Autonomous Driving Project Report

Group Member:

Shiqi Lin, U1002894184

Fanxuan Guo, U1003110288

Introduction:

In this project, we stepped closer to application of computer vision for industry. The goal is to take sets of images and correctly detect road and cars. By using technique of deep learning over computer vision, we could increase the accuracy of object detections, thus lower the risk of autonomous driving. If this is done correctly, it could help vehicle detects obstacles and navigate on the road.

We contributed evenly for all subtasks of this project, and we worked closer to research and make decisions together. Here is the rough division:

- Shiqi Lin

  Computed disparity, depth and best-fit plane. Wrote visualization of points cloud for 3D locations and plane. First method 1 of road detection classifier. Subtask 1, 2, 3, 4 and 5.
- Fanxuan Guo

  Wrote method 2 of road detection classifier, car detection, car viewpoint classifier and visualization of car viewpoint prediction. Subtask 3, 6, 7 and 8.

Subtask 1: Compute Disparity (*image_processing.py*)

For this task, we implement 2 methods.

- Method 1: *compute_disparity()* based on the logic from lecture 12

For each pixel in the left image, we create a patch and compare it with the patches on the right stereo image on the same row. The compared pixel locations are from max(0, left_pixel_x – 16) to left_pixel_x (same row, diff colm index). We choose 16 as max disparity to increase speed. We also create a matrix to store the vectorized patches that were seen in the right image for fast retrieval.

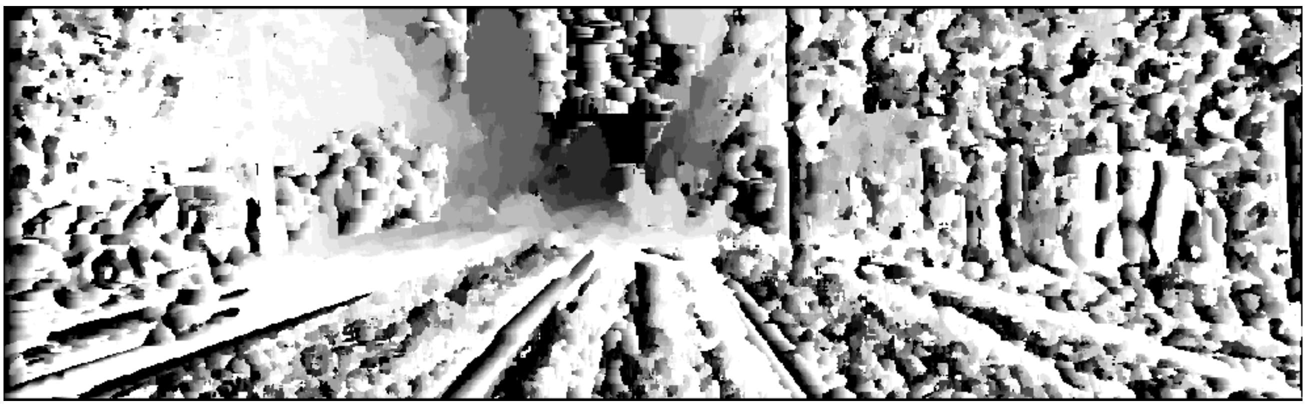


Figure 1: Method1 Disparity of *test/image_left/umm_000087.jpg*

- Method 2: *compute_disparity_by_cv2()*

We use opencv library to compute disparity. In order to create a smoother result, we smooth the disparity output by a 5 x 5 gaussian filter.

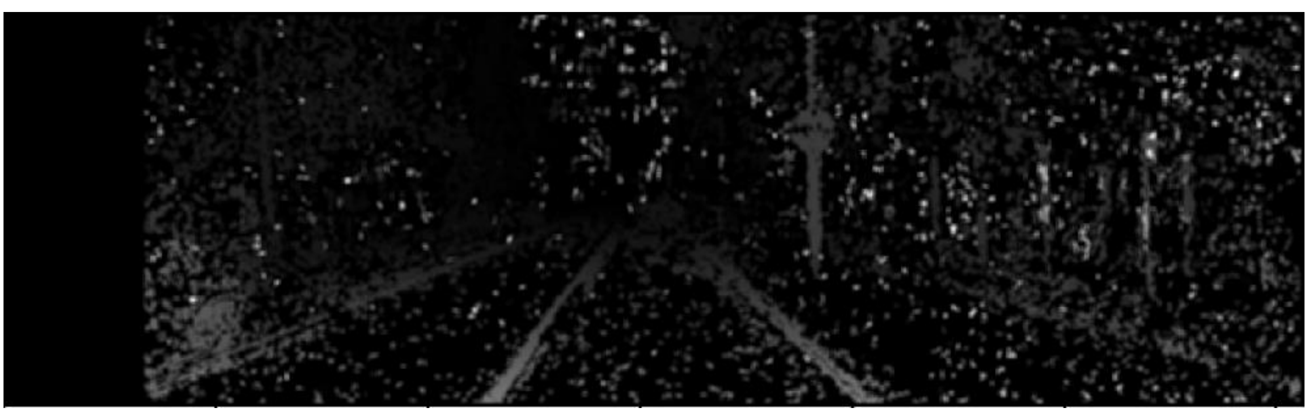


Figure 2: Method2 Disparity of *test/image_left/umm_000087.jpg*

- Comparision:

Our own method has a reasonable running time when patch size is small, but when it's too large, the method is too slow. However, method 2 runs fast under any condition.

Subtask 2: Compute Depth (*image_processings.py*)

This Task we use the code from A4 to generate the depth map.

Formula from the lecture: $Depth = \frac{\text{focal length} * \text{baseline}}{\text{disparity}}$

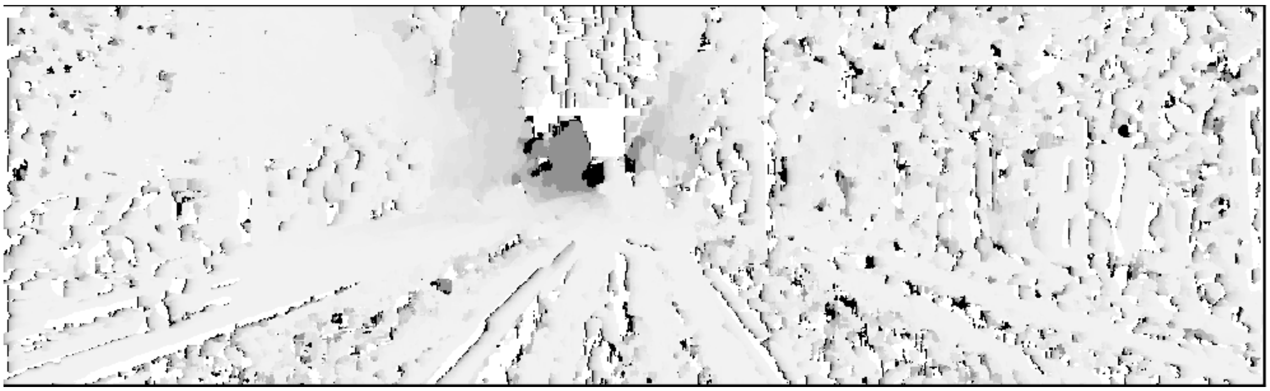


Figure 3: Method2 Disparity -> Depth Map of *test/image_left/umm_000087.jpg*

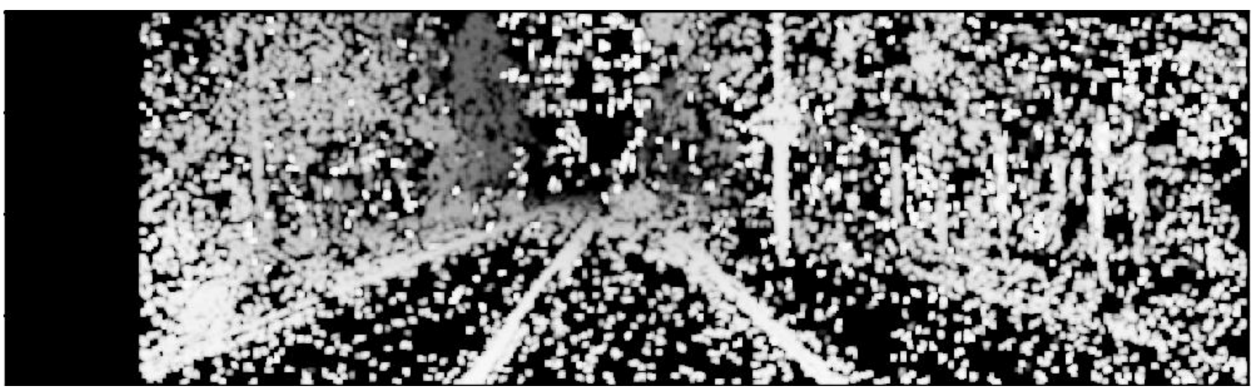


Figure 4: Method2 Disparity -> Depth Map of *test/image_left/umm_000087.jpg*

Subtask 3: Road Detection (*road_detection.py, road_segmentation.py*)

- Method 1:

We use the CNN-LSTM network structure proposed from Road Segmentation Using CNN and Distributed LSTM (https://arxiv.org/pdf/1808.04450.pdf)

Based on the pyramid prediction scheme, we firstly crop (at the center) and scale the input image to size of 600 * 160. In the image pre-processing stage, for each image, we add the x, y pixel location to the RGB channel and form a 5 channels image tensor. This way we can not only extract the color, edge, point features using NN but also predict result based on the pixel location as in the KITTI dataset, the road always appears on the bottom of the picture. We also processed each gt_mask to generate a ground truth boundary vector used in the loss calculation stage. And then following the proposed network structure, we build our own torch network model with learning rate 1e-5, Adam optimizer, L1 loss function and 80 epochs. The proposed network structure is as following: (picture from the paper) The network contains an encoder, feature processor and a decoder. It finally outputs a boundary vector of the road.

Layer	Kernel ($w_1 \times h_1 \times d_1$)	Input ($w \times h \times d$)	Output ($w_2 \times h_2 \times d_1$)
Input	–	$600 \times 160 \times 5$	–
Conv1 s=2	$3 \times 3 \times 64$	$600 \times 160 \times 5$	$600 \times 160 \times 64$
Conv2	$3 \times 3 \times 64$	$600 \times 160 \times 64$	$600 \times 160 \times 64$
Conv3 s=2	$3 \times 3 \times 64$	$300 \times 80 \times 64$	$300 \times 80 \times 64$
Conv4	$3 \times 3 \times 64$	$300 \times 80 \times 64$	$300 \times 80 \times 64$
Conv5 s=2	$3 \times 3 \times 64$	$150 \times 40 \times 64$	$150 \times 40 \times 64$
Conv6	$3 \times 3 \times 64$	$150 \times 40 \times 64$	$150 \times 40 \times 64$
Conv7	$3 \times 3 \times 64$	$75 \times 20 \times 64$	$75 \times 20 \times 64$
D-LSTM 1	64	$75 \times 20 \times 64$	$75 \times 20 \times 64$
Conv8	$3 \times 3 \times 64$	$75 \times 20 \times 64$	$75 \times 20 \times 64$
D-LSTM 2	64	$75 \times 20 \times 64$	$75 \times 20 \times 64$
Conv9	$1 \times 5 \times 64$	$75 \times 20 \times 64$	$75 \times 4 \times 64$
Conv10	$1 \times 4 \times 1$	$75 \times 4 \times 64$	$75 \times 1 \times 1$
Up-sample	$8 \times 1 \times 1$	$75 \times 1 \times 1$	$600 \times 1 \times 1$
Output	–	–	$600 \times 1 \times 1$

Figure 5: Detailed Network Blocks

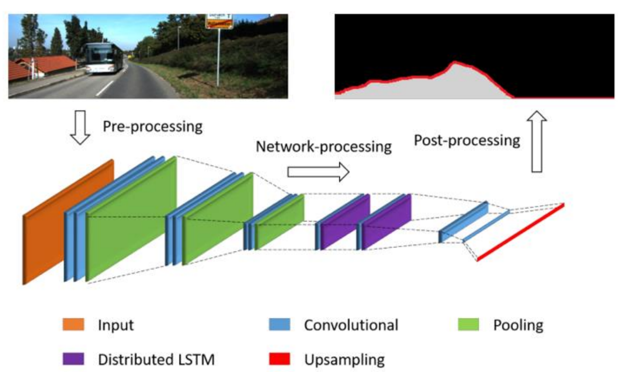


Figure 6: Proposed Network Diagram

As the paper proposed, this network should produce a result with a high accuracy rate (~89%) within a reasonably shot time. However, during real implementation, we found out that the model actually takes a long time to train.

As we take the pixel location as a road feature, the training data bias influences the final prediction a lot. Since in many images, the road appears on the bottom left corner, out network predicts that the road would appear on the left bottom more possibly. This leads to the false positive of the road prediction when the road appears on the right side of the image.

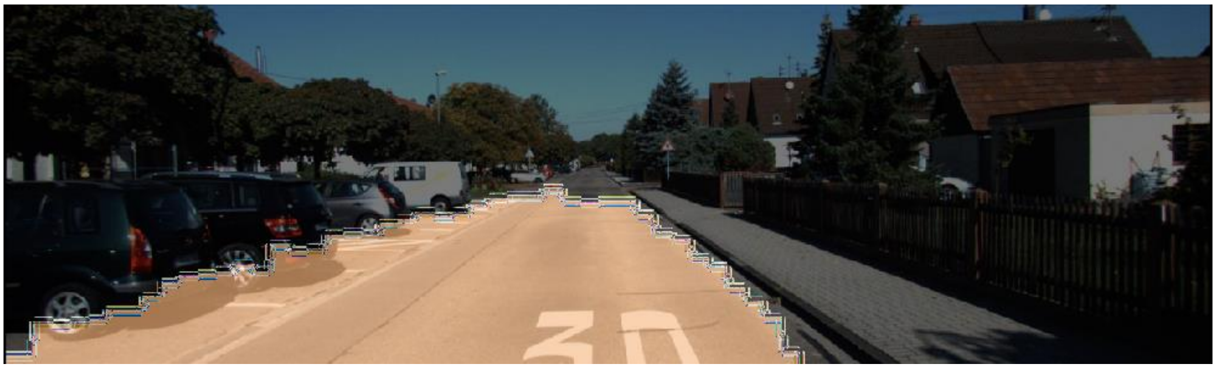


Figure 7 (top) and 8 (bottom): Bad Example of CNN-LSTM Road Classifier for

test/image_left/umm_000087.jpg

- Method 2:

We chose the U-Net convolutional network. The U-Net is designed for image segmentation. The structure contains two parts -- encoder and decoder. The encoder is used to capture image features, and the decoder is used to perform localization. For our task, we need to pass in an image as training data and classify a ground truth mask for the road. Since U-Net is an end-to-end fully convolutional network, and we need the input shape and output shape to be the same, so U-Net is a good choice.

We found the detailed structure and implementation of U-Net NN in here: https://www.depends-on-the-definition.com/unet-keras-segmenting-images/.

For the parameters of U-Net, we chose batch size to be 20, and split 80% of data to be training data set, rest of them to be validation data set. Figure 9 and Figure 10 are the loss curve and Accuracy curve during the training process. We could see that the validation accuracy is about 80%.

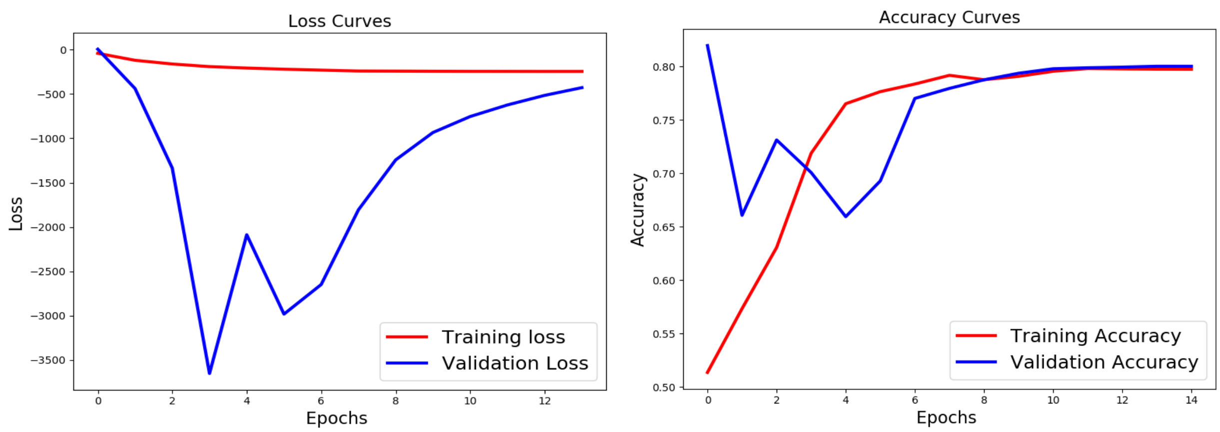


Figure 9 (left) and 10 (right): Accuracy and Loss plot for model

In Figure 13, 14, 17 and 18, we plot the ground truth mask returned from the U-Net and plot the segmentation on the image. Since the KITTI ground truth mask only covers the driving lane that the car is currently on, so most of the prediction mask gets the middle road.

- Comparision:

As the first method CNN-LSTM we tried took a very long time to train and the result is biased based on the road location. We tried the second method U-Net which is simpler and the training is faster. After comparison, we found out that although method 2 creates a smoother boundary for the road segmentation, the prediction result is unstable.

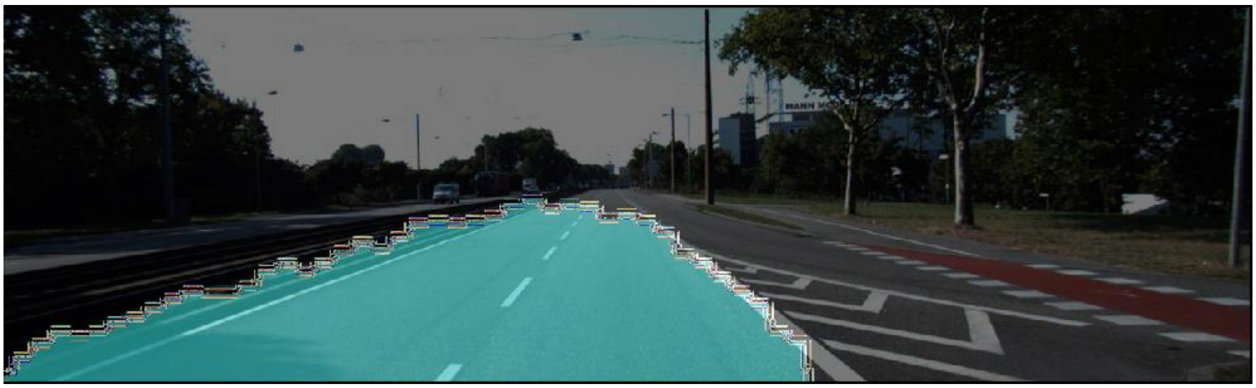


Figure 11 (top) and 12 (bottom): CNN-LSTM Road Classifier for *test/image_left/umm_000073.jpg*

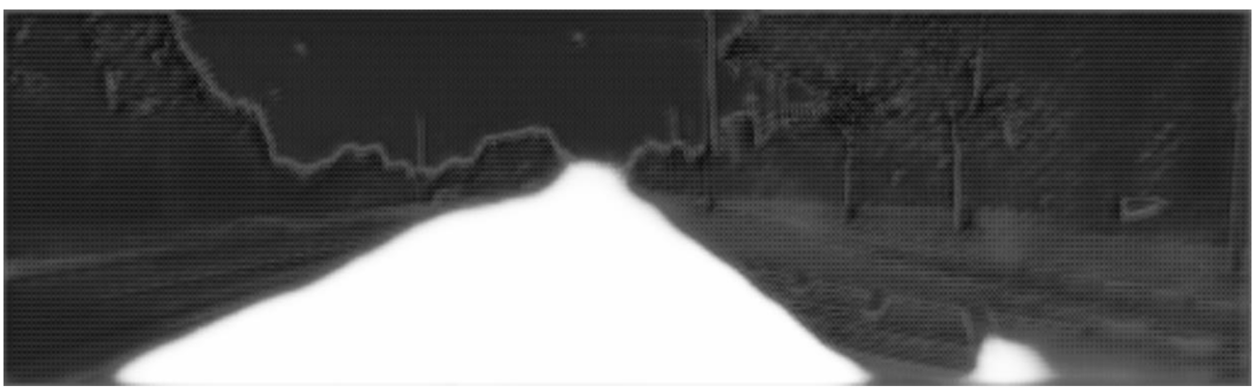


Figure 13 (top) and 14 (bottom): U-Net Road Classifier for *test/image_left/umm_000073.jpg*

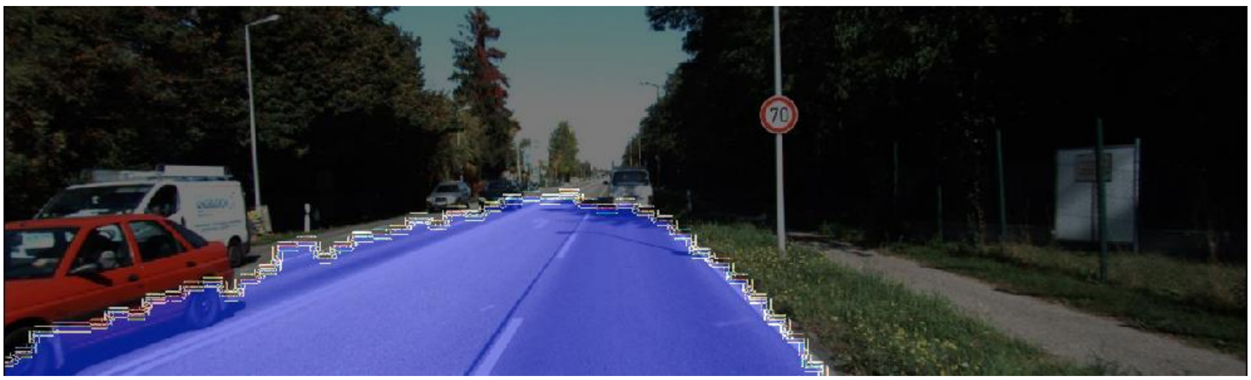


Figure 15 (top) and 16 (bottom): CNN-LSTM Road Classifier for *test/image_left/umm_000087.jpg*

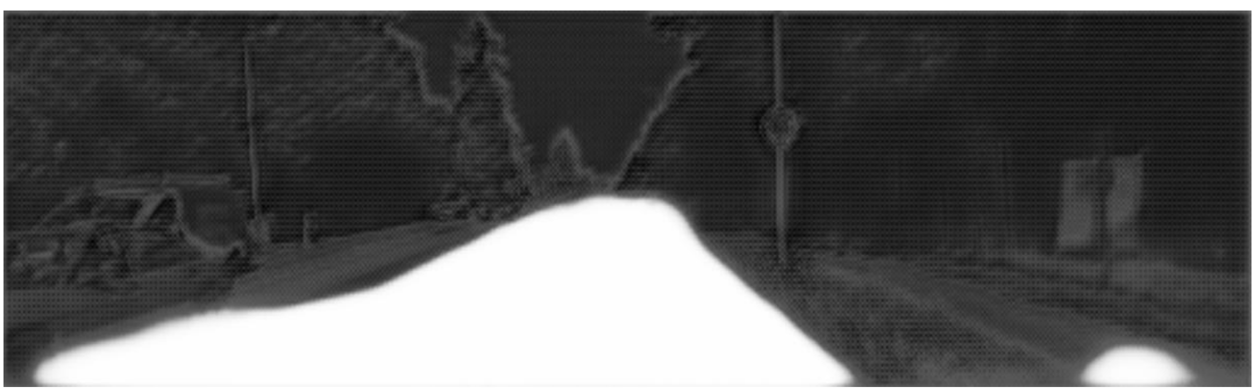


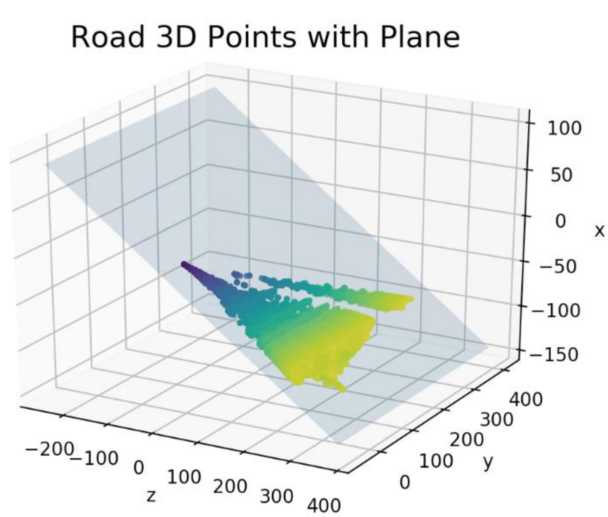
Figure 17 (top) and 18 (bottom): U-Net Road Classifier for *test/image_left/umm_000087.jpg*

Subtask 4: Fit Ground Plane (*road_visualization.py*)

This task requires depth map and ground truth mask from previous Subtask. We first computed 3D location based on depth map. We used the formula $X = \frac{\mathrm{Z}-(\mathrm{x}*\mathrm{Px})}{f}$, $Y = \frac{\mathrm{Z}-(\mathrm{y}*\mathrm{Py})}{f}$, where Z is the depth in the depth map, x and y is the image coordinate. And we found Px, Py and focal length from calibration information.

Then for all the 3D location, we used ground truth mask to filter out points that are not road. The last step is to fit a plane. Here we used a technique called Least Square to find the best fit plane. The algorithm can find a plane that minimize the least square distance to every point. In order for our algorithm to be robust to outliers, we applied RANSAC over Least Square. We had 3000 iterations, each iteration we took 5 points and got a best fit plane using Least Square. Then we computed outliers with threshold 0.02. After all the iterations, we used the plane that had the least number of outliers.

Figure 19 used matplotlib to plot all the road 3D points and the best fit plane after RANSAC. We can tell that the plane is almost touching the bottom side of points cloud from Figure 20.



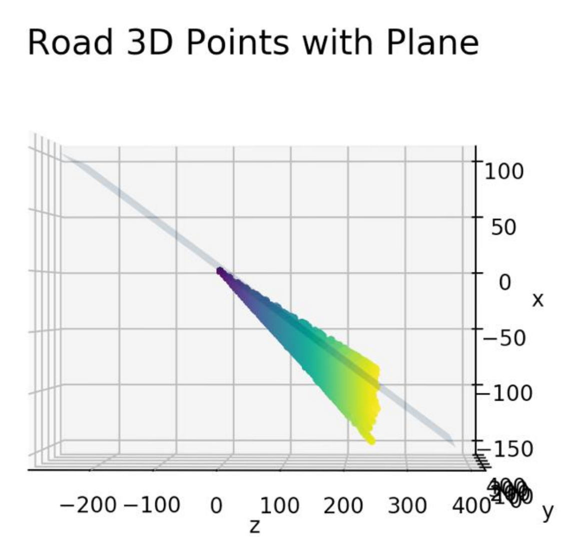


Figure 19 (left) and 20 (right): Road Plane for *test/image_left/umm_000087.jpg*

Subtask 5: 3D Point Cloud (*road_visualization.py*)

This Task requires best fit plane and 3D locations from previous Subtask. We used open3D to put the color onto 3D location points cloud, then we generated another 3D points cloud for the plane and add it to the figure.

Here is the image from *test/image_left/umm_000087.jpg* that we used to plot 3D points cloud and ground plane.

Figure 21 is captured when we rotate the model on the side. Figure 22 is the front face, and Figure 23 is viewed from top right.

Figure 21 (left) and 22 (right): 3D Point Cloud for *test/image_left/umm_000087.jpg*

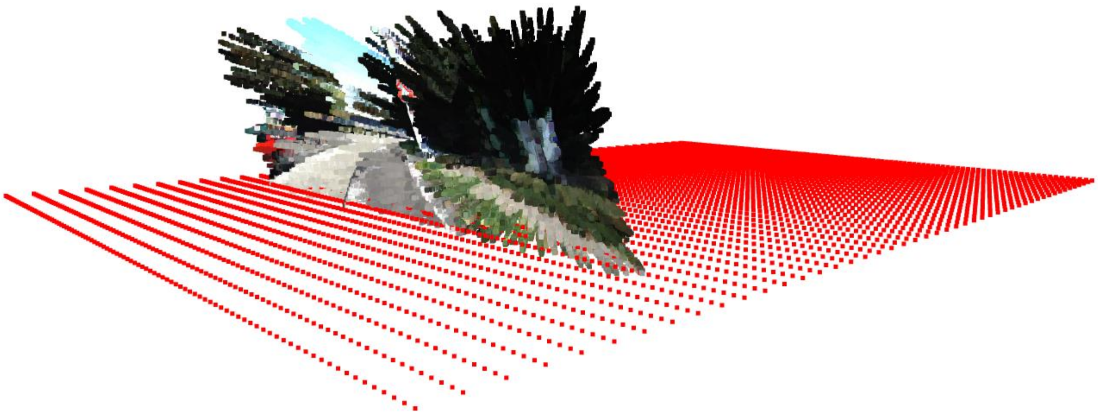


Figure 23: 3D Point Cloud with OpenCV for *test/image_left/umm_000087.jpg*

Subtask 6: Car Detection (*car_detection.py*)

At the beginning we searched a lot of pre-trained model for KITTI data set, but they have low accuracy on the test image. Finally, we used Faster RCNN from torchvision pre-trained model. It is not designed for car detection, we filtered out other objects (E.g. person). The only objects we used to detect are car, bus and truck.

The reason we chose Faster RCNN is because of its high accuracy and short computation time, compared with YOLO and SSD. YOLO and SSD have better performance for object detection in video. Here is the reference code of how to use torchvision pre-trained model: https://www.learnopencv.com/faster-r-cnn-object-detection-with-pytorch/

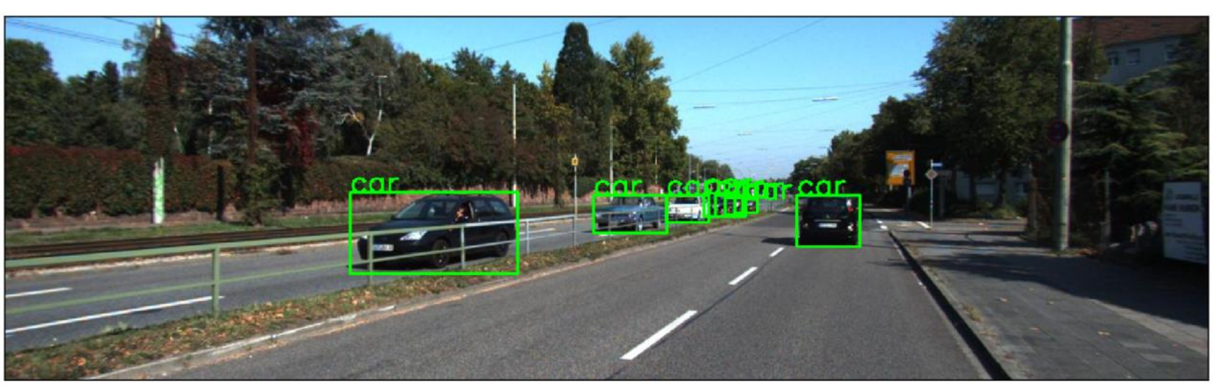


Figure 24: Car Detection for *train/image_left/um_000010.jpg*

Subtask 7: Viewpoint Classifier (*car_viewpoint.py*)

For this task, we built a simple CNN using Keras to do viewpoint classification. We used the one-hot encoder to represents the training labels. There are 12 classes, so each label has dimension 12 x 1. Figure 25 is the structure of our CNN.

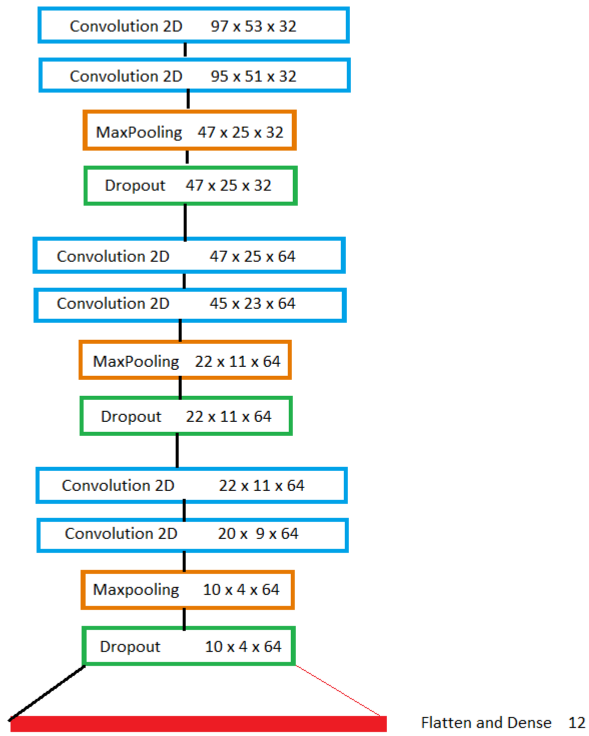


Figure 25: Structure of CNN model

Here is the reference code of simple CNN using Keras: https://www.learnopencv.com/image-classification-using-convolutional-neural-networks-in-keras/. We could see that the validation accuracy is 94%, but the training accuracy is higher than validation accuracy. The potential reason for this is that our model is overfitting.

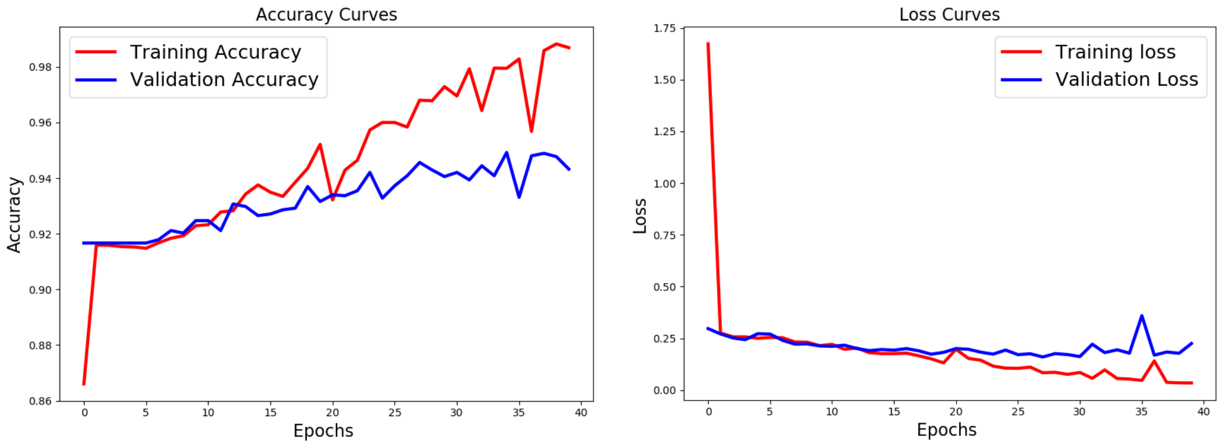


Figure 26 (left) and 27 (right): Accuracy and Loss plot for model

Subtask 8: Car and Viewpoint (*car_visualization.py*)

This task requires pre-trained viewpoint model and car detection. For each image, we first detected cars and ignore bounding box that are too small. Then we pass-in the car image patch to the pre-trained model and plot the viewpoint angle with arrows. Arrow pointing down is 0°, and arrow pointing left is 90°.

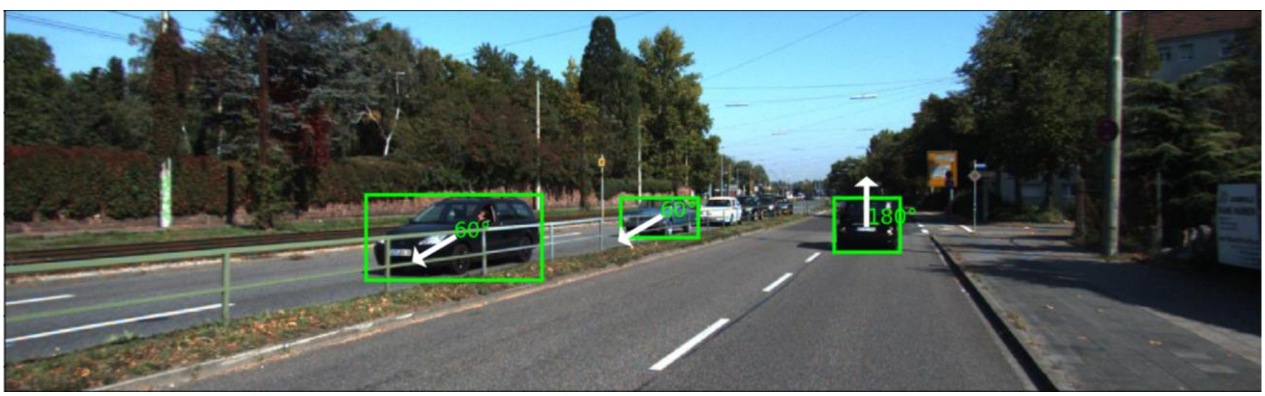


Figure 28: Car and Viewpoint for *train/image_left/um_000010.jpg*

image_processings.py

```python
import numpy as np
import cv2
from matplotlib import pyplot as plt
import skimage.io


def compute_disparity(left, right, p):
    """
    Project 2: Question 1
    Method 1

    :param left: the left image in the stereo image pairs
    :type left: numpy array
    :param right: the right image in the stereo image pair
    :type right: numpy array
    :param p: patch size
    :type p: int
    :return: disparity map of the image pair
    :rtype: numpy array of the same shape as the input images
    """
    # max disparity to check
    max_d = 16

    disp_map = np.zeros(left.shape)

    # sizes
    im_height, im_width = left.shape
    p_half = p // 2

    # zero-padding: N + p - 1 * M + p - 1 for left and right image
    padded_shape = im_height + p - 1, im_width + p - 1

    padded_im_left = np.zeros(padded_shape)
    padded_im_left[p_half:(im_height + p_half), p_half:(im_width + p_half)] = left

    padded_im_right = np.zeros(padded_shape)
    padded_im_right[p_half:(im_height + p_half), p_half:(im_width + p_half)] = right
```

```python

    # matrix of vectorized patches for right image for fast retrieval
    r_patch_vectorized = np.zeros((im_height, im_width, p * p))

    for i in range(im_height):
        for j in range(im_width):
            l_patch = padded_im_left[i: i + p, j: j + p].reshape(-1)
            r_patch = padded_im_right[i: i + p, j: j + p].reshape(-1)

            r_patch_vectorized[i, j, :] = r_patch

            # calculate absolute intensity difference
            start = max(0, j - max_d)
            abs_diff = np.abs(np.copy(r_patch_vectorized[i, start: j + 1, :]) - l_patch)
            ssd = np.sum(abs_diff, axis=1)

            # find the best match
            ssd = np.flip(ssd, 0)
            min_d = np.argmin(ssd)

            disp_map[i, j] = min_d

    cv2.normalize(disp_map, disp_map, 0, 255, cv2.NORM_MINMAX)
    return disp_map
```

```python
def compute_disparity_by_cv2(left, right, block_size):
    """
    Project 2: Question 1
    Method 2

    :param left: the left image in the stereo image pairs
    :type left: numpy array
    :param right: the right image in the stereo image pair
    :type right: numpy array
    :return: disparity map of the image pair
    :rtype: numpy array of the same shape as the input images
    """
    stereoMatcher = cv2.StereoBM_create()
    stereoMatcher.setMinDisparity(4)
    stereoMatcher.setNumDisparities(128)
    stereoMatcher.setBlockSize(block_size)
    stereoMatcher.setSpeckleRange(16)
    stereoMatcher.setSpeckleWindowSize(10)
    disp = stereoMatcher.compute(left, right)

    kernel = np.ones((5, 5), np.float32) / 25
    disp = cv2.filter2D(disp, -1, kernel)
    return disp
```

```python
def compute_depth(disp_map, f, t):
    """
    Project 2: Qestion 2

    :param disp_map: disparity map of the stereo image pair
    :type disp_map: numpy array
    :param f: focal length
    :type f: float
    :param t: baseline
    :type t: float
    :return: depth map of the image pair
    :rtype: numpy array of the same shape as the input disp_map
    """
    depth = np.divide(-f * t, disp_map, where=disp_map != 0)
    cv2.normalize(depth, depth, 0, 255, cv2.NORM_MINMAX)
    return depth
```

```python
if __name__ == "__main__":
    F = 7.215377000000e+02
    T = 0.54 * 1000
    PX, PY = 6.095593000000e+02, 1.728540000000e+02

    left_img = cv2.imread('./data/test/image_left/umm_000087.jpg', cv2.IMREAD_GRAYSCALE)
    right_img = cv2.imread('./data/test/image_right/umm_000087.jpg', cv2.IMREAD_GRAYSCALE)

    # Q1: Compute disparity
    # Method 1
    # disp = compute_disparity(left_img, right_img, 10)
    # Method 2
    disp = compute_disparity_by_cv2(left_img, right_img, 5)

    plt.imshow(disp, 'gray')
    plt.title('Disparity', fontsize=16)
    plt.show()

    # Q2: Compute depth
    depth = compute_depth(disp, F, T)
    plt.imshow(depth, 'gray')
    plt.title('Depth', fontsize=16)
    plt.show()
    skimage.io.imsave('./Output Images/depth_umm_000087.png', depth.astype('uint8'))
```

road_segmentation.py (Method 1)

```python
# ref: https://arxiv.org/pdf/1808.04450.pdf
# Method 1 for Q3
import ...

# dataloader
class KittiDataset(tu.data.Dataset):
    def __init__(self, image_path, label_path, transform):
        self.imgs, self.labels = read_files(image_path, label_path)
        self.transform = transform

    def __getitem__(self, index):
        img = self.imgs[index]
        label = self.labels[index]
        img = self.transform(img)

        dt = {'img': img,
              'gt': torch.IntTensor(label)}

        return dt

    def __len__(self):
        return len(self.imgs)

def read_files(image_path, label_path):
    # read image and pre-process them for the training data loader

    images = []
    labels = []

    for img_name in os.listdir(image_path):
        path = os.path.join(image_path, img_name)
        if "DS_Store" not in path:

            # process image with pyramid prediction scheme
            img = Image.open(path).convert('RGB')
            # crop image at center
            c_img = crop_image(img)
```

```
        # rescale image to the same size
        img = ImageOps.fit(img, (600, 160), Image.ANTIALIAS)
        # add x, y coordinate channels
        img = generate_coord_channels(img)
        c_img = generate_coord_channels(c_img)
        images.append(img)
        images.append(c_img)

        # process gt_mask with pyramid prediction scheme
        # get gt_mask name
        img_name = img_name.split('.')[0]
        img_name_l = img_name.split('_')
        img_name_l.insert(1, "road")
        gt_name = '_'.join(img_name_l)
        gt_name += '.png'

        gt_path = os.path.join(label_path, gt_name)
        gt_mask = Image.open(path).convert('L')

        c_mask = crop_image(gt_mask)
        gt_mask = ImageOps.fit(gt_mask, (600, 160), Image.ANTIALIAS)

        # generate ground truth road segmentation boundary vector
        # vector size = 600 * 1, v[i] = the row index of the boundary pixel
        gt_mask = np.array(gt_mask)
        bound = gt_mask.argmax(axis=0)
        gt_col_sum = np.sum(gt_mask, axis=0)
        bound[gt_col_sum == 0] = 160

        c_mask = np.array(c_mask)
        c_bound = c_mask.argmax(axis=0)
        c_col_sum = np.sum(c_mask, axis=0)
        c_bound[c_col_sum == 0] = 160

        labels.append(bound)
```

```python
            labels.append(c_bound)

    return images, labels

def make_coordinates_matrix(im_shape, step=1):
    """
    Return a matrix of size (im_shape[0] x im_shape[1] x 2) such that g(y,x)=[y,x]
    """
    range_x = np.arange(0, im_shape[1], step)
    range_y = np.arange(0, im_shape[0], step)
    axis_x = np.repeat(range_x[np.newaxis, ...], len(range_y), axis=0)
    axis_y = np.repeat(range_y[..., np.newaxis], len(range_x), axis=1)

    return np.dstack((axis_y, axis_x))

def generate_coord_channels(image):
    np_img = np.array(image)
    coord_m = make_coordinates_matrix(np_img.shape)
    np_img = np.concatenate((np_img, coord_m), axis=2)
    return np_img

def crop_image(image):
    center_x = image.size[0] / 2
    center_y = image.size[1] / 2
    left = center_x - 300
    top = center_y - 80
    right = center_x + 300
    bottom = center_y + 80
    c_img = image.crop((left, top, right, bottom))
    return c_img
```

```python
# training network
class CNN_LSTM(nn.Module):
    def __init__(self):
        super(CNN_LSTM, self).__init__()
        self.conv_in = nn.Conv2d(in_channels=5, out_channels=64, stride=1, kernel_size=3, padding=1)  # 2
        self.conv_en = nn.Conv2d(in_channels=64, out_channels=64, stride=1, kernel_size=3, padding=1)  # 2
        self.pool = nn.MaxPool2d(2, 2)

        self.conv_fp = nn.Conv2d(in_channels=64, out_channels=64, stride=1, kernel_size=3, padding=1)
        self.lstm = nn.LSTM(input_size=64, hidden_size=64)

        self.conv_de1 = nn.Conv2d(in_channels=64, out_channels=64, stride=(5, 1), kernel_size=(5, 1))
        self.conv_de2 = nn.Conv2d(in_channels=64, out_channels=1, stride=1, kernel_size=(4, 1))
        self.upsample = nn.Upsample(scale_factor=(1, 8), mode='nearest')

    def forward(self, image):
        # print("input shape: expect shape 5 * 160 * 600 ",image.shape)
        feature = self.conv_in(image)
        # print("conv1: expect shape 64 * 160 * 600 ",feature.shape)
        feature = self.conv_en(feature)
        # print("conv2: expect shape 64 * 160 * 600 ",feature.shape)
        feature = self.pool(feature)
        # print("pool1: expect shape 64 * 80 * 300 ",feature.shape)

        feature = self.conv_en(feature)
        # print("conv3: expect shape 64 * 80 * 300 ",feature.shape)
        feature = self.conv_en(feature)
        # print("conv4: expect shape 64 * 80 * 300 ",feature.shape)
        feature = self.pool(feature)
        # print("pool2: expect shape 64 * 40 * 150 ",feature.shape)

        feature = self.conv_en(feature)
        # print("conv5: expect shape 64 * 40 * 150 ",feature.shape)
        feature = self.conv_en(feature)
        # print("conv6: expect shape 64 * 40 * 150 ",feature.shape)
        feature = self.pool(feature)
```

```python
        output = self.conv_fp(feature)
        # print("conv7: expect shape 64 * 20 * 75 ", output.shape)
        output = output.view(output.shape[2], output.shape[3], 64)
        hidden_state = torch.randn(1, output.shape[1], 64)
        cell_state = torch.randn(1, output.shape[1], 64)
        output = self.lstm(output, (hidden_state, cell_state))[0]
        output = output.view(1, 64, output.shape[0], output.shape[1])
        # print("lstm1: expect shape 64 * 20 * 75 ", output.shape)

        output = self.conv_fp(output)
        # print("conv8: expect shape 64 * 20 * 75 ", output.shape)
        output = output.view(output.shape[2], output.shape[3], 64)
        hidden_state = torch.randn(1, output.shape[1], 64)
        cell_state = torch.randn(1, output.shape[1], 64)
        output = self.lstm(output, (hidden_state, cell_state))[0]
        output = output.view(1, 64, output.shape[0], output.shape[1])
        # print("lstm2: expect shape 64 * 20 * 75 ", output.shape)

        output = self.conv_de1(output)
        # print("conv9: expect shape 64 * 4 * 75 ", output.shape)
        output = self.conv_de2(output)
        # print("conv10: expect shape 1 * 1 * 75 ", output.shape)
        output = self.upsample(output)
        # print("final output: expect shape 1 * 1 * 600 ", output.shape)
        return output
```

```python
def train_road_classifier(md, optimizer, trainloader, criterion):
    num_epochs = 80
    total_step = 270
    losses = list()

    for epoch in range(1, num_epochs + 1):
        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            # zero the gradients
            md.zero_grad()
            optimizer.zero_grad()

            # set model into train mode
            md.train()

            images = data['img']
            gt_boundaries = data['gt']

            # Pass the inputs through the CNN-RNN model.
            outputs = md(images.float())

            # Calculate the batch loss
            loss = criterion(outputs[0, 0, :, :], gt_boundaries.float())

            # Backward pass
            loss.backward()

            # Update the parameters in the optimizer
            optimizer.step()

            losses.append(loss.item())
            running_loss += loss.item()

            # save the losses
            np.save('losses', np.array(losses))
```

```python
            # Get training statistics.
            stats = 'Epoch [%d/%d], Step [%d/%d], Loss: %.4f' % (
                epoch, num_epochs, i, total_step, loss.item())

            # Print training statistics (on same line).
            print('\r' + stats, end="")
            sys.stdout.flush()

    # Save the weights.
    print("\nSaving the model")
    torch.save(md.state_dict(), os.path.join('./data/train/results-dets', 'md.pth'))

# testing visualization
def visualize_segmentation(boundary):
    # visualize the segmentation mask for the image given the boudnary vector
    image = Image.new("RGB", (600, 160))
    draw = ImageDraw.Draw(image)
    points = generate_boundary_coord(boundary)
    points = np.vstack((points, points[0, :]))
    points = tuple(map(tuple, points))
    draw.polygon((points), fill=(255, 255, 255))
    # image.show()
    return image

def generate_boundary_coord(y, step=1):
    # generate the road boundary xy coordinate based on the boundary row index vector
    im_shape = (1, y.shape[1])

    range_x = np.arange(0, im_shape[1], step)
    range_y = np.arange(0, im_shape[0], step)
    axis_x = np.repeat(range_x[np.newaxis, ...], len(range_y), axis=0)
    result = np.dstack((axis_x, y)).reshape((im_shape[1], 2))

    return result
```

```python
def reassemble(scaled_img, cropped_img, img_size):
    # reassemble the overlayed scaled and cropped images
    center_x = img_size[1] // 2
    center_y = img_size[0] // 2
    # scale the image back to its original size
    scaled_img = ImageOps.fit(scaled_img, img_size, Image.ANTIALIAS)
    # replaced the pixel values with the cropped image at the center
    scaled_img.paste(cropped_img, (center_x, center_y))
    # scaled_img.show()
    return scaled_img
```

```python
def overlay_mask(image, gt_mask):
    gt_mask = gt_mask.astype(int)
    # code from tutorial7 to visualize segmentation
    obj_ids = np.unique(gt_mask)
    number_object = obj_ids.shape[0]


    count = 0
    for o_id in obj_ids:
        gt_mask[gt_mask == o_id] = count
        count += 1

    base_COLORS = []

    for key, value in mcolors.CSS4_COLORS.items():
        rgb = webcolors.hex_to_rgb(value)
        base_COLORS.append([rgb.blue, rgb.green, rgb.red])
    base_COLORS = np.array(base_COLORS)

    np.random.seed(99)
    base_COLORS = np.random.permutation(base_COLORS)

    colour_id = np.array([(id) % len(base_COLORS) for id in range(number_object)])
    COLORS = base_COLORS[colour_id]
    COLORS = np.vstack([[0, 0, 0], COLORS]).astype("uint8")
    mask = COLORS[gt_mask]
    output = ((0.4 * image) + (0.6 * mask)).astype("uint8")
    plt.imshow(output[:, :, ::-1])
    plt.show()
```

```python
if __name__ == "__main__":
    # generate dataset loader
    transform = transforms.Compose([transforms.ToTensor()])

    # ===== comment out this part if you just want to see the test result ======
    kd = KittiDataset(image_path='./data/train/image_left', label_path='./data/train/gt_image_left', transform=transform)

    # training
    model = CNN_LSTM().float()
    optimizer = optim.Adam(model.parameters(), lr=1e-5)
    criterion = nn.L1Loss()
    trainloader = tu.data.DataLoader(kd, batch_size=1, shuffle=False, num_workers=2)
    train_road_classifier(model, optimizer, trainloader, criterion)
    # ==========================================================================

    # testing
    md = CNN_LSTM().float()
    cmd = CNN_LSTM().float()

    md.load_state_dict(torch.load(os.path.join('./data/train/results-dets', 'md.pth')))
    cmd.load_state_dict(torch.load(os.path.join('./data/train/results-dets', 'md.pth')))

    image = Image.open('./data/test/image_left/uu_000041.jpg').convert('RGB')
    img_s = image.size

    # crop image at center with window size 600 * 160
    c_img = crop_image(image)
    c_img = generate_coord_channels(c_img)

    # scale original image to 600 * 160
    image = ImageOps.fit(image, (600, 160), Image.ANTIALIAS)
    image = generate_coord_channels(image)
```

```python
    # turn image into torch tensor for prediction
    image = transform(image).view(1, 5, 160, 600)
    c_img = transform(c_img).view(1, 5, 160, 600)

    # predict
    outputs = md(image.float())
    outputs = outputs.detach().numpy().reshape((1, 600))

    coutputs = cmd(c_img.float())
    coutputs = coutputs.detach().numpy().reshape((1, 600))

    sv = visualize_segmentation(outputs)
    cv = visualize_segmentation(coutputs)

    final = np.array(reassemble(sv, cv, img_s))[:, :, 0]
    overlay_mask(cv2.imread('./data/test/image_left/umm_000087.jpg'), final)
```

road_detection.py   (Method 2)

```python
# ref: https://www.depends-on-the-definition.com/unet-keras-segmenting-images/

import os
import cv2
import numpy as np
from keras import Model
from keras.callbacks import EarlyStopping, ReduceLROnPlateau, ModelCheckpoint
from keras.engine.saving import load_model
from keras.layers import Conv2D, BatchNormalization, Activation, MaxPooling2D, Dropout, Conv2DTranspose, concatenate
from skimage import io
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
import webcolors

im_width = 512
im_height = 128
split = 0.8

callbacks = [
    EarlyStopping(patience=10, verbose=1),
    ReduceLROnPlateau(factor=0.1, patience=3, min_lr=0.00001, verbose=1),
    ModelCheckpoint('road_model_checkpoint.h5', verbose=1, save_best_only=True, save_weights_only=True)
]
```

```python
def conv2d_block(input_tensor, n_filters, kernel_size=3, batchnorm=True):
    # first layer
    x = Conv2D(filters=n_filters, kernel_size=(kernel_size, kernel_size), kernel_initializer="he_normal",
               padding="same")(input_tensor)
    if batchnorm:
        x = BatchNormalization()(x)
    x = Activation("relu")(x)

    # second layer
    x = Conv2D(filters=n_filters, kernel_size=(kernel_size, kernel_size), kernel_initializer="he_normal",
               padding="same")(x)
    if batchnorm:
        x = BatchNormalization()(x)
    x = Activation("relu")(x)
    return x
```

```python
def get_unet(input_img, n_filters=16, dropout=0.5, batchnorm=True):
    # contracting path -- encoder
    c1 = conv2d_block(input_img, n_filters=n_filters * 1, kernel_size=3, batchnorm=batchnorm)
    p1 = MaxPooling2D((2, 2))(c1)
    p1 = Dropout(dropout * 0.5)(p1)

    c2 = conv2d_block(p1, n_filters=n_filters * 2, kernel_size=3, batchnorm=batchnorm)
    p2 = MaxPooling2D((2, 2))(c2)
    p2 = Dropout(dropout)(p2)

    c3 = conv2d_block(p2, n_filters=n_filters * 4, kernel_size=3, batchnorm=batchnorm)
    p3 = MaxPooling2D((2, 2))(c3)
    p3 = Dropout(dropout)(p3)

    c4 = conv2d_block(p3, n_filters=n_filters * 8, kernel_size=3, batchnorm=batchnorm)
    p4 = MaxPooling2D(pool_size=(2, 2))(c4)
    p4 = Dropout(dropout)(p4)

    c5 = conv2d_block(p4, n_filters=n_filters * 16, kernel_size=3, batchnorm=batchnorm)

    # expansive path -- decoder
    u6 = Conv2DTranspose(n_filters * 8, (3, 3), strides=(2, 2), padding='same')(c5)
    u6 = concatenate([u6, c4])
    u6 = Dropout(dropout)(u6)
    c6 = conv2d_block(u6, n_filters=n_filters * 8, kernel_size=3, batchnorm=batchnorm)

    u7 = Conv2DTranspose(n_filters * 4, (3, 3), strides=(2, 2), padding='same')(c6)
    u7 = concatenate([u7, c3])
    u7 = Dropout(dropout)(u7)
    c7 = conv2d_block(u7, n_filters=n_filters * 4, kernel_size=3, batchnorm=batchnorm)

    u8 = Conv2DTranspose(n_filters * 2, (3, 3), strides=(2, 2), padding='same')(c7)
    u8 = concatenate([u8, c2])
    u8 = Dropout(dropout)(u8)
    c8 = conv2d_block(u8, n_filters=n_filters * 2, kernel_size=3, batchnorm=batchnorm)

    u9 = Conv2DTranspose(n_filters * 1, (3, 3), strides=(2, 2), padding='same')(c8)
    u9 = concatenate([u9, c1], axis=3)
    u9 = Dropout(dropout)(u9)
    c9 = conv2d_block(u9, n_filters=n_filters * 1, kernel_size=3, batchnorm=batchnorm)

    outputs = Conv2D(1, (1, 1), activation='sigmoid')(c9)
    model = Model(inputs=[input_img], outputs=[outputs])
    return model
```

```python
def overlay_mask(image, mask):
    # pre-process mask
    mask[np.where(mask < 0.5)] = 0  # not road pixel
    mask[np.where(mask >= 0.5)] = 1 # road pixel
    mask = mask.astype(int)

    # code from tutorial7 to visualize segmentation
    obj_ids = np.unique(mask)
    number_object = obj_ids.shape[0]

    count = 0
    for o_id in obj_ids:
        mask[mask == o_id] = count
        count += 1

    base_COLORS = []

    for key, value in mcolors.CSS4_COLORS.items():
        rgb = webcolors.hex_to_rgb(value)
        base_COLORS.append([rgb.blue, rgb.green, rgb.red])
    base_COLORS = np.array(base_COLORS)

    np.random.seed(99)
    base_COLORS = np.random.permutation(base_COLORS)

    colour_id = np.array([(id) % len(base_COLORS) for id in range(number_object)])
    COLORS = base_COLORS[colour_id]
    COLORS = np.vstack([[0, 0, 0], COLORS]).astype("uint8")
    mask = COLORS[mask]
    output = ((0.4 * image) + (0.6 * mask)).astype("uint8")
    io.imshow(output)
    io.show()
```

```python
def visualize_performance(history):
    # Loss Curves
    plt.figure(figsize=[8, 6])
    plt.plot(history.history['loss'], 'r', linewidth=3.0)
    plt.plot(history.history['val_loss'], 'b', linewidth=3.0)
    plt.legend(['Training loss', 'Validation Loss'], fontsize=18)
    plt.xlabel('Epochs ', fontsize=16)
    plt.ylabel('Loss', fontsize=16)
    plt.title('Loss Curves', fontsize=16)
    plt.show()

    # Accuracy Curves
    plt.figure(figsize=[8, 6])
    plt.plot(history.history['accuracy'], 'r', linewidth=3.0)
    plt.plot(history.history['val_accuracy'], 'b', linewidth=3.0)
    plt.legend(['Training Accuracy', 'Validation Accuracy'], fontsize=18)
    plt.xlabel('Epochs ', fontsize=16)
    plt.ylabel('Accuracy', fontsize=16)
    plt.title('Accuracy Curves', fontsize=16)
    plt.show()
```

```python
def visualize_predict(model_name):
    model = load_model(model_name)

    images = []
    images_path = []
    images_shape = []
    for path in os.listdir('data/test/image_left'):
        img_full_path = 'data/test/image_left' + '/' + path
        if "DS_Store" not in img_full_path:
            img = io.imread(img_full_path, as_gray=True)
            images_shape.append(img.shape)
            img = cv2.resize(img, dsize=(im_width, im_height), interpolation=cv2.INTER_CUBIC)
            img = img.reshape((im_height, im_width, 1))
            images.append(img)
            images_path.append(img_full_path)

    prediction = model.predict(np.array(images), verbose=1)

    for i in range(len(images)):
        pred = prediction[i][:, :, 0]
        pred = cv2.resize(pred, dsize=(images_shape[i][1], images_shape[i][0]), interpolation=cv2.INTER_CUBIC)
        io.imshow(pred)
        io.show()
        img = io.imread(images_path[i])
        overlay_mask(img, pred)
```

```python
def load_data():
    # Read images
    images = []
    masks = []
    for path in os.listdir('data/train/image_left'):
        img_full_path = 'data/train/image_left' + '/' + path
        if "DS_Store" not in img_full_path:
            img = io.imread(img_full_path, as_gray=True)
            img = cv2.resize(img, dsize=(im_width, im_height), interpolation=cv2.INTER_CUBIC)
            img = img.reshape((im_height, im_width, 1))
            images.append(img)

            num = path.split('.')[0].split('_')[1]
            mask_full_path = 'data/train/gt_image_left' + '/um_road_' + num + '.png'
            mask = io.imread(mask_full_path, as_gray=True)
            mask = cv2.resize(mask, dsize=(im_width, im_height), interpolation=cv2.INTER_CUBIC)
            mask = mask.reshape((im_height, im_width, 1))
            masks.append(mask)

    divisor = int(len(images) * split)

    train_data, train_label = np.array(images[:divisor]), np.array(masks[:divisor])
    test_data, test_label = np.array(images[divisor:]), np.array(masks[divisor:])

    return train_data, train_label, test_data, test_label
```

```python
if __name__ == '__main__':
    input_img = Input((im_height, im_width, 1), name='img')
    model = get_unet(input_img, n_filters=16, dropout=0.05, batchnorm=True)

    model.compile(optimizer=Adam(), loss="binary_crossentropy", metrics=["accuracy"])

    train_data, train_label, test_data, test_label = load_data()
    results = model.fit(train_data, train_label, batch_size=15, epochs=40, callbacks=callbacks,
                        validation_data=(test_data, test_label))
    model.save("road_model_final.h5")

    # Visualize model performance
    visualize_performance(results)

    # Visualize predict
    visualize_predict('road_model_final.h5')
```

road_visualization.py

```python
import numpy as np
from matplotlib import pyplot as plt
from numpy.linalg import lstsq
import skimage.io
import open3d as o3d
from mpl_toolkits import mplot3d


def compute_3d_location(depth, px, py, f):
    """
    Project 2: Qestion 4 & 5
    Return a matrix of size (h x w x 3) such that m[y, x] = [X, Y, Z]

    :param depth: depth map of the stereo image pair
    :type depth: numpy array
    :param px: principal point x of camera
    :type px: float
    :param py: principal point y of camera
    :type py: float
    :param f: focal length
    :type f: float
    :return: matrix of the XYZ coordinates in world coordinate system for each pixel
    :rtype: numpy array
    """

    xy_coord = make_coordinates_matrix(depth.shape)
    z_coord = depth.reshape((depth.shape[0], depth.shape[1], 1))

    # calculate X Y coordinates: Z * (x - px) / f
    #                            Z * (y - py) / f
    xy_coord = (xy_coord - [px, py]) / f
    xy_coord = np.multiply(xy_coord, np.concatenate((z_coord, z_coord), axis=2))

    location = np.concatenate((xy_coord, z_coord), axis=2)
    return location
```

```python
def compute_distance(points, plane):
    """
    Helper for RANSAC
    return a 1d array of size N (number of points) containing the distance of a point to the given plane
    """
    # convert 3D point to Homogenous coordinate
    points = np.concatenate((points, np.ones((points.shape[0], 1))), axis=1)
    # calculate distance  = |Ax + By + Cz + d| / sqrt(A^2 + B^2 + C^2)
    dists = np.abs(np.dot(points, plane)) / np.sqrt(np.sum(plane[:3] ** 2))
    return dists
```

```python
def RANSAC(locations, iters=3000, thresh=0.02):
    """
    Project 2: Qestion 4
    RANSAC to estimate a plane roboust to outliers
    :param locations: N x 3 3D points
    :type locations: numpy array
    :param iters: number of iterations for RANSAC
    :type iters: int
    :param inlier_thresh: point-to-plane distance threshold
    :type inlier_thresh: float
    :return: 4D plane vector and a list of inlier indices in the points
    :rtype:
    """
    max_inlier_num = -1
    best_inlier_idxs = None

    # number of points
    N = locations.shape[0]

    for i in range(iters):
        # randomly generate 5 points to fit plane
        random_idx = np.random.choice(N, 5, replace=False)
        random_points = locations[random_idx, :]

        # fit plane
        curr_plane = fit_plane(random_points)

        # calculate distance and find inliers
        dists = compute_distance(locations, curr_plane)
        curr_inlier_idxs = np.where(dists < thresh)[0]
        inlier_num = curr_inlier_idxs.shape[0]
```

```python
        # update best match
        if inlier_num > max_inlier_num:
            max_inlier_num = inlier_num
            best_inlier_idxs = curr_inlier_idxs

    # using the bset match we find so far to fit the final plane
    best_points = locations[best_inlier_idxs, :]
    final_plane = fit_plane(best_points)

    # final filter for inliers
    dists = compute_distance(locations, final_plane)
    inlier_list = np.where(dists < thresh)[0]

    # [a,b,c,d] -> [a,b,-1,d]
    final_plane = final_plane / -final_plane[2]
    return final_plane, inlier_list
```

```python
def make_coordinates_matrix(im_shape):
    """
    Helper for compute_3D_location
    Return a matrix of size (im_shape[0] x im_shape[1] x 2) such that m[y, x] = [y, x]
    :param im_shape: (y, x, channel)
    :type im_shape: tuple
    :return: xy coordinate matrix of the image
    :rtype: numpy array
    """
    range_x = np.arange(0, im_shape[1], 1)
    range_y = np.arange(0, im_shape[0], 1)
    axis_x = np.repeat(range_x[np.newaxis, ...], len(range_y), axis=0)
    axis_y = np.repeat(range_y[..., np.newaxis], len(range_x), axis=1)

    return np.dstack((axis_y, axis_x))

def fit_plane(road_loc):
    """
    Project 2: Qestion 4
    Fit a plane based on the 3D road point cloud
    :param road_loc: a matrix of size (num_road_pixel x 3)
    :type road_loc: numpy array
    :return:
    :rtype:
    """
    # use LSE (least square estimation) to find the best fitting
    # num_road_pixel x 3 where A[i] = [Xi, Yi, 1]
    A = np.c_[road_loc[:, 0], road_loc[:, 1], np.ones(road_loc.shape[0])]
    # run lse to get coefficients
    C, _, _, _ = lstsq(A, road_loc[:, 2])
    plane = (C[0], C[1], -1, C[2])
    nn = np.linalg.norm(plane)
    plane = plane / nn

    return plane
```

```python
def filter_pixel(location, road_mask):
    """
    Helper function
    Filter pixel locations and return only the road pixels
    """
    # pre-processing mask for pixel classification
    road_mask[np.where(road_mask < 0.5)] = 0
    road_mask[np.where(road_mask >= 0.5)] = 1
    road_mask = road_mask.astype(int)
    # filter location points and save only road pixels
    # num_road_pixel x 3
    road_loc = np.copy(location[np.where(road_mask == 1)])

    return road_loc

def visulaize_plane(X, Y, Z, road_loc):
    """
    Visualize the estimated plane with a road point cloud
    """
    road_fig = plt.figure()
    road_ax = plt.axes(projection='3d')
    road_ax.plot_surface(Z, Y, X, rstride=1, cstride=1, alpha=0.2)
    road_ax.scatter(road_loc[:,2], road_loc[:,1], road_loc[:,0], cmap='viridis', marker='.', c=road_loc[:,2])

    road_ax.set_xlabel('z')
    road_ax.set_ylabel('y')
    road_ax.set_zlabel('x')
    plt.title('Road 3D Points with Plane', fontsize=16)
    plt.show()
```

```python
def visualize_all_with_plane(X, Y, Z, location, image):
    """
    Project 2: Qestion 5
    Visualize all the pixel 3D locations and the estimated plane
    """
    XX = X.flatten().reshape((-1,1))
    YY = Y.flatten().reshape((-1,1))
    ZZ = Z.flatten().reshape((-1,1))
    road_plane_coord = np.concatenate((XX, YY, ZZ), axis=1)

    # create point cloud for road plane
    road_pcd = o3d.geometry.PointCloud()
    road_pcd.points = o3d.utility.Vector3dVector(road_plane_coord)
    road_pcd.colors = o3d.utility.Vector3dVector([[1, 0, 0] for i in range(XX.shape[0])])

    # create point cloud for all image pixels
    pcd = o3d.geometry.PointCloud()
    pcd.points = o3d.utility.Vector3dVector(location.reshape((-1,3)))
    pcd.colors = o3d.utility.Vector3dVector(image.reshape((-1,3))/255)

    o3d.visualization.draw_geometries([road_pcd, pcd])
```

```python
def compute_XYZ(points, plane):
    """
    Helper for visualization
    """
    # find boundary
    maxx = np.max(points[:, 0])
    minx = np.min(points[:, 0])
    maxy = np.max(points[:, 1])
    miny = np.min(points[:, 1])
    # create XY meshgrid
    X, Y = np.meshgrid(np.arange(minx - 50, maxx + 100, 5), np.arange(miny - 50, maxy + 100, 5))
    # evaluate Z on XY meshgrid
    Z = plane[0] * X + plane[1] * Y + plane[3]
    return X,Y,Z

if __name__ == "__main__":
    F = 7.215377000000e+02
    PX, PY = 6.095593000000e+02, 1.728540000000e+02

    depth = skimage.io.imread('./Output Images/depth_umm_000087.png', as_gray=True)
    # Q4: Fit plane
    loc = compute_3d_location(depth, PX, PY, F)
    mask = skimage.io.imread('./Output Images/road_segmentation_mask_umm_000087.png', as_gray=True) / 255
    road_loc = filter_pixel(loc, mask)
    plane, inl = RANSAC(road_loc)

    # Q5: Visualize plane
    left_img = skimage.io.imread('./data/test/image_left/umm_000087.jpg')
    inl = road_loc[inl]
    X, Y, Z = compute_XYZ(inl, plane)

    visualize_all_with_plane(X, Y, Z, loc, left_img)
    visulaize_plane(X, Y, Z, road_loc)
```

car_detection.py

```python
# ref: https://www.learnopencv.com/faster-r-cnn-object-detection-with-pytorch/

import cv2
import torchvision
from PIL import Image
from torchvision import transforms as T
import matplotlib.pyplot as plt

model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True)
model.eval()

COCO_INSTANCE_CATEGORY_NAMES = [
    '__background__', 'person', 'bicycle', 'car', 'motorcycle', 'airplane', 'bus',
    'train', 'truck', 'boat', 'traffic light', 'fire hydrant', 'N/A', 'stop sign',
    'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse', 'sheep', 'cow',
    'elephant', 'bear', 'zebra', 'giraffe', 'N/A', 'backpack', 'umbrella', 'N/A', 'N/A',
    'handbag', 'tie', 'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball',
    'kite', 'baseball bat', 'baseball glove', 'skateboard', 'surfboard', 'tennis racket',
    'bottle', 'N/A', 'wine glass', 'cup', 'fork', 'knife', 'spoon', 'bowl',
    'banana', 'apple', 'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog', 'pizza',
    'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed', 'N/A', 'dining table',
    'N/A', 'N/A', 'toilet', 'N/A', 'tv', 'laptop', 'mouse', 'remote', 'keyboard', 'cell phone',
    'microwave', 'oven', 'toaster', 'sink', 'refrigerator', 'N/A', 'book',
    'clock', 'vase', 'scissors', 'teddy bear', 'hair drier', 'toothbrush'
]

VIHICLE_LIST = ['car', 'bus', 'truck']
```

```python
def get_prediction(img_path, threshold):
    img = Image.open(img_path)

    # Defing PyTorch transform and apply the transform to the image
    transform = T.Compose([T.ToTensor()])
    img = transform(img)

    # Pass the image to the model
    pred = model([img])

    # Get the lables, bouding boxes and prediction score
    pred_class = [COCO_INSTANCE_CATEGORY_NAMES[i] for i in list(pred[0]['labels'].numpy())]
    pred_boxes = [[(i[0], i[1]), (i[2], i[3])] for i in list(pred[0]['boxes'].detach().numpy())]
    pred_score = list(pred[0]['scores'].detach().numpy())

    # Get list of index with score greater than threshold
    pred_t = [pred_score.index(x) for x in pred_score if x > threshold][-1]
    pred_boxes = pred_boxes[:pred_t + 1]
    pred_class = pred_class[:pred_t + 1]

    return pred_boxes, pred_class
```

```python
def object_detection_api(img_path, threshold=0.5, rect_th=2, text_size=1, text_th=2):
    # Get predictions
    boxes, pred_cls = get_prediction(img_path, threshold)

    img = cv2.imread(img_path)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

    # Store only vehicle bounding boxes
    vehicle_box = []

    # Visualize bounding boxes and lables only for vehicles
    for i in range(len(boxes)):
        if pred_cls[i] in VIHICLE_LIST:
            vehicle_box.append(boxes[i])
            cv2.rectangle(img, boxes[i][0], boxes[i][1], color=(0, 255, 0), thickness=rect_th)
            cv2.putText(img, pred_cls[i], boxes[i][0], cv2.FONT_HERSHEY_SIMPLEX, text_size, (0, 255, 0),
                        thickness=text_th)

    # display the output image
    plt.figure(figsize=(20, 30))
    plt.imshow(img)
    plt.xticks([])
    plt.yticks([])
    plt.show()

    return vehicle_box


if __name__ == '__main__':
    vb = object_detection_api('./data/train/image_left/um_000011.jpg')
```

car_viewpoint.py

```python
# ref: https://www.learnopencv.com/image-classification-using-convolutional-neural-networks-in-keras/

import os
import cv2
import numpy as np
from skimage import io
import scipy.io as sio
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense, Conv2D, MaxPooling2D, Dropout, Flatten, BatchNormalization, Activation, GaussianNoise

BATCH_SIZE = 256
EPOCHS = 40
VALIDATION_SPLIT = 0.8
NUM_CLASSES = 12


def read_files(image_path, label_path):
    # Read images and labels
    images = []
    annotations = []
    for path in os.listdir(image_path):
        img_full_path = image_path + '/' + path
        if "DS_Store" not in img_full_path:
            img = io.imread(img_full_path)
            images.append(img)

            num = path.split('.')[0]
            label_full_path = label_path + '/' + num + '.mat'
            anno = sio.loadmat(label_full_path)['annotation'][0][0]
            annotations.append(anno)

    return images, annotations
```

```python
def get_labels(orient):
    # Use one hot encoder to represent labels
    divisor = int(orient // 30)
    label = [1 if i == divisor else 0 for i in range(12)]
    return np.array(label)


def filter_cars(annotations, img):
    classes = annotations[0][0]
    bboxes = annotations[3]
    orient = annotations[7]
    truncated = annotations[4][0]
    occluded = annotations[8][0]

    # Check if each car is good for training, and return bounding boxes and orientations
    patches, orients = [], []
    height, width = [], []
    for i in range(len(classes)):
        cla = classes[i][0]
        trunc = truncated[i][0]
        occ = occluded[i][0]
        if cla is not 'DontCare' and trunc <= 0.3 and occ <= 2:
            left, top = int(bboxes[i][0]), int(bboxes[i][1])
            right, bottom = int(left + bboxes[i][2]), int(top + bboxes[i][3])
            patch = img[top:bottom, left:right]
            patches.append(patch)
            orients.append(get_labels(orient[i][0]))
            width.append(bboxes[i][2])
            height.append(bboxes[i][3])

    return height, width, patches, orients
```

```python
def load_data(image_path, label_path, split):
    # Read in images and labels
    images, annotations = read_files(image_path, label_path)

    # Store patch of car and its viewpoint
    patches_list, labels_list = [], []
    heights, widths = [], []
    for i in range(len(images)):
        height, width, patches, orients = filter_cars(annotations[i], images[i])
        patches_list.extend(patches)
        labels_list.extend(orients)
        heights.extend(height)
        widths.extend(width)

    # Calculate average patch width and height
    avg_height = int(sum(heights) / len(heights))
    avg_width = int(sum(widths) / len(widths))

    # Resize patches to have same size
    patches_list = [cv2.resize(patch, dsize=(avg_height, avg_width), interpolation=cv2.INTER_CUBIC) for patch in
                    patches_list]

    # Divide the data into train and test sets with split
    divider = int(len(patches_list) * split)
    train_data, train_labels = np.array(patches_list[:divider]), np.array(labels_list[:divider])
    test_data, test_labels = np.array(patches_list[divider:]), np.array(labels_list[divider:])

    return avg_height, avg_width, train_data, train_labels, test_data, test_labels
```

```python
def createModel(input_shape, nclasses):
    model = Sequential()
    model.add(Conv2D(32, (3, 3), padding='same', activation='relu', input_shape=input_shape))
    model.add(Conv2D(32, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))

    model.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))

    model.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))

    model.add(Flatten())
    model.add(Dense(512, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(nclasses, activation='softmax'))
    return model
```

```python
def train_model(image_path, label_path):
    # Load the angle data and split it to train and test sets
    avg_height, avg_width, train_data, train_labels, test_data, test_labels = load_data(image_path, label_path,
                                                                                         VALIDATION_SPLIT)
    # Create a model
    input_shape = (avg_width, avg_height, 3)
    model = createModel(input_shape, NUM_CLASSES)

    # Compile and fit the model, evaluate the model with test data
    model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])
    history = model.fit(train_data, train_labels, batch_size=BATCH_SIZE, epochs=EPOCHS, verbose=1,
                        validation_data=(test_data, test_labels))
    model.evaluate(test_data, test_labels)

    # Save model
    model.save("model.h5")

    return history
```

```
def visualize_performance(history):
    # Loss Curves
    plt.figure(figsize=[8, 6])
    plt.plot(history.history['loss'], 'r', linewidth=3.0)
    plt.plot(history.history['val_loss'], 'b', linewidth=3.0)
    plt.legend(['Training loss', 'Validation Loss'], fontsize=18)
    plt.xlabel('Epochs ', fontsize=16)
    plt.ylabel('Loss', fontsize=16)
    plt.title('Loss Curves', fontsize=16)
    plt.show()

    # Accuracy Curves
    plt.figure(figsize=[8, 6])
    plt.plot(history.history['accuracy'], 'r', linewidth=3.0)
    plt.plot(history.history['val_accuracy'], 'b', linewidth=3.0)
    plt.legend(['Training Accuracy', 'Validation Accuracy'], fontsize=18)
    plt.xlabel('Epochs ', fontsize=16)
    plt.ylabel('Accuracy', fontsize=16)
    plt.title('Accuracy Curves', fontsize=16)
    plt.show()


if __name__ == '__main__':
    history = train_model('data/train_angle/image', 'data/train_angle/labels')
    visualize_performance(history)
```

car_visualization.py

```python
import cv2
import matplotlib.pyplot as plt
import numpy as np
from Project.car_detection import object_detection_api
from keras.engine.saving import load_model
from skimage import io

ANGLE_DICT = {0: (0, -2), 1: (-1, -np.sqrt(3)), 2: (-np.sqrt(3), -1), 3: (-2, 0), 4: (-np.sqrt(3), 1),
              5: (-1, np.sqrt(3)), 6: (0, 2), 7: (1, np.sqrt(3)), 8: (np.sqrt(3), 1), 9: (2, 0),
              10: (np.sqrt(3), -1), 11: (1, -np.sqrt(3))}

ANGLE_LABEL = {0: '0°', 1: '30°', 2: '60°', 3: '90°', 4: '120°', 5: '150°', 6: '180°', 7: '210°', 8: '240°', 9: '270°',
               10: '300°', 11: '330°'}


def visualize_car(img_path):
    # Get car detection result and trained model
    boxes = object_detection_api(img_path)
    model = load_model('model.h5')

    img = io.imread(img_path)
    fig, ax = plt.subplots()
    ax.imshow(img)

    for i in range(len(boxes)):
        # Use bounding boxes to crop patch
        left, top = int(boxes[i][0][0]), int(boxes[i][0][1])
        right, bottom = int(boxes[i][1][0]), int(boxes[i][1][1])

        # Ignore patches that are too small
        width, height = right - left, bottom - top
        if width >= 30 and height >= 30:
            patch = img[top:bottom, left:right]
            patch = cv2.resize(patch, dsize=(97, 53), interpolation=cv2.INTER_CUBIC)
            patch = patch.reshape((1, 97, 53, 3))

            # Predict viewpoint
            prediction = model.predict_classes(patch)
            dx, dy = ANGLE_DICT[prediction[0]][0], ANGLE_DICT[prediction[0]][1]

            # Visualize car bounding box and angle
            center_x, center_y = int((left + right) / 2), int((top + bottom) / 2)
            rect = plt.Rectangle((left, top), width, height, fill=False, linewidth=2, color='lime')
            plt.arrow(center_x, center_y, dx * 20, -dy * 20, color='w', linewidth=3, head_width=20, head_length=4)
            plt.text(center_x, center_y, ANGLE_LABEL[prediction[0]], fontsize=12, color='lime')
            ax.add_patch(rect)

    plt.show()


if __name__ == '__main__':
    visualize_car('./data/train/image_left/um_000011.jpg')
```