DAFTAR PUSTAKA

Bendjillali, R. I., Beladgham, M., Merit, K., & Taleb-Ahmed, A. (2020). Illumination-robust face recognition based on deep convolutional neural networks architectures. *Indonesian Journal of Electrical Engineering and Computer Science*, *18*(2), 1015–1027. https://doi.org/10.11591/ijeecs.v18.i2.pp1015-1027

Federal Automated Vehicles Policy Accelerating the Next Revolution In Roadway Safety. (2016).

Girshick, R. (2015). *Fast R-CNN*. http://arxiv.org/abs/1504.08083

Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2013). *Rich feature hierarchies for accurate object detection and semantic segmentation*. http://arxiv.org/abs/1311.2524

Grandini, M., Bagli, E., & Visani, G. (2020). *Metrics for Multi-Class Classification: an Overview*. http://arxiv.org/abs/2008.05756

Handayani, D., Ophelia, R. O., & Hartono, W. (2017). *PENGARUH PELANGGARAN LALU LINTAS TERHADAP POTENSI KECELAKAAN PADA REMAJA PENGENDARA SEPEDA MOTOR*.

Hussain, R., & Zeadally, S. (2019). Autonomous Cars: Research Results, Issues, and Future Challenges. In *IEEE Communications Surveys and Tutorials* (Vol. 21, Issue 2, pp. 1275–1313). Institute of Electrical and Electronics Engineers Inc. https://doi.org/10.1109/COMST.2018.2869360

Kohavi, R., & Provost, F. (1998). *Glossary of Terms* (Vol. 30, Issue 3).

Kong, Q., Shi, Z., Feng, Y., Yang, M., Zhang, M., Zeng, S., Li, R., Yu, K., & Shen, J. (2020). Classification Method of Ethnic Minority Patterns Based on Faster R-CNN.

Journal of Physics: Conference Series, *1575*(1). https://doi.org/10.1088/1742-6596/1575/1/012137

Ren, S., He, K., Girshick, R., & Sun, J. (2017). Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *39*(6), 1137–1149. https://doi.org/10.1109/TPAMI.2016.2577031

Sabri, M., & Fauzan, A. (2018). *Analysis of vehicle braking behaviour and distance stopping. IOP Conf. Series: Materials Science and Engineering.* https://doi.org/10.1088/1757-899X/309/1/012020

Sirbu, M. A., Baiasu, A., Bogdan, R., & Crisan-Vida, M. (2018, December 19). Smart traffic sign detection on autonomous car. *2018 13th International Symposium on Electronics and Telecommunications, ISETC 2018 - Conference Proceedings*. https://doi.org/10.1109/ISETC.2018.8583948

Sjafrie, H. (2020). *Introduction to Self-Driving Vehicle Technology*. CRC Press.

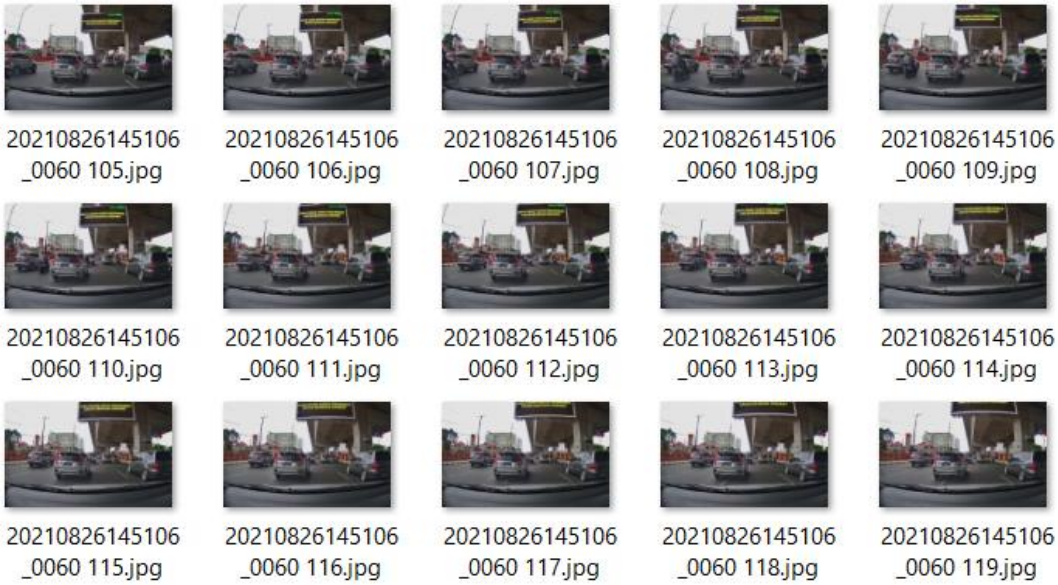
Thrun, S. (2010). Toward robotic cars. *Communications of the ACM*, *53*(4), 99–106. https://doi.org/10.1145/1721654.1721679

Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (2022). *Dive into Deep Learning Release 0.17.5*.

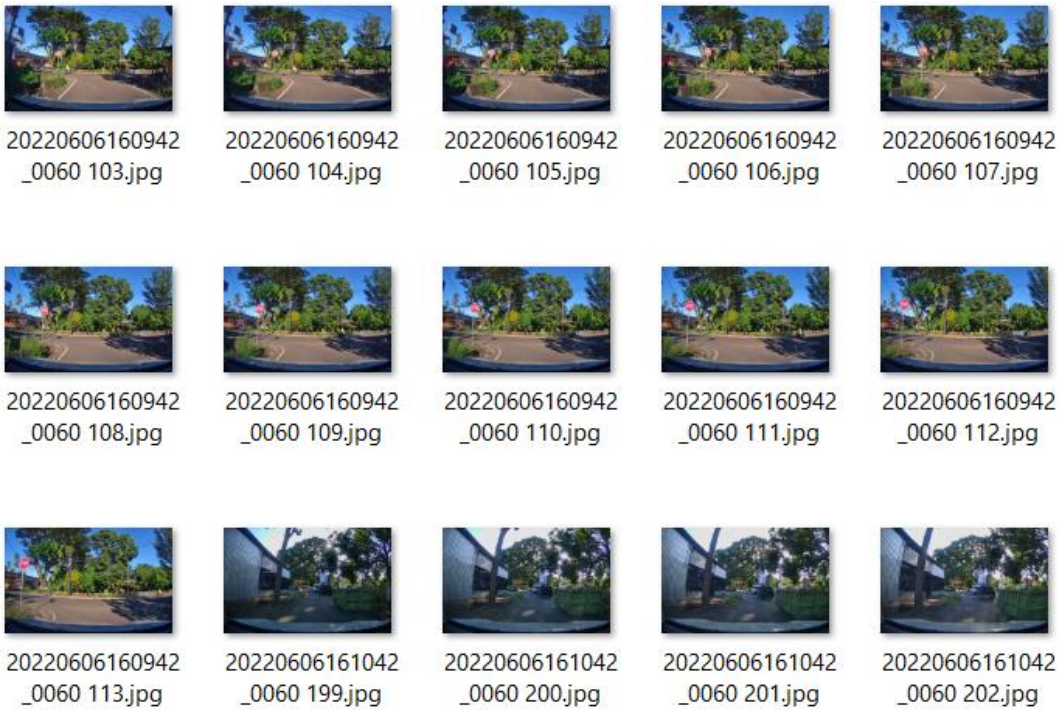
LAMPIRAN

Lampiran 1: Contoh Dataset

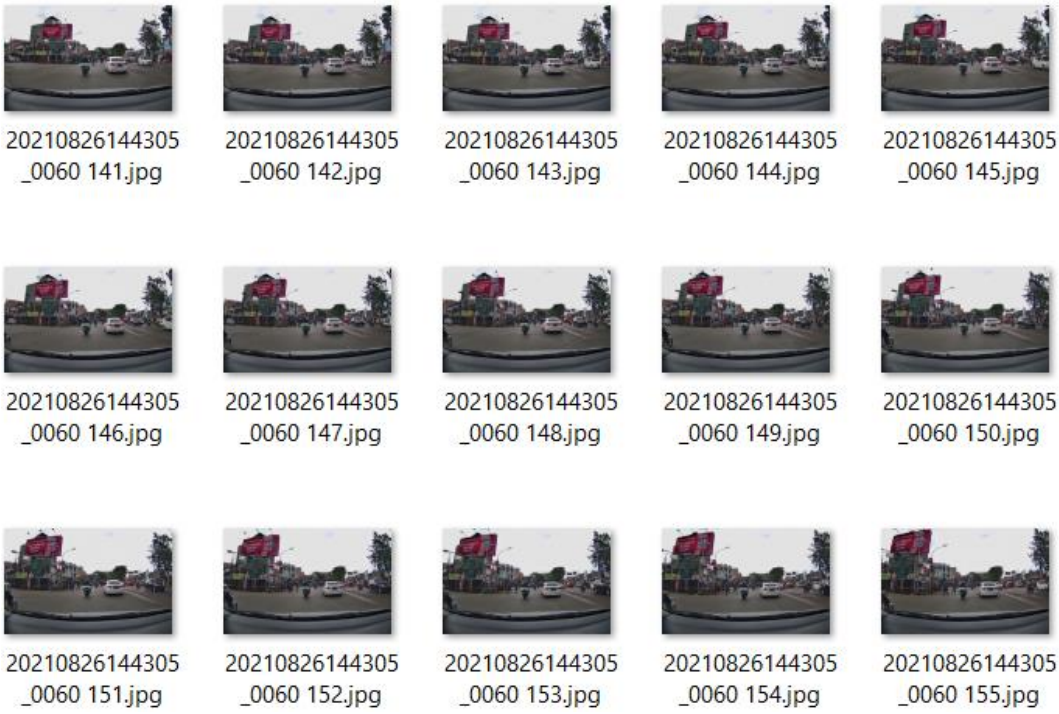
- Kelas 40km



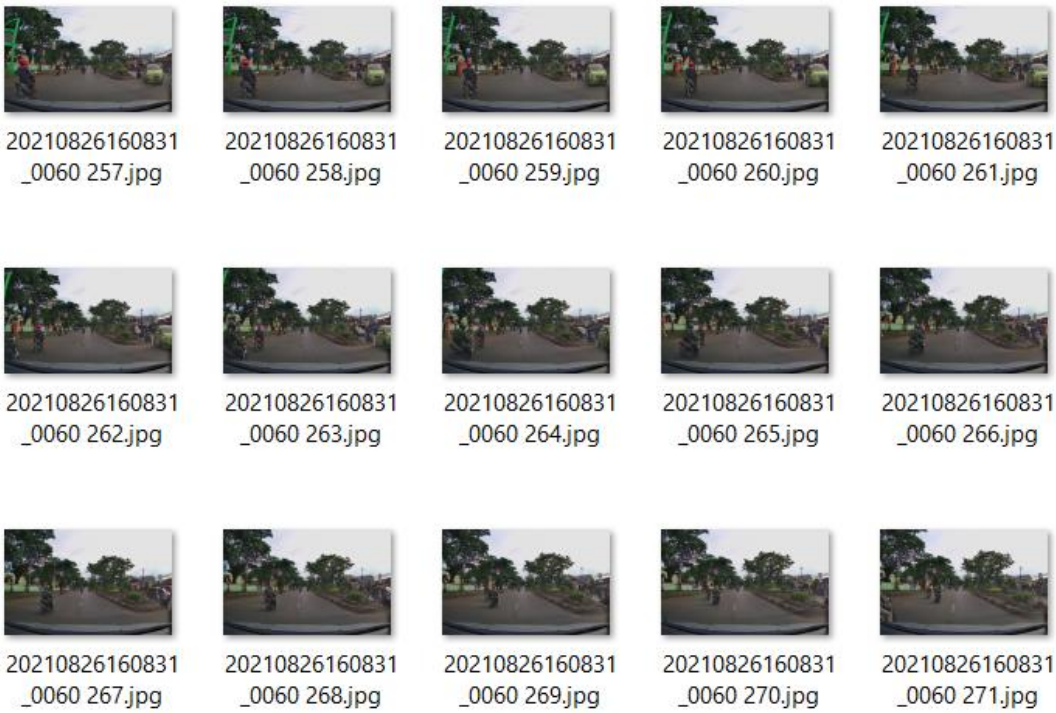
- Kelas berhenti



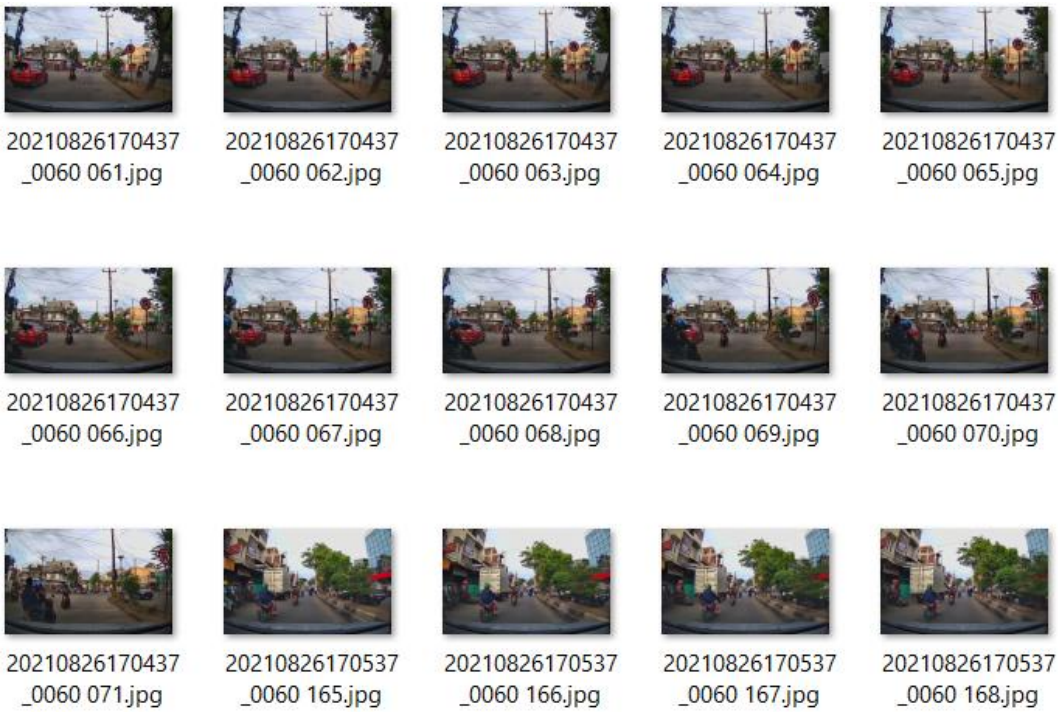
- Kelas dilarang_melintas



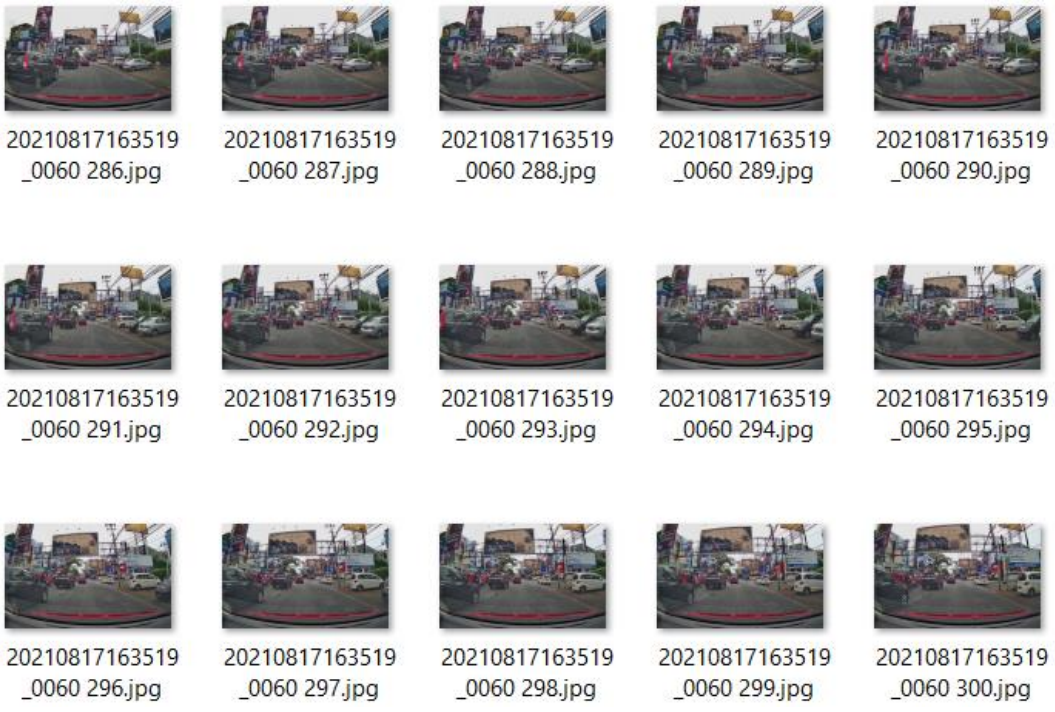
- Kelas dilarang_parkir



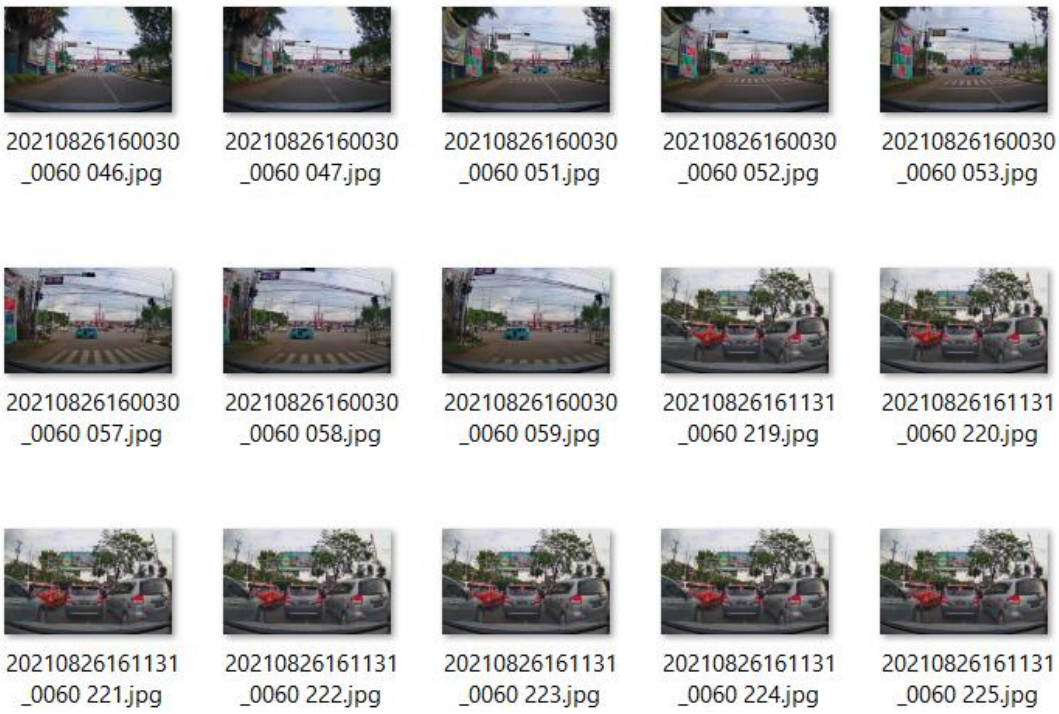
- Kelas dilarang_putar_balik



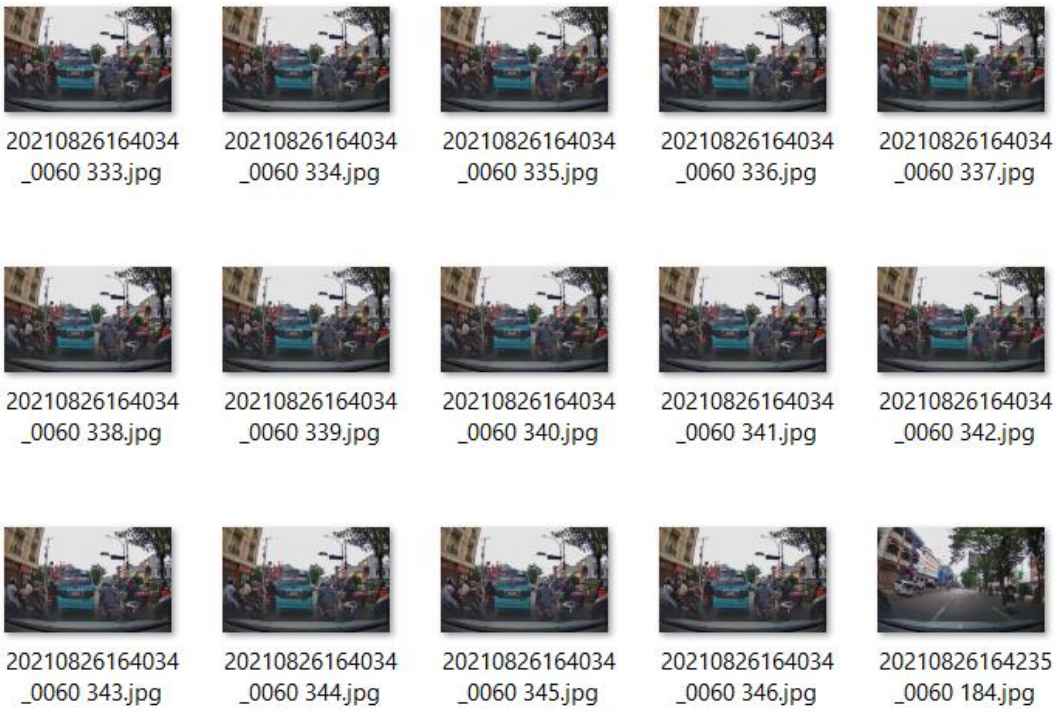
- Kelas dilarang_stop



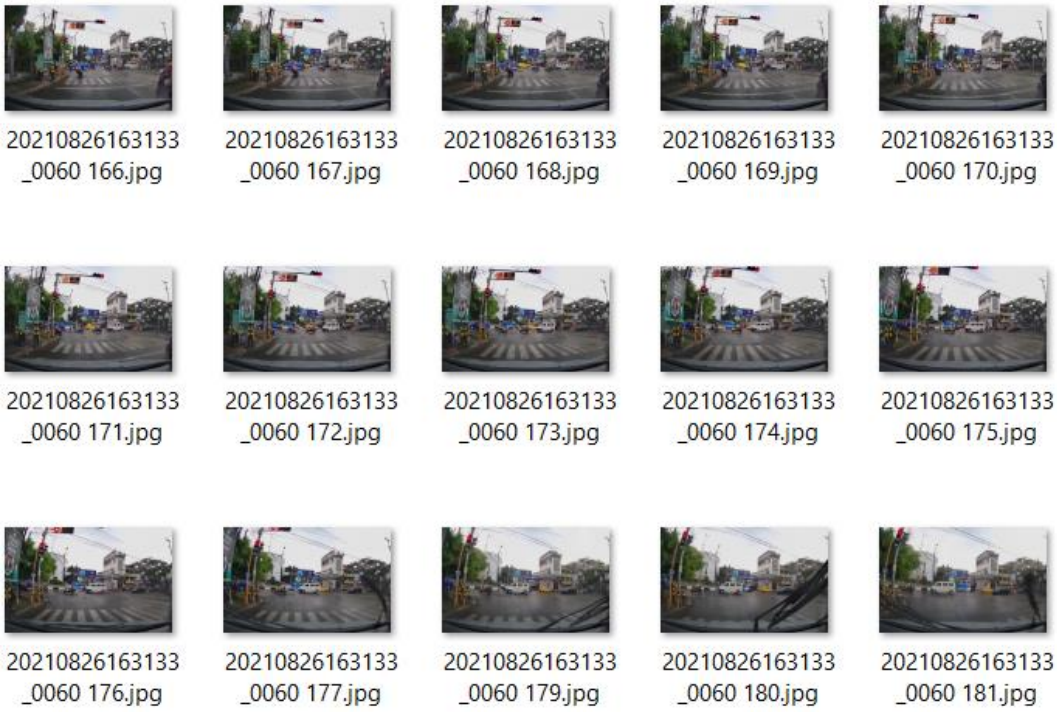
- Kelas lampu_hijau



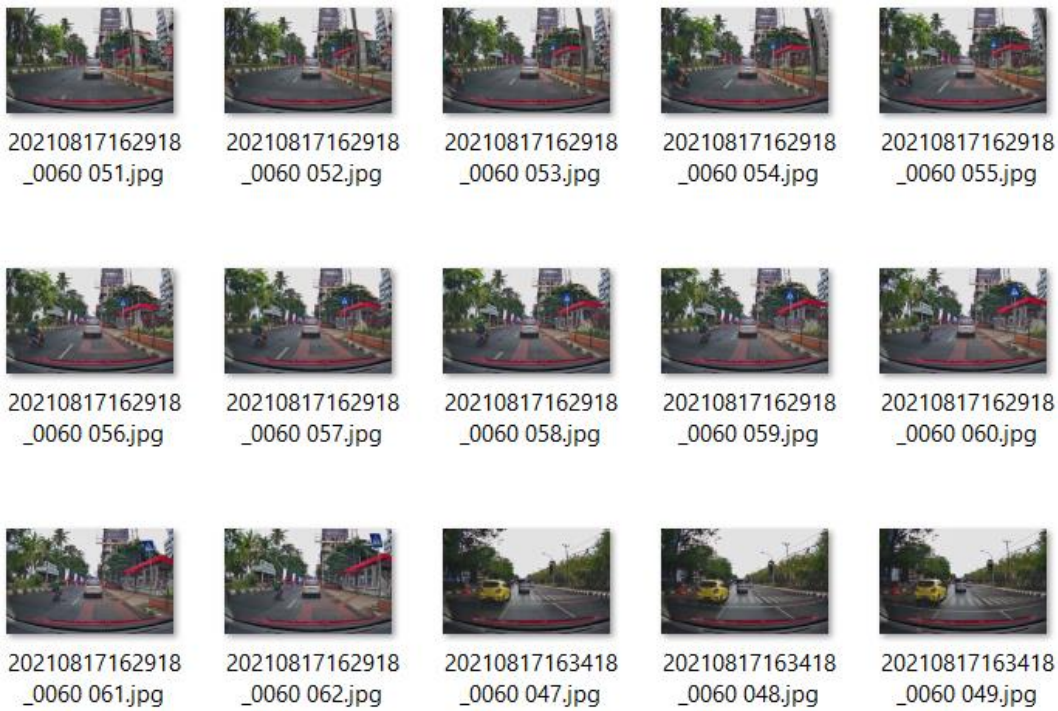
- Kelas lampu_kuning



- Kelas lampu_merah



- Kelas penyeberangan_orang



- Kelas perhatian


20210817162918_0060 031.jpg


20210817162918_0060 032.jpg


20210817162918_0060 033.jpg


20210817162918_0060 034.jpg


20210817162918_0060 035.jpg


20210817162918_0060 036.jpg


20210817162918_0060 037.jpg


20210817162918_0060 038.jpg


20210817162918_0060 039.jpg


20210817162918_0060 040.jpg


20210817162918_0060 041.jpg


20210817163218_0060 155.jpg


20210817163218_0060 156.jpg


20210817163218_0060 157.jpg


20210817163218_0060 158.jpg

- Kelas putar_balik


20210826160531_0060 139.jpg


20210826160531_0060 140.jpg


20210826160531_0060 141.jpg


20210826160531_0060 142.jpg


20210826160531_0060 143.jpg


20210826160531_0060 144.jpg


20210826160531_0060 145.jpg


20210826160531_0060 146.jpg


20210826160531_0060 147.jpg


20210826160531_0060 148.jpg


20210826160531_0060 149.jpg


20210826160531_0060 220.jpg


20210826160531_0060 221.jpg


20210826160531_0060 222.jpg


20210826160531_0060 223.jpg

Lampiran 2: Source Code

- Preparation Data

```
import numbers
from collections.abc import Sequence
from typing import Tuple, List, Optional

import torch
from torch import Tensor

try:
    import accimage
except ImportError:
    accimage = None

from ..utils import _log_api_usage_once
from . import functional as F


__all__ = [
    "Compose",
    "ToTensor",
    "ToPILImage",
    "ColorJitter",
    "RandomGrayscale",
    "RandomInvert",
    "RandomPosterize",
    "RandomSolarize",
    "RandomAdjustSharpness",
    "RandomAutocontrast",
    "RandomEqualize",
]


class Compose:
    def __init__(self, transforms):
        if not torch.jit.is_scripting() and not
torch.jit.is_tracing():
            _log_api_usage_once(self)
        self.transforms = transforms

    def __call__(self, img):
        for t in self.transforms:
            img = t(img)
        return img

    def __repr__(self) -> str:
        format_string = self.__class__.__name__ + "("
        for t in self.transforms:
            format_string += "\n"
            format_string += f"    {t}"
        format_string += "\n)"
        return format_string
```

```
class ToTensor:
    def __init__(self) -> None:
        _log_api_usage_once(self)

    def __call__(self, pic):
        return F.to_tensor(pic)

    def __repr__(self) -> str:
        return f"{self.__class__.__name__}()"


class ToPILImage:
    def __init__(self, mode=None):
        _log_api_usage_once(self)
        self.mode = mode

    def __call__(self, pic):
        return F.to_pil_image(pic, self.mode)

    def __repr__(self) -> str:
        format_string = self.__class__.__name__ + "("
        if self.mode is not None:
            format_string += f"mode={self.mode}"
        format_string += ")"
        return format_string


class ColorJitter(torch.nn.Module):
    def __init__(self, brightness=0, contrast=0, saturation=0,
hue=0):
        super().__init__()
        _log_api_usage_once(self)
        self.brightness = self._check_input(brightness,
"brightness")
        self.contrast = self._check_input(contrast, "contrast")
        self.saturation = self._check_input(saturation,
"saturation")
        self.hue = self._check_input(hue, "hue", center=0, bound=(-
0.5, 0.5), clip_first_on_zero=False)

    @torch.jit.unused
    def _check_input(self, value, name, center=1, bound=(0,
float("inf")), clip_first_on_zero=True):
        if isinstance(value, numbers.Number):
            if value < 0:
                raise ValueError(f"If {name} is a single number, it
must be non negative.")
            value = [center - float(value), center + float(value)]
            if clip_first_on_zero:
                value[0] = max(value[0], 0.0)
        elif isinstance(value, (tuple, list)) and len(value) == 2:
            if not bound[0] <= value[0] <= value[1] <= bound[1]:
                raise ValueError(f"{name} values should be between
{bound}")
        else:
```

```
            raise TypeError(f"{name} should be a single number or a
list/tuple with length 2.")

        if value[0] == value[1] == center:
            value = None
        return value

    @staticmethod
    def get_params(
        brightness: Optional[List[float]],
        contrast: Optional[List[float]],
        saturation: Optional[List[float]],
        hue: Optional[List[float]],
    ) -> Tuple[Tensor, Optional[float], Optional[float],
Optional[float], Optional[float]]:
        fn_idx = torch.randperm(4)

        b = None if brightness is None else
float(torch.empty(1).uniform_(brightness[0], brightness[1]))
        c = None if contrast is None else
float(torch.empty(1).uniform_(contrast[0], contrast[1]))
        s = None if saturation is None else
float(torch.empty(1).uniform_(saturation[0], saturation[1]))
        h = None if hue is None else
float(torch.empty(1).uniform_(hue[0], hue[1]))

        return fn_idx, b, c, s, h

    def forward(self, img):
        fn_idx, brightness_factor, contrast_factor,
saturation_factor, hue_factor = self.get_params(
            self.brightness, self.contrast, self.saturation,
self.hue
        )

        for fn_id in fn_idx:
            if fn_id == 0 and brightness_factor is not None:
                img = F.adjust_brightness(img, brightness_factor)
            elif fn_id == 1 and contrast_factor is not None:
                img = F.adjust_contrast(img, contrast_factor)
            elif fn_id == 2 and saturation_factor is not None:
                img = F.adjust_saturation(img, saturation_factor)
            elif fn_id == 3 and hue_factor is not None:
                img = F.adjust_hue(img, hue_factor)

        return img

    def __repr__(self) -> str:
        s = (
            f"{self.__class__.__name__}("
            f"brightness={self.brightness}"
            f", contrast={self.contrast}"
            f", saturation={self.saturation}"
            f", hue={self.hue})"
        )
        return s
```

```
class RandomGrayscale(torch.nn.Module):
    def __init__(self, p=0.1):
        super().__init__()
        _log_api_usage_once(self)
        self.p = p

    def forward(self, img):
        num_output_channels = F.get_image_num_channels(img)
        if torch.rand(1) < self.p:
            return F.rgb_to_grayscale(img,
num_output_channels=num_output_channels)
        return img

    def __repr__(self) -> str:
        return f"{self.__class__.__name__}(p={self.p})"


def _setup_size(size, error_msg):
    if isinstance(size, numbers.Number):
        return int(size), int(size)

    if isinstance(size, Sequence) and len(size) == 1:
        return size[0], size[0]

    if len(size) != 2:
        raise ValueError(error_msg)

    return size


def _check_sequence_input(x, name, req_sizes):
    msg = req_sizes[0] if len(req_sizes) < 2 else " or
".join([str(s) for s in req_sizes])
    if not isinstance(x, Sequence):
        raise TypeError(f"{name} should be a sequence of length
{msg}.")
    if len(x) not in req_sizes:
        raise ValueError(f"{name} should be sequence of length
{msg}.")


def _setup_angle(x, name, req_sizes=(2,)):
    if isinstance(x, numbers.Number):
        if x < 0:
            raise ValueError(f"If {name} is a single number, it
must be positive.")
        x = [-x, x]
    else:
        _check_sequence_input(x, name, req_sizes)

    return [float(d) for d in x]


class RandomInvert(torch.nn.Module):
```

```
    def __init__(self, p=0.5):
        super().__init__()
        _log_api_usage_once(self)
        self.p = p

    def forward(self, img):
        if torch.rand(1).item() < self.p:
            return F.invert(img)
        return img

    def __repr__(self) -> str:
        return f"{self.__class__.__name__}(p={self.p})"


class RandomPosterize(torch.nn.Module):
    def __init__(self, bits, p=0.5):
        super().__init__()
        _log_api_usage_once(self)
        self.bits = bits
        self.p = p

    def forward(self, img):
        if torch.rand(1).item() < self.p:
            return F.posterize(img, self.bits)
        return img

    def __repr__(self) -> str:
        return
f"{self.__class__.__name__}(bits={self.bits},p={self.p})"


class RandomSolarize(torch.nn.Module):
    def __init__(self, threshold, p=0.5):
        super().__init__()
        _log_api_usage_once(self)
        self.threshold = threshold
        self.p = p

    def forward(self, img):
        if torch.rand(1).item() < self.p:
            return F.solarize(img, self.threshold)
        return img

    def __repr__(self) -> str:
        return
f"{self.__class__.__name__}(threshold={self.threshold},p={self.p})
"


class RandomAdjustSharpness(torch.nn.Module):
    def __init__(self, sharpness_factor, p=0.5):
        super().__init__()
        _log_api_usage_once(self)
        self.sharpness_factor = sharpness_factor
        self.p = p
```

```
    def forward(self, img):
        if torch.rand(1).item() < self.p:
            return F.adjust_sharpness(img, self.sharpness_factor)
        return img

    def __repr__(self) -> str:
        return
f"{self.__class__.__name__}(sharpness_factor={self.sharpness_facto
r},p={self.p})"


class RandomAutocontrast(torch.nn.Module):
    def __init__(self, p=0.5):
        super().__init__()
        _log_api_usage_once(self)
        self.p = p

    def forward(self, img):
        if torch.rand(1).item() < self.p:
            return F.autocontrast(img)
        return img

    def __repr__(self) -> str:
        return f"{self.__class__.__name__}(p={self.p})"


class RandomEqualize(torch.nn.Module):
    def __init__(self, p=0.5):
        super().__init__()
        _log_api_usage_once(self)
        self.p = p

    def forward(self, img):
        if torch.rand(1).item() < self.p:
            return F.equalize(img)
        return img

    def __repr__(self) -> str:
        return f"{self.__class__.__name__}(p={self.p})"
```

- Training Model Faster R-CNN

```
#####CORE#####
import os
import pandas as pd
import random
import torch
import torchvision

from detecto.config import config
from detecto.utils import default_transforms,
filter_top_predictions, xml_to_csv, _is_iterable, read_image
from torchvision import transforms
```

```
from torchvision.models.detection.faster_rcnn import
FastRCNNPredictor
from tqdm import tqdm


class DataLoader(torch.utils.data.DataLoader):

    def __init__(self, dataset, **kwargs):
        super().__init__(dataset,
collate_fn=DataLoader.collate_data, **kwargs)

    @staticmethod
    def collate_data(batch):
        images, targets = zip(*batch)
        return list(images), list(targets)


class Dataset(torch.utils.data.Dataset):

    def __init__(self, label_data, image_folder=None,
transform=None):

        if os.path.isfile(label_data):
            self._csv = pd.read_csv(label_data)
        else:
            self._csv = xml_to_csv(label_data)

        if image_folder is None:
            self._root_dir = label_data
        else:
            self._root_dir = image_folder

        if transform is None:
            self.transform = default_transforms()
        else:
            self.transform = transform

    def __len__(self):
        return len(self._csv['image_id'].unique().tolist())

    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()

        object_entries = self._csv.loc[self._csv['image_id'] ==
idx]

        img_name = os.path.join(self._root_dir,
object_entries.iloc[0, 0])
        image = read_image(img_name)

        boxes = []
        labels = []
        for object_idx, row in object_entries.iterrows():
            box = self._csv.iloc[object_idx, 4:8]
            boxes.append(box)
```

```
            label = self._csv.iloc[object_idx, 3]
            labels.append(label)

        boxes = torch.tensor(boxes).view(-1, 4)

        targets = {'boxes': boxes, 'labels': labels}

        if self.transform:
            width = object_entries.iloc[0, 1]
            height = object_entries.iloc[0, 2]

            updated_transforms = []
            scale_factor = 1.0
            random_flip = 0.0
            for t in self.transform.transforms:
                updated_transforms.append(t)

                if isinstance(t, transforms.Resize):
                    original_size = min(height, width)
                    scale_factor = original_size / t.size

                elif isinstance(t,
transforms.RandomHorizontalFlip):
                    random_flip = t.p

            for t in updated_transforms:
                if isinstance(t, transforms.RandomHorizontalFlip):
                    if random.random() < random_flip:
                        image =
transforms.RandomHorizontalFlip(1)(image)
                        for idx, box in
enumerate(targets['boxes']):
                            box[0] = width - box[0]
                            box[2] = width - box[2]
                            box[[0,2]] = box[[2,0]]
                            targets['boxes'][idx] = box
                else:
                    image = t(image)

            if scale_factor != 1.0:
                for idx, box in enumerate(targets['boxes']):
                    box = (box / scale_factor).long()
                    targets['boxes'][idx] = box

        return image, targets


class Model:

    DEFAULT = 'fasterrcnn_resnet50_fpn'

    def __init__(self, classes=None, device=None, pretrained=True,
model_name=DEFAULT):

        self._device = device if device else
config['default_device']
```

```
        if model_name == self.DEFAULT:
            self._model =
torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=pr
etrained)
        else:
            raise ValueError(f'Invalid value {model_name} for
model_name. ' +
                             f'Please choose {self.DEFAULT}')

        if classes:
            in_features =
self._model.roi_heads.box_predictor.cls_score.in_features
            self._model.roi_heads.box_predictor =
FastRCNNPredictor(in_features, len(classes) + 1)
            self._disable_normalize = False
        else:
            classes = config['default_classes']
            self._disable_normalize = True

        self._model.to(self._device)

        self._classes = ['__background__'] + classes
        self._int_mapping = {label: index for index, label in
enumerate(self._classes)}

    def _get_raw_predictions(self, images):
        self._model.eval()

        with torch.no_grad():
            if not _is_iterable(images):
                images = [images]

            if not isinstance(images[0], torch.Tensor):
                if self._disable_normalize:
                    defaults =
transforms.Compose([transforms.ToTensor()])
                else:
                    defaults = default_transforms()
                images = [defaults(img) for img in images]

            images = [img.to(self._device) for img in images]

            preds = self._model(images)
            preds = [{k: v.to(torch.device('cpu')) for k, v in
p.items()} for p in preds]
            return preds

    def predict(self, images):
        is_single_image = not _is_iterable(images)
        images = [images] if is_single_image else images
        preds = self._get_raw_predictions(images)

        results = []
        for pred in preds:
```

```
            result = ([self._classes[val] for val in
pred['labels']], pred['boxes'], pred['scores'])
            results.append(result)

        return results[0] if is_single_image else results


    def predict_top(self, images):
        predictions = self.predict(images)

        if not isinstance(predictions, list):
            return filter_top_predictions(*predictions)

        results = []
        for pred in predictions:
            results.append(filter_top_predictions(*pred))

        return results

    def fit(self, dataset, val_dataset=None, epochs=10,
learning_rate=0.005, momentum=0.9,
            weight_decay=0.0005, gamma=0.1, lr_step_size=3,
verbose=True):

        if verbose and self._device == torch.device('cpu'):
            print('It looks like you\'re training your model on a
CPU. '
                  'Consider switching to a GPU; otherwise, this
method '
                  'can take hours upon hours or even days to
finish. '
                  'For more information, see
https://detecto.readthedocs.io/'
                  'en/latest/usage/quickstart.html#technical-
requirements')

        if epochs > 0:
            self._disable_normalize = False

        if not isinstance(dataset, DataLoader):
            dataset = DataLoader(dataset, shuffle=True)

        if val_dataset is not None and not isinstance(val_dataset,
DataLoader):
            val_dataset = DataLoader(val_dataset)

        losses = []
        parameters = [p for p in self._model.parameters() if
p.requires_grad]
        optimizer = torch.optim.SGD(parameters, lr=learning_rate,
momentum=momentum, weight_decay=weight_decay)
        lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
step_size=lr_step_size, gamma=gamma)

        for epoch in range(epochs):
            if verbose:
```

```
                print('Epoch {} of {}'.format(epoch + 1, epochs))

            self._model.train()

            if verbose:
                print('Begin iterating over training dataset')

            iterable = tqdm(dataset, position=0, leave=True) if
verbose else dataset
            for images, targets in iterable:
                self._convert_to_int_labels(targets)
                images, targets = self._to_device(images, targets)

                loss_dict = self._model(images, targets)
                total_loss = sum(loss for loss in
loss_dict.values())

                optimizer.zero_grad()
                total_loss.backward()
                optimizer.step()

            if val_dataset is not None:
                avg_loss = 0
                with torch.no_grad():
                    if verbose:
                        print('Begin iterating over validation
dataset')

                    iterable = tqdm(val_dataset, position=0,
leave=True) if verbose else val_dataset
                    for images, targets in iterable:
                        self._convert_to_int_labels(targets)
                        images, targets = self._to_device(images,
targets)
                        loss_dict = self._model(images, targets)
                        total_loss = sum(loss for loss in
loss_dict.values())
                        avg_loss += total_loss.item()

                avg_loss /= len(val_dataset.dataset)
                losses.append(avg_loss)

                if verbose:
                    print('Loss: {}'.format(avg_loss))

            lr_scheduler.step()

        if len(losses) > 0:
            return losses

    def get_internal_model(self):
        return self._model

    def save(self, file):
        torch.save(self._model.state_dict(), file)
```

```
    @staticmethod
    def load(file, classes):
        model = Model(classes)
        model._model.load_state_dict(torch.load(file,
map_location=model._device))
        return model

    def _convert_to_int_labels(self, targets):
        for idx, target in enumerate(targets):
            labels_array = target['labels']
            labels_int_array = [self._int_mapping[class_name] for
class_name in labels_array]
            target['labels'] = torch.tensor(labels_int_array)

    def _to_device(self, images, targets):
        images = [image.to(self._device) for image in images]
        targets = [{k: v.to(self._device) for k, v in t.items()}
for t in targets]
        return images, targets

#####FASTER R-CNN#####
import torch.nn.functional as F
from torch import nn
from torchvision.ops import MultiScaleRoIAlign

from ...ops import misc as misc_nn_ops
from ..resnet import resnet50
from ._utils import overwrite_eps
from .anchor_utils import AnchorGenerator
from .backbone_utils import _resnet_fpn_extractor,
_validate_trainable_layers, _mobilenet_extractor
from .generalized_rcnn import GeneralizedRCNN
from .roi_heads import RoIHeads
from .rpn import RPNHead, RegionProposalNetwork
from .transform import GeneralizedRCNNTransform


__all__ = [
    "FasterRCNN",
    "fasterrcnn_resnet50_fpn",
]


class FasterRCNN(GeneralizedRCNN):

    def __init__(
        self,
        backbone,
        num_classes=None,
        # transform parameters
        min_size=800,
        max_size=1333,
        image_mean=None,
        image_std=None,
        # RPN parameters
        rpn_anchor_generator=None,
```

```
        rpn_head=None,
        rpn_pre_nms_top_n_train=2000,
        rpn_pre_nms_top_n_test=1000,
        rpn_post_nms_top_n_train=2000,
        rpn_post_nms_top_n_test=1000,
        rpn_nms_thresh=0.7,
        rpn_fg_iou_thresh=0.7,
        rpn_bg_iou_thresh=0.3,
        rpn_batch_size_per_image=256,
        rpn_positive_fraction=0.5,
        rpn_score_thresh=0.0,
        # Box parameters
        box_roi_pool=None,
        box_head=None,
        box_predictor=None,
        box_score_thresh=0.05,
        box_nms_thresh=0.5,
        box_detections_per_img=100,
        box_fg_iou_thresh=0.5,
        box_bg_iou_thresh=0.5,
        box_batch_size_per_image=512,
        box_positive_fraction=0.25,
        bbox_reg_weights=None,
    ):

        if not hasattr(backbone, "out_channels"):
            raise ValueError(
                "backbone should contain an attribute out_channels
"
                "specifying the number of output channels (assumed
to be the "
                "same for all the levels)"
            )

        assert isinstance(rpn_anchor_generator, (AnchorGenerator,
type(None)))
        assert isinstance(box_roi_pool, (MultiScaleRoIAlign,
type(None)))

        if num_classes is not None:
            if box_predictor is not None:
                raise ValueError("num_classes should be None when
box_predictor is specified")
        else:
            if box_predictor is None:
                raise ValueError("num_classes should not be None
when box_predictor is not specified")

        out_channels = backbone.out_channels

        if rpn_anchor_generator is None:
            anchor_sizes = ((32,), (64,), (128,), (256,), (512,))
            aspect_ratios = ((0.5, 1.0, 2.0),) * len(anchor_sizes)
            rpn_anchor_generator = AnchorGenerator(anchor_sizes,
aspect_ratios)
        if rpn_head is None:
```

```
            rpn_head = RPNHead(out_channels,
rpn_anchor_generator.num_anchors_per_location()[0])

        rpn_pre_nms_top_n = dict(training=rpn_pre_nms_top_n_train,
testing=rpn_pre_nms_top_n_test)
        rpn_post_nms_top_n =
dict(training=rpn_post_nms_top_n_train,
testing=rpn_post_nms_top_n_test)

        rpn = RegionProposalNetwork(
            rpn_anchor_generator,
            rpn_head,
            rpn_fg_iou_thresh,
            rpn_bg_iou_thresh,
            rpn_batch_size_per_image,
            rpn_positive_fraction,
            rpn_pre_nms_top_n,
            rpn_post_nms_top_n,
            rpn_nms_thresh,
            score_thresh=rpn_score_thresh,
        )

        if box_roi_pool is None:
            box_roi_pool = MultiScaleRoIAlign(featmap_names=["0",
"1", "2", "3"], output_size=7, sampling_ratio=2)

        if box_head is None:
            resolution = box_roi_pool.output_size[0]
            representation_size = 1024
            box_head = TwoMLPHead(out_channels * resolution ** 2,
representation_size)

        if box_predictor is None:
            representation_size = 1024
            box_predictor = FastRCNNPredictor(representation_size,
num_classes)

        roi_heads = RoIHeads(
            # Box
            box_roi_pool,
            box_head,
            box_predictor,
            box_fg_iou_thresh,
            box_bg_iou_thresh,
            box_batch_size_per_image,
            box_positive_fraction,
            bbox_reg_weights,
            box_score_thresh,
            box_nms_thresh,
            box_detections_per_img,
        )

        if image_mean is None:
            image_mean = [0.485, 0.456, 0.406]
        if image_std is None:
            image_std = [0.229, 0.224, 0.225]
```

```
        transform = GeneralizedRCNNTransform(min_size, max_size,
image_mean, image_std)

        super().__init__(backbone, rpn, roi_heads, transform)


class TwoMLPHead(nn.Module):

    def __init__(self, in_channels, representation_size):
        super().__init__()

        self.fc6 = nn.Linear(in_channels, representation_size)
        self.fc7 = nn.Linear(representation_size,
representation_size)

    def forward(self, x):
        x = x.flatten(start_dim=1)

        x = F.relu(self.fc6(x))
        x = F.relu(self.fc7(x))

        return x


class FastRCNNPredictor(nn.Module):

    def __init__(self, in_channels, num_classes):
        super().__init__()
        self.cls_score = nn.Linear(in_channels, num_classes)
        self.bbox_pred = nn.Linear(in_channels, num_classes * 4)

    def forward(self, x):
        if x.dim() == 4:
            assert list(x.shape[2:]) == [1, 1]
        x = x.flatten(start_dim=1)
        scores = self.cls_score(x)
        bbox_deltas = self.bbox_pred(x)

        return scores, bbox_deltas


model_urls = {
    "fasterrcnn_resnet50_fpn_coco":
"https://download.pytorch.org/models/fasterrcnn_resnet50_fpn_coco-
258fb6c6.pth",
}


def fasterrcnn_resnet50_fpn(
    pretrained=False, progress=True, num_classes=91,
pretrained_backbone=True, trainable_backbone_layers=None, **kwargs
):
    trainable_backbone_layers = _validate_trainable_layers(
        pretrained or pretrained_backbone,
trainable_backbone_layers, 5, 3
    )
```

```
    if pretrained:
        pretrained_backbone = False

    backbone = resnet50(pretrained=pretrained_backbone,
progress=progress, norm_layer=misc_nn_ops.FrozenBatchNorm2d)
    backbone = _resnet_fpn_extractor(backbone,
trainable_backbone_layers)
    model = FasterRCNN(backbone, num_classes, **kwargs)
    if pretrained:
        state_dict =
load_state_dict_from_url(model_urls["fasterrcnn_resnet50_fpn_coco"
], progress=progress)
        model.load_state_dict(state_dict)
        overwrite_eps(model, 0.0)
    return model

#####RESNET#####

from typing import Type, Any, Callable, Union, List, Optional

import torch
import torch.nn as nn
from torch import Tensor

from .._internally_replaced_utils import load_state_dict_from_url
from ..utils import _log_api_usage_once


__all__ = [
    "ResNet",
    "resnet50",
]


model_urls = {
    "resnet50": "https://download.pytorch.org/models/resnet50-
0676ba61.pth",
}


def conv3x3(in_planes: int, out_planes: int, stride: int = 1,
groups: int = 1, dilation: int = 1) -> nn.Conv2d:
    """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: int, out_planes: int, stride: int = 1) ->
nn.Conv2d:
    """1x1 convolution"""
    return nn.Conv2d(in_planes, out_planes, kernel_size=1,
stride=stride, bias=False)


class BasicBlock(nn.Module):
    expansion: int = 1

    def __init__(
        self,
        inplanes: int,
        planes: int,
        stride: int = 1,
        downsample: Optional[nn.Module] = None,
        groups: int = 1,
        base_width: int = 64,
        dilation: int = 1,
        norm_layer: Optional[Callable[..., nn.Module]] = None,
    ) -> None:
        super().__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")
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = norm_layer(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = norm_layer(planes)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x: Tensor) -> Tensor:
        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: int = 4

   def __init__(
       self,
       inplanes: int,
       planes: int,
       stride: int = 1,
       downsample: Optional[nn.Module] = None,
       groups: int = 1,
       base_width: int = 64,
       dilation: int = 1,
       norm_layer: Optional[Callable[..., nn.Module]] = None,
   ) -> None:
       super().__init__()
       if norm_layer is None:
           norm_layer = nn.BatchNorm2d
       width = int(planes * (base_width / 64.0)) * groups
       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: Tensor) -> Tensor:
       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 ResNet(nn.Module):
   def __init__(
       self,
```

```
        block: Type[Union[BasicBlock, Bottleneck]],
        layers: List[int],
        num_classes: int = 1000,
        zero_init_residual: bool = False,
        groups: int = 1,
        width_per_group: int = 64,
        replace_stride_with_dilation: Optional[List[bool]] = None,
        norm_layer: Optional[Callable[..., nn.Module]] = None,
    ) -> None:
        super().__init__()
        _log_api_usage_once(self)
        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 "
                f"or a 3-element tuple, got
{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.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1],
stride=2, dilate=replace_stride_with_dilation[0])
        self.layer3 = self._make_layer(block, 256, layers[2],
stride=2, dilate=replace_stride_with_dilation[1])
        self.layer4 = self._make_layer(block, 512, layers[3],
stride=2, dilate=replace_stride_with_dilation[2])
        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)

        if zero_init_residual:
            for m in self.modules():
```

```
                if isinstance(m, Bottleneck):
                    nn.init.constant_(m.bn3.weight, 0)  # type:
ignore[arg-type]
                elif isinstance(m, BasicBlock):
                    nn.init.constant_(m.bn2.weight, 0)  # type:
ignore[arg-type]

    def _make_layer(
        self,
        block: Type[Union[BasicBlock, Bottleneck]],
        planes: int,
        blocks: int,
        stride: int = 1,
        dilate: bool = False,
    ) -> nn.Sequential:
        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_impl(self, x: Tensor) -> Tensor:
        # See note [TorchScript super()]
        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)
        x = self.layer4(x)

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

        return x

    def forward(self, x: Tensor) -> Tensor:
        return self._forward_impl(x)


def _resnet(
    arch: str,
    block: Type[Union[BasicBlock, Bottleneck]],
    layers: List[int],
    pretrained: bool,
    progress: bool,
    **kwargs: Any,
) -> ResNet:
    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)
    return model


def resnet50(pretrained: bool = False, progress: bool = True,
**kwargs: Any) -> ResNet:
    return _resnet("resnet50", Bottleneck, [3, 4, 6, 3],
pretrained, progress, **kwargs)

#####RPN#####
from typing import List, Optional, Dict, Tuple, cast

import torch
import torchvision
from torch import nn, Tensor
from torch.nn import functional as F
from torchvision.ops import boxes as box_ops

from . import _utils as det_utils

from .anchor_utils import AnchorGenerator
from .image_list import ImageList


@torch.jit.unused
def _onnx_get_num_anchors_and_pre_nms_top_n(ob: Tensor,
orig_pre_nms_top_n: int) -> Tuple[int, int]:
```

```
    from torch.onnx import operators

    num_anchors = operators.shape_as_tensor(ob)[1].unsqueeze(0)
    pre_nms_top_n =
torch.min(torch.cat((torch.tensor([orig_pre_nms_top_n],
dtype=num_anchors.dtype), num_anchors), 0))

    return cast(int, num_anchors), cast(int, pre_nms_top_n)


class RPNHead(nn.Module):

    def __init__(self, in_channels: int, num_anchors: int) -> None:
        super().__init__()
        self.conv = nn.Conv2d(in_channels, in_channels,
kernel_size=3, stride=1, padding=1)
        self.cls_logits = nn.Conv2d(in_channels, num_anchors,
kernel_size=1, stride=1)
        self.bbox_pred = nn.Conv2d(in_channels, num_anchors * 4,
kernel_size=1, stride=1)

        for layer in self.children():
            torch.nn.init.normal_(layer.weight, std=0.01)  # type:
ignore[arg-type]
            torch.nn.init.constant_(layer.bias, 0)  # type:
ignore[arg-type]

    def forward(self, x: List[Tensor]) -> Tuple[List[Tensor],
List[Tensor]]:
        logits = []
        bbox_reg = []
        for feature in x:
            t = F.relu(self.conv(feature))
            logits.append(self.cls_logits(t))
            bbox_reg.append(self.bbox_pred(t))
        return logits, bbox_reg


def permute_and_flatten(layer: Tensor, N: int, A: int, C: int, H:
int, W: int) -> Tensor:
    layer = layer.view(N, -1, C, H, W)
    layer = layer.permute(0, 3, 4, 1, 2)
    layer = layer.reshape(N, -1, C)
    return layer


def concat_box_prediction_layers(box_cls: List[Tensor],
box_regression: List[Tensor]) -> Tuple[Tensor, Tensor]:
    box_cls_flattened = []
    box_regression_flattened = []
    for box_cls_per_level, box_regression_per_level in zip(box_cls,
box_regression):
        N, AxC, H, W = box_cls_per_level.shape
        Ax4 = box_regression_per_level.shape[1]
        A = Ax4 // 4
        C = AxC // A
```

```
        box_cls_per_level = permute_and_flatten(box_cls_per_level,
N, A, C, H, W)
        box_cls_flattened.append(box_cls_per_level)

        box_regression_per_level =
permute_and_flatten(box_regression_per_level, N, A, 4, H, W)
        box_regression_flattened.append(box_regression_per_level)
    box_cls = torch.cat(box_cls_flattened, dim=1).flatten(0, -2)
    box_regression = torch.cat(box_regression_flattened,
dim=1).reshape(-1, 4)
    return box_cls, box_regression


class RegionProposalNetwork(torch.nn.Module):
    __annotations__ = {
        "box_coder": det_utils.BoxCoder,
        "proposal_matcher": det_utils.Matcher,
        "fg_bg_sampler": det_utils.BalancedPositiveNegativeSampler,
    }

    def __init__(
        self,
        anchor_generator: AnchorGenerator,
        head: nn.Module,
        # Faster-RCNN Training
        fg_iou_thresh: float,
        bg_iou_thresh: float,
        batch_size_per_image: int,
        positive_fraction: float,
        # Faster-RCNN Inference
        pre_nms_top_n: Dict[str, int],
        post_nms_top_n: Dict[str, int],
        nms_thresh: float,
        score_thresh: float = 0.0,
    ) -> None:
        super().__init__()
        self.anchor_generator = anchor_generator
        self.head = head
        self.box_coder = det_utils.BoxCoder(weights=(1.0, 1.0, 1.0,
1.0))

        # used during training
        self.box_similarity = box_ops.box_iou

        self.proposal_matcher = det_utils.Matcher(
            fg_iou_thresh,
            bg_iou_thresh,
            allow_low_quality_matches=True,
        )

        self.fg_bg_sampler =
det_utils.BalancedPositiveNegativeSampler(batch_size_per_image,
positive_fraction)
        # used during testing
        self._pre_nms_top_n = pre_nms_top_n
        self._post_nms_top_n = post_nms_top_n
```

```
        self.nms_thresh = nms_thresh
        self.score_thresh = score_thresh
        self.min_size = 1e-3

    def pre_nms_top_n(self) -> int:
        if self.training:
            return self._pre_nms_top_n["training"]
        return self._pre_nms_top_n["testing"]

    def post_nms_top_n(self) -> int:
        if self.training:
            return self._post_nms_top_n["training"]
        return self._post_nms_top_n["testing"]

    def assign_targets_to_anchors(
        self, anchors: List[Tensor], targets: List[Dict[str,
Tensor]]
    ) -> Tuple[List[Tensor], List[Tensor]]:

        labels = []
        matched_gt_boxes = []
        for anchors_per_image, targets_per_image in zip(anchors,
targets):
            gt_boxes = targets_per_image["boxes"]

            if gt_boxes.numel() == 0:
                device = anchors_per_image.device
                matched_gt_boxes_per_image =
torch.zeros(anchors_per_image.shape, dtype=torch.float32,
device=device)
                labels_per_image =
torch.zeros((anchors_per_image.shape[0],), dtype=torch.float32,
device=device)
            else:
                match_quality_matrix =
self.box_similarity(gt_boxes, anchors_per_image)
                matched_idxs =
self.proposal_matcher(match_quality_matrix)
                matched_gt_boxes_per_image =
gt_boxes[matched_idxs.clamp(min=0)]

                labels_per_image = matched_idxs >= 0
                labels_per_image =
labels_per_image.to(dtype=torch.float32)

                bg_indices = matched_idxs ==
self.proposal_matcher.BELOW_LOW_THRESHOLD
                labels_per_image[bg_indices] = 0.0

                inds_to_discard = matched_idxs ==
self.proposal_matcher.BETWEEN_THRESHOLDS
                labels_per_image[inds_to_discard] = -1.0

            labels.append(labels_per_image)
            matched_gt_boxes.append(matched_gt_boxes_per_image)
        return labels, matched_gt_boxes
```

```
   def _get_top_n_idx(self, objectness: Tensor,
num_anchors_per_level: List[int]) -> Tensor:
        r = []
        offset = 0
        for ob in objectness.split(num_anchors_per_level, 1):
            if torchvision._is_tracing():
                num_anchors, pre_nms_top_n =
_onnx_get_num_anchors_and_pre_nms_top_n(ob, self.pre_nms_top_n())
            else:
                num_anchors = ob.shape[1]
                pre_nms_top_n = min(self.pre_nms_top_n(),
num_anchors)
            _, top_n_idx = ob.topk(pre_nms_top_n, dim=1)
            r.append(top_n_idx + offset)
            offset += num_anchors
        return torch.cat(r, dim=1)

   def filter_proposals(
        self,
        proposals: Tensor,
        objectness: Tensor,
        image_shapes: List[Tuple[int, int]],
        num_anchors_per_level: List[int],
   ) -> Tuple[List[Tensor], List[Tensor]]:

        num_images = proposals.shape[0]
        device = proposals.device
        objectness = objectness.detach()
        objectness = objectness.reshape(num_images, -1)

        levels = [
            torch.full((n,), idx, dtype=torch.int64, device=device)
for idx, n in enumerate(num_anchors_per_level)
        ]
        levels = torch.cat(levels, 0)
        levels = levels.reshape(1, -1).expand_as(objectness)

        top_n_idx = self._get_top_n_idx(objectness,
num_anchors_per_level)

        image_range = torch.arange(num_images, device=device)
        batch_idx = image_range[:, None]

        objectness = objectness[batch_idx, top_n_idx]
        levels = levels[batch_idx, top_n_idx]
        proposals = proposals[batch_idx, top_n_idx]

        objectness_prob = torch.sigmoid(objectness)

        final_boxes = []
        final_scores = []
        for boxes, scores, lvl, img_shape in zip(proposals,
objectness_prob, levels, image_shapes):
            boxes = box_ops.clip_boxes_to_image(boxes, img_shape)
```

```
            # remove small boxes
            keep = box_ops.remove_small_boxes(boxes, self.min_size)
            boxes, scores, lvl = boxes[keep], scores[keep],
lvl[keep]

            # remove low scoring boxes
            # use >= for Backwards compatibility
            keep = torch.where(scores >= self.score_thresh)[0]
            boxes, scores, lvl = boxes[keep], scores[keep],
lvl[keep]

            # non-maximum suppression, independently done per level
            keep = box_ops.batched_nms(boxes, scores, lvl,
self.nms_thresh)

            # keep only topk scoring predictions
            keep = keep[: self.post_nms_top_n()]
            boxes, scores = boxes[keep], scores[keep]

            final_boxes.append(boxes)
            final_scores.append(scores)
        return final_boxes, final_scores

    def compute_loss(
        self, objectness: Tensor, pred_bbox_deltas: Tensor, labels:
List[Tensor], regression_targets: List[Tensor]
    ) -> Tuple[Tensor, Tensor]:

        sampled_pos_inds, sampled_neg_inds =
self.fg_bg_sampler(labels)
        sampled_pos_inds = torch.where(torch.cat(sampled_pos_inds,
dim=0))[0]
        sampled_neg_inds = torch.where(torch.cat(sampled_neg_inds,
dim=0))[0]

        sampled_inds = torch.cat([sampled_pos_inds,
sampled_neg_inds], dim=0)

        objectness = objectness.flatten()

        labels = torch.cat(labels, dim=0)
        regression_targets = torch.cat(regression_targets, dim=0)

        box_loss = (
            F.smooth_l1_loss(
                pred_bbox_deltas[sampled_pos_inds],
                regression_targets[sampled_pos_inds],
                beta=1 / 9,
                reduction="sum",
            )
            / (sampled_inds.numel())
        )

        objectness_loss =
F.binary_cross_entropy_with_logits(objectness[sampled_inds],
labels[sampled_inds])
```

```
        return objectness_loss, box_loss

    def forward(
        self,
        images: ImageList,
        features: Dict[str, Tensor],
        targets: Optional[List[Dict[str, Tensor]]] = None,
    ) -> Tuple[List[Tensor], Dict[str, Tensor]]:

        # RPN uses all feature maps that are available
        features = list(features.values())
        objectness, pred_bbox_deltas = self.head(features)
        anchors = self.anchor_generator(images, features)

        num_images = len(anchors)
        num_anchors_per_level_shape_tensors = [o[0].shape for o in
objectness]
        num_anchors_per_level = [s[0] * s[1] * s[2] for s in
num_anchors_per_level_shape_tensors]
        objectness, pred_bbox_deltas =
concat_box_prediction_layers(objectness, pred_bbox_deltas)
        proposals =
self.box_coder.decode(pred_bbox_deltas.detach(), anchors)
        proposals = proposals.view(num_images, -1, 4)
        boxes, scores = self.filter_proposals(proposals,
objectness, images.image_sizes, num_anchors_per_level)

        losses = {}
        if self.training:
            assert targets is not None
            labels, matched_gt_boxes =
self.assign_targets_to_anchors(anchors, targets)
            regression_targets =
self.box_coder.encode(matched_gt_boxes, anchors)
            loss_objectness, loss_rpn_box_reg = self.compute_loss(
                objectness, pred_bbox_deltas, labels,
regression_targets
            )
            losses = {
                "loss_objectness": loss_objectness,
                "loss_rpn_box_reg": loss_rpn_box_reg,
            }
        return boxes, losses
```

- Realtime

```
import cv2, time
from threading import Thread
from playsound import playsound
# from goprocam import GoProCamera, constants

class ThreadedCamera(object):
```

```
    def __init__(self, source = 0):

        self.capture = cv2.VideoCapture(source)

        self.thread = Thread(target = self.update, args = ())
        self.thread.daemon = True
        self.thread.start()

        self.status = False
        self.frame  = None

    def update(self):
        while True:
            if self.capture.isOpened():
                (self.status, self.frame) = self.capture.read()

    def grab_frame(self):
        if self.status:
            return self.frame
        return None
if __name__ == '__main__':
    stream_link = 'tcp://193.168.0.1:6200/'
    vid_link = 'E:/TUGAS AKHIR/testing/dilarangstop_20.mp4'
    webcam = 2
    streamer = ThreadedCamera(vid_link)

    prev_frame_time = 0
    new_frame_time = 0

    sound = {
        '40km': 'E:/TUGAS AKHIR/rambu-new/sound/40 kilometer.wav',
        'dilarang_stop': 'E:/TUGAS AKHIR/rambu-new/sound/dilarang
berhenti.wav',
        'dilarang_melintas': 'E:/TUGAS AKHIR/rambu-
new/sound/dilarang melintas.wav',
        'dilarang_parkir': 'E:/TUGAS AKHIR/rambu-new/sound/dilarang
parkir.wav',
        'dilarang_putar_balik': 'E:/TUGAS AKHIR/rambu-
new/sound/dilarang putar balik.wav',
        'lampu_hijau': 'E:/TUGAS AKHIR/rambu-new/sound/lampu
hijau.wav',
        'lampu_kuning': 'E:/TUGAS AKHIR/rambu-new/sound/lampu
kuning.wav',
        'lampu_merah': 'E:/TUGAS AKHIR/rambu-new/sound/lampu
merah.wav',
        'perhatian': 'E:/TUGAS AKHIR/rambu-
new/sound/perhatian.wav',
        'putar_balik': 'E:/TUGAS AKHIR/rambu-new/sound/putar
balik.wav',
        'penyeberangan_orang': 'E:/TUGAS AKHIR/rambu-
new/sound/penyeberangan-orang.wav',
        'Stop': 'E:/TUGAS AKHIR/rambu-new/sound/berhenti.wav'
    }

    while True:
        frame = streamer.grab_frame()
```

```
        if frame is not None:
            labels, boxes, scores = model.predict(frame)

            # time when we finish processing for this frame
            new_frame_time = time.time()
            fps = 1/(new_frame_time-prev_frame_time)
            prev_frame_time = new_frame_time
            fps = str(fps)
            # print('FPS:',fps)

            score_filter=0.5
            # Plot each box with its label and score
            for i in range(boxes.shape[0]):
                if scores[i] < score_filter:
                    continue

                box = boxes[i]
                cv2.rectangle(frame, (int(box[0]), int(box[1])),
(int(box[2]), int(box[3])), (255, 0, 0), 3)
                if labels:
                    cv2.putText(frame, '{}: {}'.format(labels[i],
round(scores[i].item(), 2)), (int(box[0]), int(box[1] - 10)),
                                cv2.FONT_HERSHEY_SIMPLEX, 0.5,
(255, 0, 0), 3)
                    playsound(sound[labels[i]])
                    print(labels[i],':', scores[i],', FPS:',fps)
            frame = cv2.resize(frame, (1080,720))
            cv2.imshow("Context", frame)

        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
    streamer.release()
    cv2.destroyAllWindows()
```

LEMBAR PERBAIKAN SKRIPSI

"SISTEM DETEKSI RAMBU DAN LAMPU LALU LINTAS UNTUK AUTONOMOUS CAR MENGGUNAKAN FASTER R-CNN"

OLEH:

KHAIRUL HIDAYAT
D121171501

Skripsi ini telah dipertahankan pada Ujian Akhir Sarjana tanggal 21 September 2022.

Telah dilakukan perbaikan penulisan dan isi skripsi berdasarkan usulan dari penguji dan pembimbing skripsi.

Persetujuan perbaikan oleh tim penguji:

	Nama	Tanda Tangan
Ketua	Dr. Indrabayu, S.T., M.T., M.Bus.sys.	
Sekretaris	Anugrayani Bustamin, S.T., M.T.	
Anggota	Dr. Ir. Ingrid Nurtanio, M.T.	
	Dr. Eng. Ady Wahyudi Paundu, S.T., M.T.	

Persetujuan Perbaikan oleh pembimbing:

Pembimbing	Nama	Tanda Tangan
I	Dr. Indrabayu, S.T., M.T., M.Bus.sys.	
II	Anugrayani Bustamin, S.T., M.T.	