MTAN_dx2222

December 9, 2022

1 Multi-Task Attention Network

1.1 COMS 4995 Sec: 006 Final Project

Daming Xing (dx2222)

12/09/2022

Video Presentation: https://youtu.be/qwgj_WAEB1s & GitHub: https://github.com/dddxing/SegNet_MTAN

1.2 Introduction

Convolutional Neural Networks (CNNs), while successful in a range of computer vision tasks, are typically designed for single-task learning. When used in real-world applications, however, where multiple tasks usually need to be worked on simultaneously, building a set of independent networks may not be the most desirable or efficient.

To enable such simultaneous performance of tasks and inturn allowing for increased memory efficiency, inference speed, and ability to share informative features between related tasks, Multi-Task Learning (MTL) [4,5,6], was proposed.

Despite its advantages, however, MTL does have its share of challenges in comparison to standard single-task learning:

1. **Network Architecture**: Both task-shared and task-specific elements should be expressed in a MTL architecture. This ensures that over or under-fitting is prevented by giving the network the chance to learn both features specific to each job and a generalizable representation.
2. **Loss Function**: MTL's loss function need to be able to assign all tasks equal importance, regardless of the difficulty of the task.

To address both challenges, Johns & Davidson (2019) [1] proposed a novel architecture called the Multi-Task Attention Network (MTAN) that allows a shared network to learn a global feature pool created across all tasks.

Inspired by John & Davidson (2019), this project aims at deriving a single network that can be used for various different computer vision tasks, specifically *senmantic segmentation and depth estimation*.

1.3 Related Work

With parallels to transfer learning and continual learning, an important aspect of Multi-Task Learning (MTL) is balancing feature sharing acorss tasks. Kendall, Gao & Cipolla (2018) [8] proposed doing so with the use of weight uncertainty, which leverages task uncertainty to modify the loss function. Chen, Badrinarayanan & Lee (2018) [9], on the other hand, manipulates gradient norms to control the training dynamics.

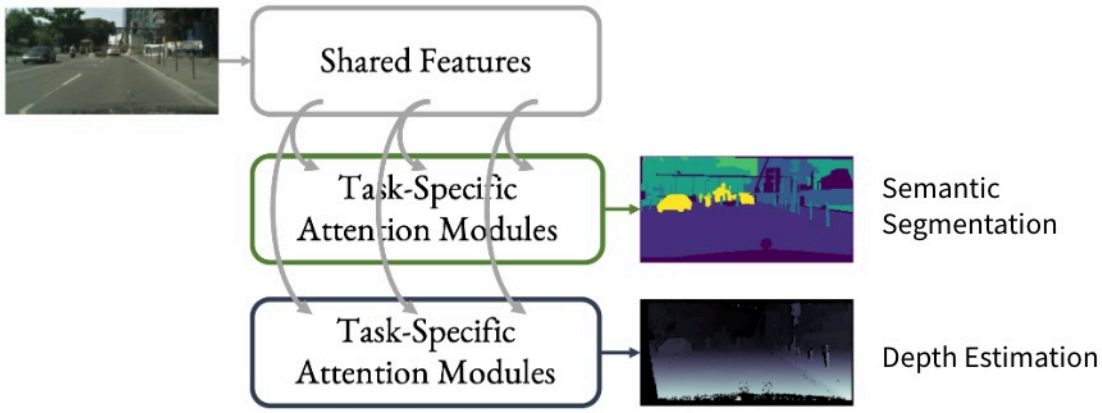
Most MTL networks are based off of exisitng CNN architectures. However, architectures like the Cross-Stitch Networks and Progressive Networks, which uses a series of networks that ar e incrementally trained for knowledge transfer between tasks One typical feed-forward network, require a large number of network parameters and will increase as the number of tasks increases[7]. On the otherhand, MTAN will require only ~10% increase in parameters for each task [1].

1.4 Methods

The goal of this project is to derive one single network that can be used to solve multiple tasks. These different tasks can be object detection, object classification and object segmentation, and so on. The constraint in this project is that I cannot have multiple networks for these different tasks. The advantage of having only one network over multiple networks is that I don't have to train multiple networks individually for the different tasks. It results in saving some time in training, as well as in inferencing.

The two tasks in this project are: - Semantic segmentation - Depth estimation

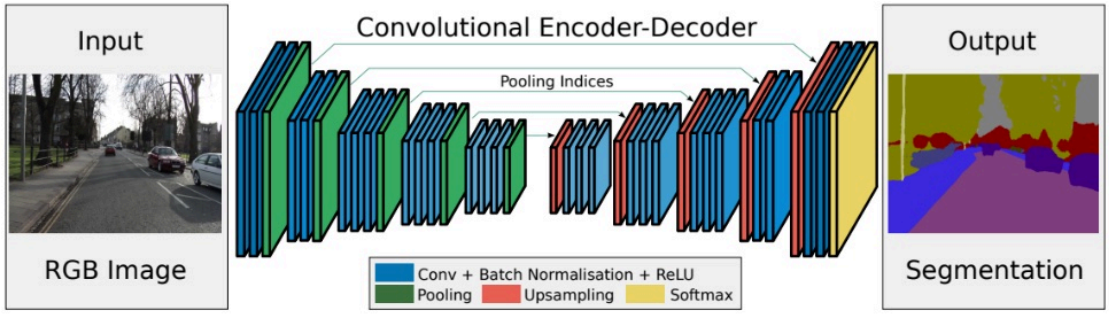
The workflow to do this, is that given an input image, the first step is to calculate the shared features. Then task-specific modules use those shared features to solve their given task. For N number of tasks, there will be N specific attention modules. These attention modules take some data from the shared features and augment it in such a way, that is useful for their particular task. *The word attention over here refers to the attention that each module gives to its particular task.*



For the MTAN technique, we can use any convolutional network to calculate the shared features. For this particular project, I used SegNet for the shared features. SegNet is a fully convolution network (FCN) that is used to solve the image segmentation task.

The first step I did is to downsample the image and derive the features from that image. Once we downsample the image and reach the bottleneck, we upsample the image to the same di-

mension as the given input image. And with the help of the features derived in the downsampling task, the output is a segmentation mask. The downsampling part of the network is also called the encoder and the upsampling part of the network is also called the decoder.

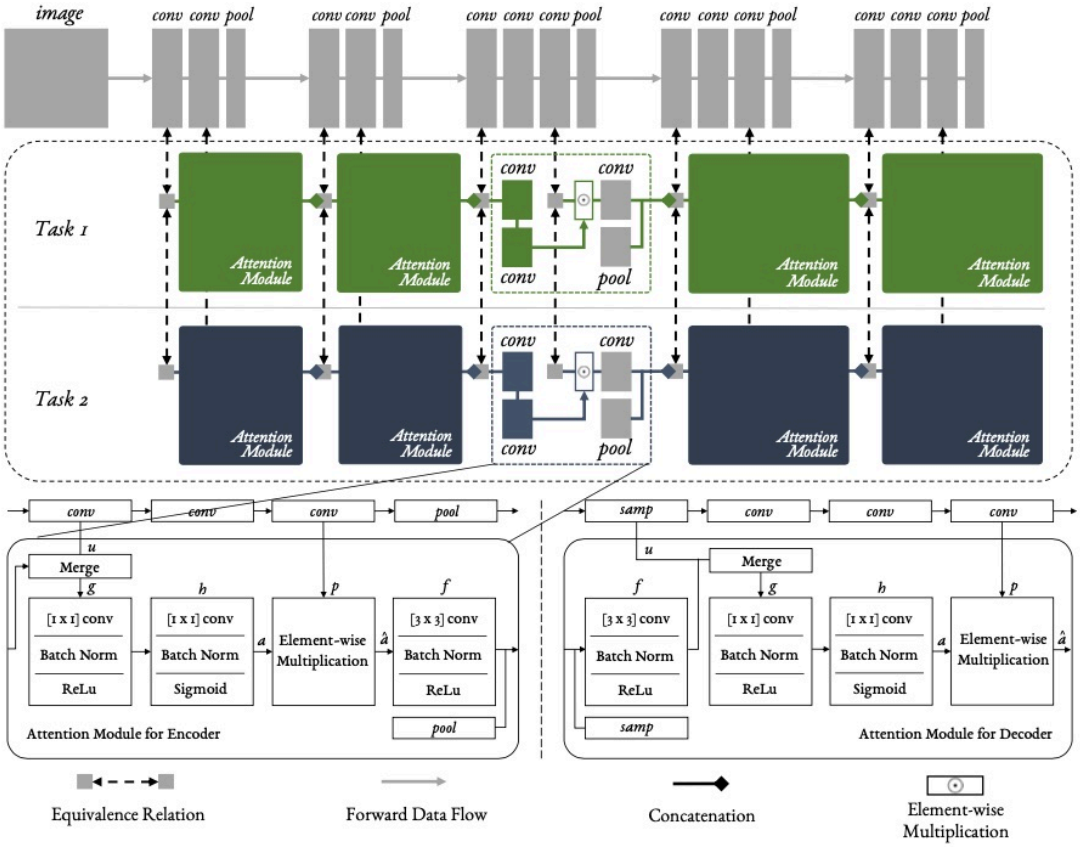


The first network (segnet_dense) was implemented without any attention modules. It worked, but with fairly low accuracy (~37%). Then I implemented the MTAN network, a network that is based on SegNet with attention modules.

MTAN is a single network, which consists of attention modules. Each task-specific network is made up of a collection of attention modules that connect to the common network, whereas the shared network can be created dependent on the individual task. To identify task-specific features, each attention module applies a soft attention mask to a specific layer of the shared network. As a result, attention masks can be thought of as feature selectors from the shared network that are automatically learnt in an end-to-end fashion, whereas the shared network learns a small global feature pool across all jobs.

The diagram below illustrates the encoder portion of SegNet and provides a thorough representation of the network based on VGG16. Thus, the SegNet decoder half is symmetric to VGG16. As demonstrated, each attention module learns a soft attention mask that depends on the shared network properties at the appropriate layer. In order to maximize the generalization of the shared features across different tasks, as well as the task-specific performance due to the attention masks, the features in the shared network and the soft attention masks can be learnt together.

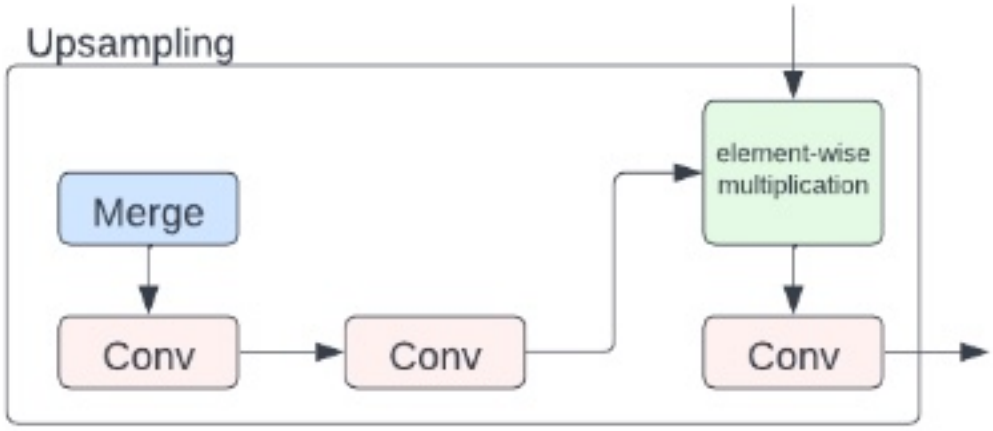
Here is the architecture is developed using SegNet as a base network with attention models. Convolution blocks comprise of convolution layers. followed by some pooling layers.



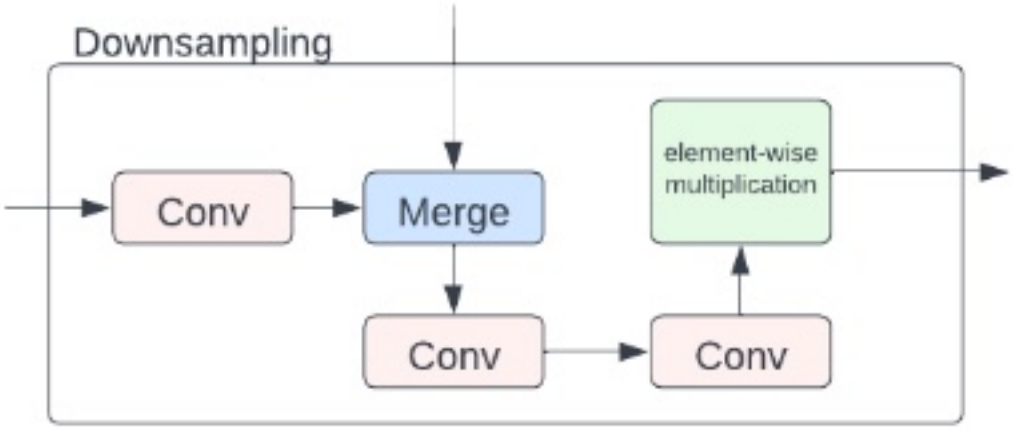
MTAN visualization based on VGG16 displaying SegNet's encoder half (with the decoder half being symmetrical to the encoder). The attention modules for tasks one (green) and two (blue) are distinct from one another and connect to the shared network (grey). The bottom section of the picture, which is extended, shows both the encoder and decoder versions of the middle attention module, which has its structure exposed for visualization. The design of each attention module is the same, but each module's weights are independently learnt.

1.4.1 Task Specific Attention Module

Upsampling filter For the upsampling attention module, the input is from the previous attention module, and another input is derived from the first layer of the convolution block. These two matrices then are merged. The convolution operation on them is computed twice, and then the elementwise multiplication of the output is calculated and the input of one of the filters of these convolution blocks. this element wise multiplication is once again sent through a convolution block and the output is returned to the next attention module.



Downsampling filter For the downsampling module, the input from the previous attention module is sent through convolution block which is then merged with the input of the first layer of the convolution block. This merged filter is then passed twice through convolution blocks. Then an elementwide multiplication is calculated for this output and the input that is taken from one of the layers of convolution block. Then, this final output is sent to the next attention module.



1.4.2 Model Objective (loss function)

For the model objective, I used mean Intersection over Union (mIoU) and prediction accuracy for the depth estimation.

$$IoU = \frac{TF}{TP + FN + FP}$$

where true-positives are those pixels that belong to the class and are correctly predicted as the class, false-negatives are those pixels that belong to the class but are incorrectly predicted as a different class and false-positives are those pixels that belong to a different class but are predicted as the class.

Recent approaches have attempted to overcome this problem since, for the majority of multi-task learning networks, training many tasks is challenging without achieving the ideal balance between those tasks. The adjustment of the weights of these losses is very important for the network to actually converge and learn the different tasks. Dynamic Weight Averaging is a technique that is used to determine these weights based on the rate of change of the gradient loss. Dynamic Weight Average (DWA) is very straightforward yet efficient adaptive weighting technique to test the suggested solution using a variety of weighting systems. By taking into account the rate of change of loss for each task, learns to average task weighting over time. GradNorm, however, needs access to the internal gradients of the network, whereas the DWA solution simply needs the numerical task loss. As a result, its implementation is much easier.

$$Loss_{total} = \lambda_1 * loss_1 + \lambda_2 * loss_2$$

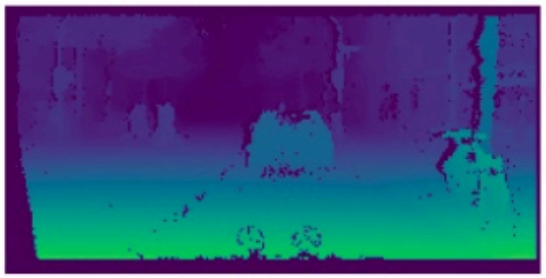
The λ_1, λ_2 and λ_3 are parameters in the networks as well.

1.5 Dataset

The City Scape dataset consists of different street view images, that is, images taken from a car that is being driven on the road. This data set is particularly used for depth estimation and semantic segmentation. Together with ground-truth inverse depth labels, the dataset includes 19 classes for pixel-wise semantic segmentation. The 19 classes' labels and the seven coarser categories' labels are the same as those in the original CityScapes dataset.

Sample Input

Sample Segmentation Label



Sample Depth Estimation Label

2-class	7-class	19-class
background	void	void
	flat	road, sidewalk
	construction	building, wall, fence
	object	pole, traffic light, traffic sign
	nature	vegetation, terrain
	sky	sky
foreground	human	person, rider
	vehicle	carm truck, bus, caravan, trailer, train, motorcycle

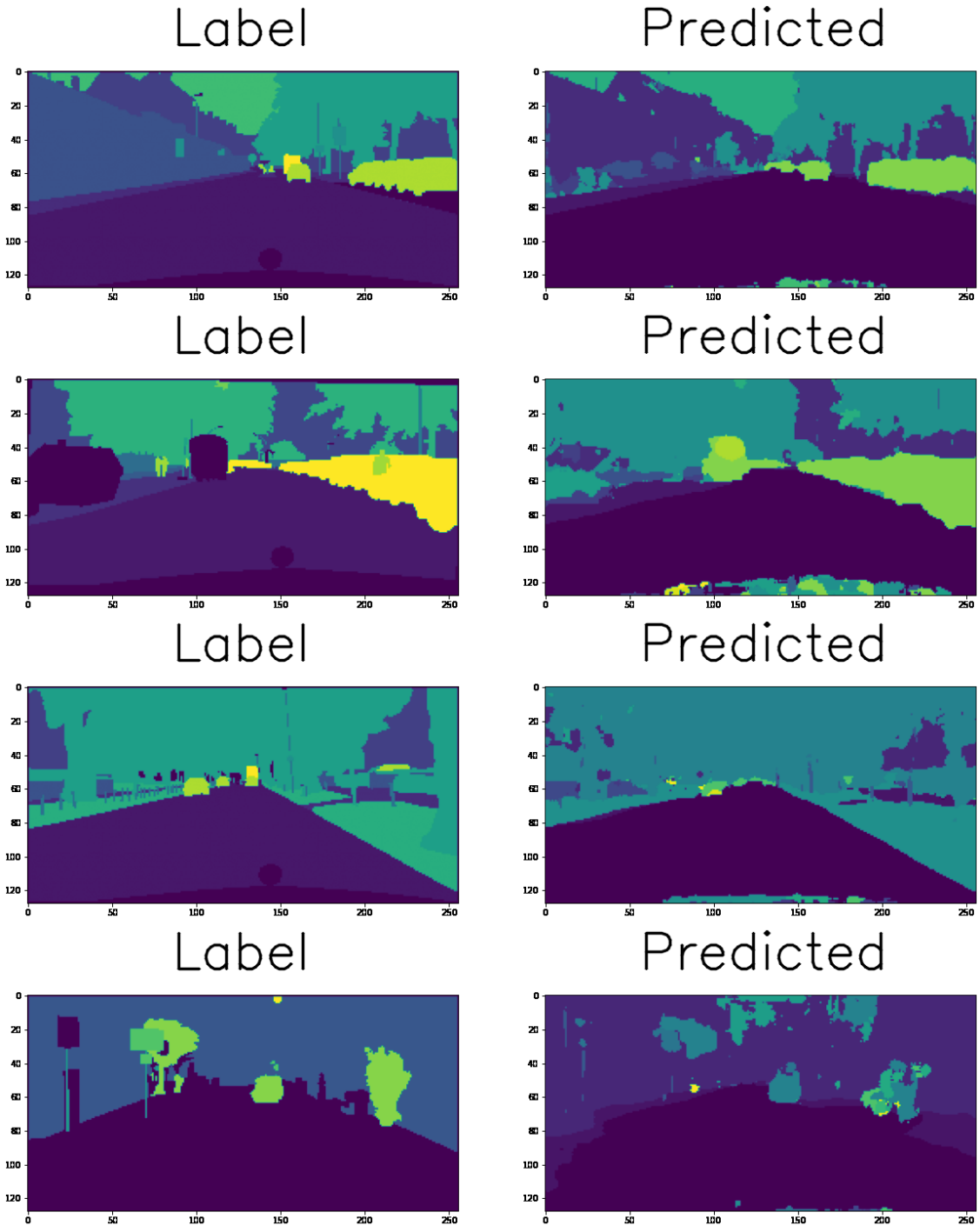
I split the training data into “training”, “validation”, and “test”.

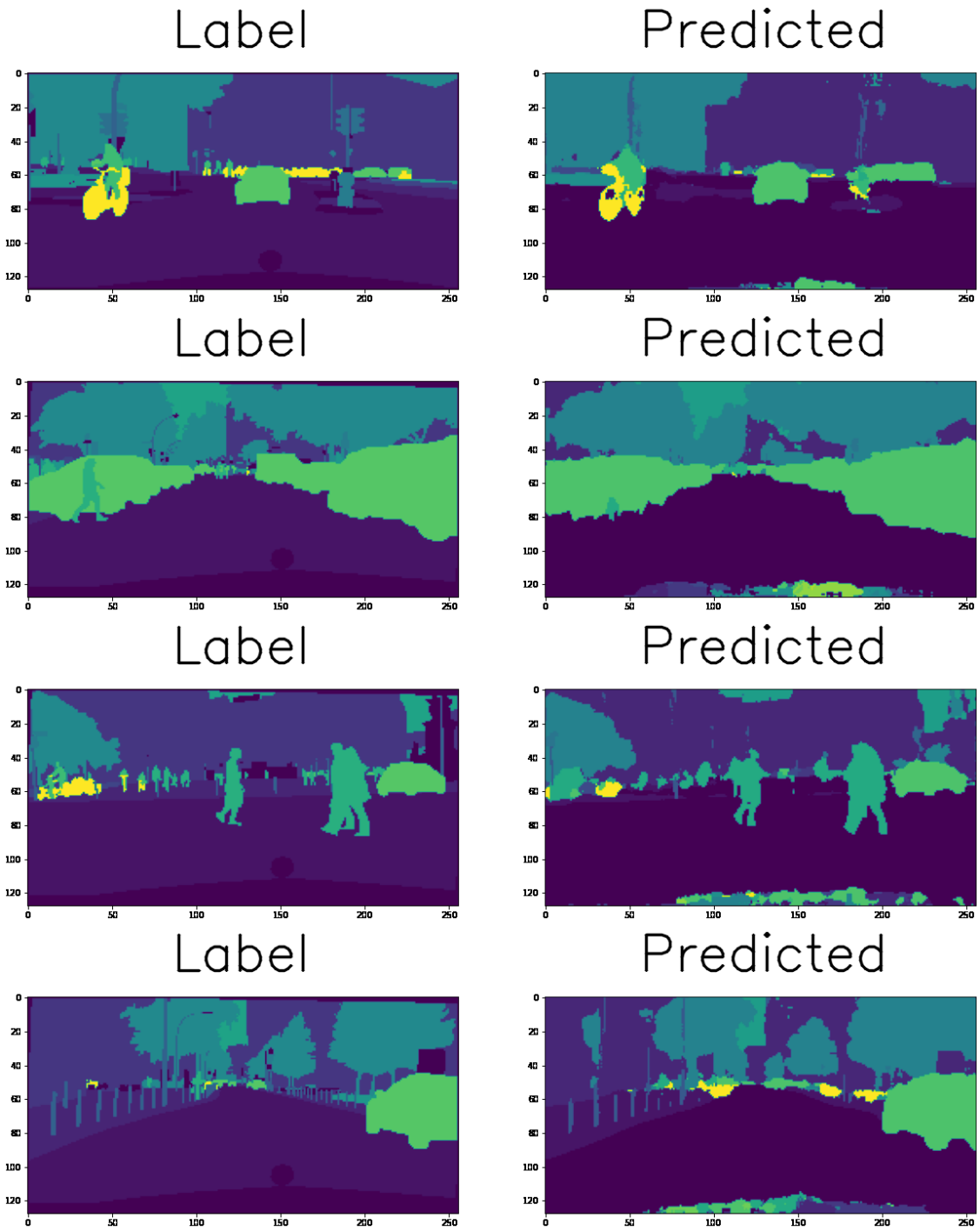
1.6 Results

Network	Accuracy
SegNet w/o attention	37%
MTAN	84%

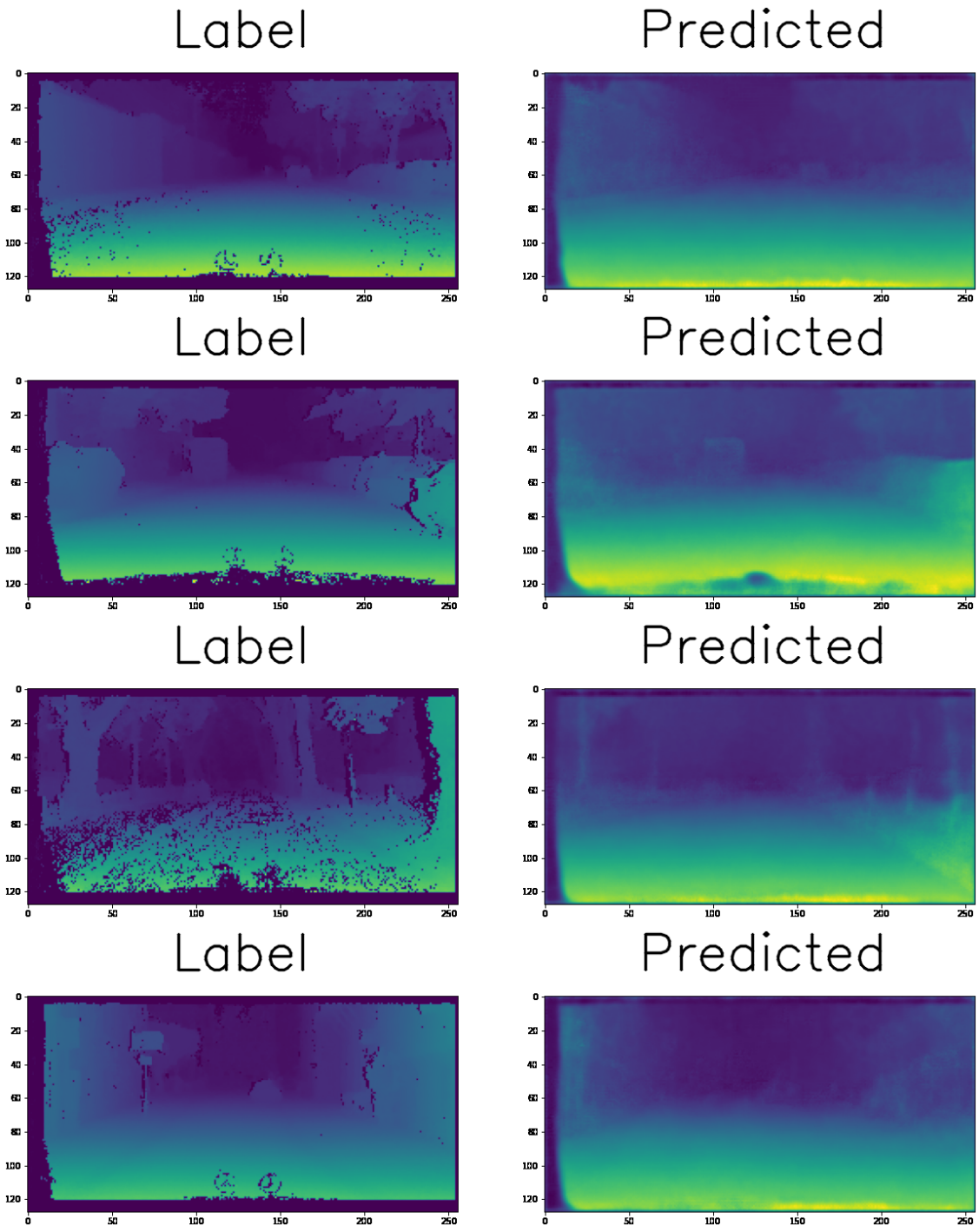
Here are some results from the testing, using the MTAN network.

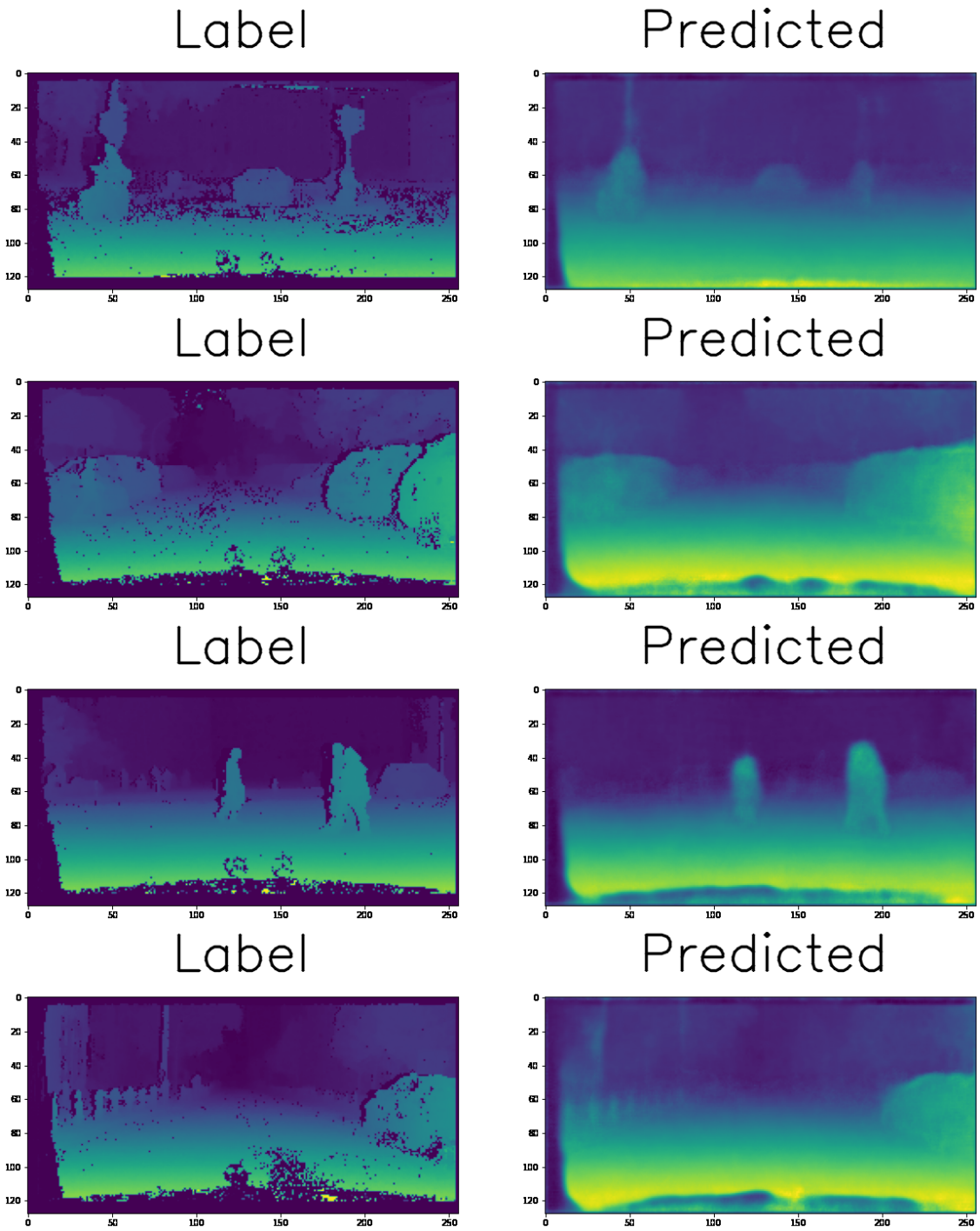
1.6.1 Segmentation - Side by Side



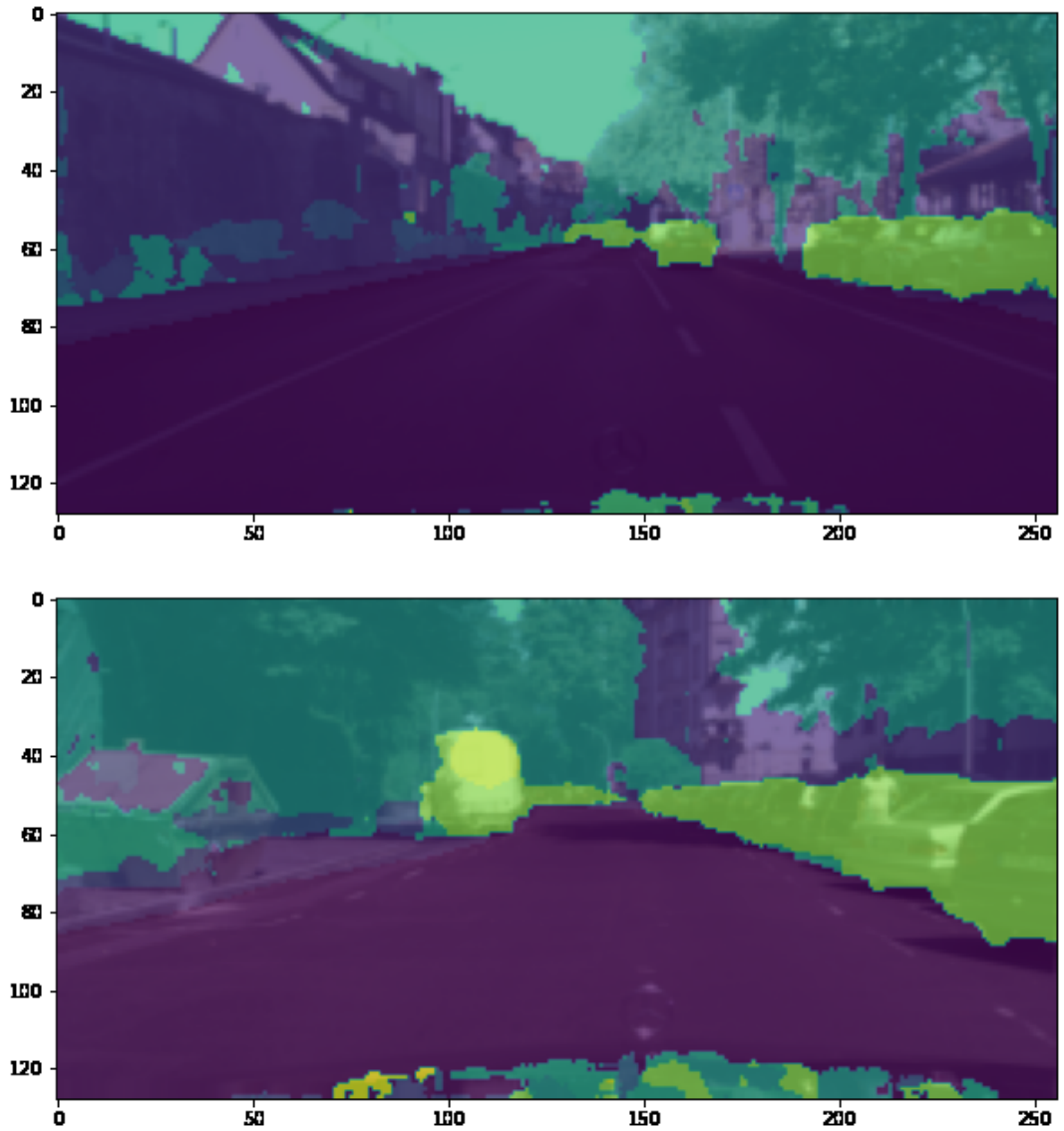

Label
Predicted
Label
Predicted
Label
Predicted
Label
Predicted

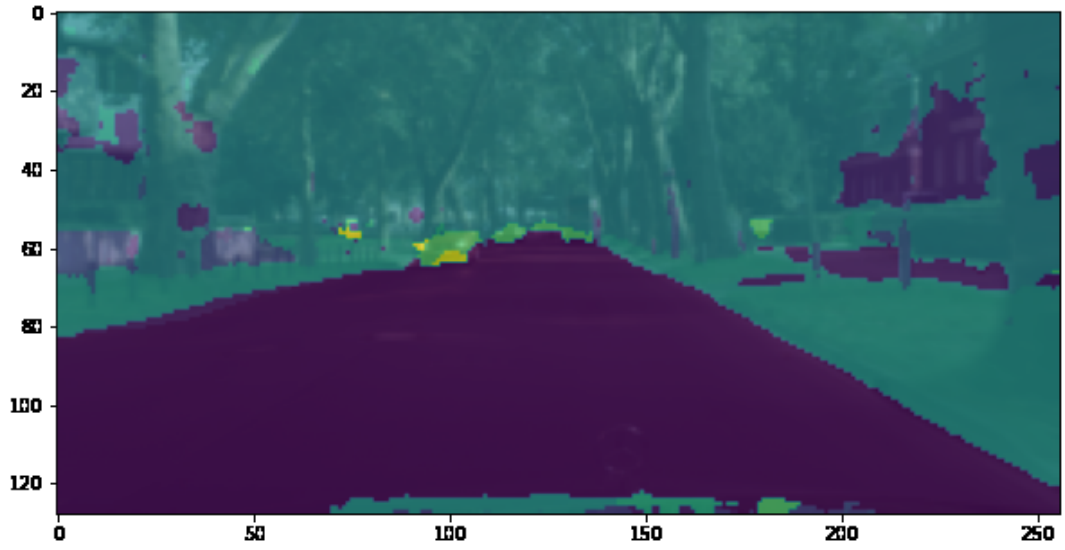
1.6.2 Depth Estimation - Side by Side

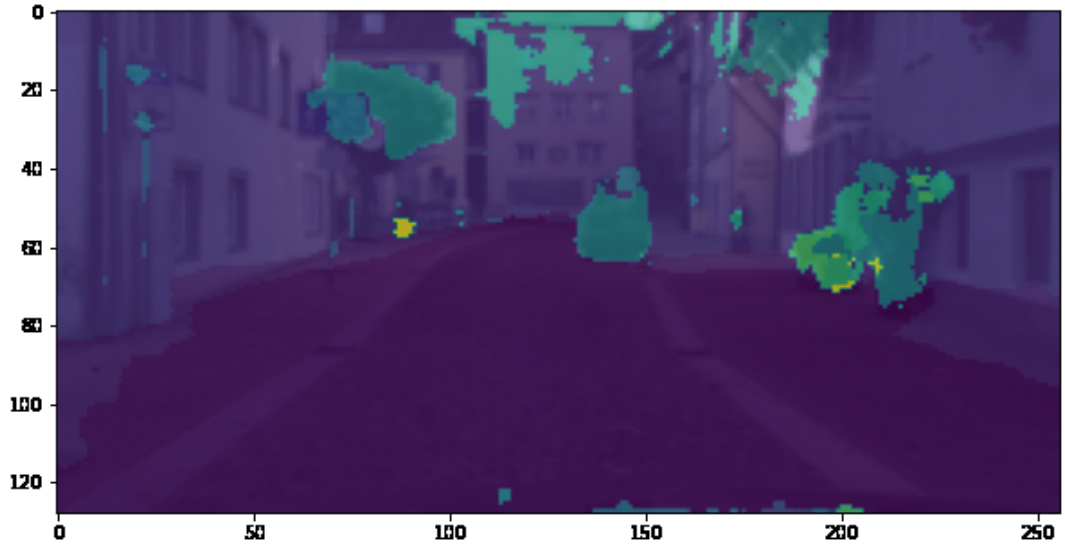


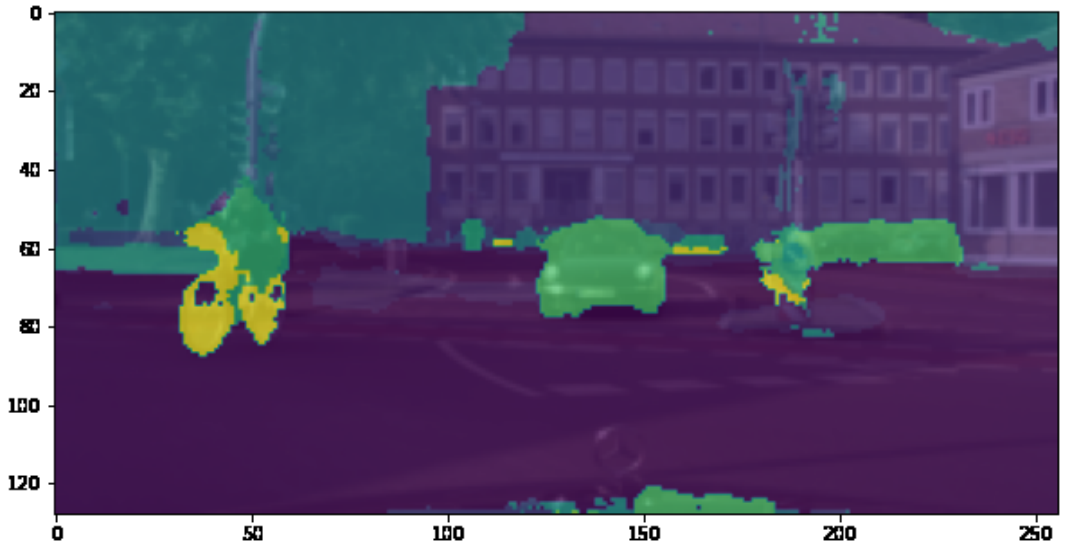

Label
Predicted
Label
Predicted
Label
Predicted
Label
Predicted

1.6.3 Segmentation - Overlay

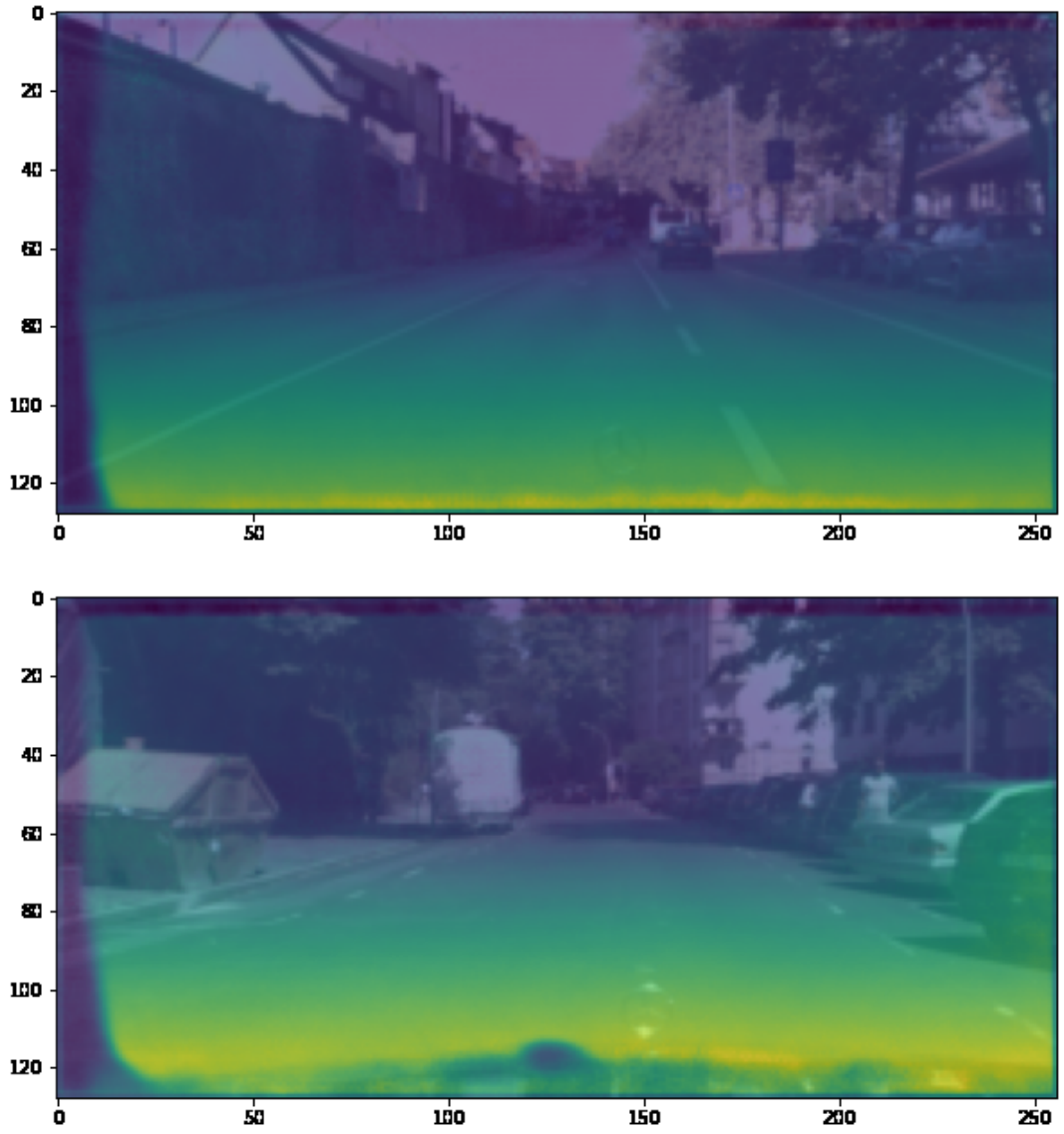


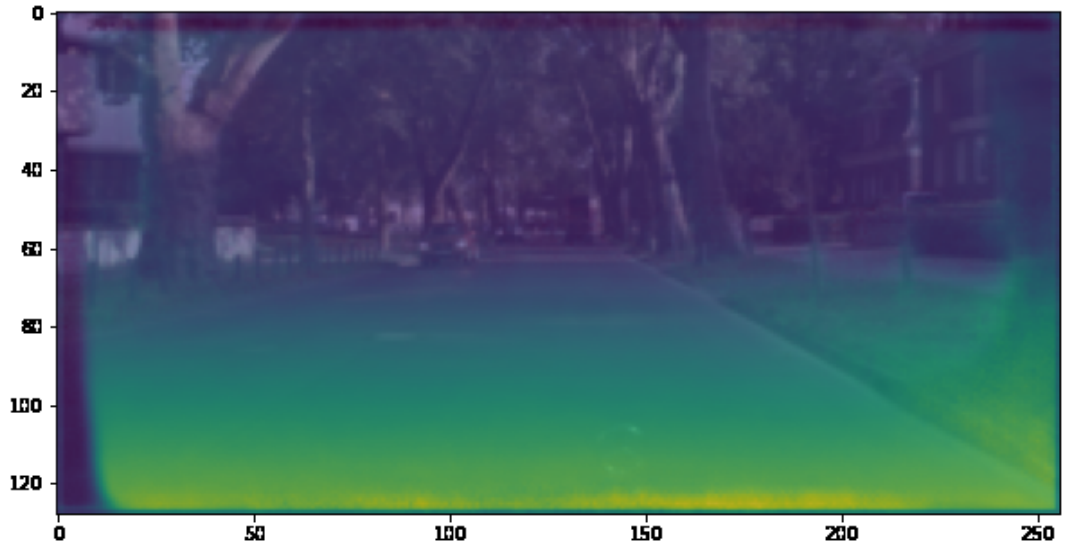

0
20
40
60
80
100
120
0
50
100
150
200
250

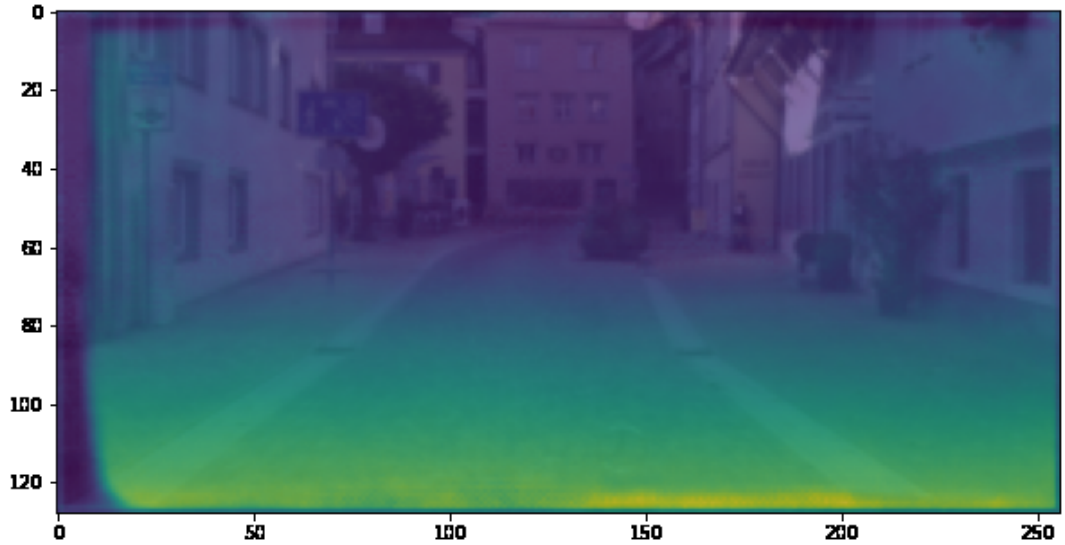

0
20
40
60
80
100
120
0
50
100
150
200
250

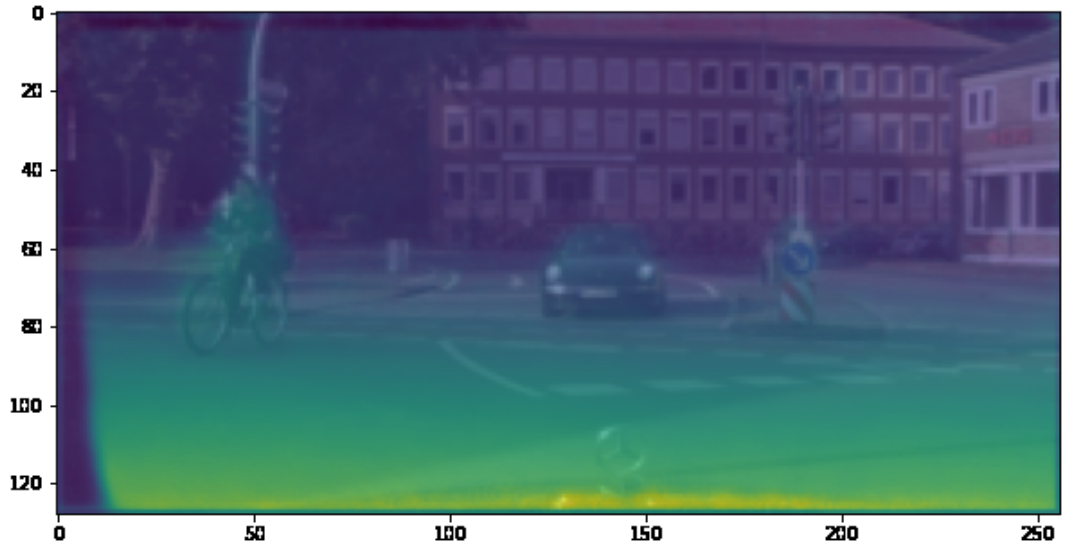

0
20
40
60
80
100
120
0
50
100
150
200
250

1.6.4 Depth Estimation - Overlay (not as clear as side by side)




0
20
40
60
80
100
120
0
50
100
150
200
250


0
20
40
60
80
100
120
0
50
100
150
200
250

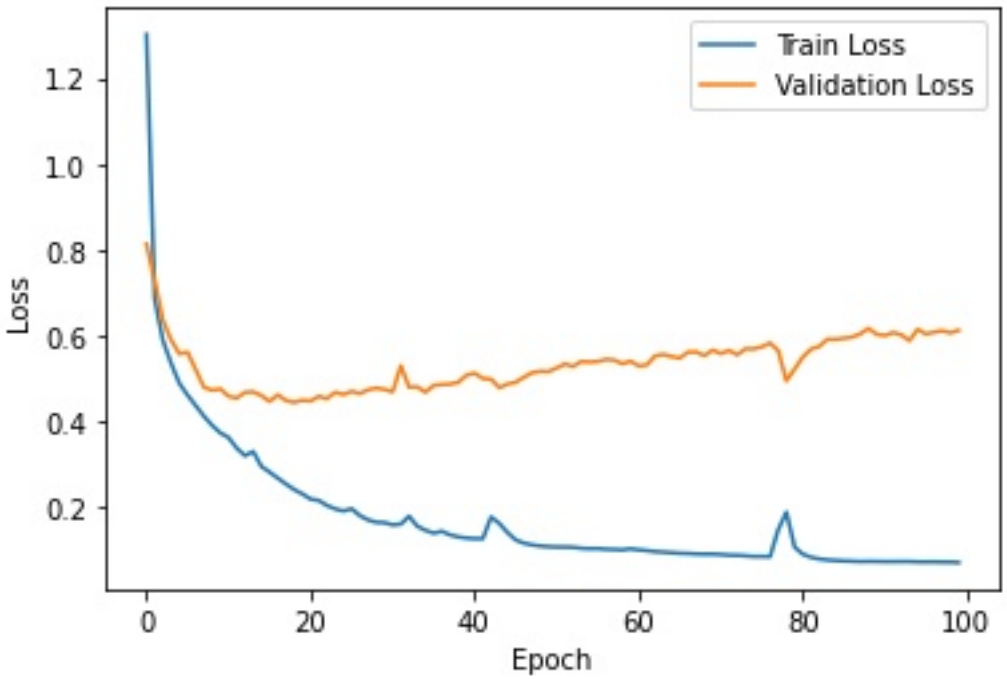


1.7 Conclusion

MTAN, as an extension of MTL, is made up of a global feature pool and task-specific attention modules, enabling automatic learning of both task-shared and task-specific characteristics in an end-to-end way. The technique performs well in experiments that utilized the CityScapes datasets with several tasks, including semantic segmentation and depth estimation, and it also demonstrates robustness to the specific task weighting schemes employed in the loss function using DWA. The total accuracy (segmentation & depth estimation) during traning is around 84%.

1.7.1 Learning Curve

I trained the network from scratch to 100 epochs. It took more than 7 hours to complete the training process.



1.7.2 Future Work

Although I mentioned a single Multi-task Learning network would be faster during both train inference than multiple single-task networks, I did not get the chance to confirm this experimentally. Therefore, I hope to test it out in the future by using multiple single networks as a baseline to compare with a single multi-tasks network in training and inference speed.

In addition to which, in the paper [1], the authors also implemented different flavors of MTAN. In this project, I only implemented the most basic MTAN. In the future, I want to test out different versions of MTAN networks, such as Multi-Task, Split (Wide, Deep), Multi-Task, Dense and Multi-Task, Cross-Stitch.

Also, by looking at the result, I think training the network with more epochs will result in a better model. However, the Google Cloud Platform (GCP) with class voucher was not too friendly with PyTorch, so I had to implement this project on AWS EC2 GPU. Unfortunately, I did not train this network for more than 100 epochs.

1.8 Citation

[1] S. Liu, E. Johns and A. J. Davison, "End-To-End Multi-Task Learning With Attention," 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2019, pp. 1871-1880, doi: 10.1109/CVPR.2019.00197.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In Proceed- ings of the IEEE conference on computer vision and pattern recognition, pages 770–778, 2016.

[3] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. In European Conference on Computer Vision, pages 694–711. Springer, 2016.

[4] IshanMisra, AbhinavShrivastava, AbhinavGupta, and Martial Hebert. Cross-stitch networks for multi-task learning. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 3994–4003, 2016.

[5] Carl Doersch and Andrew Zisserman. Multi-task self-supervised visual learning. In The IEEE International Con- ference on Computer Vision (ICCV), Oct 2017.

[7] Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Raz- van Pascanu, and Raia Hadsell. Progressive neural networks. arXiv preprint arXiv:1606.04671, 2016.

[8] Alex Kendall, Yarin Gal, and Roberto Cipolla. Multi-task learning using uncertainty to weigh losses for scene geome- try and semantics. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 7482– 7491, 2018.

[9] Zhao Chen, Vijay Badrinarayanan, Chen-Yu Lee, and An- drew Rabinovich. Gradnorm: Gradient normalization for adaptive loss balancing in deep multitask networks. In Inter- national Conference on Machine Learning, pages 793–802, 2018.

1.9 Code

[2]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data.dataset import Dataset
import torch.utils.data.sampler as sampler

import os
import fnmatch
import numpy as np
import random
import matplotlib.pyplot as plt
```

1.9.1 Dataset Loader

[2]:
```
class CityScapes(Dataset):
    def __init__(self, root, train=True):
        self.train = train
        self.root = os.path.expanduser(root)

        # read the data file
        if train:
            self.data_path = root + '/train'
        else:
            self.data_path = root + '/val'

        # calculate data length
        self.data_len = len(fnmatch.filter(os.listdir(self.data_path + '/
 ↪image'), '*.npy'))
```

```
    def __getitem__(self, index):
        # load data from the pre-processed npy files
        image = torch.from_numpy(np.moveaxis(np.load(self.data_path + '/image/{:
↪d}.npy'.format(index)), -1, 0))
        semantic = torch.from_numpy(np.load(self.data_path + '/label/{:d}.npy'.
↪format(index)))
        depth = torch.from_numpy(np.moveaxis(np.load(self.data_path + '/depth/{:
↪d}.npy'.format(index)), -1, 0))

        return image.float(), semantic.float(), depth.float()

    def __len__(self):
        return self.data_len
```

1.9.2 Utility Functions

Define Task Metrics, Loss Functions and Model Trainer

[3]:
```
def model_fit(x_pred, x_output, task_type):
    device = x_pred.device

    # binary mark to mask out undefined pixel space
    binary_mask = (torch.sum(x_output, dim=1) != 0).float().unsqueeze(1).
↪to(device)

    if task_type == 'semantic':
        # semantic loss: depth-wise cross entropy
        loss = F.nll_loss(x_pred, x_output, ignore_index=-1)

    if task_type == 'depth':
        # depth loss: l1 norm
        loss = torch.sum(
            torch.abs(x_pred - x_output) * binary_mask) / torch.
↪nonzero(binary_mask, as_tuple=False).size(0)

    return loss
```

[4]:
```
# mIoU and Acc. formula: accumulate every pixel and average across all pixels␣
↪in all images
class ConfMatrix(object):
    def __init__(self, num_classes):
        self.num_classes = num_classes
        self.mat = None

    def update(self, pred, target):
        n = self.num_classes
```

```
        if self.mat is None:
            self.mat = torch.zeros((n, n), dtype=torch.int64, device=pred.
 ↪device)
        with torch.no_grad():
            k = (target >= 0) & (target < n)
            inds = n * target[k].to(torch.int64) + pred[k]
            self.mat += torch.bincount(inds, minlength=n ** 2).reshape(n, n)

    def get_metrics(self):
        h = self.mat.float()
        acc = torch.diag(h).sum() / h.sum()
        iu = torch.diag(h) / (h.sum(1) + h.sum(0) - torch.diag(h))
        return torch.mean(iu), acc
```

[3]:
```
def depth_error(x_pred, x_output):
    device = x_pred.device
    binary_mask = (torch.sum(x_output, dim=1) != 0).unsqueeze(1).to(device)
    x_pred_true = x_pred.masked_select(binary_mask)
    x_output_true = x_output.masked_select(binary_mask)
    abs_err = torch.abs(x_pred_true - x_output_true)
    rel_err = torch.abs(x_pred_true - x_output_true) / x_output_true
    return (torch.sum(abs_err) / torch.nonzero(binary_mask, as_tuple=False).
 ↪size(0)).item(), \
           (torch.sum(rel_err) / torch.nonzero(binary_mask, as_tuple=False).
 ↪size(0)).item()

_loss = []
def multi_task_trainer(train_loader, test_loader, multi_task_model, device,␣
 ↪optimizer, scheduler, config, total_epoch=200):
    train_batch = len(train_loader)
    test_batch = len(test_loader)

    T = config['temp']
    avg_cost = np.zeros([total_epoch, 12], dtype=np.float32)
    lambda_weight = np.ones([2, total_epoch])

    for index in range(total_epoch):
        cost = np.zeros(12, dtype=np.float32)

        # apply Dynamic Weight Average
        if config['weight'] == 'dwa':
            if index == 0 or index == 1:
                lambda_weight[:, index] = 1.0
            else:
                w_1 = avg_cost[index - 1, 0] / avg_cost[index - 2, 0]
                w_2 = avg_cost[index - 1, 3] / avg_cost[index - 2, 3]
```

```
                lambda_weight[0, index] = 2 * np.exp(w_1 / T) / (np.exp(w_1 /␣
↪T) + np.exp(w_2 / T))
                lambda_weight[1, index] = 2 * np.exp(w_2 / T) / (np.exp(w_1 /␣
↪T) + np.exp(w_2 / T))

        # iteration for all batches
        multi_task_model.train()
        train_dataset = iter(train_loader)
        conf_mat = ConfMatrix(multi_task_model.class_nb)
        for k in range(train_batch):
            train_data, train_label, train_depth = next(train_dataset)
            train_data, train_label = train_data.to(device), train_label.long().
↪to(device)
            train_depth = train_depth.to(device)

            train_pred, logsigma = multi_task_model(train_data)

            optimizer.zero_grad()
            train_loss = [model_fit(train_pred[0], train_label, 'semantic'),
                          model_fit(train_pred[1], train_depth, 'depth')]

            if config['weight'] == 'equal' or config['weight'] == 'dwa':
                loss = sum([lambda_weight[i, index] * train_loss[i] for i in␣
↪range(2)])
            else:
                loss = sum(1 / (2 * torch.exp(logsigma[i])) * train_loss[i] +␣
↪logsigma[i] / 2 for i in range(2))

            loss.backward()
            optimizer.step()

            # accumulate label prediction for every pixel in training images
            conf_mat.update(train_pred[0].argmax(1).flatten(), train_label.
↪flatten())

            cost[0] = train_loss[0].item()
            cost[3] = train_loss[1].item()
            cost[4], cost[5] = depth_error(train_pred[1], train_depth)
            avg_cost[index, :6] += cost[:6] / train_batch

        # compute mIoU and acc
        mIoU, acc = conf_mat.get_metrics()
        tmp = (mIoU.cpu(), mIoU.cpu())
        avg_cost[index, 1:3] = tmp

        # evaluating test data
```

```
        multi_task_model.eval()
        conf_mat = ConfMatrix(multi_task_model.class_nb)
        with torch.no_grad():  # operations inside don't track history
            test_dataset = iter(test_loader)
            for k in range(test_batch):
                test_data, test_label, test_depth = next(test_dataset)
                test_data, test_label = test_data.to(device), test_label.long().
↪to(device)
                test_depth = test_depth.to(device)

                test_pred, _ = multi_task_model(test_data)
                test_loss = [model_fit(test_pred[0], test_label, 'semantic'),
                             model_fit(test_pred[1], test_depth, 'depth')]

                conf_mat.update(test_pred[0].argmax(1).flatten(), test_label.
↪flatten())

                cost[6] = test_loss[0].item()
                cost[9] = test_loss[1].item()
                cost[10], cost[11] = depth_error(test_pred[1], test_depth)
                avg_cost[index, 6:] += cost[6:] / test_batch

            # compute mIoU and acc
            mIoU, acc = conf_mat.get_metrics()
            tmp = (mIoU.cpu(), mIoU.cpu())
            avg_cost[index, 7:9] = tmp

        scheduler.step()
        _loss.append(avg_cost[index, :])
        print('Epoch: {:04d} | TRAIN: {:.4f} {:.4f} | {:.4f} {:.4f} ||'
            'VALIDATION: {:.4f} {:.4f} | {:.4f} {:.4f} '
            .format(index, avg_cost[index, 0], avg_cost[index, 1],
                    avg_cost[index, 4], avg_cost[index, 5], avg_cost[index, 6],␣
↪avg_cost[index, 7],
                    avg_cost[index, 10], avg_cost[index, 11]))
```

1.9.3 Network

[4]:
```
class SegNet(nn.Module):
    def __init__(self):
        super(SegNet, self).__init__()


        # initialise network parameters
        filter = [64, 128, 256, 512, 512]
        self.class_nb = 19
```

```
        # define encoder decoder layers
        self.encoder_block = nn.ModuleList([self.conv_layer([3, filter[0]])])
        self.decoder_block = nn.ModuleList([self.conv_layer([filter[0],␣
↪filter[0]])])
        for i in range(4):
            self.encoder_block.append(self.conv_layer([filter[i], filter[i +␣
↪1]]))
            self.decoder_block.append(self.conv_layer([filter[i + 1],␣
↪filter[i]]))

        # define convolution layer
        self.conv_block_enc = nn.ModuleList([self.conv_layer([filter[0],␣
↪filter[0]])])
        self.conv_block_dec = nn.ModuleList([self.conv_layer([filter[0],␣
↪filter[0]])])
        for i in range(4):
            if i == 0:
                self.conv_block_enc.append(self.conv_layer([filter[i + 1],␣
↪filter[i + 1]]))
                self.conv_block_dec.append(self.conv_layer([filter[i],␣
↪filter[i]]))
            else:
                self.conv_block_enc.append(nn.Sequential(self.
↪conv_layer([filter[i + 1], filter[i + 1]]),
                                                         self.
↪conv_layer([filter[i + 1], filter[i + 1]])))
                self.conv_block_dec.append(nn.Sequential(self.
↪conv_layer([filter[i], filter[i]]),
                                                         self.
↪conv_layer([filter[i], filter[i]])))

        # define task attention layers
        self.encoder_att = nn.ModuleList([nn.ModuleList([self.
↪att_layer([filter[0], filter[0], filter[0]])])])
        self.decoder_att = nn.ModuleList([nn.ModuleList([self.att_layer([2 *␣
↪filter[0], filter[0], filter[0]])])])
        self.encoder_block_att = nn.ModuleList([self.conv_layer([filter[0],␣
↪filter[1]])])
        self.decoder_block_att = nn.ModuleList([self.conv_layer([filter[0],␣
↪filter[0]])])

        for j in range(2):
            if j < 2:
                self.encoder_att.append(nn.ModuleList([self.
↪att_layer([filter[0], filter[0], filter[0]])]))
```

```
                self.decoder_att.append(nn.ModuleList([self.att_layer([2 *␣
↪filter[0], filter[0], filter[0]])]))
            for i in range(4):
                self.encoder_att[j].append(self.att_layer([2 * filter[i + 1],␣
↪filter[i + 1], filter[i + 1]]))
                self.decoder_att[j].append(self.att_layer([filter[i + 1] +␣
↪filter[i], filter[i], filter[i]]))

        for i in range(4):
            if i < 3:
                self.encoder_block_att.append(self.conv_layer([filter[i + 1],␣
↪filter[i + 2]]))
                self.decoder_block_att.append(self.conv_layer([filter[i + 1],␣
↪filter[i]]))
            else:
                self.encoder_block_att.append(self.conv_layer([filter[i + 1],␣
↪filter[i + 1]]))
                self.decoder_block_att.append(self.conv_layer([filter[i + 1],␣
↪filter[i + 1]]))

        self.pred_task1 = self.conv_layer([filter[0], self.class_nb], pred=True)
        self.pred_task2 = self.conv_layer([filter[0], 1], pred=True)

        # define pooling and unpooling functions
        self.down_sampling = nn.MaxPool2d(kernel_size=2, stride=2,␣
↪return_indices=True)
        self.up_sampling = nn.MaxUnpool2d(kernel_size=2, stride=2)

        self.logsigma = nn.Parameter(torch.FloatTensor([-0.5, -0.5, -0.5]))

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.xavier_normal_(m.weight)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.xavier_normal_(m.weight)
                nn.init.constant_(m.bias, 0)

    def conv_layer(self, channel, pred=False):
        if not pred:
            conv_block = nn.Sequential(
                nn.Conv2d(in_channels=channel[0], out_channels=channel[1],␣
↪kernel_size=3, padding=1),
```

```
                nn.BatchNorm2d(num_features=channel[1]),
                nn.ReLU(inplace=True),
            )
        else:
            conv_block = nn.Sequential(
                nn.Conv2d(in_channels=channel[0], out_channels=channel[0],␣
↪kernel_size=3, padding=1),
                nn.Conv2d(in_channels=channel[0], out_channels=channel[1],␣
↪kernel_size=1, padding=0),
            )
        return conv_block

    def att_layer(self, channel):
        att_block = nn.Sequential(
            nn.Conv2d(in_channels=channel[0], out_channels=channel[1],␣
↪kernel_size=1, padding=0),
            nn.BatchNorm2d(channel[1]),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=channel[1], out_channels=channel[2],␣
↪kernel_size=1, padding=0),
            nn.BatchNorm2d(channel[2]),
            nn.Sigmoid(),
        )
        return att_block

    def forward(self, x):
        g_encoder, g_decoder, g_maxpool, g_upsampl, indices = ([0] * 5 for _ in␣
↪range(5))
        for i in range(5):
            g_encoder[i], g_decoder[-i - 1] = ([0] * 2 for _ in range(2))

        # define attention list for tasks
        atten_encoder, atten_decoder = ([0] * 3 for _ in range(2))
        for i in range(2):
            atten_encoder[i], atten_decoder[i] = ([0] * 5 for _ in range(2))
        for i in range(2):
            for j in range(5):
                atten_encoder[i][j], atten_decoder[i][j] = ([0] * 3 for _ in␣
↪range(2))

        # define global shared network
        for i in range(5):
            if i == 0:
                g_encoder[i][0] = self.encoder_block[i](x)
                g_encoder[i][1] = self.conv_block_enc[i](g_encoder[i][0])
                g_maxpool[i], indices[i] = self.down_sampling(g_encoder[i][1])
```

```
            else:
                g_encoder[i][0] = self.encoder_block[i](g_maxpool[i - 1])
                g_encoder[i][1] = self.conv_block_enc[i](g_encoder[i][0])
                g_maxpool[i], indices[i] = self.down_sampling(g_encoder[i][1])

        for i in range(5):
            if i == 0:
                g_upsampl[i] = self.up_sampling(g_maxpool[-1], indices[-i - 1])
                g_decoder[i][0] = self.decoder_block[-i - 1](g_upsampl[i])
                g_decoder[i][1] = self.conv_block_dec[-i - 1](g_decoder[i][0])
            else:
                g_upsampl[i] = self.up_sampling(g_decoder[i - 1][-1],␣
↪indices[-i - 1])
                g_decoder[i][0] = self.decoder_block[-i - 1](g_upsampl[i])
                g_decoder[i][1] = self.conv_block_dec[-i - 1](g_decoder[i][0])

        # define task dependent attention module
        for i in range(2):
            for j in range(5):
                if j == 0:
                    atten_encoder[i][j][0] = self.
↪encoder_att[i][j](g_encoder[j][0])
                    atten_encoder[i][j][1] = (atten_encoder[i][j][0]) *␣
↪g_encoder[j][1]
                    atten_encoder[i][j][2] = self.
↪encoder_block_att[j](atten_encoder[i][j][1])
                    atten_encoder[i][j][2] = F.
↪max_pool2d(atten_encoder[i][j][2], kernel_size=2, stride=2)
                else:
                    atten_encoder[i][j][0] = self.encoder_att[i][j](torch.
↪cat((g_encoder[j][0], atten_encoder[i][j - 1][2]), dim=1))
                    atten_encoder[i][j][1] = (atten_encoder[i][j][0]) *␣
↪g_encoder[j][1]
                    atten_encoder[i][j][2] = self.
↪encoder_block_att[j](atten_encoder[i][j][1])
                    atten_encoder[i][j][2] = F.
↪max_pool2d(atten_encoder[i][j][2], kernel_size=2, stride=2)

            for j in range(5):
                if j == 0:
                    atten_decoder[i][j][0] = F.
↪interpolate(atten_encoder[i][-1][-1], scale_factor=2, mode='bilinear',␣
↪align_corners=True)
                    atten_decoder[i][j][0] = self.decoder_block_att[-j -␣
↪1](atten_decoder[i][j][0])
```

```
                    atten_decoder[i][j][1] = self.decoder_att[i][-j - 1](torch.
↪cat((g_upsampl[j], atten_decoder[i][j][0]), dim=1))
                    atten_decoder[i][j][2] = (atten_decoder[i][j][1]) *␣
↪g_decoder[j][-1]
                else:
                    atten_decoder[i][j][0] = F.interpolate(atten_decoder[i][j -␣
↪1][2], scale_factor=2, mode='bilinear', align_corners=True)
                    atten_decoder[i][j][0] = self.decoder_block_att[-j -␣
↪1](atten_decoder[i][j][0])
                    atten_decoder[i][j][1] = self.decoder_att[i][-j - 1](torch.
↪cat((g_upsampl[j], atten_decoder[i][j][0]), dim=1))
                    atten_decoder[i][j][2] = (atten_decoder[i][j][1]) *␣
↪g_decoder[j][-1]

        # define task prediction layers
        t1_pred = F.log_softmax(self.pred_task1(atten_decoder[0][-1][-1]),␣
↪dim=1)
        t2_pred = self.pred_task2(atten_decoder[1][-1][-1])

        return [t1_pred, t2_pred], self.logsigma
```

[3]:
```
class SegNet_dense(nn.Module):
    def __init__(self):
        super(SegNet_dense, self).__init__()
        # initialise network parameters
        filter = [64, 128, 256, 512, 512]
        self.class_nb = 13

        # define encoder decoder layers
        self.encoder_block = nn.ModuleList([self.conv_layer([3, filter[0],␣
↪filter[0]], bottle_neck=True)])
        self.decoder_block = nn.ModuleList([self.conv_layer([filter[0],␣
↪filter[0], self.class_nb], bottle_neck=True)])

        self.encoder_block_t = nn.ModuleList([nn.ModuleList([self.
↪conv_layer([3, filter[0], filter[0]], bottle_neck=True)])])
        self.decoder_block_t = nn.ModuleList([nn.ModuleList([self.conv_layer([2␣
↪* filter[0], 2 * filter[0], filter[0]], bottle_neck=True)])])

        for i in range(4):
            if i == 0:
                self.encoder_block.append(self.conv_layer([filter[i], filter[i␣
↪+ 1], filter[i + 1]], bottle_neck=True))
                self.decoder_block.append(self.conv_layer([filter[i + 1],␣
↪filter[i], filter[i]], bottle_neck=True))
            else:
```

```
                self.encoder_block.append(self.conv_layer([filter[i], filter[i␣
↪+ 1], filter[i + 1]], bottle_neck=False))
                self.decoder_block.append(self.conv_layer([filter[i + 1],␣
↪filter[i], filter[i]], bottle_neck=False))

        for j in range(3):
            if j < 2:
                self.encoder_block_t.append(nn.ModuleList([self.conv_layer([3,␣
↪filter[0], filter[0]], bottle_neck=True)]))
                self.decoder_block_t.append(nn.ModuleList([self.conv_layer([2 *␣
↪filter[0], 2 * filter[0], filter[0]], bottle_neck=True)]))
            for i in range(4):
                if i == 0:
                    self.encoder_block_t[j].append(self.conv_layer([2 *␣
↪filter[i], filter[i + 1], filter[i + 1]], bottle_neck=True))
                    self.decoder_block_t[j].append(self.conv_layer([2 *␣
↪filter[i + 1], filter[i], filter[i]], bottle_neck=True))
                else:
                    self.encoder_block_t[j].append(self.conv_layer([2 *␣
↪filter[i], filter[i + 1], filter[i + 1]], bottle_neck=False))
                    self.decoder_block_t[j].append(self.conv_layer([2 *␣
↪filter[i + 1], filter[i], filter[i]], bottle_neck=False))

        # define pooling and unpooling functions
        self.down_sampling = nn.MaxPool2d(kernel_size=2, stride=2,␣
↪return_indices=True)
        self.up_sampling = nn.MaxUnpool2d(kernel_size=2, stride=2)

        self.pred_task1 = self.conv_layer([filter[0], self.class_nb],␣
↪bottle_neck=True, pred_layer=True)
        self.pred_task2 = self.conv_layer([filter[0], 1], bottle_neck=True,␣
↪pred_layer=True)
        self.pred_task3 = self.conv_layer([filter[0], 3], bottle_neck=True,␣
↪pred_layer=True)

        self.logsigma = nn.Parameter(torch.FloatTensor([-0.5, -0.5, -0.5]))

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.xavier_uniform_(m.weight)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.xavier_uniform_(m.weight)
```

```
                nn.init.constant_(m.bias, 0)

    def conv_layer(self, channel, bottle_neck, pred_layer=False):
        if bottle_neck:
            if not pred_layer:
                conv_block = nn.Sequential(
                    nn.Conv2d(in_channels=channel[0], out_channels=channel[1],␣
↪kernel_size=3, padding=1),
                    nn.BatchNorm2d(channel[1]),
                    nn.ReLU(inplace=True),
                    nn.Conv2d(in_channels=channel[1], out_channels=channel[2],␣
↪kernel_size=3, padding=1),
                    nn.BatchNorm2d(channel[2]),
                    nn.ReLU(inplace=True),
                )
            else:
                conv_block = nn.Sequential(
                    nn.Conv2d(in_channels=channel[0], out_channels=channel[0],␣
↪kernel_size=3, padding=1),
                    nn.Conv2d(in_channels=channel[0], out_channels=channel[1],␣
↪kernel_size=1, padding=0),
                )

        else:
            conv_block = nn.Sequential(
                nn.Conv2d(in_channels=channel[0], out_channels=channel[1],␣
↪kernel_size=3, padding=1),
                nn.BatchNorm2d(channel[1]),
                nn.ReLU(inplace=True),
                nn.Conv2d(in_channels=channel[1], out_channels=channel[1],␣
↪kernel_size=3, padding=1),
                nn.BatchNorm2d(channel[1]),
                nn.ReLU(inplace=True),
                nn.Conv2d(in_channels=channel[1], out_channels=channel[2],␣
↪kernel_size=3, padding=1),
                nn.BatchNorm2d(channel[2]),
                nn.ReLU(inplace=True),
            )

        return conv_block

    def forward(self, x):
        encoder_conv, decoder_conv, encoder_samp, decoder_samp, indices = ([0]␣
↪* 5 for _ in range(5))
        encoder_conv_t, decoder_conv_t, encoder_samp_t, decoder_samp_t,␣
↪indices_t = ([0] * 3 for _ in range(5))
```

```
        for i in range(3):
            encoder_conv_t[i], decoder_conv_t[i], encoder_samp_t[i],␣
↪decoder_samp_t[i], indices_t[i] = ([0] * 5 for _ in range(5))

        # global shared encoder-decoder network
        for i in range(5):
            if i == 0:
                encoder_conv[i] = self.encoder_block[i](x)
                encoder_samp[i], indices[i] = self.
↪down_sampling(encoder_conv[i])
            else:
                encoder_conv[i] = self.encoder_block[i](encoder_samp[i - 1])
                encoder_samp[i], indices[i] = self.
↪down_sampling(encoder_conv[i])

        for i in range(5):
            if i == 0:
                decoder_samp[i] = self.up_sampling(encoder_samp[-1],␣
↪indices[-1])
                decoder_conv[i] = self.decoder_block[-i - 1](decoder_samp[i])
            else:
                decoder_samp[i] = self.up_sampling(decoder_conv[i - 1],␣
↪indices[-i - 1])
                decoder_conv[i] = self.decoder_block[-i - 1](decoder_samp[i])

        # define task prediction layers
        for j in range(3):
            for i in range(5):
                if i == 0:
                    encoder_conv_t[j][i] = self.encoder_block_t[j][i](x)
                    encoder_samp_t[j][i], indices_t[j][i] = self.
↪down_sampling(encoder_conv_t[j][i])
                else:
                    encoder_conv_t[j][i] = self.encoder_block_t[j][i](torch.
↪cat((encoder_samp_t[j][i - 1], encoder_samp[i - 1]), dim=1))
                    encoder_samp_t[j][i], indices_t[j][i] = self.
↪down_sampling(encoder_conv_t[j][i])

            for i in range(5):
                if i == 0:
                    decoder_samp_t[j][i] = self.
↪up_sampling(encoder_samp_t[j][-1], indices_t[j][-1])
                    decoder_conv_t[j][i] = self.decoder_block_t[j][-i -␣
↪1](torch.cat((decoder_samp_t[j][i], decoder_samp[i]), dim=1))
                else:
```

```
                    decoder_samp_t[j][i] = self.up_sampling(decoder_conv_t[j][i-
↪- 1], indices_t[j][-i - 1])
                    decoder_conv_t[j][i] = self.decoder_block_t[j][-i -
↪1](torch.cat((decoder_samp_t[j][i], decoder_samp[i]), dim=1))

        t1_pred = F.log_softmax(self.pred_task1(decoder_conv_t[0][-1]), dim=1)
        t2_pred = self.pred_task2(decoder_conv_t[1][-1])
        t3_pred = self.pred_task3(decoder_conv_t[2][-1])
        t3_pred = t3_pred / torch.norm(t3_pred, p=2, dim=1, keepdim=True)

        return [t1_pred, t2_pred, t3_pred], self.logsigma
```

1.9.4 Training

[5]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
config = {
    'temp': 2.0,
    'weight': 'dwa'
}
device
```

[5]:
```
device(type='cuda', index=0)
```

1.9.5 Model, Optimizer and Scheduler

[6]:
```
SegNet_MTAN = SegNet().to(device)
optimizer = optim.Adam(SegNet_MTAN.parameters(), lr=1e-4)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=100, gamma=0.5)

print('LOSS FORMAT: SEMANTIC_LOSS MEAN_IOU PIX_ACC | DEPTH_LOSS ABS_ERR REL_ERR-
↪<11.25 <22.5')
```

```
LOSS FORMAT: SEMANTIC_LOSS MEAN_IOU PIX_ACC | DEPTH_LOSS ABS_ERR REL_ERR <11.25
<22.5
```

1.9.6 Dataset Loading

[7]:
```
dataset_path = '.'
train_set = CityScapes(root=dataset_path, train=True)
test_set = CityScapes(root=dataset_path, train=False)

batch_size = 16
train_loader = torch.utils.data.DataLoader(
                dataset=train_set,
                batch_size=batch_size,
                shuffle=True)
```

```
test_loader = torch.utils.data.DataLoader(
              dataset=test_set,
              batch_size=batch_size,
              shuffle=False)
```

1.10 Train

[8]:
```
multi_task_trainer(train_loader,
                   test_loader,
                   SegNet_MTAN,
                   device,
                   optimizer,
                   scheduler,
                   config,
                   100)
```

```
Epoch: 0000 | TRAIN: 1.3019 0.1110 | 0.1000 264.8485 ||VALIDATION: 0.8140 0.1792
| 0.0389 115.6575
Epoch: 0001 | TRAIN: 0.6849 0.2068 | 0.0329 78.9241 ||VALIDATION: 0.7282 0.2000
| 0.0302 74.1709
Epoch: 0002 | TRAIN: 0.5887 0.2311 | 0.0270 63.5757 ||VALIDATION: 0.6361 0.2272
| 0.0262 75.9499
Epoch: 0003 | TRAIN: 0.5354 0.2490 | 0.0255 57.3513 ||VALIDATION: 0.5901 0.2378
| 0.0250 60.7712
Epoch: 0004 | TRAIN: 0.4891 0.2693 | 0.0238 53.3895 ||VALIDATION: 0.5579 0.2496
| 0.0227 62.1998
Epoch: 0005 | TRAIN: 0.4616 0.2835 | 0.0223 53.5165 ||VALIDATION: 0.5616 0.2591
| 0.0224 65.7789
Epoch: 0006 | TRAIN: 0.4371 0.2944 | 0.0218 51.2193 ||VALIDATION: 0.5208 0.2663
| 0.0253 63.9843
Epoch: 0007 | TRAIN: 0.4126 0.3069 | 0.0224 50.9415 ||VALIDATION: 0.4800 0.2819
| 0.0218 53.6568
Epoch: 0008 | TRAIN: 0.3911 0.3207 | 0.0201 48.2632 ||VALIDATION: 0.4732 0.2895
| 0.0225 50.5156
Epoch: 0009 | TRAIN: 0.3737 0.3397 | 0.0194 48.0168 ||VALIDATION: 0.4761 0.3003
| 0.0229 51.3667
Epoch: 0010 | TRAIN: 0.3629 0.3536 | 0.0194 46.7382 ||VALIDATION: 0.4590 0.3094
| 0.0225 51.6895
Epoch: 0011 | TRAIN: 0.3379 0.3731 | 0.0189 46.8156 ||VALIDATION: 0.4548 0.3200
| 0.0197 56.2042
Epoch: 0012 | TRAIN: 0.3206 0.3953 | 0.0184 45.4736 ||VALIDATION: 0.4680 0.3182
| 0.0234 50.1836
Epoch: 0013 | TRAIN: 0.3304 0.3890 | 0.0179 45.8754 ||VALIDATION: 0.4692 0.3225
| 0.0220 50.0194
Epoch: 0014 | TRAIN: 0.2957 0.4229 | 0.0182 44.6354 ||VALIDATION: 0.4606 0.3241
| 0.0210 49.2309
Epoch: 0015 | TRAIN: 0.2823 0.4458 | 0.0171 43.8950 ||VALIDATION: 0.4469 0.3361
| 0.0231 44.2406
```

```
Epoch: 0016 | TRAIN: 0.2683 0.4592 | 0.0177 45.3321 ||VALIDATION: 0.4623 0.3365
| 0.0215 40.1080
Epoch: 0017 | TRAIN: 0.2545 0.4850 | 0.0167 43.4926 ||VALIDATION: 0.4491 0.3474
| 0.0214 44.9611
Epoch: 0018 | TRAIN: 0.2415 0.5103 | 0.0161 42.4425 ||VALIDATION: 0.4447 0.3423
| 0.0217 43.4268
Epoch: 0019 | TRAIN: 0.2316 0.5243 | 0.0155 42.0251 ||VALIDATION: 0.4497 0.3557
| 0.0211 47.7155
Epoch: 0020 | TRAIN: 0.2194 0.5574 | 0.0156 41.4331 ||VALIDATION: 0.4479 0.3501
| 0.0237 40.7830
Epoch: 0021 | TRAIN: 0.2163 0.5697 | 0.0162 41.5235 ||VALIDATION: 0.4590 0.3631
| 0.0251 36.1114
Epoch: 0022 | TRAIN: 0.2046 0.5906 | 0.0153 40.3568 ||VALIDATION: 0.4534 0.3674
| 0.0222 42.4895
Epoch: 0023 | TRAIN: 0.1967 0.6096 | 0.0154 40.3272 ||VALIDATION: 0.4682 0.3664
| 0.0188 50.8038
Epoch: 0024 | TRAIN: 0.1917 0.6244 | 0.0143 39.9910 ||VALIDATION: 0.4631 0.3650
| 0.0227 43.7742
Epoch: 0025 | TRAIN: 0.1970 0.6165 | 0.0145 40.3034 ||VALIDATION: 0.4703 0.3711
| 0.0236 47.9545
Epoch: 0026 | TRAIN: 0.1810 0.6476 | 0.0142 39.0699 ||VALIDATION: 0.4655 0.3721
| 0.0213 39.8665
Epoch: 0027 | TRAIN: 0.1710 0.6671 | 0.0143 40.3049 ||VALIDATION: 0.4740 0.3746
| 0.0192 42.8023
Epoch: 0028 | TRAIN: 0.1655 0.6797 | 0.0137 38.5538 ||VALIDATION: 0.4780 0.3763
| 0.0261 37.7207
Epoch: 0029 | TRAIN: 0.1646 0.6831 | 0.0136 38.1611 ||VALIDATION: 0.4750 0.3800
| 0.0195 42.0940
Epoch: 0030 | TRAIN: 0.1594 0.6916 | 0.0137 38.3342 ||VALIDATION: 0.4693 0.3775
| 0.0237 38.7875
Epoch: 0031 | TRAIN: 0.1612 0.6902 | 0.0137 38.6823 ||VALIDATION: 0.5299 0.3532
| 0.0201 43.4458
Epoch: 0032 | TRAIN: 0.1796 0.6624 | 0.0141 38.2893 ||VALIDATION: 0.4803 0.3684
| 0.0190 43.0758
Epoch: 0033 | TRAIN: 0.1563 0.6998 | 0.0135 37.0530 ||VALIDATION: 0.4808 0.3748
| 0.0249 37.2589
Epoch: 0034 | TRAIN: 0.1468 0.7174 | 0.0130 37.0337 ||VALIDATION: 0.4683 0.3837
| 0.0178 45.0979
Epoch: 0035 | TRAIN: 0.1401 0.7273 | 0.0133 37.3820 ||VALIDATION: 0.4843 0.3704
| 0.0187 44.5619
Epoch: 0036 | TRAIN: 0.1445 0.7241 | 0.0138 36.5691 ||VALIDATION: 0.4863 0.3771
| 0.0209 34.0273
Epoch: 0037 | TRAIN: 0.1361 0.7354 | 0.0128 37.0186 ||VALIDATION: 0.4874 0.3747
| 0.0182 40.1617
Epoch: 0038 | TRAIN: 0.1315 0.7431 | 0.0138 36.9096 ||VALIDATION: 0.4914 0.3744
| 0.0195 50.1061
Epoch: 0039 | TRAIN: 0.1287 0.7467 | 0.0129 36.7394 ||VALIDATION: 0.5080 0.3790
| 0.0182 37.6284
```

```
Epoch: 0040 | TRAIN: 0.1273 0.7490 | 0.0121 36.2838 ||VALIDATION: 0.5127 0.3726
| 0.0193 39.9457
Epoch: 0041 | TRAIN: 0.1274 0.7483 | 0.0121 35.7626 ||VALIDATION: 0.5004 0.3778
| 0.0184 41.4431
Epoch: 0042 | TRAIN: 0.1778 0.6498 | 0.0130 36.2954 ||VALIDATION: 0.4986 0.3637
| 0.0199 40.3716
Epoch: 0043 | TRAIN: 0.1629 0.6869 | 0.0129 36.3024 ||VALIDATION: 0.4788 0.3778
| 0.0204 36.6542
Epoch: 0044 | TRAIN: 0.1416 0.7307 | 0.0132 37.1072 ||VALIDATION: 0.4872 0.3805
| 0.0222 36.7270
Epoch: 0045 | TRAIN: 0.1243 0.7569 | 0.0128 35.3702 ||VALIDATION: 0.4917 0.3808
| 0.0169 46.0673
Epoch: 0046 | TRAIN: 0.1168 0.7694 | 0.0124 35.8297 ||VALIDATION: 0.5033 0.3826
| 0.0223 39.4670
Epoch: 0047 | TRAIN: 0.1128 0.7755 | 0.0125 34.7830 ||VALIDATION: 0.5138 0.3755
| 0.0189 44.5367
Epoch: 0048 | TRAIN: 0.1102 0.7782 | 0.0125 35.1318 ||VALIDATION: 0.5172 0.3796
| 0.0216 35.0466
Epoch: 0049 | TRAIN: 0.1087 0.7803 | 0.0132 35.1761 ||VALIDATION: 0.5164 0.3774
| 0.0196 36.7302
Epoch: 0050 | TRAIN: 0.1079 0.7807 | 0.0115 33.9191 ||VALIDATION: 0.5247 0.3716
| 0.0187 38.8457
Epoch: 0051 | TRAIN: 0.1078 0.7817 | 0.0117 34.8306 ||VALIDATION: 0.5342 0.3716
| 0.0174 39.6552
Epoch: 0052 | TRAIN: 0.1072 0.7824 | 0.0120 35.0610 ||VALIDATION: 0.5285 0.3755
| 0.0208 32.8390
Epoch: 0053 | TRAIN: 0.1049 0.7866 | 0.0115 34.0707 ||VALIDATION: 0.5399 0.3772
| 0.0220 44.8364
Epoch: 0054 | TRAIN: 0.1040 0.7887 | 0.0114 33.7073 ||VALIDATION: 0.5388 0.3778
| 0.0199 33.8728
Epoch: 0055 | TRAIN: 0.1043 0.7876 | 0.0113 33.6653 ||VALIDATION: 0.5400 0.3723
| 0.0188 38.9162
Epoch: 0056 | TRAIN: 0.1027 0.7894 | 0.0110 33.5291 ||VALIDATION: 0.5447 0.3767
| 0.0211 33.6164
Epoch: 0057 | TRAIN: 0.1021 0.7900 | 0.0111 33.4637 ||VALIDATION: 0.5435 0.3726
| 0.0177 35.5778
Epoch: 0058 | TRAIN: 0.1013 0.7921 | 0.0121 33.6279 ||VALIDATION: 0.5345 0.3799
| 0.0176 36.9470
Epoch: 0059 | TRAIN: 0.1036 0.7888 | 0.0112 32.7778 ||VALIDATION: 0.5408 0.3776
| 0.0213 31.4420
Epoch: 0060 | TRAIN: 0.1012 0.7924 | 0.0112 33.2757 ||VALIDATION: 0.5298 0.3739
| 0.0165 42.9021
Epoch: 0061 | TRAIN: 0.0995 0.7949 | 0.0119 32.9750 ||VALIDATION: 0.5320 0.3781
| 0.0228 37.3481
Epoch: 0062 | TRAIN: 0.0971 0.7990 | 0.0115 33.1480 ||VALIDATION: 0.5531 0.3781
| 0.0167 40.7018
Epoch: 0063 | TRAIN: 0.0959 0.8016 | 0.0105 32.2071 ||VALIDATION: 0.5564 0.3760
| 0.0197 34.2328
```

```
Epoch: 0064 | TRAIN: 0.0948 0.8035 | 0.0108 32.2383 ||VALIDATION: 0.5515 0.3774
| 0.0188 36.2876
Epoch: 0065 | TRAIN: 0.0932 0.8058 | 0.0109 32.0719 ||VALIDATION: 0.5478 0.3810
| 0.0234 35.6319
Epoch: 0066 | TRAIN: 0.0928 0.8066 | 0.0112 32.4975 ||VALIDATION: 0.5617 0.3777
| 0.0194 40.4667
Epoch: 0067 | TRAIN: 0.0917 0.8083 | 0.0108 32.8287 ||VALIDATION: 0.5622 0.3729
| 0.0197 40.1693
Epoch: 0068 | TRAIN: 0.0912 0.8091 | 0.0108 32.2909 ||VALIDATION: 0.5536 0.3744
| 0.0174 39.8022
Epoch: 0069 | TRAIN: 0.0914 0.8087 | 0.0108 31.7918 ||VALIDATION: 0.5665 0.3748
| 0.0198 36.4007
Epoch: 0070 | TRAIN: 0.0904 0.8105 | 0.0109 32.3295 ||VALIDATION: 0.5584 0.3759
| 0.0183 35.2194
Epoch: 0071 | TRAIN: 0.0889 0.8128 | 0.0102 31.6086 ||VALIDATION: 0.5656 0.3743
| 0.0175 33.2019
Epoch: 0072 | TRAIN: 0.0885 0.8133 | 0.0110 31.6361 ||VALIDATION: 0.5556 0.3789
| 0.0204 37.0924
Epoch: 0073 | TRAIN: 0.0874 0.8149 | 0.0107 31.3159 ||VALIDATION: 0.5693 0.3687
| 0.0193 34.7826
Epoch: 0074 | TRAIN: 0.0859 0.8185 | 0.0102 31.3003 ||VALIDATION: 0.5692 0.3807
| 0.0183 34.7771
Epoch: 0075 | TRAIN: 0.0860 0.8178 | 0.0103 31.6093 ||VALIDATION: 0.5741 0.3748
| 0.0182 43.8593
Epoch: 0076 | TRAIN: 0.0853 0.8197 | 0.0112 31.7269 ||VALIDATION: 0.5830 0.3750
| 0.0196 31.6178
Epoch: 0077 | TRAIN: 0.1483 0.7070 | 0.0111 31.9078 ||VALIDATION: 0.5653 0.3132
| 0.0230 53.5168
Epoch: 0078 | TRAIN: 0.1890 0.6267 | 0.0119 32.9628 ||VALIDATION: 0.4949 0.3808
| 0.0194 33.3732
Epoch: 0079 | TRAIN: 0.1075 0.7798 | 0.0109 31.6577 ||VALIDATION: 0.5226 0.3706
| 0.0188 36.2996
Epoch: 0080 | TRAIN: 0.0911 0.8066 | 0.0107 31.2675 ||VALIDATION: 0.5503 0.3735
| 0.0181 35.2135
Epoch: 0081 | TRAIN: 0.0839 0.8224 | 0.0104 31.5854 ||VALIDATION: 0.5688 0.3759
| 0.0203 33.4177
Epoch: 0082 | TRAIN: 0.0800 0.8299 | 0.0102 30.8398 ||VALIDATION: 0.5749 0.3739
| 0.0169 35.9931
Epoch: 0083 | TRAIN: 0.0777 0.8330 | 0.0098 30.4700 ||VALIDATION: 0.5919 0.3696
| 0.0174 37.0607
Epoch: 0084 | TRAIN: 0.0765 0.8352 | 0.0098 30.6686 ||VALIDATION: 0.5915 0.3732
| 0.0188 37.9473
Epoch: 0085 | TRAIN: 0.0754 0.8371 | 0.0105 30.4827 ||VALIDATION: 0.5941 0.3723
| 0.0168 35.7745
Epoch: 0086 | TRAIN: 0.0749 0.8376 | 0.0099 30.3093 ||VALIDATION: 0.5966 0.3734
| 0.0175 34.9044
Epoch: 0087 | TRAIN: 0.0740 0.8394 | 0.0095 30.1162 ||VALIDATION: 0.6042 0.3766
| 0.0198 33.9777
```

```
Epoch: 0088 | TRAIN: 0.0745 0.8384 | 0.0096 29.9573 ||VALIDATION: 0.6170 0.3673
| 0.0188 33.1941
Epoch: 0089 | TRAIN: 0.0742 0.8383 | 0.0100 30.2804 ||VALIDATION: 0.6035 0.3763
| 0.0169 34.8666
Epoch: 0090 | TRAIN: 0.0739 0.8390 | 0.0101 30.0585 ||VALIDATION: 0.6004 0.3722
| 0.0197 37.8226
Epoch: 0091 | TRAIN: 0.0738 0.8392 | 0.0098 29.6652 ||VALIDATION: 0.6073 0.3696
| 0.0180 34.5080
Epoch: 0092 | TRAIN: 0.0740 0.8387 | 0.0099 29.7259 ||VALIDATION: 0.6023 0.3734
| 0.0156 35.1581
Epoch: 0093 | TRAIN: 0.0740 0.8383 | 0.0096 29.8060 ||VALIDATION: 0.5883 0.3769
| 0.0177 33.7239
Epoch: 0094 | TRAIN: 0.0732 0.8400 | 0.0093 28.9101 ||VALIDATION: 0.6150 0.3717
| 0.0175 40.7610
Epoch: 0095 | TRAIN: 0.0731 0.8400 | 0.0091 29.1579 ||VALIDATION: 0.6040 0.3749
| 0.0184 36.5037
Epoch: 0096 | TRAIN: 0.0732 0.8404 | 0.0092 29.2137 ||VALIDATION: 0.6083 0.3711
| 0.0176 40.9713
Epoch: 0097 | TRAIN: 0.0727 0.8416 | 0.0096 29.1449 ||VALIDATION: 0.6114 0.3691
| 0.0184 36.9366
Epoch: 0098 | TRAIN: 0.0727 0.8410 | 0.0098 29.0584 ||VALIDATION: 0.6066 0.3699
| 0.0171 33.3590
Epoch: 0099 | TRAIN: 0.0717 0.8426 | 0.0095 28.7218 ||VALIDATION: 0.6128 0.3727
| 0.0180 38.4418
```

[ ]:
```
len(_loss)
```

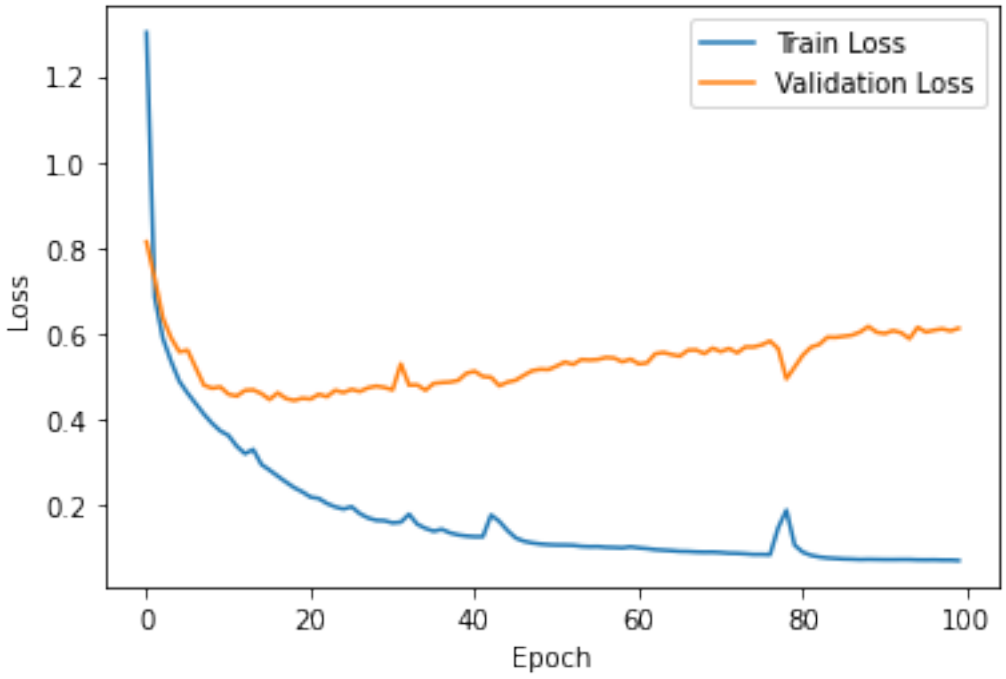
[30]:
```
len(_loss[:][0])
```

[30]:
```
10
```

[9]:
```
train_loss = [_loss[x][0] for x in range(len(_loss))]
val_loss = [_loss[x][6] for x in range(len(_loss))]
```

[10]:
```
plt.plot(train_loss, label="Train Loss")
plt.plot(val_loss, label="Validation Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
```

[10]:
```
<matplotlib.legend.Legend at 0x7f27ca83cc40>
```



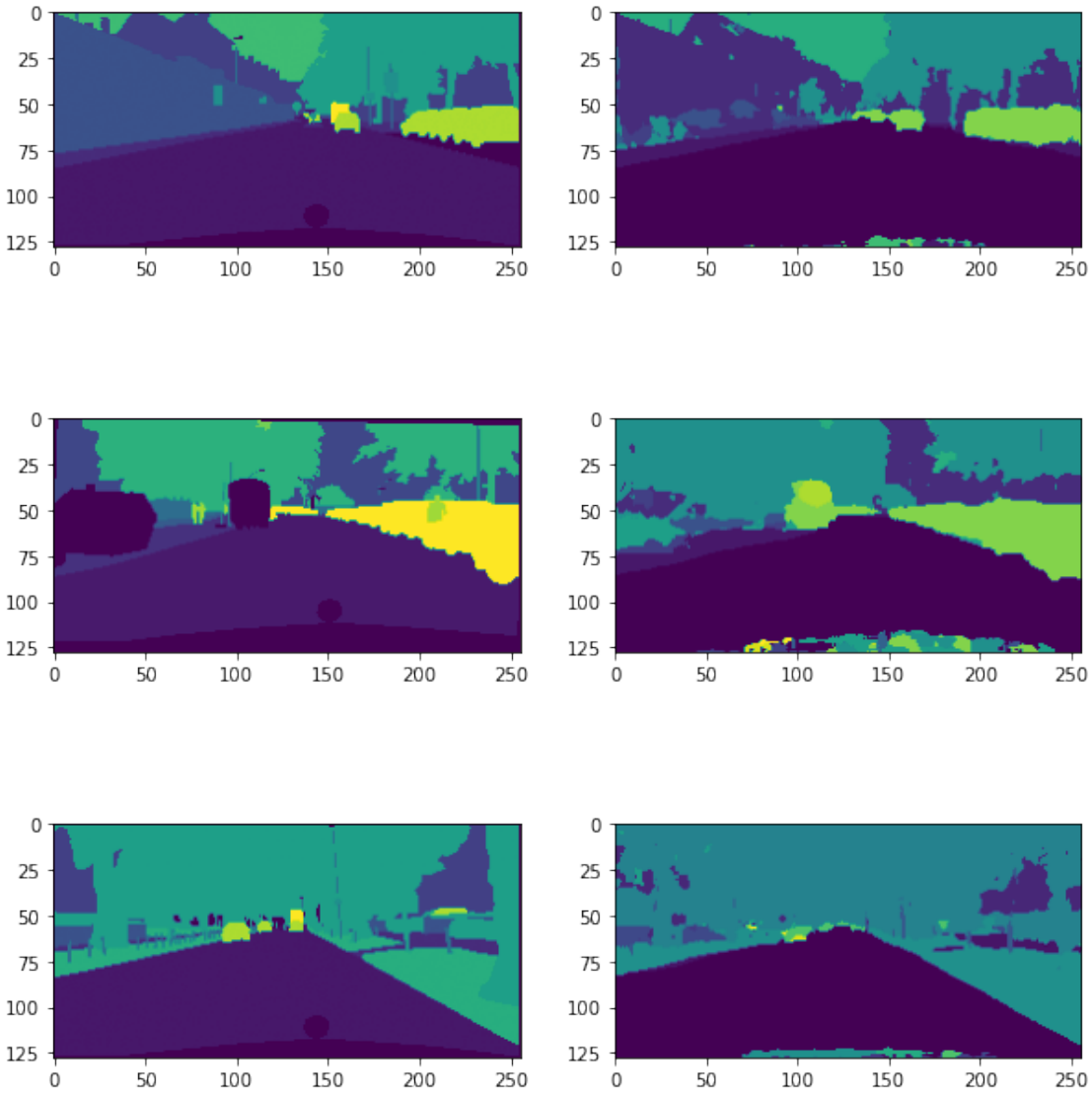
1.10.1 Testing

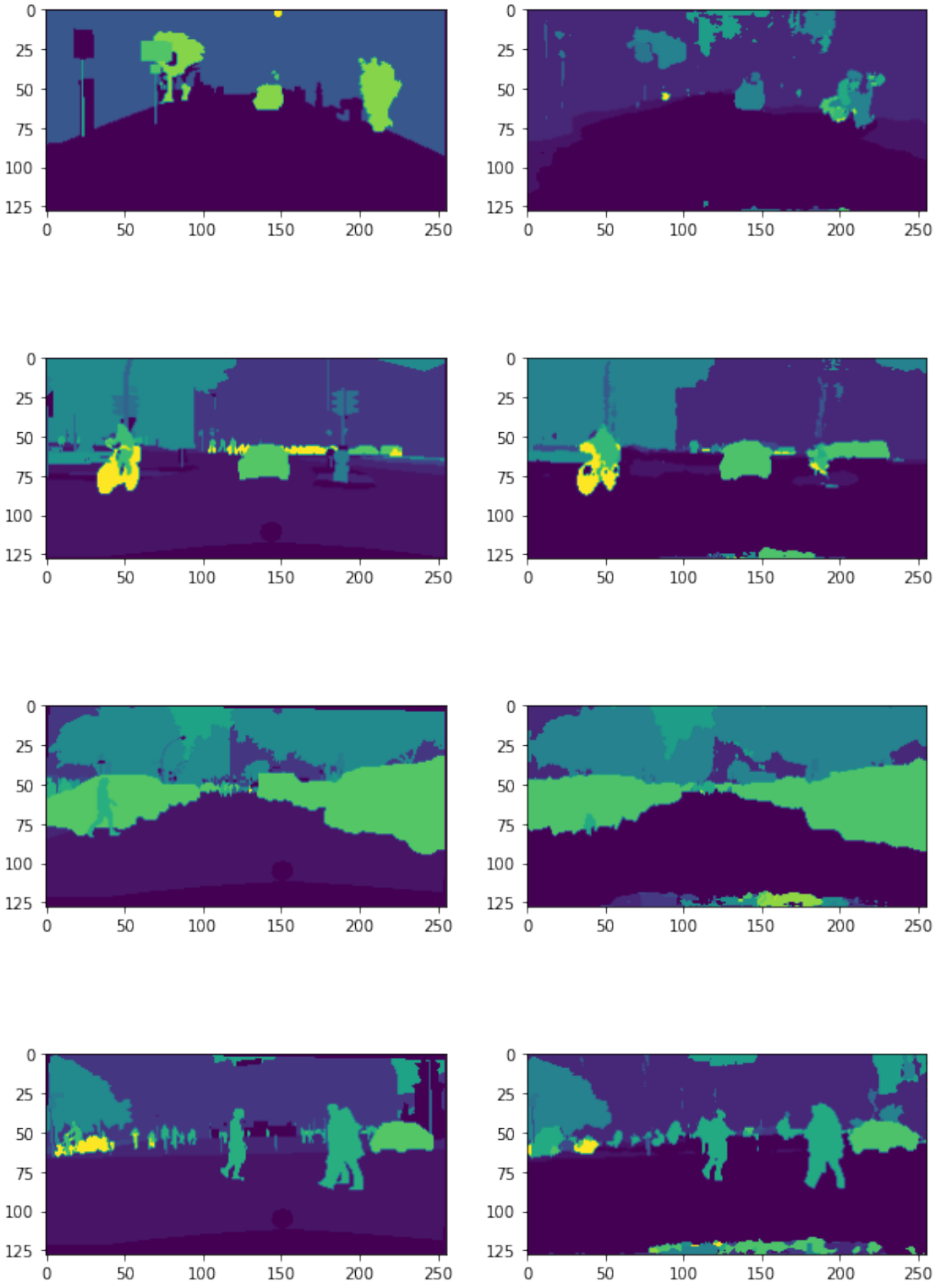
[11]:
```
VALID_DIR = 'val/'
n_files = 10
files = random.sample(os.listdir(VALID_DIR + 'image'), n_files)
```

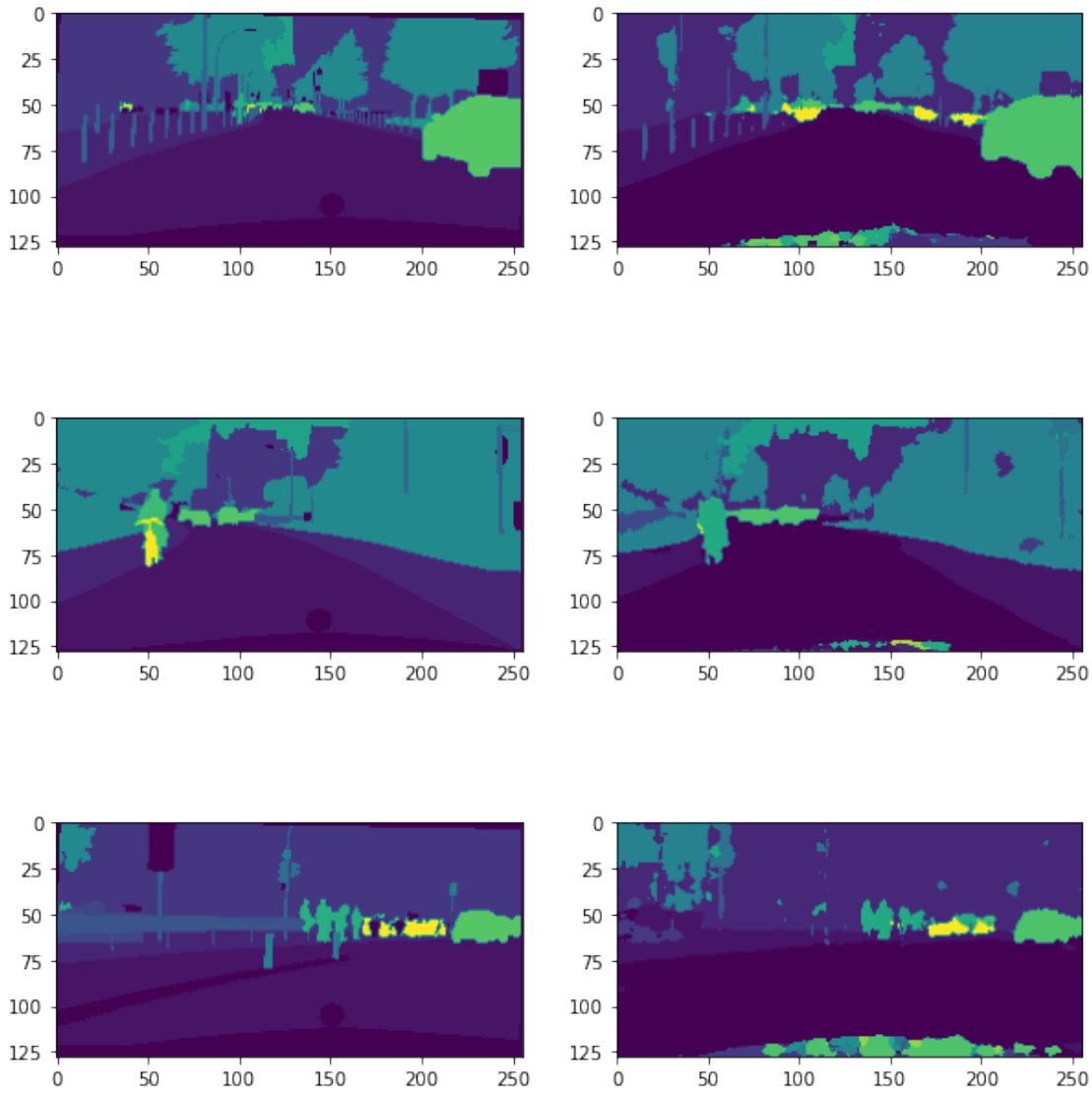
[12]:
```
segmentations = []
depths = []

for file in files:
    img = torch.from_numpy(np.expand_dims(np.moveaxis(np.load(VALID_DIR +␣
 ↪'image/' + file), -1, 0), axis = 0))
    img = img.to(device)
    prediction, _ = SegNet_MTAN(img.float())

    seg_prediction = np.moveaxis(prediction[0][0].cpu().detach().numpy(), 0,␣
 ↪-1).argmax(2)
    depth_prediction = np.moveaxis(prediction[1][0].cpu().detach().numpy(), 0,␣
 ↪-1)

    segmentations.append(seg_prediction)
    depths.append(depth_prediction)
```

1.10.2 Segmentation Labels

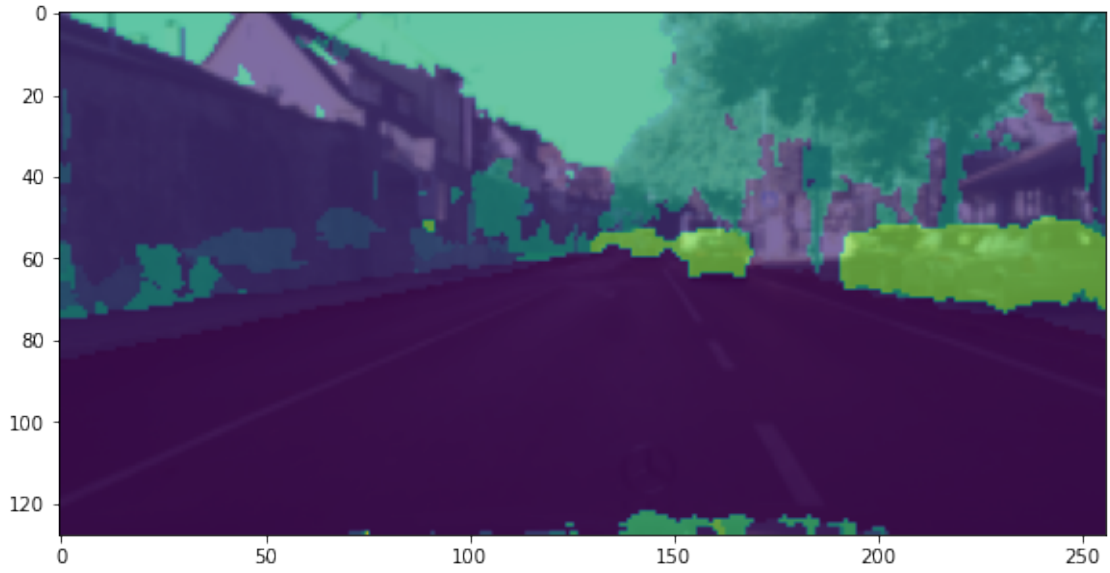
[33]:
```
i = 1
for file, seg_pred in zip(files, segmentations):
    seg_label = np.load(VALID_DIR + 'label/' + file)

    plt.figure(figsize = (10,5))
    plt.subplot(1, 2, 1)
    plt.imshow(seg_label)

    plt.subplot(1, 2, 2)
    plt.imshow(seg_pred)
#     plt.savefig(f"segmentation/seg_demo_{i}.png")
    i += 1
```

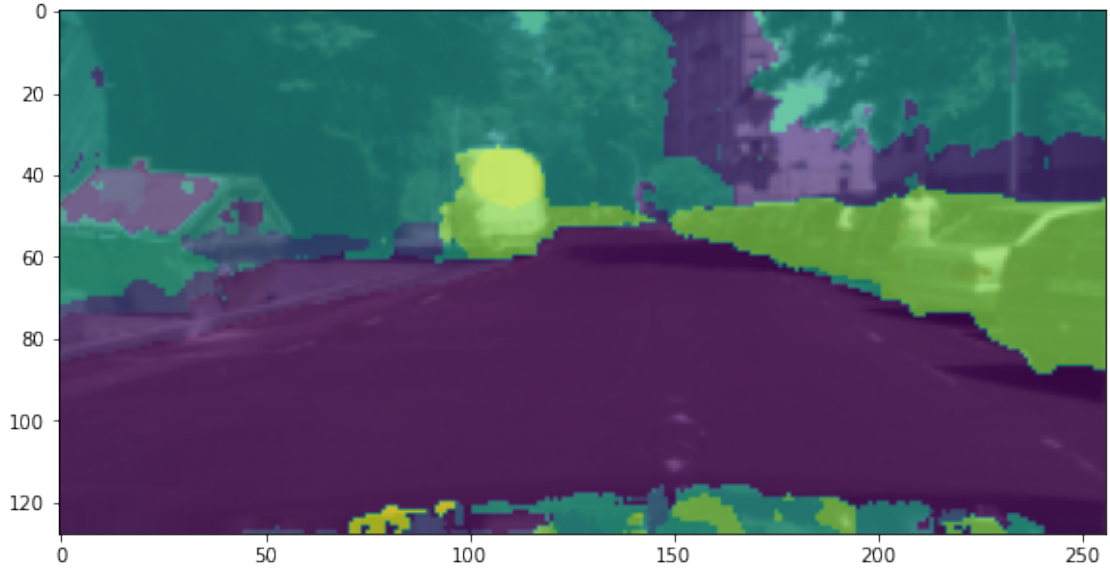


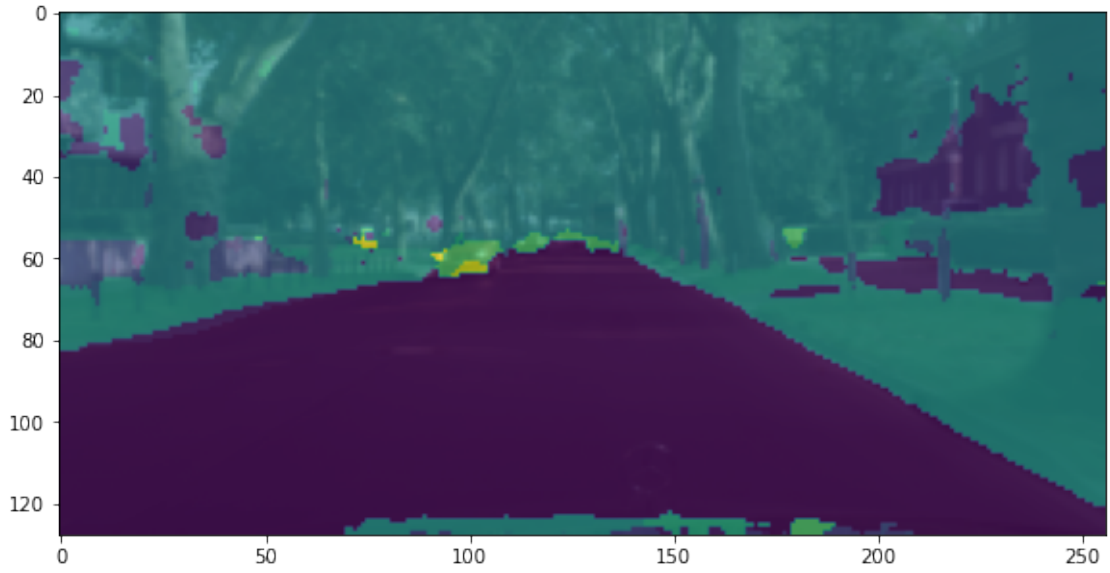

0
25
50
75
100
125
0
50
100
150
200
250

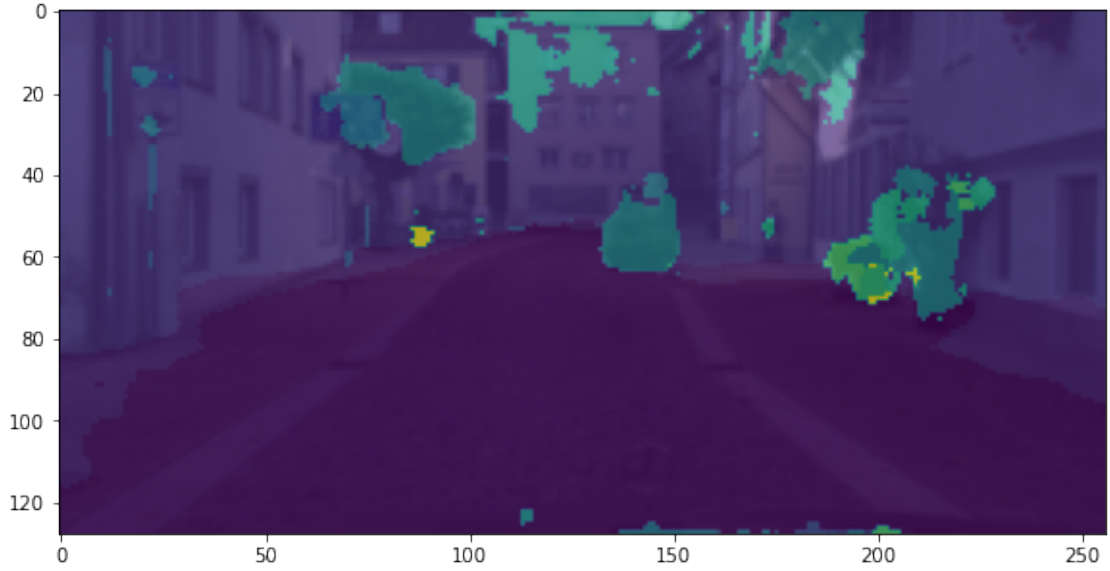


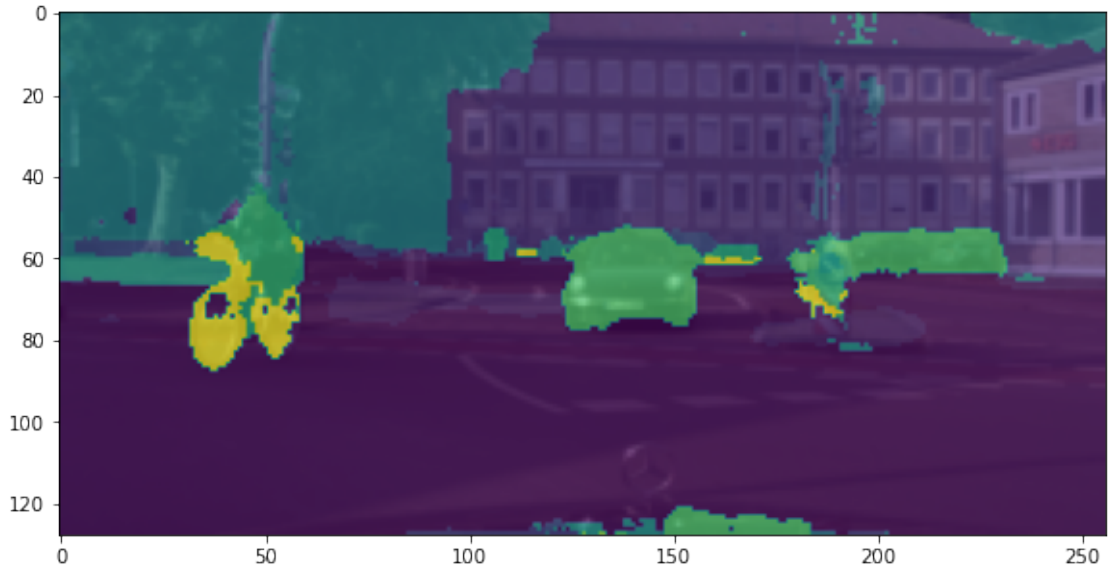
[34]:
```
i = 1
for file, seg_pred in zip(files, segmentations):
    img = np.load(VALID_DIR + 'image/' + file)

    plt.figure(figsize = (10,5))
    plt.imshow(img)
    plt.imshow(seg_pred, alpha = 0.7)
#     plt.savefig(f"segmentation/seg_overlay_{i}.png")
    i += 1
```

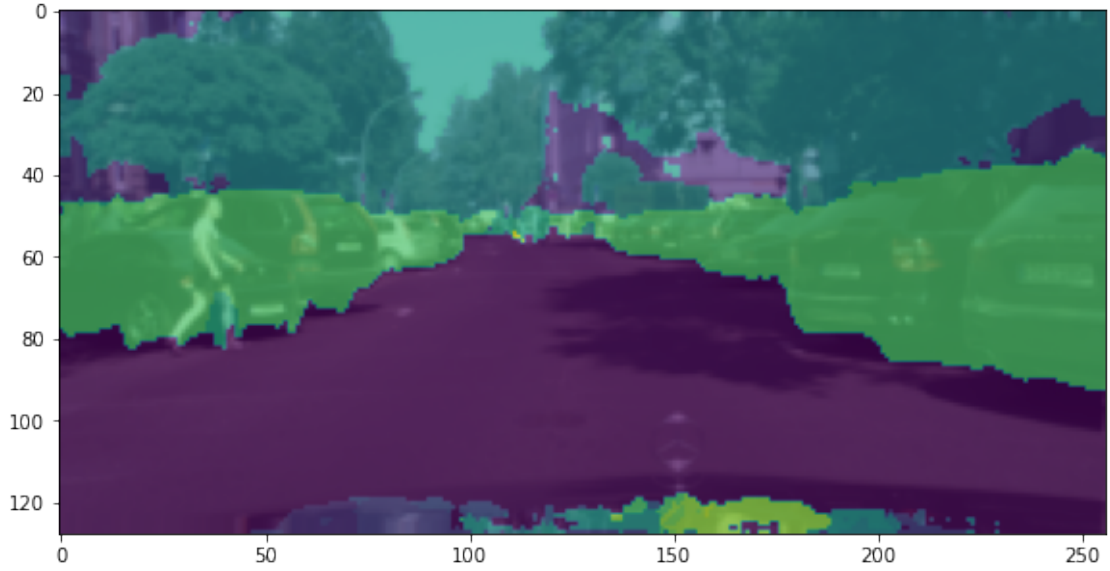

0
20
40
60
80
100
120
0
50
100
150
200
250

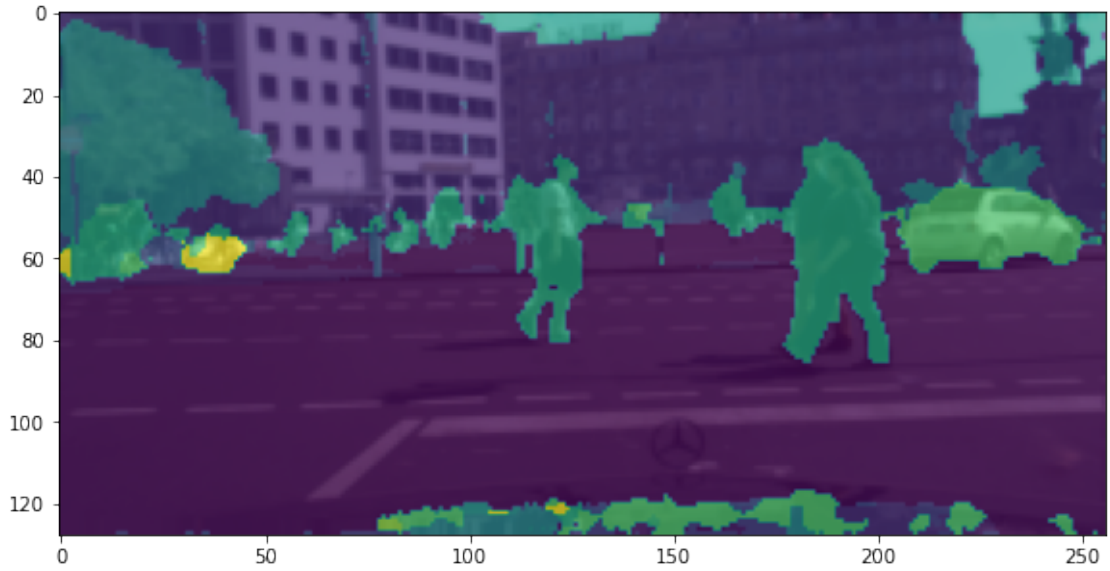

0
20
40
60
80
100
120
0
50
100
150
200
250

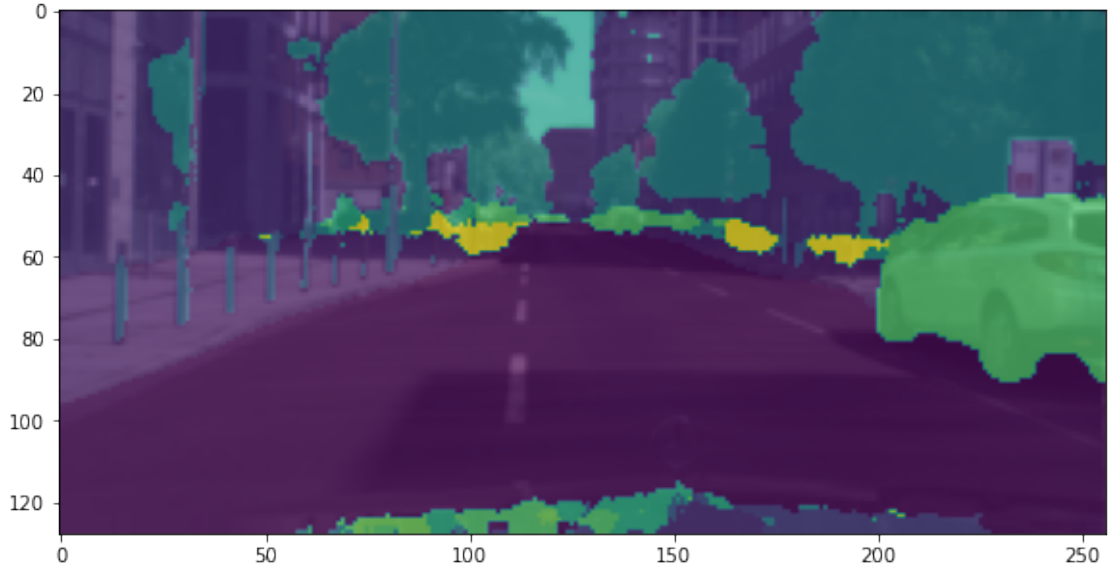

0
20
40
60
80
100
120
0
50
100
150
200
250

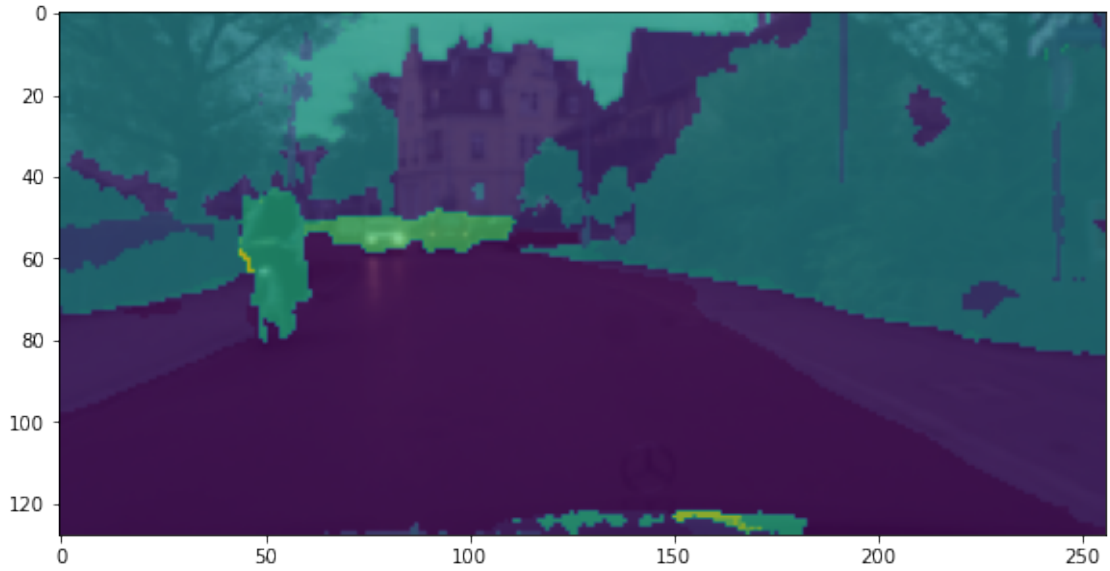

0
20
40
60
80
100
120
0
50
100
150
200
250

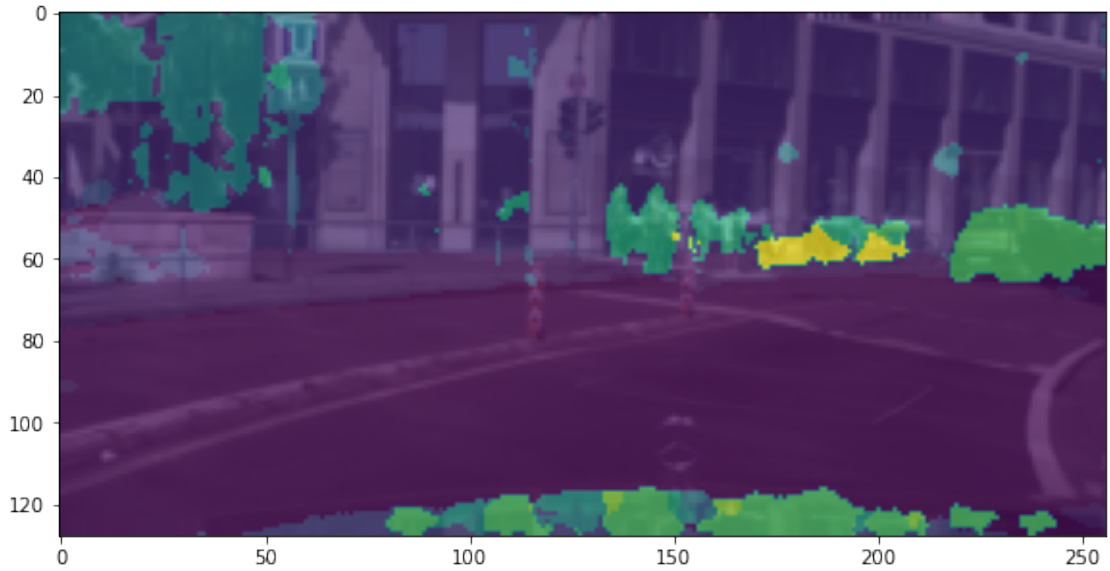

0
20
40
60
80
100
120
0
50
100
150
200
250


0
20
40
60
80
100
120
0
50
100
150
200
250


0
20
40
60
80
100
120
0
50
100
150
200
250


0
20
40
60
80
100
120
0
50
100
150
200
250

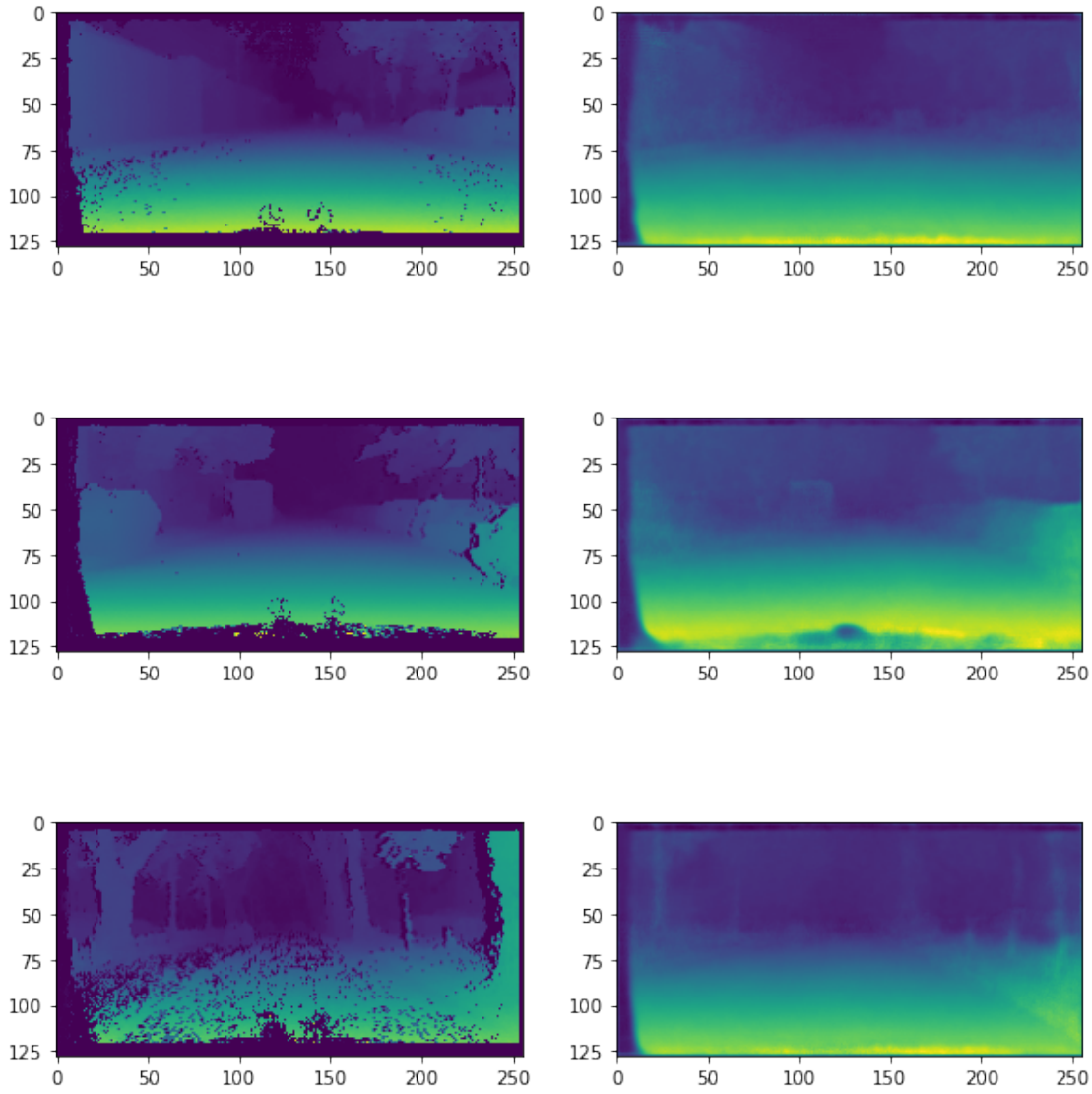


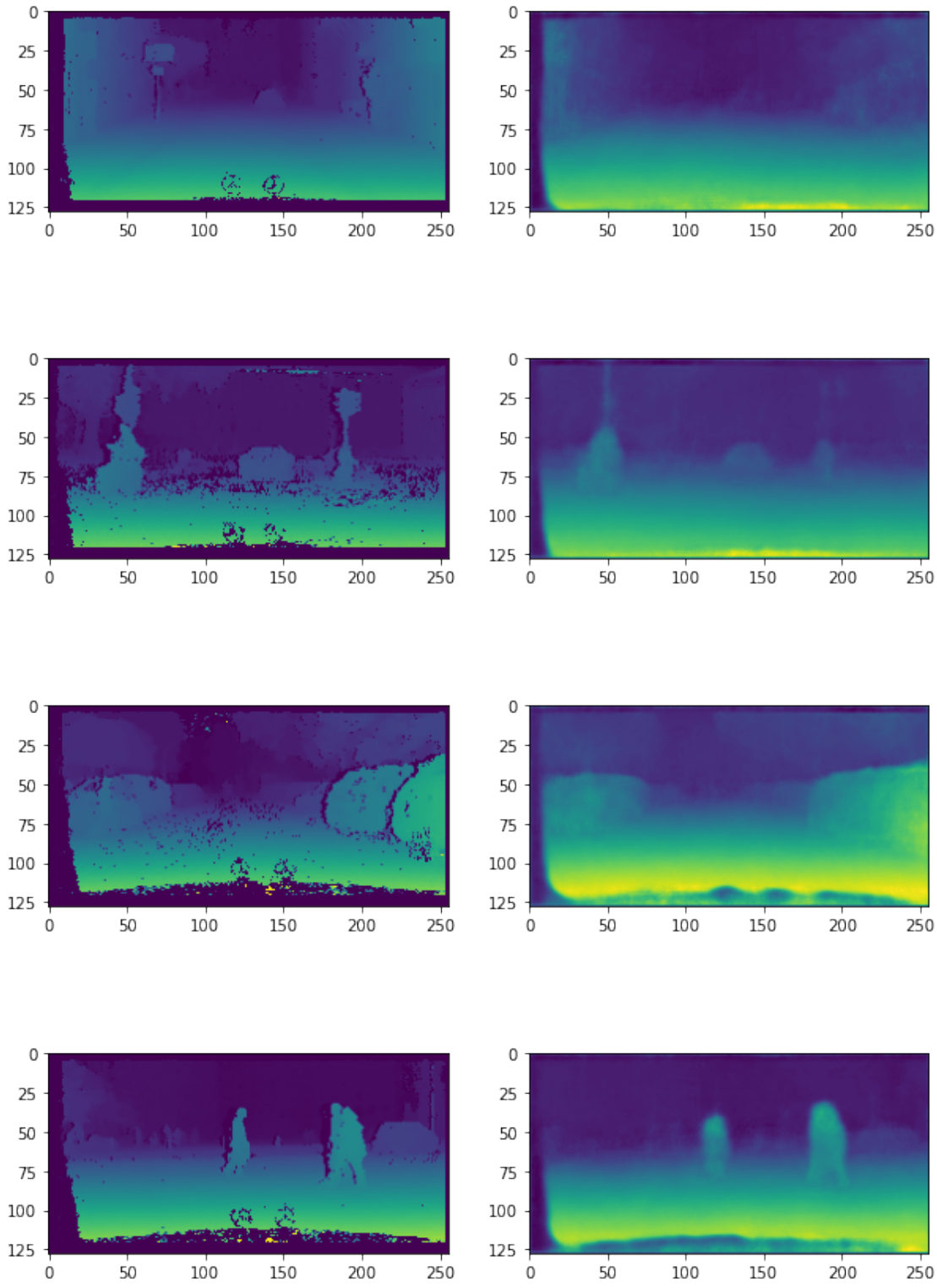


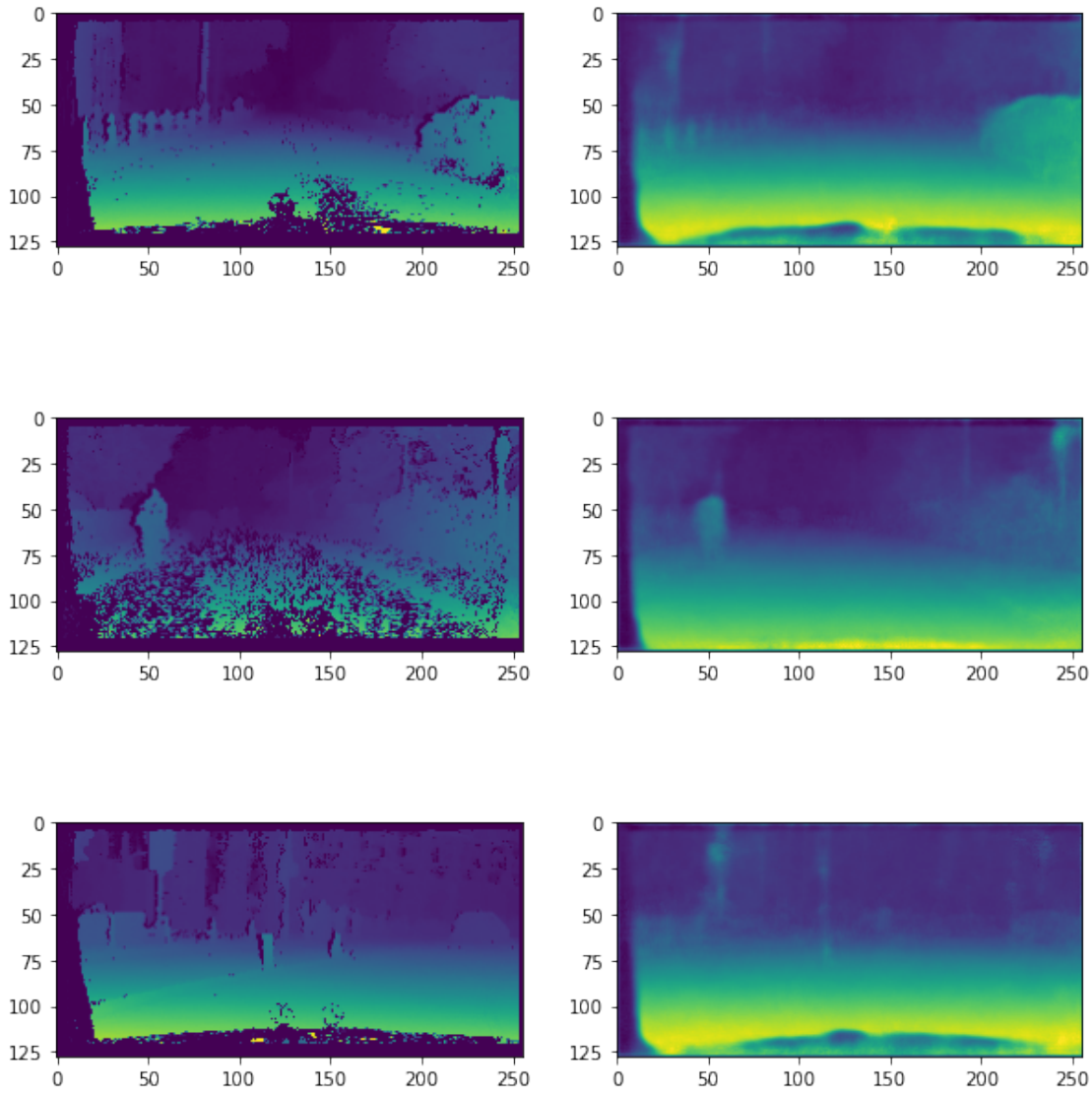
1.10.3 Depth Maps

[35]:
```
i = 1
for file, depth_pred in zip(files, depths):
    depth_label = np.load(VALID_DIR + 'depth/' + file)

    plt.figure(figsize = (10,5))
    plt.subplot(1, 2, 1)
```

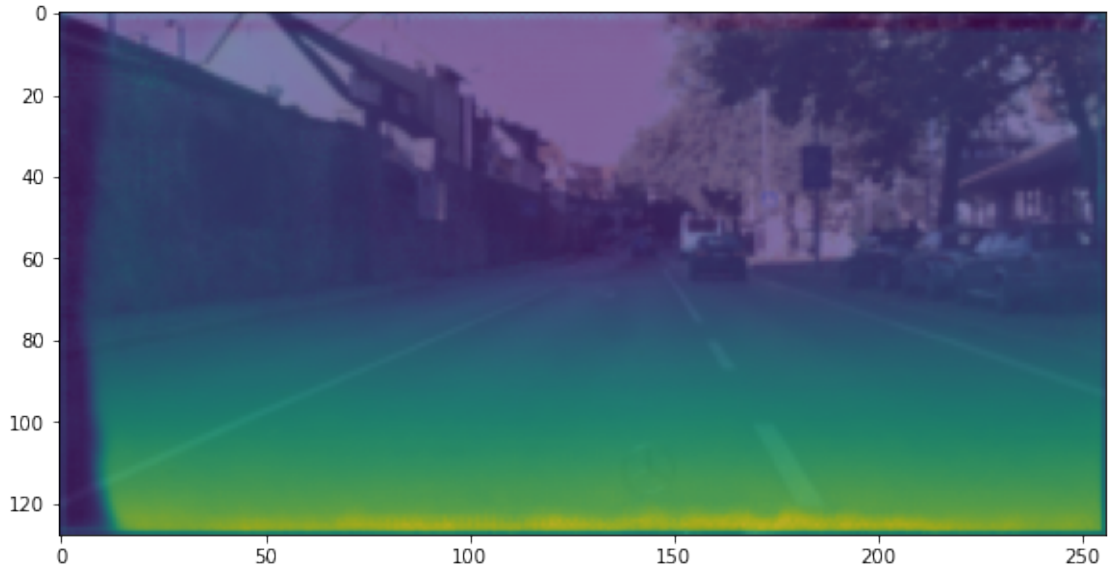
```
    plt.imshow(depth_label)

    plt.subplot(1, 2, 2)
    plt.imshow(depth_pred)
#     plt.savefig(f"depth_est/depth_demo_{i}.png")
    i += 1
```

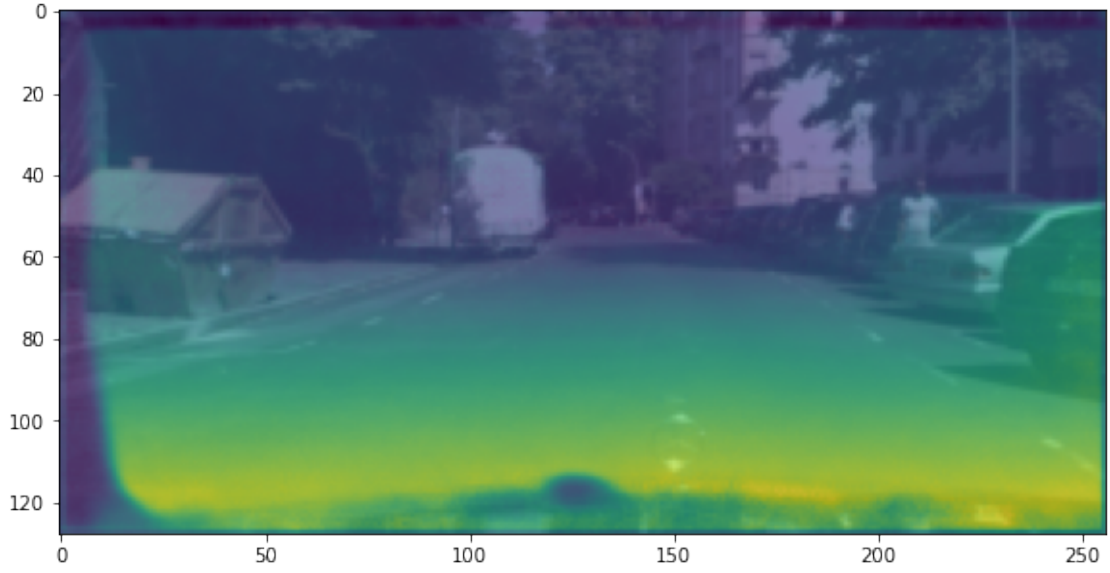


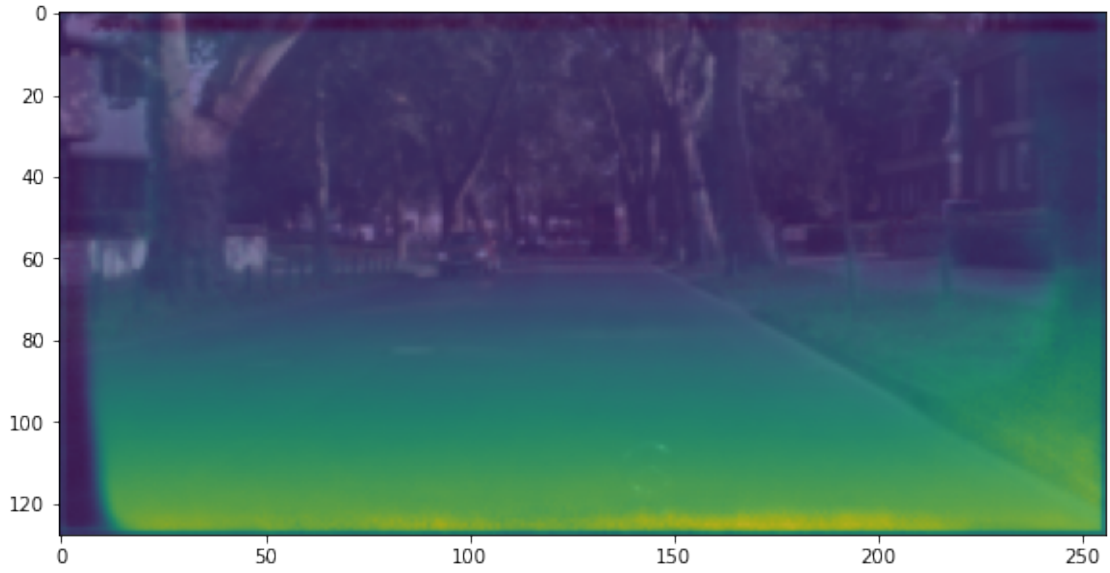

0
25
50
75
100
125
0
50
100
150
200
250

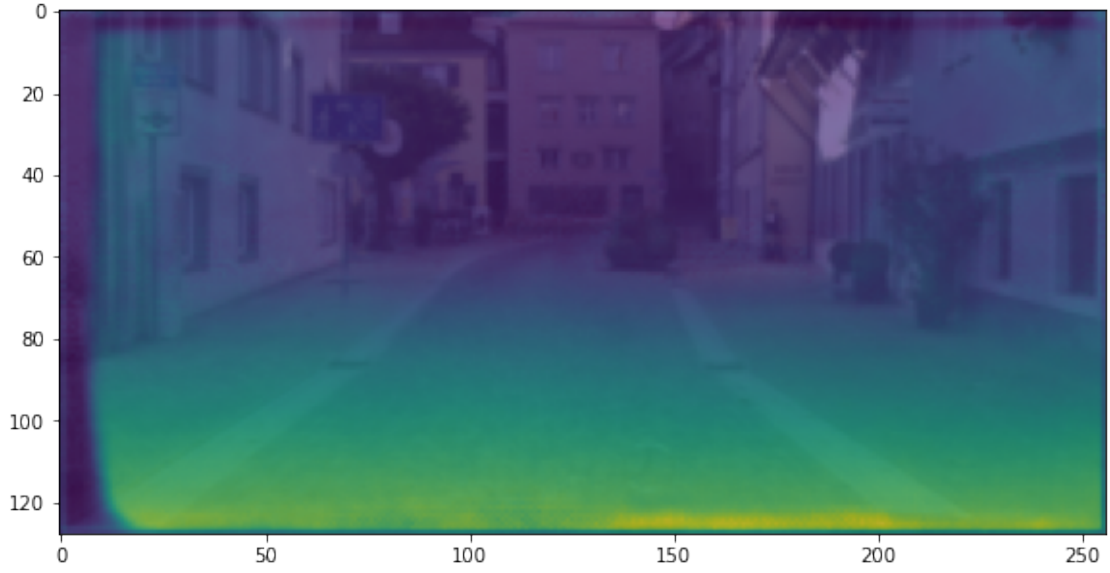


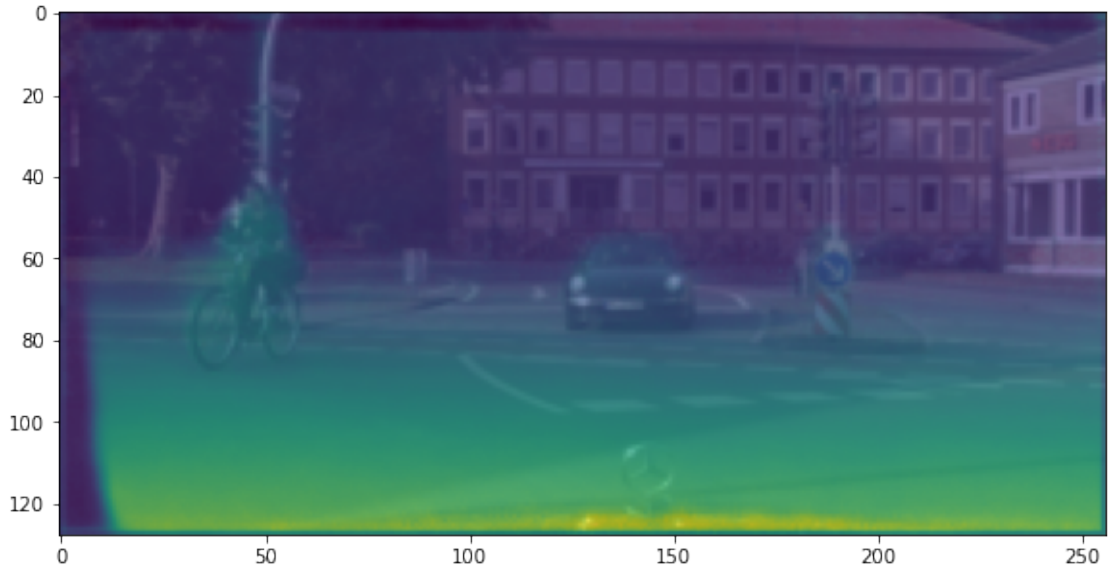
[36]:
```
i = 1
for file, depth_pred in zip(files, depths):
    img = np.load(VALID_DIR + 'image/' + file)

    plt.figure(figsize = (10,5))
    plt.imshow(img)
    plt.imshow(depth_pred, alpha = 0.7)
#     plt.savefig(f"depth_est/depth_overlay_{i}.png")
    i += 1
```

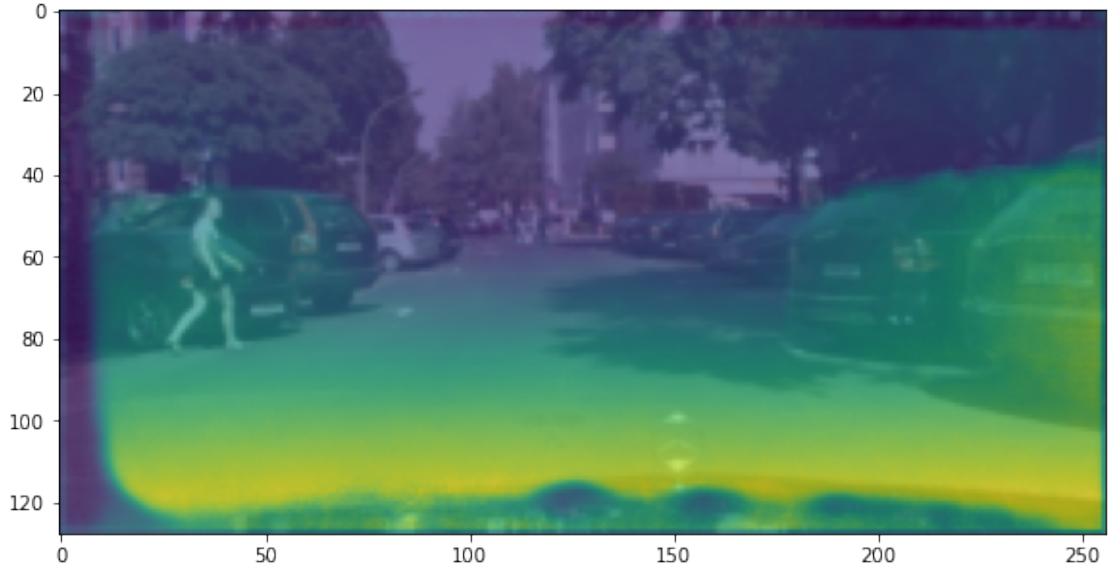

0
20
40
60
80
100
120
0
50
100
150
200
250

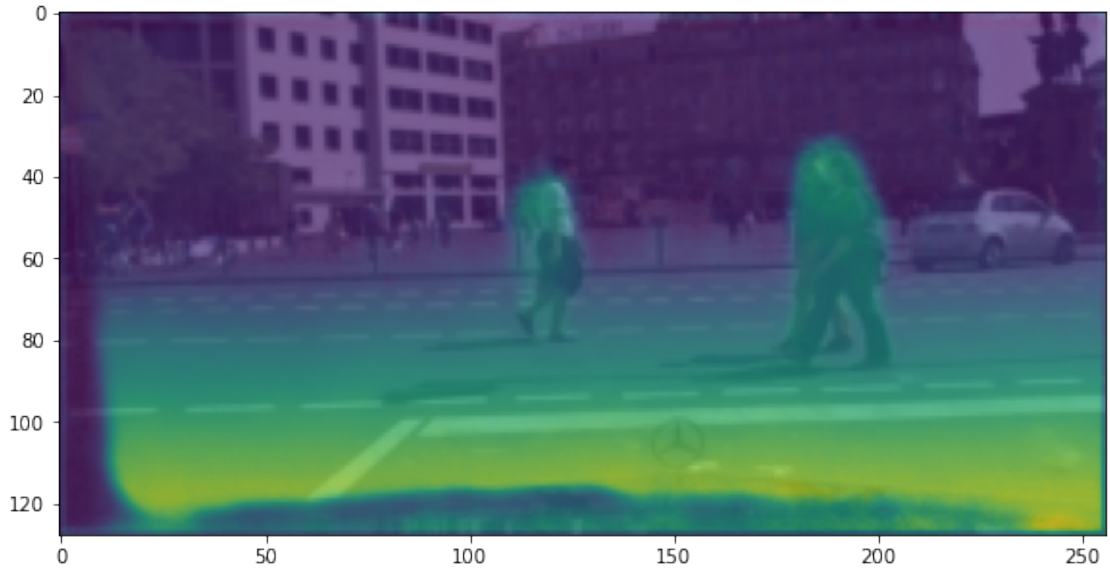

0
20
40
60
80
100
120
0
50
100
150
200
250

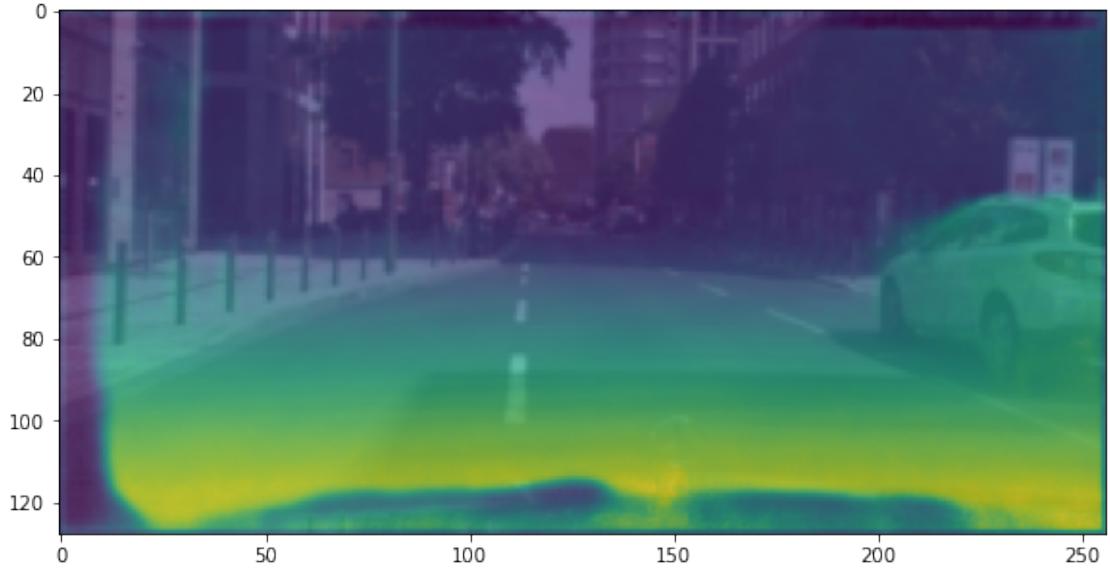

0
20
40
60
80
100
120
0
50
100
150
200
250

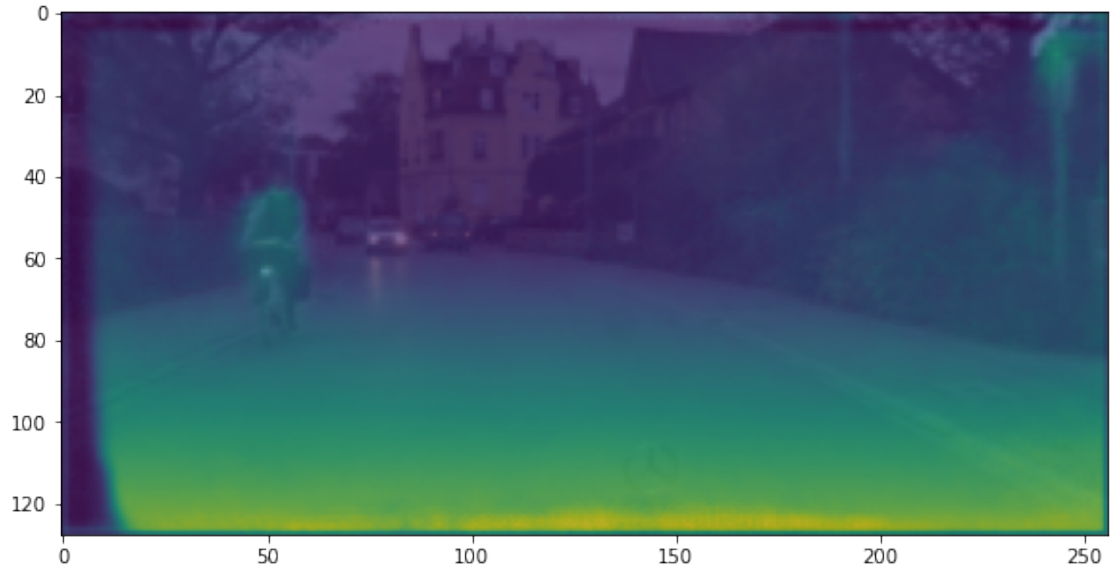

0
20
40
60
80
100
120
0
50
100
150
200
250

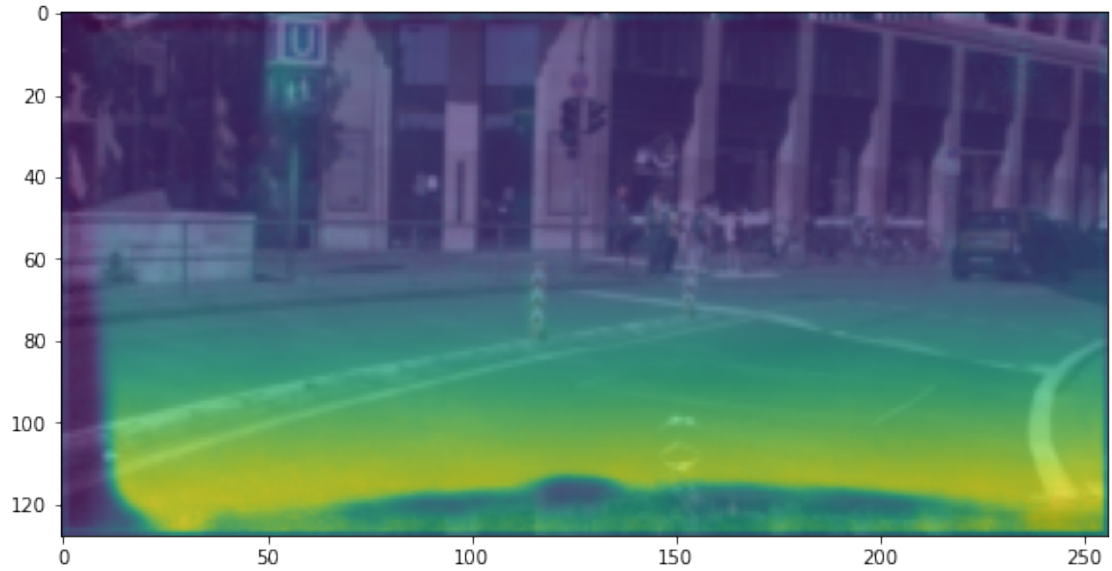

0
20
40
60
80
100
120
0
50
100
150
200
250


0
20
40
60
80
100
120
0
50
100
150
200
250


0
20
40
60
80
100
120
0
50
100
150
200
250


0
20
40
60
80
100
120
0
50
100
150
200
250


0
20
40
60
80
100
120
0
50
100
150
200
250


0
20
40
60
80
100
120
0
50
100
150
200
250

[ ]: