            conv_vert1 = nn.Conv2d(
                chan, chan, (1, conv_stride),
                padding=(0, conv_stride//2), groups=1, bias=False)
            conv_vert2 = nn.Conv2d(
                chan, chan, (1, conv_stride),
                padding=(0, conv_stride//2), groups=1, bias=False)

            setattr(self, 'conv_d'+str(i), conv_vert1)
            setattr(self, 'conv_u'+str(i), conv_vert2)

            conv_hori1 = nn.Conv2d(
                chan, chan, (conv_stride, 1),
                padding=(conv_stride//2, 0), groups=1, bias=False)
            conv_hori2 = nn.Conv2d(
                chan, chan, (conv_stride, 1),
                padding=(conv_stride//2, 0), groups=1, bias=False)

            setattr(self, 'conv_r'+str(i), conv_hori1)
            setattr(self, 'conv_l'+str(i), conv_hori2)

            idx_d = (torch.arange(self.height) + self.height //
                     2**(self.iter - i)) % self.height
```

```
            setattr(self, 'idx_d'+str(i), idx_d)

            idx_u = (torch.arange(self.height) - self.height //
                     2**(self.iter - i)) % self.height
            setattr(self, 'idx_u'+str(i), idx_u)

            idx_r = (torch.arange(self.width) + self.width //
                     2**(self.iter - i)) % self.width
            setattr(self, 'idx_r'+str(i), idx_r)

            idx_l = (torch.arange(self.width) - self.width //
                     2**(self.iter - i)) % self.width
            setattr(self, 'idx_l'+str(i), idx_l)

    def forward(self, x):
        x = x.clone()

        for direction in ['d', 'u']:
            for i in range(self.iter):
                conv = getattr(self, 'conv_' + direction + str(i))
                idx = getattr(self, 'idx_' + direction + str(i))
                x.add_(self.alpha * F.relu(conv(x[..., idx, :])))

        for direction in ['r', 'l']:
            for i in range(self.iter):
                conv = getattr(self, 'conv_' + direction + str(i))
                idx = getattr(self, 'idx_' + direction + str(i))
                x.add_(self.alpha * F.relu(conv(x[..., idx])))

        return x



class ExistHead(nn.Module):
    def __init__(self, cfg=None):
        super(ExistHead, self).__init__()
        self.cfg = cfg

        self.dropout = nn.Dropout2d(0.1)  # ???
        self.conv8 = nn.Conv2d(128, cfg.num_classes, 1)

        stride = cfg.backbone.fea_stride * 2
        self.fc9 = nn.Linear(
            int(cfg.num_classes * cfg.img_width / stride * cfg.img_height /␣
 ↪stride), 128)
        self.fc10 = nn.Linear(128, cfg.num_classes-1)
```

```
    def forward(self, x):
        x = self.dropout(x)
        x = self.conv8(x)

        x = F.softmax(x, dim=1)
        x = F.avg_pool2d(x, 2, stride=2, padding=0)
        x = x.view(-1, x.numel() // x.shape[0])
        x = self.fc9(x)
        x = F.relu(x)
        x = self.fc10(x)
        x = torch.sigmoid(x)

        return x


@NET.register_module
class RESANet(nn.Module):
    def __init__(self, cfg):
        super(RESANet, self).__init__()
        self.cfg = cfg
        self.backbone = ResNetWrapper(cfg)
        self.resa = RESA_module(cfg)
        self.decoder = eval(cfg.decoder)(cfg)
        self.heads = ExistHead(cfg)

    def forward(self, batch):
        #print( "Input batch data is : " + str( batch ))
        fea = self.backbone(batch)
        fea = self.resa(fea)
        seg = self.decoder(fea)
        exist = self.heads(fea)

        output = {'seg': seg, 'exist': exist}

        return output
```

[13]:
```
import torch


_optimizer_factory = {
    'adam': torch.optim.Adam,
    'sgd': torch.optim.SGD
}


def build_optimizer(cfg, net):
    params = []
```

```
    lr = cfg.optimizer.lr
    weight_decay = cfg.optimizer.weight_decay

    for key, value in net.named_parameters():
        if not value.requires_grad:
            continue
        params += [{"params": [value], "lr": lr, "weight_decay": weight_decay}]

    if 'adam' in cfg.optimizer.type:
        optimizer = _optimizer_factory[cfg.optimizer.type](params, lr,␣
↪weight_decay=weight_decay)
    else:
        optimizer = _optimizer_factory[cfg.optimizer.type](
                params, lr, weight_decay=weight_decay, momentum=cfg.optimizer.
↪momentum)

    return optimizer
```

[14]:
```
import logging

logger_initialized = {}

def get_logger(name, log_file=None, log_level=logging.INFO):
    """Initialize and get a logger by name.
    If the logger has not been initialized, this method will initialize the
    logger by adding one or two handlers, otherwise the initialized logger will
    be directly returned. During initialization, a StreamHandler will always be
    added. If `log_file` is specified and the process rank is 0, a FileHandler
    will also be added.
    Args:
        name (str): Logger name.
        log_file (str | None): The log filename. If specified, a FileHandler
            will be added to the logger.
        log_level (int): The logger level. Note that only the process of
            rank 0 is affected, and other processes will set the level to
            "Error" thus be silent most of the time.
    Returns:
        logging.Logger: The expected logger.
    """
    logger = logging.getLogger(name)
    if name in logger_initialized:
        return logger
    # handle hierarchical names
    # e.g., logger "a" is initialized, then logger "a.b" will skip the
    # initialization since it is a child of "a".
    for logger_name in logger_initialized:
        if name.startswith(logger_name):
```

```
            return logger

    stream_handler = logging.StreamHandler()
    handlers = [stream_handler]

    if log_file is not None:
        file_handler = logging.FileHandler(log_file, 'w')
        handlers.append(file_handler)

    formatter = logging.Formatter(
        '%(asctime)s - %(name)s - %(levelname)s - %(message)s')
    for handler in handlers:
        handler.setFormatter(formatter)
        handler.setLevel(log_level)
        logger.addHandler(handler)

    logger.setLevel(log_level)

    logger_initialized[name] = True

    return logger
```

[15]:
```
from collections import deque, defaultdict
import torch
import os
import datetime


class SmoothedValue(object):
    """Track a series of values and provide access to smoothed values over a
    window or the global series average.
    """

    def __init__(self, window_size=20):
        self.deque = deque(maxlen=window_size)
        self.total = 0.0
        self.count = 0

    def update(self, value):
        self.deque.append(value)
        self.count += 1
        self.total += value

    @property
    def median(self):
        d = torch.tensor(list(self.deque))
        return d.median().item()
```

```
    @property
    def avg(self):
        d = torch.tensor(list(self.deque))
        return d.mean().item()

    @property
    def global_avg(self):
        return self.total / self.count


class Recorder(object):
    def __init__(self, cfg):
        self.cfg = cfg
        self.work_dir = self.get_work_dir()
        cfg.work_dir = self.work_dir
        self.log_path = os.path.join(self.work_dir, 'log.txt')

        self.logger = get_logger('resa', self.log_path)
        self.logger.info('Config: \n' + cfg.text)

        # scalars
        self.epoch = 0
        self.step = 0
        self.loss_stats = defaultdict(SmoothedValue)
        self.batch_time = SmoothedValue()
        self.data_time = SmoothedValue()
        self.max_iter = self.cfg.total_iter
        self.lr = 0.

    def get_work_dir(self):
        now = datetime.datetime.now().strftime('%Y%m%d_%H%M%S')
        hyper_param_str = '_lr_%1.0e_b_%d' % (self.cfg.optimizer.lr, self.cfg.
 ↪batch_size)
        work_dir = os.path.join(self.cfg.work_dirs, now + hyper_param_str)
        if not os.path.exists(work_dir):
            os.makedirs(work_dir)
        return work_dir

    def update_loss_stats(self, loss_dict):
        for k, v in loss_dict.items():
            self.loss_stats[k].update(v.detach().cpu())

    def record(self, prefix, step=-1, loss_stats=None, image_stats=None):
        self.logger.info(self)
        # self.write(str(self))
```

```
    def write(self, content):
        with open(self.log_path, 'a+') as f:
            f.write(content)
            f.write('\n')

    def state_dict(self):
        scalar_dict = {}
        scalar_dict['step'] = self.step
        return scalar_dict

    def load_state_dict(self, scalar_dict):
        self.step = scalar_dict['step']

    def __str__(self):
        loss_state = []
        for k, v in self.loss_stats.items():
            loss_state.append('{}: {:.4f}'.format(k, v.avg))
        loss_state = '  '.join(loss_state)

        recording_state = '  '.join(['epoch: {}', 'step: {}', 'lr: {:.4f}',␣
 ↪'{}', 'data: {:.4f}', 'batch: {:.4f}', 'eta: {}'])
        eta_seconds = self.batch_time.global_avg * (self.max_iter - self.step)
        eta_string = str(datetime.timedelta(seconds=int(eta_seconds)))
        return recording_state.format(self.epoch, self.step, self.lr,␣
 ↪loss_state, self.data_time.avg, self.batch_time.avg, eta_string)


def build_recorder(cfg):
    return Recorder(cfg)
```

[16]:
```
import torch


_optimizer_factory = {
    'adam': torch.optim.Adam,
    'sgd': torch.optim.SGD
}


def build_optimizer(cfg, net):
    params = []
    lr = cfg.optimizer.lr
    weight_decay = cfg.optimizer.weight_decay

    for key, value in net.named_parameters():
        if not value.requires_grad:
            continue
```

```
        params += [{"params": [value], "lr": lr, "weight_decay": weight_decay}]

    if 'adam' in cfg.optimizer.type:
        optimizer = _optimizer_factory[cfg.optimizer.type](params, lr,␣
 ↪weight_decay=weight_decay)
    else:
        optimizer = _optimizer_factory[cfg.optimizer.type](
                params, lr, weight_decay=weight_decay, momentum=cfg.optimizer.
 ↪momentum)

    return optimizer
```

[17]:
```
import torch
import math


_scheduler_factory = {
    'LambdaLR': torch.optim.lr_scheduler.LambdaLR,
}


def build_scheduler(cfg, optimizer):

    assert cfg.scheduler.type in _scheduler_factory

    cfg_cp = cfg.scheduler.copy()
    cfg_cp.pop('type')

    scheduler = _scheduler_factory[cfg.scheduler.type](optimizer, **cfg_cp)


    return scheduler
```

[18]:
```
import torch
import os
from torch import nn
import numpy as np
import torch.nn.functional
from termcolor import colored

def save_model(net, optim, scheduler, recorder, is_best=False):
    model_dir = os.path.join(recorder.work_dir, 'ckpt')
    os.system('mkdir -p {}'.format(model_dir))
    epoch = recorder.epoch
    ckpt_name = 'best' if is_best else epoch
    torch.save({
        'net': net.state_dict(),
```

```
        'optim': optim.state_dict(),
        'scheduler': scheduler.state_dict(),
        'recorder': recorder.state_dict(),
        'epoch': epoch
    }, os.path.join(model_dir, '{}.pth'.format(ckpt_name)))


def load_network_specified(net, model_dir, logger=None):
    pretrained_net = torch.load(model_dir)['net']
    net_state = net.state_dict()
    state = {}
    for k, v in pretrained_net.items():
        if k not in net_state.keys() or v.size() != net_state[k].size():
            if logger:
                logger.info('skip weights: ' + k)
            continue
        state[k] = v
    net.load_state_dict(state, strict=False)


def load_network(net, model_dir, finetune_from=None, logger=None):
    if finetune_from:
        if logger:
            logger.info('Finetune model from: ' + finetune_from)
        load_network_specified(net, finetune_from, logger)
        return
    pretrained_model = torch.load(model_dir)
    net.load_state_dict(pretrained_model['net'], strict=True)
```

[19]:
```
import time
import torch
import numpy as np
from tqdm import tqdm
!pip install pytorch_warmup
import pytorch_warmup as warmup

DEVICE = "cpu"

class Runner(object):
    def __init__(self, cfg):
        self.cfg = cfg
        self.recorder = build_recorder(self.cfg)
        self.net = build_net(self.cfg)
        if DEVICE == "cuda" :
            self.net = torch.nn.parallel.DataParallel(
                    self.net, device_ids = range(self.cfg.gpus)).cuda()
        self.recorder.logger.info('Network: \n' + str(self.net))
```

```
        self.resume()
        self.optimizer = build_optimizer(self.cfg, self.net)
        self.scheduler = build_scheduler(self.cfg, self.optimizer)
        self.evaluator = build_evaluator(self.cfg)
        self.warmup_scheduler = warmup.LinearWarmup(
            self.optimizer, warmup_period=5000)
        self.metric = 0.

    def resume(self):
        if not self.cfg.load_from and not self.cfg.finetune_from:
            return
        load_network(self.net, self.cfg.load_from,
                finetune_from=self.cfg.finetune_from, logger=self.recorder.
↪logger)

    def to_cuda(self, batch):
        for k in batch:
            if k == 'meta':
                continue
            batch[k] = torch.tensor( batch[k] ).cuda()
        return batch

    def train_epoch(self, epoch, train_loader):
        self.net.train()
        end = time.time()
        max_iter = len(train_loader)
        for i, data in enumerate(train_loader):
            if self.recorder.step >= self.cfg.total_iter:
                break
            date_time = time.time() - end
            self.recorder.step += 1
            #print( "Data of dataloader is : " + str( data ))
            #data = self.to_cuda(data)
            output = self.trainer.forward(self.net, data)
            self.optimizer.zero_grad()
            loss = output['loss']
            loss.backward()
            self.optimizer.step()
            self.scheduler.step()
            self.warmup_scheduler.dampen()
            batch_time = time.time() - end
            end = time.time()
            self.recorder.update_loss_stats(output['loss_stats'])
            self.recorder.batch_time.update(batch_time)
            self.recorder.data_time.update(date_time)

            if i % self.cfg.log_interval == 0 or i == max_iter - 1:
```

```
            lr = self.optimizer.param_groups[0]['lr']
            self.recorder.lr = lr
            self.recorder.record('train')

    def train(self , train_dataloader , val_dataloader ):
        self.recorder.logger.info('start training...')
        self.trainer = build_trainer(self.cfg)
        train_loader = train_dataloader #build_dataloader(self.cfg.dataset.
↪train, self.cfg, is_train=True)
        val_loader = val_dataloader #build_dataloader(self.cfg.dataset.val,␣
↪self.cfg, is_train=False)

        for epoch in range(self.cfg.epochs):
            self.recorder.epoch = epoch
            self.train_epoch(epoch, train_loader)
            if (epoch + 1) % self.cfg.save_ep == 0 or epoch == self.cfg.epochs␣
↪- 1:
                self.save_ckpt()
            if (epoch + 1) % self.cfg.eval_ep == 0 or epoch == self.cfg.epochs␣
↪- 1:
                self.validate(val_loader)
            if self.recorder.step >= self.cfg.total_iter:
                break

    def validate(self, val_loader):
        self.net.eval()
        for i, data in enumerate(tqdm(val_loader, desc=f'Validate')):
            if DEVICE == "cuda" :

                data = self.to_cuda(data)
            with torch.no_grad():
                output = self.net(data['img'])
                self.evaluator.evaluate(val_loader.dataset, output, data)

        metric = self.evaluator.summarize()
        if not metric:
            return
        if metric > self.metric:
            self.metric = metric
            self.save_ckpt(is_best=True)
        self.recorder.logger.info('Best metric: ' + str(self.metric))

    def save_ckpt(self, is_best=False):
        save_model(self.net, self.optimizer, self.scheduler,
                self.recorder, is_best)
```

```
WARNING: Retrying (Retry(total=4, connect=None, read=None, redirect=None,
status=None)) after connection broken by
'NewConnectionError('<pip._vendor.urllib3.connection.HTTPSConnection object at
0x785bb420ca00>: Failed to establish a new connection: [Errno -3] Temporary
failure in name resolution')': /simple/pytorch-warmup/
WARNING: Retrying (Retry(total=3, connect=None, read=None,
redirect=None, status=None)) after connection broken by
'NewConnectionError('<pip._vendor.urllib3.connection.HTTPSConnection object at
0x785bb420d690>: Failed to establish a new connection: [Errno -3] Temporary
failure in name resolution')': /simple/pytorch-warmup/
WARNING: Retrying (Retry(total=2, connect=None, read=None,
redirect=None, status=None)) after connection broken by
'NewConnectionError('<pip._vendor.urllib3.connection.HTTPSConnection object at
0x785bb420da50>: Failed to establish a new connection: [Errno -3] Temporary
failure in name resolution')': /simple/pytorch-warmup/
WARNING: Retrying (Retry(total=1, connect=None, read=None,
redirect=None, status=None)) after connection broken by
'NewConnectionError('<pip._vendor.urllib3.connection.HTTPSConnection object at
0x785bb420dc00>: Failed to establish a new connection: [Errno -3] Temporary
failure in name resolution')': /simple/pytorch-warmup/
WARNING: Retrying (Retry(total=0, connect=None, read=None,
redirect=None, status=None)) after connection broken by
'NewConnectionError('<pip._vendor.urllib3.connection.HTTPSConnection object at
0x785bb420ddb0>: Failed to establish a new connection: [Errno -3] Temporary
failure in name resolution')': /simple/pytorch-warmup/
ERROR: Could not find a version that satisfies the requirement
pytorch_warmup (from versions: none)
ERROR: No matching distribution found for pytorch_warmup
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[19], line 6
```

```
      4 from tqdm import tqdm
      5 get_ipython().system('pip install pytorch_warmup')
----> 6 import pytorch_warmup as warmup
      8 DEVICE = "cpu"
     10 class Runner(object):

ModuleNotFoundError: No module named 'pytorch_warmup'
```

[20]:
```
# Copyright (c) Open-MMLab. All rights reserved.
import ast
import os.path as osp
import shutil
import sys
import tempfile
from argparse import Action, ArgumentParser
from collections import abc
from importlib import import_module

!pip install addict
from addict import Dict
from yapf.yapflib.yapf_api import FormatCode


BASE_KEY = '_base_'
DELETE_KEY = '_delete_'
RESERVED_KEYS = ['filename', 'text', 'pretty_text']

def check_file_exist(filename, msg_tmpl='file "{}" does not exist'):
    if not osp.isfile(filename):
        raise FileNotFoundError(msg_tmpl.format(filename))



class ConfigDict(Dict):

    def __missing__(self, name):
        raise KeyError(name)

    def __getattr__(self, name):
        try:
            value = super(ConfigDict, self).__getattr__(name)
        except KeyError:
            ex = AttributeError(f"'{self.__class__.__name__}' object has no "
                                f"attribute '{name}'")
        except Exception as e:
            ex = e
        else:
```

```
            return value
        raise ex


def add_args(parser, cfg, prefix=''):
    for k, v in cfg.items():
        if isinstance(v, str):
            parser.add_argument('--' + prefix + k)
        elif isinstance(v, int):
            parser.add_argument('--' + prefix + k, type=int)
        elif isinstance(v, float):
            parser.add_argument('--' + prefix + k, type=float)
        elif isinstance(v, bool):
            parser.add_argument('--' + prefix + k, action='store_true')
        elif isinstance(v, dict):
            add_args(parser, v, prefix + k + '.')
        elif isinstance(v, abc.Iterable):
            parser.add_argument('--' + prefix + k, type=type(v[0]), nargs='+')
        else:
            print(f'cannot parse key {prefix + k} of type {type(v)}')
    return parser


class Config:
    """A facility for config and config files.
    It supports common file formats as configs: python/json/yaml. The interface
    is the same as a dict object and also allows access config values as
    attributes.
    Example:
        >>> cfg = Config(dict(a=1, b=dict(b1=[0, 1])))
        >>> cfg.a
        1
        >>> cfg.b
        {'b1': [0, 1]}
        >>> cfg.b.b1
        [0, 1]
        >>> cfg = Config.fromfile('tests/data/config/a.py')
        >>> cfg.filename
        "/home/kchen/projects/mmcv/tests/data/config/a.py"
        >>> cfg.item4
        'test'
        >>> cfg
        "Config [path: /home/kchen/projects/mmcv/tests/data/config/a.py]: "
        "{'item1': [1, 2], 'item2': {'a': 0}, 'item3': True, 'item4': 'test'}"
    """

    @staticmethod
```

```
    def _validate_py_syntax(filename):
        with open(filename) as f:
            content = f.read()
        try:
            ast.parse(content)
        except SyntaxError:
            raise SyntaxError('There are syntax errors in config '
                              f'file {filename}')

    @staticmethod
    def _file2dict(filename):
        filename = osp.abspath(osp.expanduser(filename))
        check_file_exist(filename)
        if filename.endswith('.py'):
            with tempfile.TemporaryDirectory() as temp_config_dir:
                temp_config_file = tempfile.NamedTemporaryFile(
                    dir=temp_config_dir, suffix='.py')
                temp_config_name = osp.basename(temp_config_file.name)
                shutil.copyfile(filename,
                                osp.join(temp_config_dir, temp_config_name))
                temp_module_name = osp.splitext(temp_config_name)[0]
                sys.path.insert(0, temp_config_dir)
                Config._validate_py_syntax(filename)
                mod = import_module(temp_module_name)
                sys.path.pop(0)
                cfg_dict = {
                    name: value
                    for name, value in mod.__dict__.items()
                    if not name.startswith('__')
                }
                # delete imported module
                del sys.modules[temp_module_name]
                # close temp file
                temp_config_file.close()
        elif filename.endswith(('.yml', '.yaml', '.json')):
            import mmcv
            cfg_dict = mmcv.load(filename)
        else:
            raise IOError('Only py/yml/yaml/json type are supported now!')

        cfg_text = filename + '\n'
        with open(filename, 'r') as f:
            cfg_text += f.read()

        if BASE_KEY in cfg_dict:
            cfg_dir = osp.dirname(filename)
            base_filename = cfg_dict.pop(BASE_KEY)
```

```
            base_filename = base_filename if isinstance(
                base_filename, list) else [base_filename]

            cfg_dict_list = list()
            cfg_text_list = list()
            for f in base_filename:
                _cfg_dict, _cfg_text = Config._file2dict(osp.join(cfg_dir, f))
                cfg_dict_list.append(_cfg_dict)
                cfg_text_list.append(_cfg_text)

            base_cfg_dict = dict()
            for c in cfg_dict_list:
                if len(base_cfg_dict.keys() & c.keys()) > 0:
                    raise KeyError('Duplicate key is not allowed among bases')
                base_cfg_dict.update(c)

            base_cfg_dict = Config._merge_a_into_b(cfg_dict, base_cfg_dict)
            cfg_dict = base_cfg_dict

            # merge cfg_text
            cfg_text_list.append(cfg_text)
            cfg_text = '\n'.join(cfg_text_list)

        return cfg_dict, cfg_text

    @staticmethod
    def _merge_a_into_b(a, b):
        # merge dict `a` into dict `b` (non-inplace). values in `a` will
        # overwrite `b`.
        # copy first to avoid inplace modification
        b = b.copy()
        for k, v in a.items():
            if isinstance(v, dict) and k in b and not v.pop(DELETE_KEY, False):
                if not isinstance(b[k], dict):
                    raise TypeError(
                        f'{k}={v} in child config cannot inherit from base '
                        f'because {k} is a dict in the child config but is of '
                        f'type {type(b[k])} in base config. You may set '
                        f'`{DELETE_KEY}=True` to ignore the base config')
                b[k] = Config._merge_a_into_b(v, b[k])
            else:
                b[k] = v
        return b

    @staticmethod
    def fromfile(filename):
        cfg_dict, cfg_text = Config._file2dict(filename)
```

```
        return Config(cfg_dict, cfg_text=cfg_text, filename=filename)

    @staticmethod
    def auto_argparser(description=None):
        """Generate argparser from config file automatically (experimental)
        """
        partial_parser = ArgumentParser(description=description)
        partial_parser.add_argument('config', help='config file path')
        cfg_file = partial_parser.parse_known_args()[0].config
        cfg = Config.fromfile(cfg_file)
        parser = ArgumentParser(description=description)
        parser.add_argument('config', help='config file path')
        add_args(parser, cfg)
        return parser, cfg

    def __init__(self, cfg_dict=None, cfg_text=None, filename=None):
        if cfg_dict is None:
            cfg_dict = dict()
        elif not isinstance(cfg_dict, dict):
            raise TypeError('cfg_dict must be a dict, but '
                            f'got {type(cfg_dict)}')
        for key in cfg_dict:
            if key in RESERVED_KEYS:
                raise KeyError(f'{key} is reserved for config file')

        super(Config, self).__setattr__('_cfg_dict', ConfigDict(cfg_dict))
        super(Config, self).__setattr__('_filename', filename)
        if cfg_text:
            text = cfg_text
        elif filename:
            with open(filename, 'r') as f:
                text = f.read()
        else:
            text = ''
        super(Config, self).__setattr__('_text', text)

    @property
    def filename(self):
        return self._filename

    @property
    def text(self):
        return self._text

    @property
    def pretty_text(self):
```

```
        indent = 4

        def _indent(s_, num_spaces):
            s = s_.split('\n')
            if len(s) == 1:
                return s_
            first = s.pop(0)
            s = [(num_spaces * ' ') + line for line in s]
            s = '\n'.join(s)
            s = first + '\n' + s
            return s

        def _format_basic_types(k, v, use_mapping=False):
            if isinstance(v, str):
                v_str = f"'{v}'"
            else:
                v_str = str(v)

            if use_mapping:
                k_str = f"'{k}'" if isinstance(k, str) else str(k)
                attr_str = f'{k_str}: {v_str}'
            else:
                attr_str = f'{str(k)}={v_str}'
            attr_str = _indent(attr_str, indent)

            return attr_str

        def _format_list(k, v, use_mapping=False):
            # check if all items in the list are dict
            if all(isinstance(_, dict) for _ in v):
                v_str = '[\n'
                v_str += '\n'.join(
                    f'dict({_indent(_format_dict(v_), indent)}),'
                    for v_ in v).rstrip(',')
                if use_mapping:
                    k_str = f"'{k}'" if isinstance(k, str) else str(k)
                    attr_str = f'{k_str}: {v_str}'
                else:
                    attr_str = f'{str(k)}={v_str}'
                attr_str = _indent(attr_str, indent) + ']'
            else:
                attr_str = _format_basic_types(k, v, use_mapping)
            return attr_str

        def _contain_invalid_identifier(dict_str):
            contain_invalid_identifier = False
            for key_name in dict_str:
```

```
                contain_invalid_identifier |= \
                    (not str(key_name).isidentifier())
            return contain_invalid_identifier

        def _format_dict(input_dict, outest_level=False):
            r = ''
            s = []

            use_mapping = _contain_invalid_identifier(input_dict)
            if use_mapping:
                r += '{'
            for idx, (k, v) in enumerate(input_dict.items()):
                is_last = idx >= len(input_dict) - 1
                end = '' if outest_level or is_last else ','
                if isinstance(v, dict):
                    v_str = '\n' + _format_dict(v)
                    if use_mapping:
                        k_str = f"'{k}'" if isinstance(k, str) else str(k)
                        attr_str = f'{k_str}: dict({v_str}'
                    else:
                        attr_str = f'{str(k)}=dict({v_str}'
                    attr_str = _indent(attr_str, indent) + ')' + end
                elif isinstance(v, list):
                    attr_str = _format_list(k, v, use_mapping) + end
                else:
                    attr_str = _format_basic_types(k, v, use_mapping) + end

                s.append(attr_str)
            r += '\n'.join(s)
            if use_mapping:
                r += '}'
            return r

        cfg_dict = self._cfg_dict.to_dict()
        text = _format_dict(cfg_dict, outest_level=True)
        # copied from setup.cfg
        yapf_style = dict(
            based_on_style='pep8',
            blank_line_before_nested_class_or_def=True,
            split_before_expression_after_opening_paren=True)
        text, _ = FormatCode(text, style_config=yapf_style, verify=True)

        return text

    def __repr__(self):
        return f'Config (path: {self.filename}): {self._cfg_dict.__repr__()}'
```

```
    def __len__(self):
        return len(self._cfg_dict)

    def __getattr__(self, name):
        return getattr(self._cfg_dict, name)

    def __getitem__(self, name):
        return self._cfg_dict.__getitem__(name)

    def __setattr__(self, name, value):
        if isinstance(value, dict):
            value = ConfigDict(value)
        self._cfg_dict.__setattr__(name, value)

    def __setitem__(self, name, value):
        if isinstance(value, dict):
            value = ConfigDict(value)
        self._cfg_dict.__setitem__(name, value)

    def __iter__(self):
        return iter(self._cfg_dict)

    def dump(self, file=None):
        cfg_dict = super(Config, self).__getattribute__('_cfg_dict').to_dict()
        if self.filename.endswith('.py'):
            if file is None:
                return self.pretty_text
            else:
                with open(file, 'w') as f:
                    f.write(self.pretty_text)
        else:
            import mmcv
            if file is None:
                file_format = self.filename.split('.')[-1]
                return mmcv.dump(cfg_dict, file_format=file_format)
            else:
                mmcv.dump(cfg_dict, file)

    def merge_from_dict(self, options):
        """Merge list into cfg_dict
        Merge the dict parsed by MultipleKVAction into this cfg.
        Examples:
            >>> options = {'model.backbone.depth': 50,
            ...            'model.backbone.with_cp':True}
            >>> cfg = Config(dict(model=dict(backbone=dict(type='ResNet'))))
            >>> cfg.merge_from_dict(options)
            >>> cfg_dict = super(Config, self).__getattribute__('_cfg_dict')
```

```
            >>> assert cfg_dict == dict(
            ...     model=dict(backbone=dict(depth=50, with_cp=True)))
        Args:
            options (dict): dict of configs to merge from.
        """
        option_cfg_dict = {}
        for full_key, v in options.items():
            d = option_cfg_dict
            key_list = full_key.split('.')
            for subkey in key_list[:-1]:
                d.setdefault(subkey, ConfigDict())
                d = d[subkey]
            subkey = key_list[-1]
            d[subkey] = v

        cfg_dict = super(Config, self).__getattribute__('_cfg_dict')
        super(Config, self).__setattr__(
            '_cfg_dict', Config._merge_a_into_b(option_cfg_dict, cfg_dict))


class DictAction(Action):
    """
    argparse action to split an argument into KEY=VALUE form
    on the first = and append to a dictionary. List options should
    be passed as comma separated values, i.e KEY=V1,V2,V3
    """

    @staticmethod
    def _parse_int_float_bool(val):
        try:
            return int(val)
        except ValueError:
            pass
        try:
            return float(val)
        except ValueError:
            pass
        if val.lower() in ['true', 'false']:
            return True if val.lower() == 'true' else False
        return val

    def __call__(self, parser, namespace, values, option_string=None):
        options = {}
        for kv in values:
            key, val = kv.split('=', maxsplit=1)
            val = [self._parse_int_float_bool(v) for v in val.split(',')]
            if len(val) == 1:
```

```
                val = val[0]
            options[key] = val
        setattr(namespace, self.dest, options)
```

```
WARNING: Retrying (Retry(total=4, connect=None, read=None, redirect=None,
status=None)) after connection broken by
'NewConnectionError('<pip._vendor.urllib3.connection.HTTPSConnection object at
0x7ebf5616d300>: Failed to establish a new connection: [Errno -3] Temporary
failure in name resolution')': /simple/addict/
WARNING: Retrying (Retry(total=3, connect=None, read=None,
redirect=None, status=None)) after connection broken by
'NewConnectionError('<pip._vendor.urllib3.connection.HTTPSConnection object at
0x7ebf5616d4b0>: Failed to establish a new connection: [Errno -3] Temporary
failure in name resolution')': /simple/addict/
WARNING: Retrying (Retry(total=2, connect=None, read=None,
redirect=None, status=None)) after connection broken by
'NewConnectionError('<pip._vendor.urllib3.connection.HTTPSConnection object at
0x7ebf5616da50>: Failed to establish a new connection: [Errno -3] Temporary
failure in name resolution')': /simple/addict/
WARNING: Retrying (Retry(total=1, connect=None, read=None,
redirect=None, status=None)) after connection broken by
'NewConnectionError('<pip._vendor.urllib3.connection.HTTPSConnection object at
0x7ebf5616dc00>: Failed to establish a new connection: [Errno -3] Temporary
failure in name resolution')': /simple/addict/
WARNING: Retrying (Retry(total=0, connect=None, read=None,
redirect=None, status=None)) after connection broken by
'NewConnectionError('<pip._vendor.urllib3.connection.HTTPSConnection object at
0x7ebf5616ddb0>: Failed to establish a new connection: [Errno -3] Temporary
failure in name resolution')': /simple/addict/
ERROR: Could not find a version that satisfies the requirement addict
(from versions: none)
ERROR: No matching distribution found for addict
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[20], line 12
      9 from importlib import import_module
     11 get_ipython().system('pip install addict')
---> 12 from addict import Dict
     13 from yapf.yapflib.yapf_api import FormatCode
     16 BASE_KEY = '_base_'

ModuleNotFoundError: No module named 'addict'
```

[21]:
```python
import numpy as np
from sklearn.linear_model import LinearRegression
import json as json


class LaneEval(object):
    lr = LinearRegression()
    pixel_thresh = 20
    pt_thresh = 0.85

    @staticmethod
    def get_angle(xs, y_samples):
        xs, ys = xs[xs >= 0], y_samples[xs >= 0]
        if len(xs) > 1:
            LaneEval.lr.fit(ys[:, None], xs)
            k = LaneEval.lr.coef_[0]
            theta = np.arctan(k)
        else:
            theta = 0
        return theta

    @staticmethod
    def line_accuracy(pred, gt, thresh):
        pred = np.array([p if p >= 0 else -100 for p in pred])
        gt = np.array([g if g >= 0 else -100 for g in gt])
        return np.sum(np.where(np.abs(pred - gt) < thresh, 1., 0.)) / len(gt)

    @staticmethod
    def bench(pred, gt, y_samples, running_time):
        if any(len(p) != len(y_samples) for p in pred):
            raise Exception('Format of lanes error.')
        if running_time > 200 or len(gt) + 2 < len(pred):
            return 0., 0., 1.
        angles = [LaneEval.get_angle(
            np.array(x_gts), np.array(y_samples)) for x_gts in gt]
```

```
        threshs = [LaneEval.pixel_thresh / np.cos(angle) for angle in angles]
        line_accs = []
        fp, fn = 0., 0.
        matched = 0.
        for x_gts, thresh in zip(gt, threshs):
            accs = [LaneEval.line_accuracy(
                np.array(x_preds), np.array(x_gts), thresh) for x_preds in pred]
            max_acc = np.max(accs) if len(accs) > 0 else 0.
            if max_acc < LaneEval.pt_thresh:
                fn += 1
            else:
                matched += 1
            line_accs.append(max_acc)
        fp = len(pred) - matched
        if len(gt) > 4 and fn > 0:
            fn -= 1
        s = sum(line_accs)
        if len(gt) > 4:
            s -= min(line_accs)
        return s / max(min(4.0, len(gt)), 1.), fp / len(pred) if len(pred) > 0␣
↪else 0., fn / max(min(len(gt), 4.), 1.), matched

    @staticmethod
    def bench_one_submit(pred_file, gt_file):
        try:
            json_pred = [json.loads(line)
                         for line in open(pred_file).readlines()]
        except BaseException as e:
            raise Exception('Fail to load json file of the prediction.')
        json_gt = [json.loads(line) for line in open(gt_file).readlines()]
        if len(json_gt) != len(json_pred):
            raise Exception(
                'We do not get the predictions of all the test tasks')
        gts = {l['raw_file']: l for l in json_gt}
        accuracy, fp, fn , tp= 0., 0., 0. , 0.
        for pred in json_pred:
            if 'raw_file' not in pred or 'lanes' not in pred or 'run_time' not␣
↪in pred:
                raise Exception(
                    'raw_file or lanes or run_time not in some predictions.')
            raw_file = pred['raw_file']
            pred_lanes = pred['lanes']
            run_time = pred['run_time']
            if raw_file not in gts:
                raise Exception(
                    'Some raw_file from your predictions do not exist in the␣
↪test tasks.')
```

```
            gt = gts[raw_file]
            gt_lanes = gt['lanes']
            y_samples = gt['h_samples']
            try:
                a, p, n , matched= LaneEval.bench(
                    pred_lanes, gt_lanes, y_samples, run_time)
            except BaseException as e:
                raise Exception('Format of lanes error.')
            accuracy += a
            fp += p
            fn += n
            tp = tp + matched
        num = len(gts)
        # the first return parameter is the default ranking parameter
        recall = tp/( tp + fp )
        precision = tp/( tp + fn )
        return json.dumps([
            {'name': 'Accuracy', 'value': accuracy / num, 'order': 'desc'},
            {'name': 'FP', 'value': fp / num, 'order': 'asc'},
            {'name': 'FN', 'value': fn / num, 'order': 'asc'},
            {"name" : "F1- Score" , "value" : 2/(1/recall + 1/precision) if tp␣
 ↪> 0 else 0 , "order" : "asc" }
        ]), accuracy / num

def split_path(path):
    """split path tree into list"""
    folders = []
    while True:
        path, folder = os.path.split(path)
        if folder != "":
            folders.insert(0, folder)
        else:
            if path != "":
                folders.insert(0, path)
            break
    return folders


@EVALUATOR.register_module
class Tusimple(nn.Module):
    def __init__(self, cfg):
        super(Tusimple, self).__init__()
        self.cfg = cfg
        exp_dir = os.path.join(self.cfg.work_dir, "output")
        if not os.path.exists(exp_dir):
            os.mkdir(exp_dir)
        self.out_path = os.path.join(exp_dir, "coord_output")
```

```
        if not os.path.exists(self.out_path):
            os.mkdir(self.out_path)
        self.dump_to_json = []
        self.thresh = cfg.evaluator.thresh
        self.logger = get_logger('resa')
        if cfg.view:
            self.view_dir = os.path.join(self.cfg.work_dir, 'vis')

    def evaluate_pred(self, dataset, seg_pred, exist_pred, batch):
        img_name = batch['meta']['img_name']
        img_path = batch['meta']['full_img_path']
        for b in range(len(seg_pred)):
            seg = seg_pred[b]
            exist = [1 if exist_pred[b, i] >
                     0.5 else 0 for i in range(self.cfg.num_classes-1)]
            lane_coords = dataset.probmap2lane(seg, exist, thresh = self.thresh)
            for i in range(len(lane_coords)):
                lane_coords[i] = sorted(
                    lane_coords[i], key=lambda pair: pair[1])

            path_tree = split_path(img_name[b])
            save_dir, save_name = path_tree[-3:-1], path_tree[-1]
            save_dir = os.path.join(self.out_path, *save_dir)
            save_name = save_name[:-3] + "lines.txt"
            save_name = os.path.join(save_dir, save_name)
            if not os.path.exists(save_dir):
                os.makedirs(save_dir, exist_ok=True)

            with open(save_name, "w") as f:
                for l in lane_coords:
                    for (x, y) in l:
                        print("{} {}".format(x, y), end=" ", file=f)
                    print(file=f)

            json_dict = {}
            json_dict['lanes'] = []
            json_dict['h_sample'] = []
            json_dict['raw_file'] = os.path.join(*path_tree[-4:])
            json_dict['run_time'] = 0
            for l in lane_coords:
                if len(l) == 0:
                    continue
                json_dict['lanes'].append([])
                for (x, y) in l:
                    json_dict['lanes'][-1].append(int(x))
            for (x, y) in lane_coords[0]:
                json_dict['h_sample'].append(y)
```

```
            self.dump_to_json.append(json.dumps(json_dict))
            if self.cfg.view:
                img = cv2.imread(img_path[b])
                new_img_name = img_name[b].replace('/', '_')
                save_dir = os.path.join(self.view_dir, new_img_name)
                dataset.view(img, lane_coords, save_dir)


    def evaluate(self, dataset, output, batch):
        seg_pred, exist_pred = output['seg'], output['exist']
        seg_pred = F.softmax(seg_pred, dim=1)
        seg_pred = seg_pred.detach().cpu().numpy()
        exist_pred = exist_pred.detach().cpu().numpy()
        self.evaluate_pred(dataset, seg_pred, exist_pred, batch)

    def summarize(self):
        best_acc = 0
        output_file = os.path.join(self.out_path, 'predict_test.json')
        with open(output_file, "w+") as f:
            for line in self.dump_to_json:
                print(line, end="\n", file=f)

        eval_result, acc = LaneEval.bench_one_submit(output_file,
                            self.cfg.test_json_file)

        self.logger.info(eval_result)
        self.dump_to_json = []
        best_acc = max(acc, best_acc)
        return best_acc
```

[22]:
```
def dice_loss(input, target):
    input = input.contiguous().view(input.size()[0], -1)
    target = target.contiguous().view(target.size()[0], -1).float()

    a = torch.sum(input * target, 1)
    b = torch.sum(input * input, 1) + 0.001
    c = torch.sum(target * target, 1) + 0.001
    d = (2 * a) / (b + c)
    return (1-d).mean()

@TRAINER.register_module
class RESA(nn.Module):
    def __init__(self, cfg):
        super(RESA, self).__init__()
        self.cfg = cfg
        self.loss_type = cfg.loss_type
        if self.loss_type == 'cross_entropy':
```

```
            weights = torch.ones(cfg.num_classes)
            weights[0] = cfg.bg_weight
            if DEVICE == "cuda" :
                weights = weights.cuda()
            if DEVICE == "cuda" :
                self.criterion = torch.nn.NLLLoss(ignore_index=self.cfg.
↪ignore_label,
                                                  weight=weights).cuda()
            else :
                self.criterion = torch.nn.NLLLoss(ignore_index=self.cfg.
↪ignore_label,
                                                  weight=weights).cpu()

        if DEVICE == "cuda" :
            self.criterion_exist = torch.nn.BCEWithLogitsLoss().cuda()
        else :
            self.criterion_exist = torch.nn.BCEWithLogitsLoss().cpu()

    def forward(self, net, batch):
        output = net(batch['img'])

        loss_stats = {}
        loss = 0.

        if self.loss_type == 'dice_loss':
            target = F.one_hot(batch['label'], num_classes=self.cfg.
↪num_classes).permute(0, 3, 1, 2)
            seg_loss = dice_loss(F.softmax(
                output['seg'], dim=1)[:, 1:], target[:, 1:])
        else:
            seg_loss = self.criterion(F.log_softmax(
                output['seg'], dim=1), batch['label'].long())

        loss += seg_loss * self.cfg.seg_loss_weight

        loss_stats.update({'seg_loss': seg_loss})

        if 'exist' in output:
            exist_loss = 0.1 * \
                self.criterion_exist(output['exist'], batch['exist'].float())
            loss += exist_loss
            loss_stats.update({'exist_loss': exist_loss})

        ret = {'loss': loss, 'loss_stats': loss_stats}

        return ret
```

3 3. Process TUSimple Dataset Lane Detection

[23]:
```
dataset = LaneDataset(size=(880, 368))

train_size = int(0.85 * len(dataset))
val_size = len(dataset) - train_size
train_ds, val_ds = torch.utils.data.random_split(dataset, [train_size,␣
 ↪val_size], generator=torch.Generator().manual_seed(42))

train_dataloader = torch.utils.data.DataLoader(train_ds, batch_size=8,␣
 ↪shuffle=False)
val_dataloader = torch.utils.data.DataLoader(val_ds, batch_size=8,␣
 ↪shuffle=False)
```

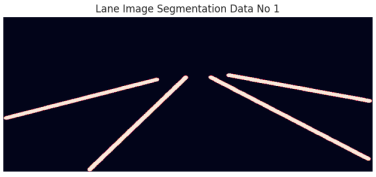
[24]:
```
print(train_dataloader.__len__())
print(val_dataloader.__len__())
```

```
386
68
```

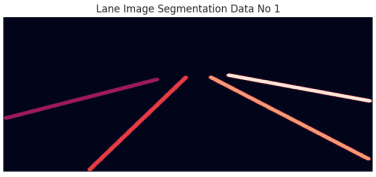
[25]:
```
# Show sample image of TUSimple Lane Detection Dataset

dataset._show_sample_dataset( number_samples= 10 )
```

Lane Image Data No 1

Lane Image Segmentation Data No 1

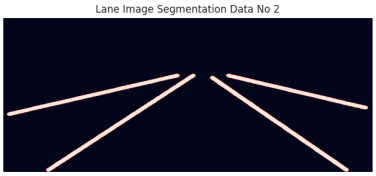


Lane Image Segmentation Data No 1

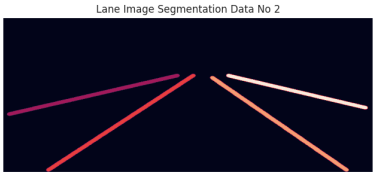


Lane Image Data No 2

Lane Image Segmentation Data No 2

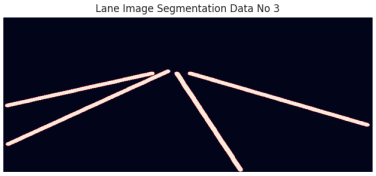


Lane Image Segmentation Data No 2

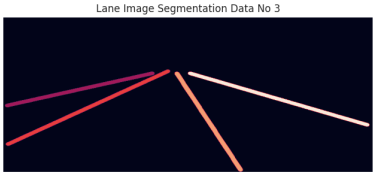


Lane Image Data No 3

Lane Image Segmentation Data No 3

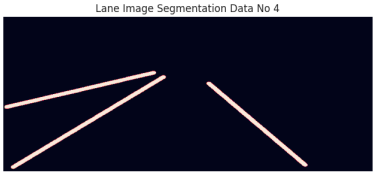


Lane Image Segmentation Data No 3

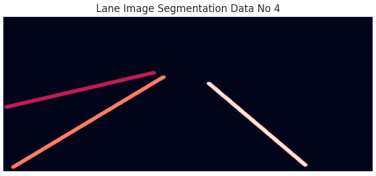


Lane Image Data No 4

Lane Image Segmentation Data No 4

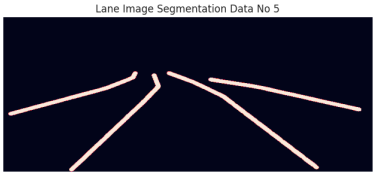


Lane Image Segmentation Data No 4

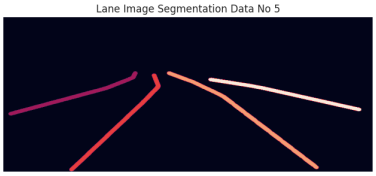


Lane Image Data No 5

Lane Image Segmentation Data No 5

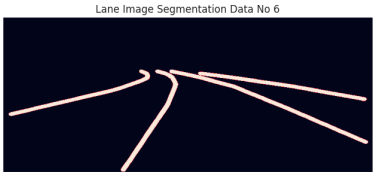


Lane Image Segmentation Data No 5

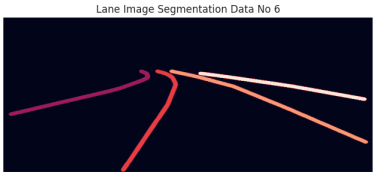


Lane Image Data No 6

Lane Image Segmentation Data No 6

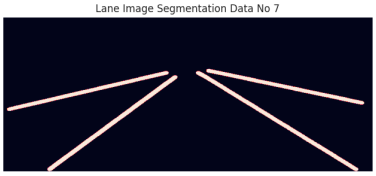


Lane Image Segmentation Data No 6

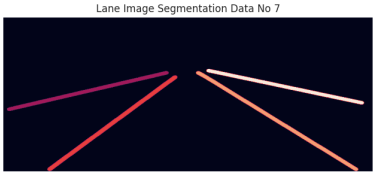


Lane Image Data No 7

Lane Image Segmentation Data No 7

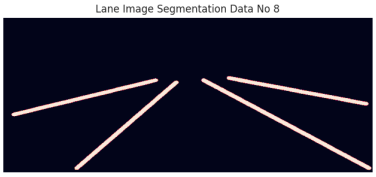


Lane Image Segmentation Data No 7

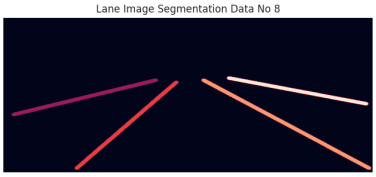


Lane Image Data No 8

Lane Image Segmentation Data No 8

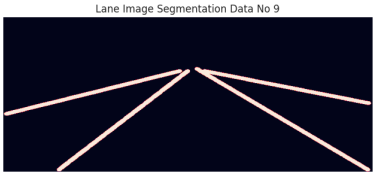


Lane Image Segmentation Data No 8

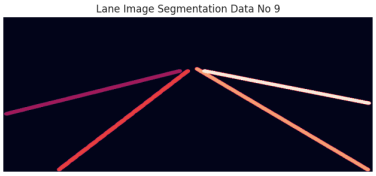


Lane Image Data No 9

Lane Image Segmentation Data No 9

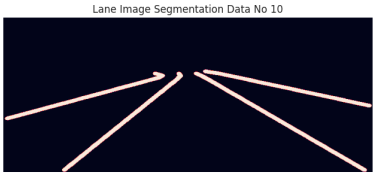


Lane Image Segmentation Data No 9

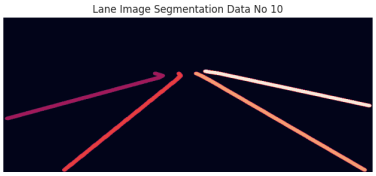


Lane Image Data No 10

Lane Image Segmentation Data No 10



Lane Image Segmentation Data No 10



[ ]:

4 4. Training Reccurrent Shift Aggregator CNN Lane Detection

[26]:
```
import math

dataset_path = './data/tusimple',
test_json_file = './data/tusimple/test_label.json',
epochs = 72 #300
total_iter = 20000 #80000

training_configs = dict( net = dict(
    type='RESANet',
) ,

backbone = dict(
    type='ResNetWrapper',
    resnet='resnet50',
    pretrained=True,
    replace_stride_with_dilation=[False, True, True],
    out_conv=True,
    fea_stride=8,
),

resa = dict(
    type='RESA',
    alpha=2.0,
    iter=5,
    input_channel=128,
    conv_stride=9,
),

decoder = 'BUSD',

trainer = dict(
    type='RESA'
),

evaluator = dict(
    type='Tusimple',
    thresh = 0.60
),

optimizer = dict(
```

```
  type='sgd',
  lr=0.020,
  weight_decay=1e-4,
  momentum=0.9
),

total_iter = 80000,

scheduler = dict(
    type = 'LambdaLR',
    lr_lambda = lambda _iter : math.pow(1 - _iter/total_iter, 0.9)
),

bg_weight = 0.4,

img_norm = dict(
    mean=[103.939, 116.779, 123.68],
    std=[1., 1., 1.]
),

img_height = 368, #368,
img_width = 880 , #640,
cut_height = 160,
seg_label = "seg_label",

dataset_path = './data/tusimple',
test_json_file = './data/tusimple/test_label.json',

dataset = dict(
    train=dict(
        type='TuSimple',
        img_path=dataset_path,
        data_list='train_val_gt.txt',
    ),
    val=dict(
        type='TuSimple',
        img_path=dataset_path,
        data_list='test_gt.txt'
    ),
    test=dict(
        type='TuSimple',
        img_path=dataset_path,
        data_list='test_gt.txt'
    )
),
```

```
loss_type = 'cross_entropy',
seg_loss_weight = 1.0,


batch_size = 4,
workers = 12,
num_classes = 6 + 1,
ignore_label = 255,
epochs = 36, #300,
log_interval = 100,
eval_ep = 1,
save_ep = epochs,
log_note = ''
                        )
```

[27]:
```
import numpy as np
import argparse
import torch.nn.parallel
import torch.backends.cudnn as cudnn


def main( is_training = True ) :
    #rgs = parse_args()
    #s.environ["CUDA_VISIBLE_DEVICES"] = ','.join(str(gpu) for gpu in args.gpus)

    #cfg = Config.fromfile(args.config)
    cfg = Config( training_configs )
    cfg.gpus = 1 #(args.gpus)

    cfg.load_from = None #d_from
    cfg.finetune_from = None #inetune_from
    cfg.view = True #iew

    cfg.work_dirs = "."#.work_dirs + '/' + cfg.dataset.train.type

    cudnn.benchmark = True
    cudnn.fastest = True

    runner = Runner(cfg)

    if not is_training :
        val_loader = build_dataloader(cfg.dataset.val, cfg, is_train=False)
        runner.validate(val_loader)
    else:
        runner.train( train_dataloader = train_dataloader ,
                      val_dataloader= val_dataloader )
```

```
"""
def parse_args():
    parser = argparse.ArgumentParser(description='Train a detector')
    parser.add_argument('config', help='train config file path')
    parser.add_argument(
        '--work_dirs', type=str, default='work_dirs',
        help='work dirs')
    parser.add_argument(
        '--load_from', default=None,
        help='the checkpoint file to resume from')
    parser.add_argument(
        '--finetune_from', default=None,
        help='whether to finetune from the checkpoint')
    parser.add_argument(
        '--validate',
        action='store_true',
        help='whether to evaluate the checkpoint during training')
    parser.add_argument(
        '--view',
        action='store_true',
        help='whether to show visualization result')
    parser.add_argument('--gpus', nargs='+', type=int, default='0')
    parser.add_argument('--seed', type=int,
                        default=None, help='random seed')
    args = parser.parse_args()

    return args
"""
```

[27]:
```
"\ndef parse_args():\n    parser = argparse.ArgumentParser(description='Train a
detector')\n    parser.add_argument('config', help='train config file path')\n
parser.add_argument(\n        '--work_dirs', type=str, default='work_dirs',\n
help='work dirs')\n    parser.add_argument(\n        '--load_from',
default=None,\n        help='the checkpoint file to resume from')\n
parser.add_argument(\n        '--finetune_from', default=None,\n
help='whether to finetune from the checkpoint')\n    parser.add_argument(\n
'--validate',\n        action='store_true',\n        help='whether to evaluate
the checkpoint during training')\n    parser.add_argument(\n        '--view',\n
action='store_true',\n        help='whether to show visualization result')\n
parser.add_argument('--gpus', nargs='+', type=int, default='0')\n
parser.add_argument('--seed', type=int,\n                        default=None,
help='random seed')\n    args = parser.parse_args()\n\n    return args\n"
```

[28]:
```
#!watch -n 1 nvidia-smi
```

[29]:
```
main()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[29], line 1
----> 1 main()

Cell In[27], line 12, in main(is_training)
      7 def main( is_training = True ) :
      8     #rgs = parse_args()
      9     #s.environ["CUDA_VISIBLE_DEVICES"] = ','.join(str(gpu) for gpu in␣
 ↪args.gpus)
     10
     11     #cfg = Config.fromfile(args.config)
---> 12     cfg = Config( training_configs )
     13     cfg.gpus = 1 #(args.gpus)
     15     cfg.load_from = None #d_from

NameError: name 'Config' is not defined
```

[ ]: