دستیار تشخیص حداکثر سرعت مجاز

بهار ۱۴۰۱

تیم شماره ۱

اعضای گروه:

یاشار ظروفچی

سپهر صفری

فهرست

مقدمه

شرح پروژه

"دستیار تشخیص حداکثر سرعت مجاز" محصولی است که حداکثر سرعت مجاز خودرو را به با تصویر برداری از تابلوهای راهنمایی و رانندگی در حین حرکت و سپس تحلیل آن‌ها بدست آورده و به کمک تکنولوژی GPS سرعت فعلی خودرو را محاسبه می‌کند. در نهایت با مقایسه سرعت فعلی و حداکثر سرعت مجاز در صورت نیاز به راننده هشدار لازم را مبنی بر اینکه سرعت مناسب نیست، می‌دهد.

این محصول همچنین به همراه یک برنامه اندروید است که باید بر روی گوشی هوشمند راننده نصب شود. به کمک این برنامه، گوشی می‌تواند به برد اصلی محصول که در خودرو قرار دارد متصل شود و از آن اطلاعات مختلفی را دریافت کند. این اطلاعات علاوه بر هشدارها شامل تاریخچه سرعت‌های خودرو، تابلوهای دیده شده و ... نیز می‌شود.

به طور کلی می‌توان ویژگی‌ها و قابلیت‌های این محصول را در موارد زیر فهرست کرد:

۱. عکس برداری از تابلوهای راهنمایی در حین حرکت خودرو و پردازش آن‌ها برای بدست آوردن حداکثر سرعت مجاز
۲. تحلیل مکان خودرو در لحظات مختلف برای محاسبه سرعت فعلی آن
۳. ارسال اطلاعات مختلف اعم از هشدارها، تاریخچه سرعت و ... به گوشی راننده از طریق برنامه اندروید نصب شده روی آن

مزایای رقابتی

از مزایای رقابتی این محصول می‌توان به موارد زیر اشاره کرد:

۱. محاسبه مکان فعلی خودرو از طریق ماژول GPS خود محصول و نبود نیاز به استفاده از GPS گوشی راننده
۲. قابلیت اتصال گوشی راننده به برد از طریق ارتباط بی‌سیم
۳. پردازش تصاویر تابلوهای راهنمایی برای محاسبه سرعت مجاز برخلاف محصولات دیگر که از طریق API سرعت مجاز را بدست می‌آورند و اگر اطلاعات مسیر فعلی در سرور موجود نباشد، نمی‌توانند سرعت را محاسبه کنند.
۴. قابلیت دریافت اطلاعات از برد اصلی به کمک رابط کاربری مناسب که بر روی گوشی همراه موجود است.

قطعات

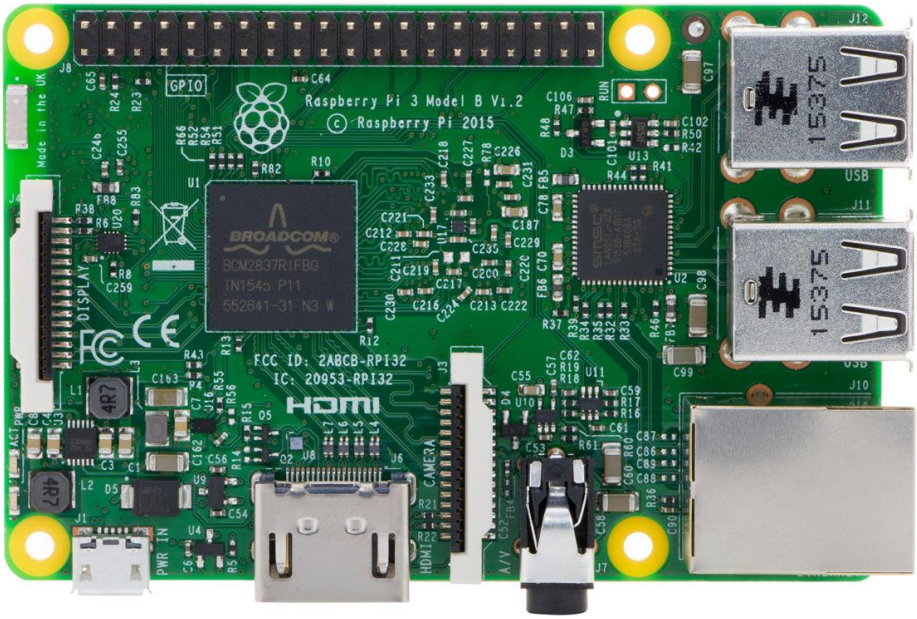
در این بخش قطعات سخت‌افزاری استفاده شده در این محصول بررسی می‌شوند.

برد Rasbperry pi 3 مدل B

این قطعه، برد اصلی محصول است که نقش اصلی را برعهده دارد. کارهایی نظیر دریافت داده از ماژول‌ها، فرستادن اطلاعات به گوشی راننده، پردازش اطلاعات و ... در این قطعه انجام می‌شود. از قابلیت‌های این برد می‌توان به موارد زیر اشاره کرد:

۱. پشتیبانی از سیستم عامل لینوکس
۲. پشتیبانی از اتصال WiFi
۳. قابلیت اتصال به ماژول‌های مختلف
۴. و ...

تصویری از این قطعه در زیر مشاهده می‌شود.



شکل ۱: برد رزبری پای 3 مدل B

در این لینک می‌توان این محصول را خریداری کرد.

ماژول دوربین رزبری پای کد IMX219

به کمک این ماژول می‌توان از محیط عکس گرفت و آن را به برد رزبری فرستاد. از قابلیت‌های این ماژول می‌توان به موارد زیر اشاره کرد:

۱. سینتکس آسان در نوشتن کد برای برقراری ارتباط با رزبری و کار با دوربین

۲. قابلیت تنظیم رزولوشن، میزان روشنایی و ...

در زیر عکسی از این قطعه را می‌توانید مشاهده کنید:

شکل ۲: ماژول دوربین

در این لینک می‌توان این محصول را خریداری کرد.

ماژول موقعیت NEO-7M-C GPS

برد رزبری به بر روی خود ماژول GPS از پیش تعبیه شده ندارد. به همین خاطر برای استفاده از تکنولوژی موقعیت یابی باید از ماژول‌های جانبی مانند ماژول موقعیت NEO-7M-C GPS استفاده نمود. این ماژول به رزبری وصل می‌شود و به صورت streaming داده‌ها موقعیت مکانی را برای رزبری می‌فرستد. البته این ماژول در ابتدای روشن شدن دقت پایینی دارد و باید کمی منتظر بود تا دقت مناسب را پیدا کند.

در زیر عکسی از این قطعه را می‌توانید مشاهده کنید:

شکل ۳: ماژول GPS

در این لینک می‌توان این قطعه را خریداری کرد.

سیم

برای این محصول به 3 مدل سیم نیاز بود:

۱. نری – نری
۲. نری – مادگی
۳. مادگی – مادگی

تصویر این 3 نوع سیم را در زیر مشاهده می‌کنید:

شکل ٤: سیم نری - نری

شکل ٥: سیم نری - مادگی

شکل ٦: سیم مادگی – مادگی

در این لینک می‌توانید سیم نری – نری را خریداری کنید.

در این لینک می‌توانید سیم نری – مادگی را خریداری کنید.

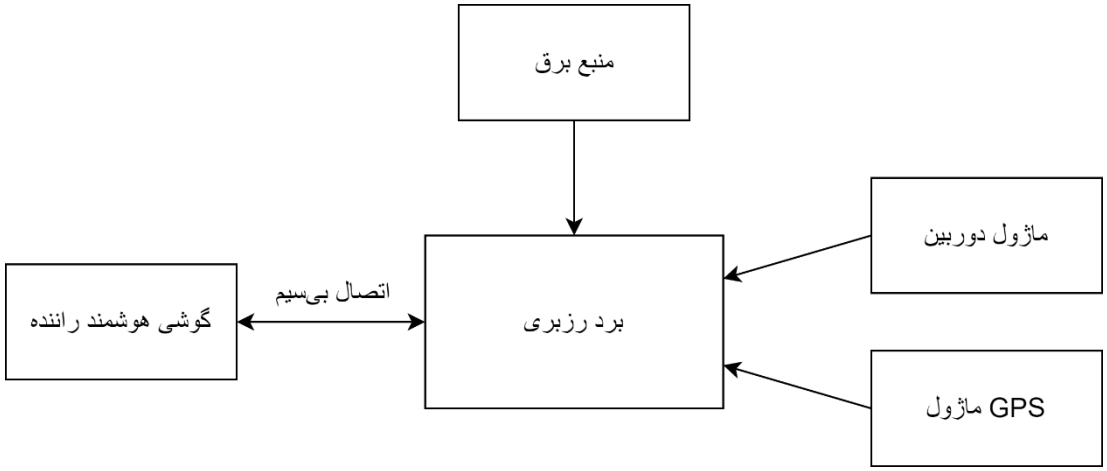
در این لینک می‌توانید سیم مادگی – مادگی را خریداری کنید.

معماری سیستم

در این بخش دو معماری سخت‌افزاری و نرم‌افزاری محصول را به طور مختصر بررسی می‌کنیم. در فصل‌های بعد کدهای برد رزبری و برنامه اندروید به طور دقیق‌تر بررسی خواهند شد.

معماری سخت‌افزاری

نمودار بلوکی معماری در زیر قرار داده شده است:

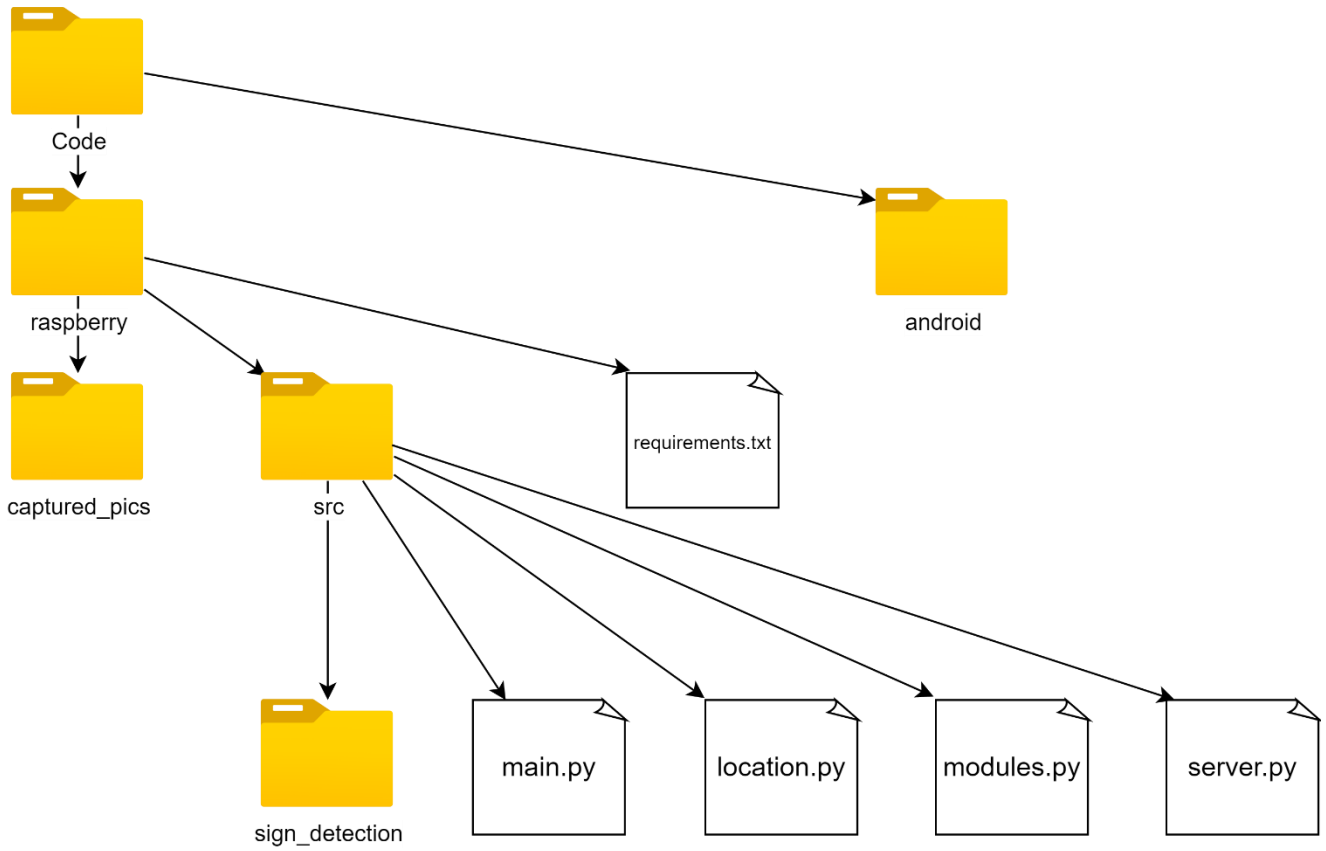


شکل ۷: نمودار بلوکی سخت‌افزار محصول

همان طور که در شکل بالا مشاهده می‌شود، برد رزبری از طریق اتصال سیمی به 2 ماژول دوربین و GPS متصل می‌شود و از آن‌ها داده می‌گیرد. همچنین از منبع برق هم انرژی دریافت می‌کند. در نهایت هم با گوشی هوشمند راننده از طریق اتصال بی‌سیم ارتباط برقرار کرده و به درخواست‌های آن پاسخ می‌دهد و به آن اطلاعات می‌فرستد.

معماری نرم‌افزاری

تصویر کلی پوشه‌بندی کد محصول در زیر نمایش داده شده است:



شکل ۸: نمودار پوشه‌بندی کد محصول

android: این پوشه شامل کد برنامه اندروید است.

raspberry: این پوشه شامل کدهای برد رزبری است.

در پوشه raspberry بخش‌های زیر وجود دارد:

۱. **captured_pics**: عکس‌هایی که دوربین می‌گیرد در این جا ذخیره می‌شود.

۲. **requirements.txt**: نیازمندی‌های محصول است که همگی از نوع کتابخانه‌های پایتون هستند. برای نصب آن‌ها کافی است پس از ساخت محیط مجازی (virtual environment) دستور `pip install –r requirements.txt` را اجرا کرد.

۳. **src**: کدهای رزبری در این مسیر قرار دارند.

در پوشه src پوشه‌ها و فایل‌های زیر وجود دارند:

۱. **sign_detection**: این پوشه حاوی کدهای مربوط به پردازش تصویر و تشخیص تابلوهای راهنمایی است. توابع محاسبه سرعت مجاز نیز در این بخش قرار دارند. به دلیل پیچیدگی آن این بخش را در قسمت‌های بعد به طور مفصل توضیح خواهیم داد.

۲. **main.py**: این فایل، فایل اصلی محصول است. در واقع برای اجرای کد روی رزبری، این فایل اجرا می‌شود.

۳. **location.py**: توابع مربوط به تعامل با ماژول GPS و محاسبه مکان و سرعت فعلی خودرو در این جا قرار دارند.

۴. **modules.py**: این فایل، فایل اصلی است که با ماژول‌های مختلف رزبری تعامل می‌کند و به طور متناوب از آن‌ها داده خوانده، سپس آن داده‌ها را پردازش می‌کند.

۵. **server.py**: در این فایل کدی قرار دارد که با اجرای آن سروری ساده بر روی رزبری بالا می‌آید. وظیفه این سرور پاسخگویی به درخواست‌های برنامه اندروید است.

زیربخش‌های سیستم

در این قسمت، زیربخش‌های مختلف نرم‌افزار محصول را بررسی می‌کنیم که شامل کد رزبری و کد برنامه اندروید می‌شود.

بدنه اصلی کد رزبری

این زیربخش آن قسمت از کد رزبری را شامل می‌شود که سیستم را راه‌اندازی می‌کند و با بقیه بخش‌های کد تعامل می‌کند و بین آن‌ها ارتباط برقرار می‌کند. این زیربخش در فایل های main.py و modules.py که در بخش معماری نرم‌افزار بررسی شدند، قرار دارد. در ادامه کدهای آن را بررسی می‌کنیم.

همان طور که در بخش‌های قبل‌تر گفتیم اجرای کد روی رزبری با اجرای تابع main() در فایل main.py شروع می‌شود. در این تابع ابتدا تابع init() صدا زده تا زیربخش‌های پردازش تصویر و مکان یابی راه‌اندازی شوند، سپس تابع create_threads() صدا زده می‌شود که 2 تا ریسه در آن ساخته می‌شود. ریسه اول modules-thread نام دارد که وظیفه تعامل با ماژول‌ها را برعهده دارد. ریسه دوم server-thread نام دارد که وظیفه پاسخ به درخواست‌های گوشی همراه راننده را دارد. بخش‌های مهم فایل main.py در زیر قرار داده شده است:

```
# Setups subsystems
def init():
    sign_detection.init()
    location.init()


# Creates threads. One will handle the modules and the other will handle the server.
def create_threads():
    modules_thread = Thread(
        target=handle_modules, args=None, name='modules-thread', daemon=True)
    server_thread = Thread(
        target=run_server, args=(SERVER_PORT,), name='server-thread', daemon=True)

    modules_thread.start()
    server_thread.start()

    modules_thread.join()
    server_thread.join()


def main():
    init()
    create_threads()
```

بقیه بدنه اصلی در فایل modules.py قرار دارد. آن قسمت از این فایل که مربوط به این زیربخش است، تابع handle_modules() است. این تابع همان تابعی است که ریسه modules-thread در لحظه شروع خود، آن را شروع به اجرا می‌کند و همان طور که در زیر پیدا است از یک حلقه while تشکیل شده که هیچگاه از آن بیرون نمی‌آییم. در واقع در هر CLK_PERIOD ثانیه کل این حلقه یک بار اجرا می‌شود که در آن صورت نیاز با ماژول‌های دوربین و GPS تعامل می‌شود، بدین معنا که از آن‌ها داده می‌گیریم، داده‌ها را پردازش می‌کنیم و اطلاعات لازم را ذخیره می‌کنیم. این تابع در زیر قابل مشاهده است:

```
# Main module handling function which calls other functions periodically
def handle_modules():
    last_camera_clk = 0

    # Each loop cycle will last CLK_PERIOD seconds
    while True:
        start = get_time()

        handle_gps(start)
        if start - last_camera_clk > CAMERA_PERIOD:
            last_camera_clk = start
            handle_camera(start)

        now = get_time()
        sleep(CLK_PERIOD - (now - start))
```

تشخیص تابلوهای راهنمایی و رانندگی

این زیربخش مسئولیت پردازش تصاویر ثبت شده توسط ماژول دوربین و استخراج تابلوها از آن‌ها است. این زیربخش پوشه sign_detection و محتویات درون آن را شامل می‌شود.

این پوشه و محتویات آن در واقع تغییر یافته پروژه‌ای بر روی گیتهاب است که در این لینک قرار دارد. این پروژه از کتابخانه pytorch زبان python استفاده کرده است که مخصوص پردازش تصویر به کمک شبکه‌های عصبی است.

حال محتویات sign_detection را بررسی می‌کنیم. مدل از پیش آموزش دیده این شبکه عصبی در پوشه models در فایلی به نام model_40.pth قرار دارد. ما برای پردازش تصاویر این مدل را بارگذاری کرده و از آن استفاده می‌کنیم.

2 فایل data.py و model.py فایل‌هایی هستند که سازنده اصلی این پروژه برای طراحی شبکه عصبی و تعامل با آن استفاده کرده است. ما هم صرفا از آن‌ها استفاده کردیم و وارد جزئیات نشدیم.

خروجی این شبکه عصبی در نهایت عددی بین 0 تا 42 خواهد بود که هر عدد نشان دهنده یکی از 43 نوع تابلویی است که شبکه عصبی پوشش می‌دهد. فایل labels.csv مشخص می‌کند که هر عدد مربوط به کدام تابلو است. بدین شکل که دو ستون دارد، ID و NAME، ستون ID عدد را نشان می‌دهد و ستون NAME اسم تابلوی مورد نظر را.

در نهایت به فایل main.py می‌رسیم که فایل اصلی sign_detection است و ما آن را با تغییر فایلی در پروژه اصلی بدست آوردیم. وظیفه این فایل به طور کلی دریافت یک عکس و پیدا کردن تابلو به کمک فایل‌های دیگر است. همچنین برخی محاسبات ساده دیگر هم در این فایل انجام می‌شود. در ادامه این فایل را به طور عمیق‌تر بررسی می‌کنیم.

توابع load_model() و set_labels() به ترتیب وظیفه بارگذاری مدل از پیش آموزش دیده و بارگذاری اطلاعات فایل labels.csv در یک متغیر سراسری به نام labels را دارند. کد آن‌ها در زیر نمایش داده شده است:

```
# Loads pre-trained model from model_40.pth
def load_model():
    global model
    state_dict = torch.load(MODEL_PATH)
    model = Net()
    model.load_state_dict(state_dict)
    model.eval()


# Loads labels from labels.csv
def set_labels():
    global labels
    with open(LABELS_PATH, 'r') as f:
        reader_obj = csv.reader(f)
        labels = []
        for row in reader_obj:
            labels.append(row[1])
    labels.pop(0)
```

تابع predict_sign() اسم یک عکس را می‌گیرد، سپس محل آن را در حافظه پیدا می‌کند (دوربین عکس را در حافظه ذخیره می‌کند) و سپس با بارگذاری آن از حافظه، درون آن تابلو پیدا می‌کند. درون این فایل تعدادی تابع دیگر تعریف شده است که predict_sig() از آن‌ها استفاده می‌کند.

تابع load_pic() با گرفتن مسیر عکس، آن را بارگذاری می‌کند و آن را به صورت یک عکس از کتابخانه Pillow برمی‌گرداند.

تابع eval_pic() با گرفتن یک عکس، تابلوی تشخیص داده شده در آن را به همراه احتمال درست بودن تشخیص برمی‌گرداند. این تابع در واقع با شبکه عصبی تعامل می‌کند.

تابع process_pic() همه پیکسل‌هایی از عکس که تقریبا قرمز هستند را به پیکس تقریبا آبی تغییر می‌دهد.

تابع does_have_red() چک می‌کند که در عکس داده شده به عنوان ورودی، حداقل درصد مشخصی از پیکسل‌ها تقریبا قرمز هستند یا نه.

با توجه به این که شبکه عصبی توان پیدا کردن تابلو در عکسی را دارد که تابلو در کل چهارچوب عکس قرار داشته باشد و نه در بخشی از آن، کلیت روشی که ما برای پیدا کردن یک تابلو در عکسی که از خیابان گرفته شده است استفاده کردیم، به این صورت است:

- ابتدا مربعی با ابعادی کوچک در نظر می‌گیریم
- این مربع را بر روی عکس گرفته شده حرکت می‌دهیم، قسمتی از عکس که در این مربع قرار دارد را **قالب** می‌نامیم.
- اگر حداقل تعدادی از پیکس‌های قاب در عکس اصلی قرمز نبود (این را به کمک does_have_red() انجام می‌دهیم)، این قالب را رد می‌کنیم و مربع را دوباره حرکت می‌دهیم.
- اگر شرط بالا برقرار بود، قالب را به eval_pic() می‌دهیم تا درون آن تابلو پیدا کند. اگر احتمال درست بودن تابلوی تشخیص داده شده کم بود، مانند مرحله قبل، قالب را رد می‌کنیم و مربع را دوباره حرکت می‌دهیم.
- اما اگر احتمال تابلوی تشخیص داده شده زیاد بود، آن را به عنوان جواب تابع predict_sign() برمی‌گردانیم.
- در صورتی که با مربع با اندازه فعلی کل عکس را پیمایش کردیم و تابلویی پیدا نشد و اندازه مربع را بزرگ‌تر می‌کنیم و پیماش عکس را دوباره انجام می‌دهیم تا جایی که مربع از خود عکس بزرگ‌تر شود.

لازم به ذکر است ما همیشه یک نسخه اصلی عکس را داریم و یک نسخه پردازش شده که پیکسل‌های قرمز آن آبی شده‌اند (به کمک تابع process_pic()). دلیل آن اینست که مجموعه دادگان استفاده شده در شبکه عصبی مجموعه دادگان GTSRB است که از تابلوهای آلمان است. در آلمان تابلوهای حداکثر سرعت مجاز، دورشان آبی است، اما در ایران دورشان قرمز است. به همین خاطر پردازش اصلی بر روی عکس پردازش شده انجام می‌شود، یعنی عکسی که دور تابلوها در آن آبی شده است. در غیر این صورت شبکه عصبی نمی‌تواند تابلو را تشخیص دهد.

دقت شود دلیل این که قالبی که حداقل تعدادی از پیکسل‌های آن در تصویر پردازش شده، قرمز نیست، رد می‌شود اینست که سرعت پیدا کردن تابلو افزایش یابد. دلیل این کار اینست که اگر تابلو در یک قالب باشد، باید حداقل تعدادی از پیکسل‌های آن که نشان‌دهنده دور تابلو هستند، قرمز باشند.

کد تابع predict_sign() که مهم‌ترین تابع این بخش است در زیر قرار داده شده است:

```
# Finds a traffic sign in image file with name PIC_NAME and returns the sign's label,
probability and name.
```

```
# Returns None if no sign with high enough probability was found.
def predict_sign(pic_name):
    global model, labels

    # Loads image in PATH
    def load_pic(path):
        with Image.open(path) as img:
            img.load()
            return img.convert('RGB')

    # Returns sign detected in PIC
    def eval_pic(pic: Image):
        output = torch.zeros(size=[1, 43], dtype=torch.float32)
        for i in range(0, len(transforms)):
            data = transforms[i](pic)
            data = data.view(1, data.size(0), data.size(1), data.size(2))
            data = Variable(data)
            output = output.add(model(data))
        # Remove unimportant classes in a crude way
        for i in [e for e in list(range(43)) if e not in ACCEPTED_SIGN_LABELS]:
            output.data[0][i] = -999999
        prediction = output.data.max(1, keepdim=True)
        pred = int(prediction[1])
        prob = 1 + float(prediction[0])
        return pred, prob, labels[pred]

    # Converts red pixels to blue in PIC.
    def process_pic(pic: Image):
        img = Image.new('RGB', (pic.size))
        for x in range(pic.width):
            for y in range(pic.height):
                r, g, b = pic.getpixel((x, y))
                if r > 50 and g < 200 and b < 200:
                    img.putpixel((x, y), (b, g, r))
                else:
                    img.putpixel((x, y), (r, g, b))
        return img

    # Checks if atleast a specified percentage of PIC's pixels are red.
    def does_have_red(pic: Image):
        red_cnt = 0
        for x in range(pic.width):
            for y in range(pic.height):
                r, g, b = pic.getpixel((x, y))
                if r > 50 and g < 200 and b < 200:
```

```
                    red_cnt += 1

        pic_size = pic.width * pic.height
        return red_cnt / pic_size > RED_THRESHOLD

    transforms = [data_transforms, data_jitter_hue, data_jitter_brightness, data_jit-
ter_saturation,
                  data_jitter_contrast, data_rotate, data_hvflip, data_shear,
data_translate, data_center]

    orig_pic = load_pic(PICS_DIR + pic_name)
    pic = process_pic(orig_pic)
    grid_size = GRID_INIT_SIZE
    grid_step = int(grid_size * GRID_STEP_TO_SIZE_RATIO)
    # Iterate over grids to find a sign.
    while True:
        for y in range(0, pic.height - grid_size, grid_step):
            for x in range(0, pic.width - grid_size, grid_step):
                # Skip the grid if orginal pic doesn't have enough red pixels.
                orig_cropped_pic = orig_pic.crop(
                    (x, y, x + grid_size, y + grid_size))
                if not does_have_red(orig_cropped_pic):
                    continue

                # Check grid to find sign.
                cropped_pic = pic.crop((x, y, x + grid_size, y + grid_size))
                pred_label, prob, name = eval_pic(cropped_pic)
                if prob > PROB_THRESHOLD:
                    return pred_label, prob, name

        # Update grid attributes.
        grid_size = int(grid_size * GRID_SIZE_MUL)
        grid_step = int(grid_size * GRID_STEP_TO_SIZE_RATIO)
        if grid_size > pic.width or grid_size > pic.height:
            break

    return None
```

تابع get_speed_limit() با دریافت آرایه تابلوهای تشخیص داده شده در طی زمان، با توجه به آخرین تابلو، حداکثر سرعت مجاز را برمی‌گرداند.

تابع get_random_sign()یک تابلوی تصادفی را در برخی مواقع به ما برمی‌گرداند. در بقیه مواقع هیچ تابلویی برنمی‌گرداند. می‌توان این که به چه احتمالی تابلویی برگرادند یا نه را مشخص کرد.

لازم به ذکر است پس از اجرای روشی که در نظر گرفته بودیم متوجه شدیم که این روش با این که با دقت معمولی کار می‌کند، از سرعت عمل کافی برای برطرف کردن نیاز پروژه که حدودا باید در هر 1 ثانیه یک تصویر از خیابان پردازش شود و در آن تابلو پیدا شود، برخوردار نیست. به همین خاطر از تابع get_random_sign() برای اجرای پروژه و گرفتن خروجی از آن استفاده کردیم.

لازم به ذکر است که پروژه‌های مختلفی را در این حوزه بررسی کردیم، اما به نظر می‌رسد چالش اصلی که باعث می‌شود این روش به اندازه کافی سریع نباشد، کند بودن ذاتی شبکه‌های عصبی نسبت به دیگر روش‌ها است. همچنین این که رزبری کارت گرافیک ندارد چالش استفاده از این روش را بیش‌تر می‌کند. به نظر می‌رسد راه مناسب‌تر برای پردازش تصویر در این پروژه، استفاده از روش‌های سریع‌تر مانند تبدیل‌ها، فیلترها و روش گرافی باشد.

تشخیص سرعت و مکان خودرو

این زیربخش مسئولیت محاسبه سرعت و مکان خودرو و مدیریت این اطلاعات را برعهده دارد. کد آن کل فایل location.py و قسمتی از modules.py را دربرمی‌گیرد. آن بخشی که در فایل modules.py قرار دارد، شامل تابع handle_gps() و متغیرهای سراسری locations،distances و prev_dist می‌شود. کد این اجزا در زیر قرار داده شده است:

```
lock = threading.Lock()

# These are used to store data as time passes.
locations = []  # Locations of the car in different timestamps
speeds = [] # Speeds of the car in different timestamps
prev_speed = 0  # Last calculated speed
signs = [] # Traffic signs detected in different timestamps
speed_limits = []   # Speed limits in different timestamps
```

```
# Handle GPS module and do the jobs related to the car's location and speed
def handle_gps(cur_time):
    global locations, speeds, prev_speed, lock

    lock.acquire()
    loc = location.get_location()
    locations.append((cur_time, loc))
    cur_speed = location.get_speed(locations, prev_speed)
    speeds.append((cur_time, cur_speed))
    prev_speed = cur_speed
    lock.release()
```

لازم به ذکر است که متغیرهای سراسری دیگری که در بالا مشاهده می‌شود برای زیربخش‌های دیگر (به طور دقیق‌تر زیربخش تشخیص تابلوهای راهنمایی و رانندگی) استفاده می‌شود. همچنین متغیر سراسری lock برای همگام سازی ریسه‌های modules-thread و server-thread در استفاده از این متغیرهای سراسری است. چرا که modules-thread به این متغیرها که از نوع لیست هستند اطلاعات جدید اضافه می‌کند و server-thread برای جواب دادن به درخواست گوشی راننده، از این متغیرها اطلاعات می‌خواند و نباید بین این دو مسابقه (race) پیش بیاید. در نهایت به این اشاره می‌کنیم که ساختار متغیرهای سراسری از نوع لیست که اطلاعات را در خود نگه می‌دارند به این شکل است که هر خانه از این لیست‌ها یک چندتایی (tuple) است که خانه اول آن زمان ثبت آن داده به صورت unix timestamp و خانه دوم آن خود داده می‌باشد، به این شکل با داشتن هر خانه از لیست می‌توان خود داده و زمان دقیق ثبت آن را بدست آورد.

در حقیقت، تنها وظیفه‌ی رزبری نگه‌داری مسافت‌های پیموده شده است. زیرا هم رزبری و هم کد اندروید (که در ادامه خواهیم دید)، در بازه‌های ۱ ثانیه‌ای اقدام به ثبت و دریافت جابه‌جایی (به متر) می‌کنند در نتیجه سرعت لحظه‌ای تقریبی همان جابه‌جایی بر ثانیه است. در حالت اصلی، رزبری و گوشی به یکدیگر متصل‌اند، اما چون تست رزبری و اندروید در دو زمان متفاوت صورت گرفت، ابتدا رکوردهایی از

مسافت‌های پیموده شده به همراه محدودیت سرعت در تعدادی فایل با نام‌های trajectory_history_limit ذخیره شده‌اند و کد اندروید با کمک این فایل‌های لاگ تست شده است.

سرور برد رزبری

این زیربخش مسئولیت بالا آوردن سروری ساده بر روی رزبری را برعهده دارد که باید به درخواست‌های گوشی راننده پاسخ دهد و اطلاعات خواسته شده اعم از هشدارها، تاریخچه تابلوهای دیده شده، تاریخچه سرعت‌ها و ... را به آن بدهد. کد زیربخش کل فایل server.py را شامل می‌شود. برای بالا آوردن سرور از کتابخانه BaseHTTPRequestHandler استفاده کردیم که یک کتابخانه ساده و ابتدایی پایتون برای مدیریت درخواست‌های http است. مستند این کتابخانه را می‌توانید در این لینک مشاهده کنید.

روند کلی استفاده ما از این کتابخانه به این صورت بوده است که یک کلاس به نام MyHandler تعریف کرده‌ایم که کلاس BaseHTTPRequestHandler را ارث می‌برد. در این کلاس یک تابع اصلی do_GET() وجود دارد که همه درخواست‌های از نوع GET که به این سرور می‌رسند، این تابع برایشان اجرا می‌شود. کد این تابع در زیر قابل مشاهده است:

```
# HTTP request handler class
class MyHandler(BaseHTTPRequestHandler):
    # Main request handler function
    def do_GET(self):
        self.__parse_get_params()
        if self.path.startswith('/get-locations'):
            self.__send_response(locations)
        elif self.path.startswith('/get-speeds'):
            self.__send_response(speeds)
        elif self.path.startswith('/get-signs'):
            self.__send_response(signs)
        elif self.path.startswith('/speed-limits'):
            self.__send_response(speed_limits)
        else:
            self.__send_400_response()
```

پیاده‌سازی APIهای این سرور هم ساده است و به طور کلی به این شکل است که گوشی راننده نوع داده‌ای که می‌خواهد (مثلا تاریخچه سرعت‌ها) و timestamp آخرین داده‌ای که دریافت کرده را برای ما می‌فرستد و ما از آن لحظه به بعد هر داده جدیدی که بدست آورده‌ایم را برای او می‌فرستیم. لازم به ذکر است اولین بار که گوشی درخواست می‌دهد timestamp را برابر با 1- قرار می‌دهد. تابع __send_response() مسئول مدیریت APIها با توجه به نوع آن‌ها است. کد آن در زیر مشاهده می‌شود:

```
# Sends response to android app.
    def __send_response(self, arr):
        timestamp = int(self.params['timestamp'][0])
        data_lock.acquire()
        if timestamp == -1: # First request from android app will have TIMESTAMP equal
to -1.
            index = -1
```

```
        else:   # Other requests will set TIMESTAMP equal to the timestamp of the last
cell they have gotten.
            index = self.__find_timestamp_index(timestamp, arr)
        self.send_response(200)
        self.send_header('Content-type', 'text/html')
        self.end_headers()
        self.wfile.write(json.dumps(arr[index+1:len(arr)]).encode())
        data_lock.release()
```

با توجه به این که متغیرهای سراسری که داده‌ها در آن‌ها ذخیره می‌شوند و در زیربخش‌های تشخیص تابلوهای راهنمایی و رانندگی و تشخیص سرعت و مکان خودرو آن‌ها را بررسی کردیم، براساس زمان ثبت‌شده داده از کوچک به بزرگ مرتب شده‌اند، برای پیدا کردن اولین داده‌ای که از آن به بعد باید برای گوشی فرستاده شود، از جستجوی دودویی استفاده کردیم. تابع __find_timestamp_index() مسئول این کار است. کد آن در زیر مشاهده می‌شود:

```
# Returns the index of the first cell with the timestamp greater or equal to TIMESTAMP
in ARR.
    # Uses binary search algorithm for better performance.
    def __find_timestamp_index(timestamp: int, arr: [tuple]):
        # Will find TIMESTAMP in ARR using bin search.
        def bin_search(start, end):
            if start == end:
                return start
            mid = (start + end) // 2
            mid_timestamp = arr[mid][0]
            if mid_timestamp < timestamp:
                return bin_search(mid+1, end)
            else:
                return bin_search(start, mid)

        return bin_search(0, len(arr)-1)
```

برنامه اندروید

۱. رویه اصلی (Main Activity)

توابع اصلی برنامه در این قسمت قرار دارند که به توضیح مختصر آنان می‌پردازیم:

```
private TrajectoryDTO fetch_trajectory() {
    try {
        TrajectoryDTO trajectoryDTO = apiService.getTrajectoryInfo();
        TrajectoryRecord record = modelConverter.getTrajectoryEntity(trajectoryDTO);
        trajectoryRepository.updateTrajectories(record);
        return trajectoryDTO;
    } catch (ApiConnectivityException e) {
        return TrajectoryDTO.getFailed();
    }
}
```

این تابع وظیفه‌ی صدا کردن تابع دریافت اطلاعات را دارد و سپس اطلاعات پایگاه داده را به روزرسانی می‌کند.

```
private void alertDriver(float speed, int limit, TextView speedInfo) {
    String notifText;
    if (speed > limit) {
        speedInfo.setTextColor(Color.RED);
        notifText = "Exceeding speed: " + speed + " while the limit is " + limit;
    } else {
        speedInfo.setTextColor(Color.GRAY);
        notifText = "Speed: " + speed + " limit: " + limit;
    }
    Intent intent = new Intent( packageContext: this, MainActivity.class);
    TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);
    stackBuilder.addNextIntentWithParentStack(intent);
    PendingIntent resultPendingIntent = stackBuilder.getPendingIntent( requestCode: 0,
            flags: PendingIntent.FLAG_UPDATE_CURRENT | PendingIntent.FLAG_IMMUTABLE);
    NotificationCompat.Builder builder = new NotificationCompat.Builder( context: this, C
            .setSmallIcon(R.drawable.notification_icon)
            .setContentTitle("Speed notification")
            .setContentText(notifText)
            .setContentIntent(resultPendingIntent)
            .setAutoCancel(false);
    notificationManagerCompat.notify(NOTIF_ID, builder.build());
}
```

این تابع هشدار را از طریق اعلان به راننده اعلام می‌کند و درصورت کلیک روی آن اعلان به رویه‌ی اصلی منتقل می‌شود.

۲. رویه‌ی نمودار (Chart Activity)

این قسمت شامل دو تابع configureLineChart و setLineChartData است که با کمک توابع این کتابخانه پیاده‌سازی شده‌اند.

۳. بخش REST

در این قسمت، درخواست دریافت اطلاعات GPS ارسال می‌شود و پاسخ آن در تابع getTrajectoryInfo پردازش می‌شود.

```
public TrajectoryDTO getTrajectoryInfo() throws ApiConnectivityException {
    Request request = buildFetchCoinsInfoRequest();
    CompletableFuture<TrajectoryDTO> lockCompletableFuture = new CompletableFuture<>();
    client.newCall(request).enqueue(new Callback() {
        @Override
        public void onFailure(@NotNull Call call, @NotNull IOException e) {
            Log.wtf( tag: "Api",  msg: "getCoinsInfo->onFailure: ", e);
            lockCompletableFuture.complete( value: null);
        }

        @Override
        public void onResponse(@NotNull Call call, @NotNull Response response) throws IOException {
            if (response.code() == HttpURLConnection.HTTP_OK) {
                String responseBody = Objects.requireNonNull(response.body()).string();
                GpsDTO gpsDTO = objectMapper.reader().readValue(responseBody, GpsDTO.class);
                TrajectoryDTO trajectoryDTO = converter.getTrajectoryDTO(gpsDTO);
                lockCompletableFuture.complete(trajectoryDTO);
            } else {
                Log.e( tag: "Api",  msg: "getCoinsInfo->onResponse code: " + response.code());
                lockCompletableFuture.complete( value: null);
            }
        }
    });
    TrajectoryDTO result = lockCompletableFuture.join();
    if (result == null)
        throw new ApiConnectivityException();
    return result;
}
```

۴. بخش پایگاه داده

این قسمت با محوریت SQLite توسعه داده شده است. با کمک Entryها و DBHel per و همچنین ماهیت TrajectoryRecord، اطلاعات مربوط به پیمایش شخودرو در پایگاه داه ذخیره می‌شوند و با کلاس TrajectoryRepository عمل‌کرد ORM شبیه‌سازی شده است.

در تابع زیر رکوردهای مربوط به سه روز گذشته استخراج می‌شود.

```
public List<TrajectoryRecord> getPastSevenDaysInfo(long epochs) {
    SQLiteDatabase db = trajectoryDBHelper.getReadableDatabase();
    String[] columns = {_ID, RECORD_DATE, DISTANCE, AVERAGE_SPEED,
            DRIVING_TIME_SECONDS};
    Cursor cursor = db.query(TABLE_NAME, columns, selection: RECORD_DATE + " >= ?",
            new String[]{String.valueOf(epochs)}, groupBy: null, having: null, orderBy: null, limit: null);
    List<TrajectoryRecord> records = new ArrayList<>();
    while (cursor.moveToNext()) {
        records.add(readTrajectory(cursor));
    }
    cursor.close();
    return records;
}
```

همچنین برای به روزرسانی داده‌ها و یا ثبت داده‌ی جدید از توابع زیر استفاده می‌شود.

```
public void putTrajectory(TrajectoryRecord record) {
    SQLiteDatabase db = trajectoryDBHelper.getWritableDatabase();
    db.insert(TABLE_NAME, nullColumnHack: null, setTrajectoryValues(record));
}

public void updateTrajectories(TrajectoryRecord record) {
    SQLiteDatabase db = trajectoryDBHelper.getWritableDatabase();
    String selection = RECORD_DATE + " = ?";
    String[] args = new String[]{String.valueOf(record.getRecordDateEpochs())};
    Cursor sameDayRecordCursor = db.query(TABLE_NAME, columns: null, selection, args,
            groupBy: null, having: null, orderBy: null);
    if (sameDayRecordCursor.moveToLast()){
        float distance = sameDayRecordCursor.getFloat(sameDayRecordCursor.
                getColumnIndexOrThrow(DISTANCE));
        int drivingTimeSeconds = sameDayRecordCursor.getInt(sameDayRecordCursor.
                getColumnIndexOrThrow(DRIVING_TIME_SECONDS));
        record.setDistance(record.getDistance() + distance);
        record.setDrivingTimeSeconds(1 + drivingTimeSeconds);
        record.setAverageSpeed((float) (record.getDistance() * 3.6 /
                (record.getDrivingTimeSeconds())));
        db.update(TABLE_NAME, setTrajectoryValues(record), selection, args);
    } else {
        record.setDrivingTimeSeconds(1);
        putTrajectory(record);
    }
}
```

۵. ابزارهای کمکی

برای تبدیل مدل داده‌ها به یکدیگر از کلاس ModelConvertor استفاده شده است که در آن توابع زیر پیاده‌سازی شده‌اند.

```
public TrajectoryRecord getTrajectoryEntity(TrajectoryDTO trajectoryDTO) {
    TrajectoryRecord trajectoryRecord = new TrajectoryRecord();
    trajectoryRecord.setAverageSpeed(Float.parseFloat(trajectoryDTO.getAverageSpeed()));
    trajectoryRecord.setDistance(Float.parseFloat(trajectoryDTO.getDistance()));
    trajectoryRecord.setRecordDateEpochs(DateConversionUtil.getTodayEpoch());
    return trajectoryRecord;
}

@SuppressLint("DefaultLocale")
public TrajectoryDTO getTrajectoryDTO(GpsDTO gpsDTO) {
    return new TrajectoryDTO(String.valueOf(gpsDTO.getDistance()),
            String.format("%.2f", gpsDTO.getDistance() * 3.6), String.valueOf(gpsDTO.getLimit()));
}
```

شایان ذکر است که GpsDTO داده‌ی دریافتی از سرور و TrajectoryDTO داده‌ی قابل انتقال در لایه‌های MVC برنامه‌ی اندروید است. همچنین در کلاس DateConvertorUtil نیز توابعی جهت یافتن epoch ابتدای روز (جهت تعیین رکورد در پایگاه داده) و یافتن epoch سه روز گذشته (برای نمایش در نمودار) تعبیه شده‌اند.

```
public static Long getTodayEpoch() {
    long currentEpoch = System.currentTimeMillis();
    return (currentEpoch - currentEpoch % DAY_IN_MS);
}

public static Long getThreeDaysAgo() {
    LocalDateTime now = LocalDateTime.now();
    return now.with(LocalTime.MIN).atZone(ZoneId.systemDefault()).toInstant().
            toEpochMilli() - (3 * DAY_IN_MS);
}
```

بسته‌بندی

نمونه‌هایی که در بازار وجود دارد اغلب سیستم‌های جامع ناوبری هستند. به صورت کلی یا خودرو دارای این سامانه به صورت داخلی است و یا از نمونه‌های قابل حمل استفاده می‌گردد. که روی داشبورد نصب می‌گردد. پیشنهاد ما برای بسته‌بندی این است که رزبری به همراه دوربین پشت شیشه و روی داشبورد قرار گیرد، از پایه‌های آهنربایی موبایل نیز جهت نگهداری گوشی استفاده شود. می‌توان نمونه‌ی جامعی از سیستم ناوبری (بدون دوربین) را در این لینک مشاهده کرد.

جمع بندی

برآورد هزینه ساخت یک عدد از این محصول در جدول زیر نشان داده شده است:

نام قطعه	تعداد	قیمت
برد رزبری پای	۱	۴,۰۰۰,۰۰۰ تومان
ماژول دوربین	۱	۱۸۰,۰۰۰ تومان
ماژول GPS	۱	۱۳۰,۰۰۰ تومان
ماژول بازر	۱	۸,۰۰۰ تومان
سیم	به تعداد لازم	۴۰,۰۰۰ تومان
برد بورد	۱	۳۵,۰۰۰ تومان
		مجموع ۴,۳۹۳,۰۰۰ تومان

لینک وبسایت‌های که قطعات لازم را می‌توان از آن‌ها خریداری کرد، در بخش قطعات ذکر شده است.