Self-Supervised Single-Image (Monocular) Depth-Estimation

CS 484: Computational Vision

By: Matthew Lam

Email: m39lam@uwaterloo.ca

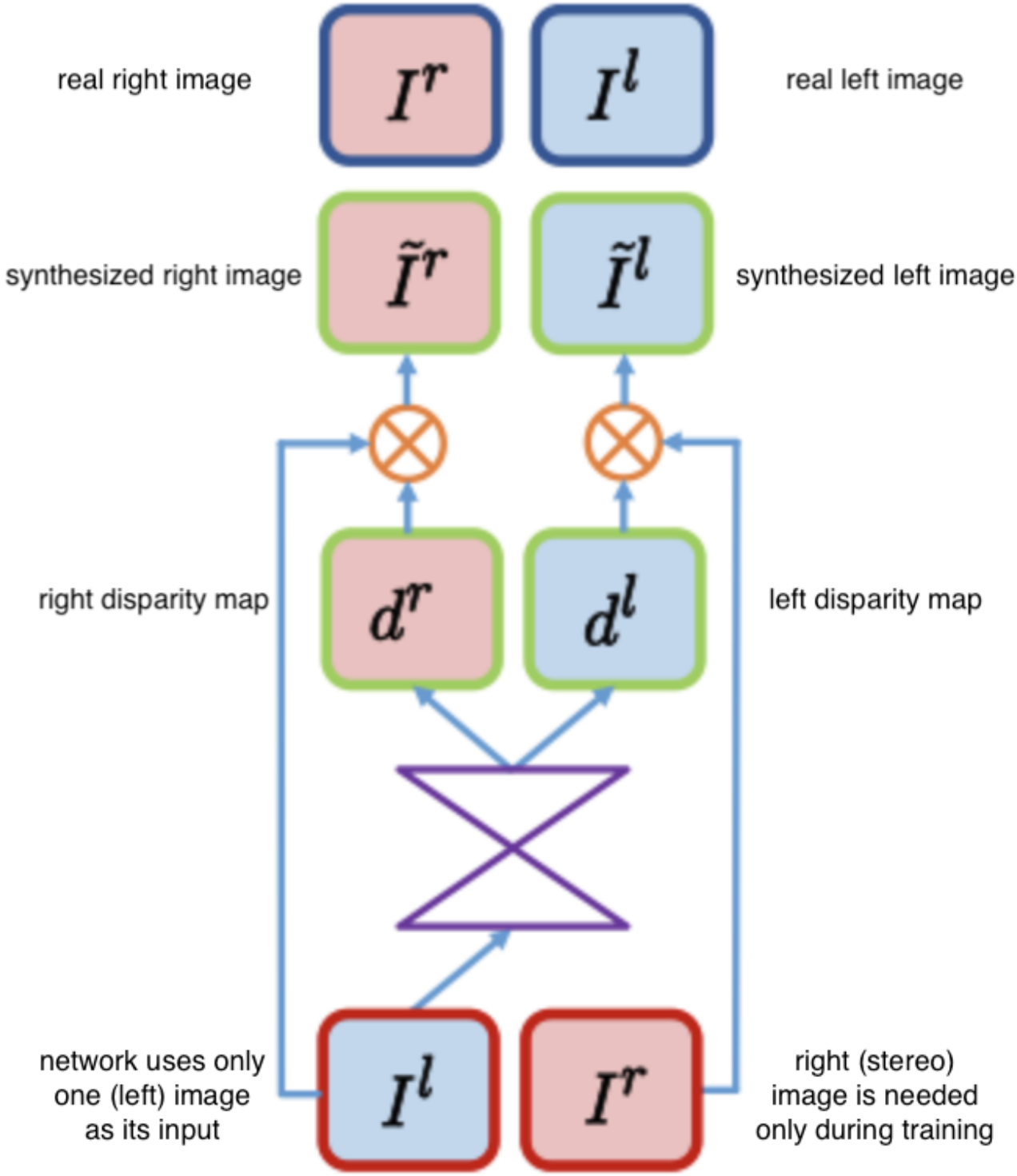
Abstract

This notebook explores self-supervised monocular depth estimation networks proposed by Godard et al. in their 2017 paper, "Unsupervised Monocular Depth Estimation with Left-Right Consistency" (https://arxiv.org/pdf/1609.03677v3.pdf). Most existing techniques treat depth prediction as a supervised learning problem, which requires large amounts of ground truth depth data for training. Deep learning for monocular depth estimation faces this same problem as semantic segmentation, where there is a lack of readily available and accurate ground truth due to cumbersome labelling, which impedes the application of deep learning to this problem. This paper proposes a self-supervised method with a unique loss function that allows the convolutional neural network to perform single-image depth estimation without a ground truth by replacing the explicit depth data during training with readily available binocular stereo images.

The implementation follows the official "Monodepth" repository on GitHub (https://github.com/mrharicot/monodepth). This approach uses the model's architecture in the paper for the decoder and transfer learning using ResNet for the encoder with the KITTI dataset to extract features from the RGB image correlated with depth information to allow the network to estimate depth from a single image. The convolutional neural network gets trained to predict the disparity map between the left and right images using only the left image as input. The scene's depth gets reconstructed using the estimated disparity map and the camera's intrinsic (focal length and optical center) and extrinsic (position and orientation in the world coordinate system) parameters. The depth gets computed using the formula:

$$Depth = \frac{(b \cdot f)}{d}$$

where b (baseline) is the distance between the two cameras, f is the camera's focal length, and d is the disparity of the corresponding point in the stereo pair.



During training, the network learns to predict both the left-to-right and right-to-left disparity maps, which enforces the consistency between the two disparity maps to reduce artifacts in the final image. Additionally, two ground truths are used during training, one for the left image and one for the right image, by creating synthesized images using the disparity map and the corresponding image as shown in the diagram above. The training loss consists of comparing the target images, synthesized images and disparity maps using photo-consistency (appearance matching) and disparity map regularization (disparity smoothness and left-right disparity consistency).

In [ ]:
```
%matplotlib inline
import numpy as np
```

```
import matplotlib.pyplot as plt
import matplotlib.image as image
from tqdm import tqdm
import os
import random
import copy
from PIL import Image

import torch
import torch.nn.functional as F
from torch import nn
from torch.utils.data import DataLoader
import torchvision.models as models
import torchvision.transforms as transforms
import torchvision.transforms.functional as tF

from torch.utils.data import Dataset, DataLoader, ConcatDataset
```

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

In [ ]:
```
ROOT_DIR = '/content/drive/My Drive'
TRAIN_DIR = ROOT_DIR + '/train_data'
VAL_DIR = ROOT_DIR + '/val_data'
TEST_DIR = ROOT_DIR + '/test_data'
IMAGES_DIR = ROOT_DIR + '/images'
WEIGHTS_DIR = ROOT_DIR + '/weights'

device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

Dataset and Transformations

The dataset samples a small subset of the KITTI dataset.

Training Set consists of 3318 left-right image pairs:

- https://s3.eu-central-1.amazonaws.com/avg-kitti/raw_data/2011_09_26_drive_0117/2011_09_26_drive_0117_sync.zip
- https://s3.eu-central-1.amazonaws.com/avg-kitti/raw_data/2011_09_28_drive_0001/2011_09_28_drive_0001_sync.zip
- https://s3.eu-central-1.amazonaws.com/avg-kitti/raw_data/2011_09_29_drive_0026/2011_09_29_drive_0026_sync.zip
- https://s3.eu-central-1.amazonaws.com/avg-kitti/raw_data/2011_09_30_drive_0034/2011_09_30_drive_0034_sync.zip
- https://s3.eu-central-1.amazonaws.com/avg-kitti/raw_data/2011_10_03_drive_0042/2011_10_03_drive_0042_sync.zip

Validation Set consists of 661 left-right image pairs:

- https://s3.eu-central-1.amazonaws.com/avg-kitti/raw_data/2011_09_28_drive_0045/2011_09_28_drive_0045_sync.zip

- https://s3.eu-central-1.amazonaws.com/avg-kitti/raw_data/2011_09_29_drive_0004/2011_09_29_drive_0004_sync.zip
- https://s3.eu-central-1.amazonaws.com/avg-kitti/raw_data/2011_09_30_drive_0016/2011_09_30_drive_0016_sync.zip

Test Set consists of 1 left-right image pairs:

- https://vision.middlebury.edu/stereo/data/scenes2021/data/chess1

The KITTI dataset images were downloaded and extracted from the links above and uploaded to Google Drive in the following file structure:

- /content/drive/My Drive
  - train_data
    - 2011_09_26_drive_0117
      - left_images
        - 0000000000.png
        - ...
      - right_images
        - 0000000000.png
        - ...
    - ...
  - val_data
    - 2011_09_28_drive_0045
      - left_images
        - 0000000000.png
        - ...
      - right_images
        - 0000000000.png
        - ...
    - ...
  - test_data
    - chess
      - left_images
        - chessL.png
      - right_images
        - chessR.png

Each of the dataloaders finds the images in the left_images and right_images folders under the associated parent folder under train_data, val_data and test_data respectively.

Note: test_data only contains one left-right image pair used for testing the neural network. This test data comes from the 2021 mobile datasets for chess1 by Scharstein et al.

There are six different transformations that will be applied during training including: random flip, resizing to 256 x 512 (input size), random gamma shift, random brightness shift, random color shift or

saturation. Resizing the image to a smaller size makes training faster. Note that other transformations like cropping doesn't make sense as the network is trying to construct disparity maps and requires the left and right images to be similar.

In [ ]:

```
# Create joint transformations for a sample from the dataset.
# It is either a imL or (imL, imR).

class JointToTensor(object):
    def __call__(self, target):
        if isinstance(target, list) or isinstance(target, tuple):
            return tF.to_tensor(target[0]), tF.to_tensor(target[1])
        return tF.to_tensor(target)

class JointRandomFlip(object):
    def __call__(self, target):
        if random.random() > 0.5:
            if isinstance(target, list) or isinstance(target, tuple):
                return tF.hflip(target[0]), tF.hflip(target[1])
            return tF.hflip(target)
        return target

class JointResize(object):
    def __init__(self, size=(256, 512)):
        self.resize = transforms.Resize(size)

    def __call__(self, target):
        if isinstance(target, list) or isinstance(target, tuple):
            return self.resize(target[0]), self.resize(target[1])
        return self.resize(target)

class JointRandomGammaShift(object):
    def __init__(self, gamma_low=0.8, gamma_high=1.3):
        self.gamma_low = gamma_low
        self.gamma_high = gamma_high

    def __call__(self, target):
        if isinstance(target, list) or isinstance(target, tuple):
            random_gamma = np.random.uniform(self.gamma_low, self.gamma_high)
            augL = target[0] ** random_gamma
            augR = target[1] ** random_gamma
            return augL, augR
        return target

class JointRandomBrightness(object):
    def __init__(self, brightness_low=0.5, brightness_high=2.0):
        self.brightness_low = brightness_low
        self.brightness_high = brightness_high

    def __call__(self, target):
        if isinstance(target, list) or isinstance(target, tuple):
            random_brightness = np.random.uniform(self.brightness_low, self.brightness_h
            augL = target[0] * random_brightness
            augR = target[1] * random_brightness
            return augL, augR
        return target

class JointRandomColorShift(object):
    def __init__(self, color_low=0.8, color_high=1.2):
        self.color_low = color_low
        self.color_high = color_high
```

```
    def __call__(self, target):
        if isinstance(target, list) or isinstance(target, tuple):
            random_colors = np.random.uniform(self.color_low, self.color_high, 3)
            augL = target[0]
            augR = target[1]
            # Randomly shifts colors for each channel
            for i in range(3):
                augL[i, :, :] *= random_colors[i]
                augR[i, :, :] *= random_colors[i]

            return augL, augR
        return target

class JointSaturate(object):
    def __call__(self, target):
        if isinstance(target, list) or isinstance(target, tuple):
            augL = torch.clamp(target[0], 0, 1)
            augR = torch.clamp(target[1], 0, 1)
            return augL, augR
        return target
```

In [ ]:
```
# Data Transformations
# Only augment randomized resizing, gamma shifts, brightness, color shift, saturate
# and flipping during training.
train_transforms = transforms.Compose([
                                JointToTensor(),
                                JointResize(),
                                JointRandomGammaShift(),
                                JointRandomBrightness(),
                                JointRandomColorShift(),
                                JointSaturate(),
                                JointRandomFlip()])

val_transforms = transforms.Compose([
                                JointToTensor(),
                                JointResize()])

test_transforms = transforms.Compose([
                                JointToTensor(),
                                JointResize()])
```

In [ ]:
```
class KITTIDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        """
        Creates a dataset.
        Args:
            root_dir (string): root directory of the KITTI Dataset
            transforms: A transform that takes in either a imL or a (imL, imR)
                tuple and returns a transformed version.
        """
        self.transform = transform
        self.imagesL = self.sortFilenames(os.path.join(root_dir, 'left_images'))
        self.imagesR = self.sortFilenames(os.path.join(root_dir, 'right_images'))
        assert len(self.imagesL) == len(self.imagesR)

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

    def __getitem__(self, idx):
        """
        Args:
```

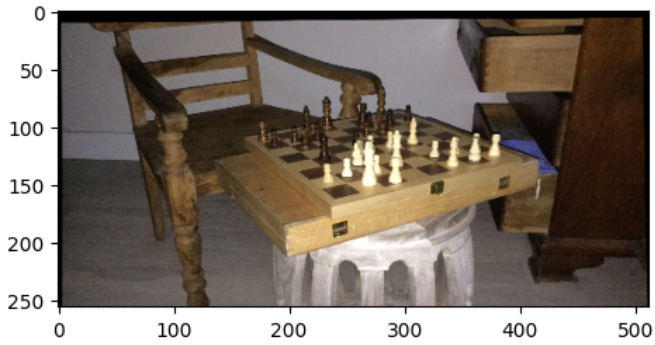
```
            idx (int): corresponding index of an image in the dataset.
        """
        imL = Image.open(self.imagesL[idx])
        imR = Image.open(self.imagesR[idx])

        if self.transform:
            return self.transform((imL, imR))
        return imL, imR

    def sortFilenames(self, dir):
        """
        Args:
            dir (str): directory where the file names are sorted
        """
        return sorted([os.path.join(dir, file) for file in os.listdir(dir)])
```

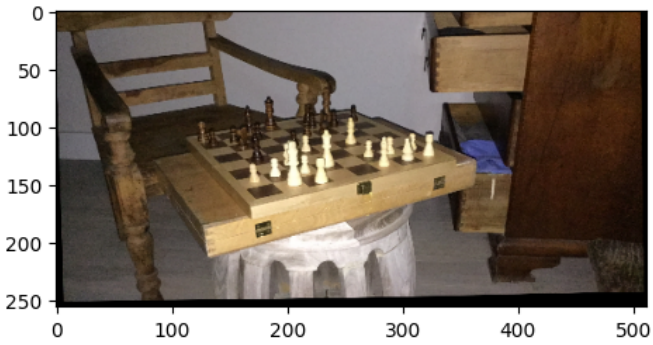
In [ ]:
```
# Generates the training dataset and dataloader
train_datasets = []
for data_dir in os.listdir(TRAIN_DIR):
    train_dataset = KITTIDataset(os.path.join(TRAIN_DIR, data_dir), train_transforms)
    train_datasets.append(train_dataset)
train_dataset = ConcatDataset(train_datasets)
print('Training dataset with', len(train_dataset), 'images')
train_loader = DataLoader(train_dataset, batch_size=24, num_workers=4, shuffle=True)

# Generates the validation dataset and dataloader
val_datasets = []
for data_dir in os.listdir(VAL_DIR):
    val_dataset = KITTIDataset(os.path.join(VAL_DIR, data_dir), val_transforms)
    val_datasets.append(val_dataset)
val_dataset = ConcatDataset(val_datasets)
print('Validation dataset with', len(val_dataset), 'images')
val_loader = DataLoader(val_dataset, batch_size=24, num_workers=4, shuffle=True)

# Generates the test dataset and dataloader
test_datasets = []
for data_dir in os.listdir(TEST_DIR):
    test_dataset = KITTIDataset(os.path.join(TEST_DIR, data_dir), test_transforms)
    test_datasets.append(test_dataset)
test_dataset = ConcatDataset(test_datasets)
print('Validation dataset with', len(test_dataset), 'images')
test_loader = DataLoader(test_dataset, batch_size=24, num_workers=4, shuffle=True)
```

```
Training dataset with 3318 images
Validation dataset with 661 images
Validation dataset with 1 images
```

In [ ]:
```
imL, imR = test_loader.dataset[0]
fig = plt.figure(figsize=(12,6))
ax1 = fig.add_subplot(2,2,1)
ax1.imshow(imL.permute(1, 2, 0))
ax2 = fig.add_subplot(2,2,2)
ax2.imshow(imR.permute(1, 2, 0))
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7f12204f7700>
```





Monocular Depth Loss Function

The proposed loss function is composed of the photo-consistency loss (appearance matching) and disparity map regularization loss (disparity smoothness and left-right disparity consistency).

The appearance matching loss C_{ap} compares the reconstructed image with the original image using SSIM (structural similarity index measure) and L1 norm.

$$C_{ap} = \frac{1}{N}\sum_{ij}\alpha\frac{1-SSIM(I_{ij}\tilde{I_{ij}})}{2} + (1-\alpha)\|I_{ij}-\tilde{I_{ij}}\|$$

Here, N is the number of pixels, I_{ij} is the input image, $\tilde{I_{ij}}$ is the reconstructed image and $\alpha = 0.85$.

The disparity smoothness loss C_{ds} uses an L1 penalty on the disparity gradients to encourage local smoothness and to prevent edge discontunity from image gradients.

$$C_{ds} = \frac{1}{N}\sum_{ij}|\partial_x d_{ij}|e^{-\|\partial_x I_{ij}\|} + |\partial_y d_{ij}|e^{-\|\partial_y I_{ij}\|}$$

Here, d_{ij} is the disparity, ∂d_{ij} is the disparity gradient, N is the number of pixels, I_{ij} is the input image, and $\tilde{I_{ij}}$ is the reconstructed image.

The left-right disparity loss C_{lr} uses L1 norm to encourage consistency between the left disparity map and the synthesized right disparity map.

$$C_{lr} = \frac{1}{N}\sum_{ij}|d^l_{ij} - d^r_{ij+d^l_{ij}}|$$

Here d^l_{ij} is the left image disparity and $d^r_{ij+d^l_{ij}}$ is the right image disparity.

The overall loss is expressed as:

$$C_s = \alpha_{ap}(C^l_a p + C^r_a p) + \alpha_{ds}(C^l_d s + C^r_d s) + \alpha_{lr}(C^l_l r + C^r_l r)$$

Here, $\alpha_{ap}, \alpha_{ds}, \alpha_{lr}$ are the weighting variable for each type of loss. For the neural net, $\alpha_{ap} = \alpha_{lr} = 1$ and $\alpha_{ds} = \frac{1}{r}$ where r is the downscaling factor for each disparity map.

In [ ]:
```
class MonocularDepthLoss(nn.modules.Module):
    def __init__(self, device='cpu', alphaAP = 1, alphaDS = 1, alphaLR = 1):
```

```
        super(MonocularDepthLoss, self).__init__()

        self.alphaAP = alphaAP
        self.alphaDS = alphaDS
        self.alphaLR = alphaLR

    def scaleImg(self, img):
        """
        Scales the images using bilinear interpolation by a factor of 1, 0.5,
        0.25, and 0.125.

        Args:
            img: input image
        Return:
            (list): list of images that are scaled
        """
        images = []
        _, _, h, w = img.size()

        for s in [1, 0.5, 0.25, 0.125]:
            newSize = (int(h * s), int(w * s))
            scaledImg = F.interpolate(img, size=newSize, mode='bilinear', align_corners=
            images.append(scaledImg)
        return images

    def generateImageFromDmap(self, imgL, left2RightDmap):
        """
        Generates a right image given a left image and a left to right disparity map
        args:
            imgL: tensor of size [batchSize, channels, height, width]
            left2RightDmap: tensor of size [batchSize, channels, height, width]
        Return:
            (tensor): generated img using a disparity map
        """
        batchSize, channels, h, w = imgL.shape
        left2RightDmap = left2RightDmap[:, 0, :, :].to(device)

        # Normalize pixel positions
        x = torch.linspace(0, 1, h)
        y = torch.linspace(0, 1, w)
        meshy, meshx = torch.meshgrid(x, y)
        meshx = meshx.repeat(batchSize, 1, 1).to(device)
        meshy = meshy.repeat(batchSize, 1, 1).to(device)

        newR = meshx + left2RightDmap
        flowfield = torch.stack((newR, meshy), dim=3).type_as(imgL)
        grid = 2 * flowfield - 1
        return F.grid_sample(imgL, grid, mode='bilinear', padding_mode='zeros')

    def appearanceMatchingLoss(self, x, y, alpha=0.85):
        """
        Compares the reconstructed image with the original image using SSIM
        (structural similarity index measure).
        Args:
            dmap: disparity map
            img: input image
            alpha: float from 0-1 for SSIM
        Return:
            (float): appearance matching loss
        """
        # This SSIM used was adapted from
        # https://github.com/mrharicot/monodepth/blob/master/monodepth_model.py#L91
```

```
        c1 = 0.01
        c2 = 0.03

        muX = nn.AvgPool2d(3, 1)(x)
        muY = nn.AvgPool2d(3, 1)(y)

        sigmaX = nn.AvgPool2d(3, 1)(x * x) - (muX * muX)
        sigmaY = nn.AvgPool2d(3, 1)(y * y) - (muY * muY)
        sigmaXY = nn.AvgPool2d(3, 1)(x * y) - (muX * muY)

        SSIM_n = (2 * muX * muY + c1*c1) * (2 * sigmaXY + c2*c2)
        SSIM_d = (muX * muX + muY * muY + c1*c1) * (sigmaX + sigmaY + c2*c2)
        SSIM = SSIM_n / SSIM_d

        ssim = torch.clamp((1 - SSIM) / 2, 0, 1)
        ssim_loss = alpha * torch.mean(ssim)
        mae = (1 - alpha) * torch.mean(torch.abs(x - y))

        return ssim_loss + mae

    def disparitySmoothnessLoss(self, dmap, img):
        """
        Penalizes gradient discontinuities. Uses weights from image gradients
        to prevent edge discontunity penalization.
        Args:
            dmap: disparity map
            img: input image
        Return:
            (float): disparity smoothness loss
        """
        dmapDX = F.pad(dmap, (0, 1, 0, 0), mode="replicate")
        dmapDX = torch.abs(dmapDX[:, :, :, :-1] - dmapDX[:, :, :, 1:])
        dmapDY = F.pad(dmap, (0, 0, 0, 1), mode="replicate")
        dmapDY = torch.abs(dmapDY[:, :, :-1, :] - dmapDY[:, :, 1:, :])

        imgDX = F.pad(img, (0, 1, 0, 0), mode="replicate")
        imgDX = torch.abs(imgDX[:, :, :, :-1] - imgDX[:, :, :, 1:])
        imgDY = F.pad(img, (0, 0, 0, 1), mode="replicate")
        imgDY = torch.abs(imgDY[:, :, :-1, :] - imgDY[:, :, 1:, :])

        dslDX = dmapDX * torch.exp(-torch.mean(imgDX, 1, keepdim=True))
        dslDY = dmapDY * torch.exp(-torch.mean(imgDY, 1, keepdim=True))
        return torch.mean(dslDX + dslDY)

    def LRConsistencyLoss(self, dmapL, dmapR):
        """
        Uses MAE to encourage consistency between the left disparity map and the
        synthesized right disparity map.
        Args:
            dmapL: left disparity map
            dmapR: right disparity map
        Return:
            (float): left-right consistency loss
        """
        dmapSynthR = self.generateImageFromDmap(dmapR, -dmapL)
        dmapSynthL = self.generateImageFromDmap(dmapL, dmapR)
        maeSynthR = torch.mean(torch.abs(dmapSynthR  - dmapL))
        maeSynthL = torch.mean(torch.abs(dmapSynthL  - dmapR))
        return maeSynthL + maeSynthR

    def __call__(self, input, target):
        """
```

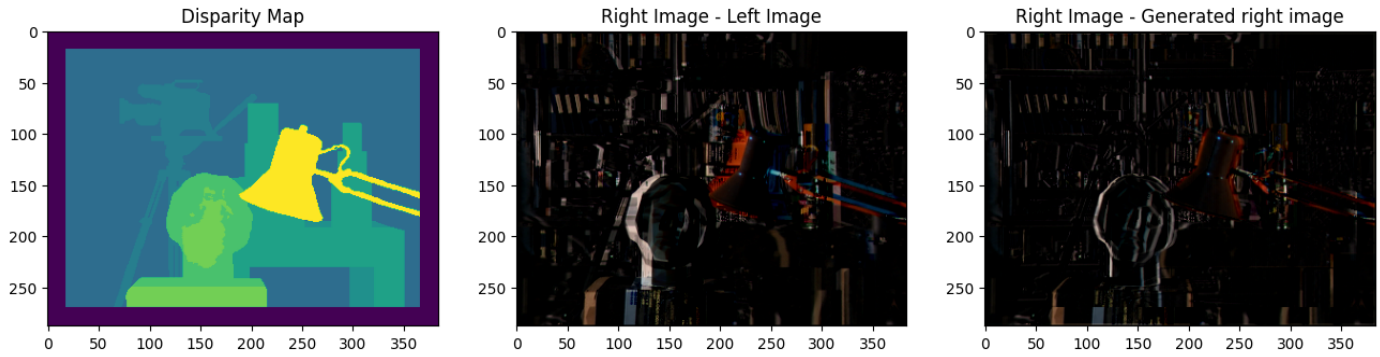
```
        Calculates the total loss using appearance matching, disparity
        smoothness, and left-right consistency.
        Args:
            input [disp1, disp2, disp3, disp4]
            target [left, right]
        Return:
            (float): monocular depth loss
        """

        imL, imR = target
        imScaledL = self.scaleImg(imL)
        imScaledR = self.scaleImg(imR)

        dmapL = [disp[:, 0, :, :].unsqueeze(1) for disp in input]
        dmapR = [disp[:, 1, :, :].unsqueeze(1) for disp in input]

        # Dmap has values [0,0.3]. Need to shift LEFT (negative) when using the left dma
        imReconstructedL = [self.generateImageFromDmap(im, -disp) for im, disp in zip(im
        imReconstructedR = [self.generateImageFromDmap(im, disp) for im, disp in zip(imS

        # Appearance Matching Loss
        apLoss = []
        for i in range(len(imReconstructedL)):
            apLossL = self.appearanceMatchingLoss(imReconstructedL[i], imScaledL[i])
            apLossR = self.appearanceMatchingLoss(imReconstructedR[i], imScaledR[i])
            apLoss.append(apLossL)
            apLoss.append(apLossR)
        self.lossAP = self.alphaAP * sum(apLoss)

        # Disparity Smoothness Loss
        dsLoss = []
        for i in range(4):
            dsLoss.append(self.disparitySmoothnessLoss(dmapL[i], dmapR[i]) / 2 ** i)
            dsLoss.append(self.disparitySmoothnessLoss(dmapR[i], dmapL[i]) / 2 ** i)
        self.lossDS = self.alphaDS * sum(dsLoss)

        # Left Right Consistency Loss
        lrLoss = []
        for L, R in zip(dmapL, dmapR):
            lrLoss.append(self.LRConsistencyLoss(L, R))
        self.lossLR = self.alphaLR * sum(lrLoss)

        return self.lossAP + self.lossDS + self.lossLR
```

In [ ]:

```
# Import images from class
imL = tF.to_tensor(image.imread(IMAGES_DIR + "/sceneL.ppm"))
imR = tF.to_tensor(image.imread(IMAGES_DIR + "/sceneR.ppm"))
imGT = torch.from_numpy(image.imread(IMAGES_DIR + "/truedisp.pgm") / 16)
imGT = imGT / imGT.shape[0]

# Generate image using disparity map
mdl = MonocularDepthLoss()
imGenerated = mdl.generateImageFromDmap(imL[None], imGT[None, None])

# Calculate disparities
dispRL = (imR - imL).permute(1, 2, 0)
dispRGen = (imR - imGenerated[0]).permute(1, 2, 0)

fig = plt.figure(figsize=(16,8))
ax = fig.add_subplot(1,3,1)
plt.title('Disparity Map')
ax.imshow(imGT)
```

```
ax = fig.add_subplot(1,3,2)
plt.title('Right Image - Left Image')
ax.imshow(dispRL)

ax = fig.add_subplot(1,3,3)
plt.title('Right Image - Generated right image')
ax.imshow(dispRGen)
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7f1220151f10>
```



In [ ]:
```
# Used to verify the correctness of the loss functions
print("Appearance Matching Loss btwn right image and generated image:")
print(mdl.appearanceMatchingLoss(imR[None], imGenerated))
print("Disparity Smoothness Loss btwn left image (dmap) and ground truth (img):")
print(mdl.disparitySmoothnessLoss(imGT[None, None], imL[None]))
print("Disparity Smoothness Loss btwn ground truth (dmap) and left image (img):")
print(mdl.disparitySmoothnessLoss(imL[None], imGT[None, None]))
print("Left-Right Consistency Loss btwn two ground truths:")
print(mdl.LRConsistencyLoss(imGT[None, None], imGT[None, None]))

# Test the loss
test = [torch.stack((dmap[:,0], dmap[:,0]), dim=1) for dmap in mdl.scaleImg(imGT[None, N
print("Test Image Shape: ", test[0].shape)
print("Test Prediction: ", mdl(test, [imL[None], imR[None]]))
```

```
Appearance Matching Loss btwn right image and generated image:
tensor(0.1623)
Disparity Smoothness Loss btwn left image (dmap) and ground truth (img):
tensor(0.0006, dtype=torch.float64)
Disparity Smoothness Loss btwn ground truth (dmap) and left image (img):
tensor(0.0545, dtype=torch.float64)
Left-Right Consistency Loss btwn two ground truths:
tensor(0.0042, dtype=torch.float64)
Test Image Shape:  torch.Size([1, 2, 288, 384])
Test Prediction:  tensor(1.3256, dtype=torch.float64)
```

The Network

The architecture of monocular depth estimation follows a conventional encoder-decoder framework, where the encoder uses ResNet18. This slightly differs from the implementation details from the paper as the authors use ResNet50 for the encoder. Four skip connections from the ResNet18's encoder are used to enhance the resolution of the output. The decoder estimates diparities at four resolutions, each having a double scale in resolution. The network provides two disparity map outputs, left-to-right and right-to-left. The kernel size, stride, number of input and output channels for each layer follow the diagram below from the paper by Godard et al.

1. Model architecture

"Encoder"						
layer	**k**	**s**	**chns**	**in**	**out**	**input**
conv1	7	2	3/32	1	2	left
conv1b	7	1	32/32	2	2	conv1
conv2	5	2	32/64	2	4	conv1b
conv2b	5	1	64/64	4	4	conv2
conv3	3	2	64/128	4	8	conv2b
conv3b	3	1	128/128	8	8	conv3
conv4	3	2	128/256	8	16	conv3b
conv4b	3	1	256/256	16	16	conv4
conv5	3	2	256/512	16	32	conv4b
conv5b	3	1	512/512	32	32	conv5
conv6	3	2	512/512	32	64	conv5b
conv6b	3	1	512/512	64	64	conv6
conv7	3	2	512/512	64	128	conv6b
conv7b	3	1	512/512	128	128	conv7

"Decoder"						
layer	**k**	**s**	**chns**	**in**	**out**	**input**
upconv7	3	2	512/512	128	64	conv7b
iconv7	3	1	1024/512	64	64	upconv7+conv6b
upconv6	3	2	512/512	64	32	iconv7
iconv6	3	1	1024/512	32	32	upconv6+conv5b
upconv5	3	2	512/256	32	16	iconv6
iconv5	3	1	512/256	16	16	upconv5+conv4b
upconv4	3	2	256/128	16	8	iconv5
iconv4	3	1	128/128	8	8	upconv4+conv3b
disp4	3	1	128/2	8	8	iconv4
upconv3	3	2	128/64	8	4	iconv4
iconv3	3	1	130/64	4	4	upconv3+conv2b+disp4*
disp3	3	1	64/2	4	4	iconv3
upconv2	3	2	64/32	4	2	iconv3
iconv2	3	1	66/32	2	2	upconv2+conv1b+disp3*
disp2	3	1	32/2	2	2	iconv2
upconv1	3	2	32/16	2	1	iconv2
iconv1	3	1	18/16	1	1	upconv1+disp2*
disp1	3	1	16/2	1	1	iconv1

Table 1: Our network architecture, where **k** is the kernel size, **s** the stride, **chns** the number of input and output channels for each layer, **in**put and **out**put is the downscaling factor for each layer relative to the input image, and **input** corresponds to the input of each layer where $+$ is a concatenation and $*$ is a $2\times$ upsampling of the layer.

In [ ]:
```
# Helper nn.Modules adapted from
# https://github.com/mrharicot/monodepth/blob/master/monodepth_model.py#L161
class iconv(nn.Module):
    def __init__(self, in_layers, out_layers, kernel_size, stride):
        super(iconv, self).__init__()
        padding = int(np.floor((kernel_size-1)/2))

        self.layers = nn.Sequential(
            nn.Conv2d(in_layers, out_layers, kernel_size, stride, padding),
            nn.BatchNorm2d(out_layers),
            nn.ELU(inplace=True),
        )

    def forward(self, x):
        return self.layers(x)

class upconv(nn.Module):
    def __init__(self, in_layers, out_layers, kernel_size, scale):
        super(upconv, self).__init__()
        self.scale = scale
        self.layers = iconv(in_layers, out_layers, kernel_size, 1)

    def forward(self, x):
        x = F.interpolate(x, scale_factor=self.scale, mode='bilinear', align_corners=Tru
        return self.layers(x)

class disp(nn.Module):
    def __init__(self, in_layers):
        super(disp, self).__init__()

        self.layers = nn.Sequential(
            nn.Conv2d(in_layers, 2, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(2),
            nn.Sigmoid(),
        )
```

```
    def forward(self, x):
        # Maximum disparity of 1/3 the image
        return 0.3 * self.layers(x)
```

In [ ]:

```
# Network adapted from:
# https://github.com/mrharicot/monodepth/blob/master/monodepth_model.py#L171

# Note: TorchVision's Resnet18 slightly differs that the one used in the paper
# as it has faster downsampling, hence fewer number of encoding layers.

class MyNet(nn.Module):
    def __init__(self):
        super(MyNet, self).__init__()

        # Used for graphing the network's training results
        self.loss_graph = []
        self.ap = []
        self.lr = []
        self.ds = []

        # Encoding layers
        # Transfer learning for faster training using the pretrained weights
        # from Resnet18.
        self.resnet = models.resnet18(weights=models.ResNet18_Weights.DEFAULT)
        ds5 = nn.Sequential(
                    nn.Conv2d(512, 512, kernel_size=(1, 1), stride=(2, 2), bias=False),
                    nn.BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
        )
        # Additional layers 5 and 6 for the encoder to follow similarly to the
        # paper as the resnet block goes from depth 512 -> 512.
        self.layer5 = models.resnet.BasicBlock(512, 512, stride=(2, 2), downsample=ds5)
        self.layer6 = models.resnet.BasicBlock(512, 512)

        # Decoding layers
        self.upconv1 = upconv(32, 16, 3, 2)
        self.upconv2 = upconv(64, 32, 3, 2)
        self.upconv3 = upconv(128, 64, 3, 1)
        self.upconv4 = upconv(256, 128, 3, 2)
        self.upconv5 = upconv(512, 256, 3, 2)
        self.upconv6 = upconv(512, 512, 3, 2)
        self.upconv7 = upconv(512, 512, 3, 2)

        self.iconv1 = iconv(16 + 2, 16, 3, 1)
        self.iconv2 = iconv(64 + 32 + 2, 32, 3, 1)
        self.iconv3 = iconv(64 + 64 + 2, 64, 3, 1)
        self.iconv4 = iconv(64 + 128, 128, 3, 1)
        self.iconv5 = iconv(128 + 256, 256, 3, 1)
        self.iconv6 = iconv(256 + 512, 512, 3, 1)
        self.iconv7 = iconv(512 + 512, 512, 3, 1)


        self.disp1_layer = disp(16)
        self.disp2_layer = disp(32)
        self.disp3_layer = disp(64)
        self.disp4_layer = disp(128)

    def forward(self, imL):
        """
        Args:
            imL: left image
        Return:
```

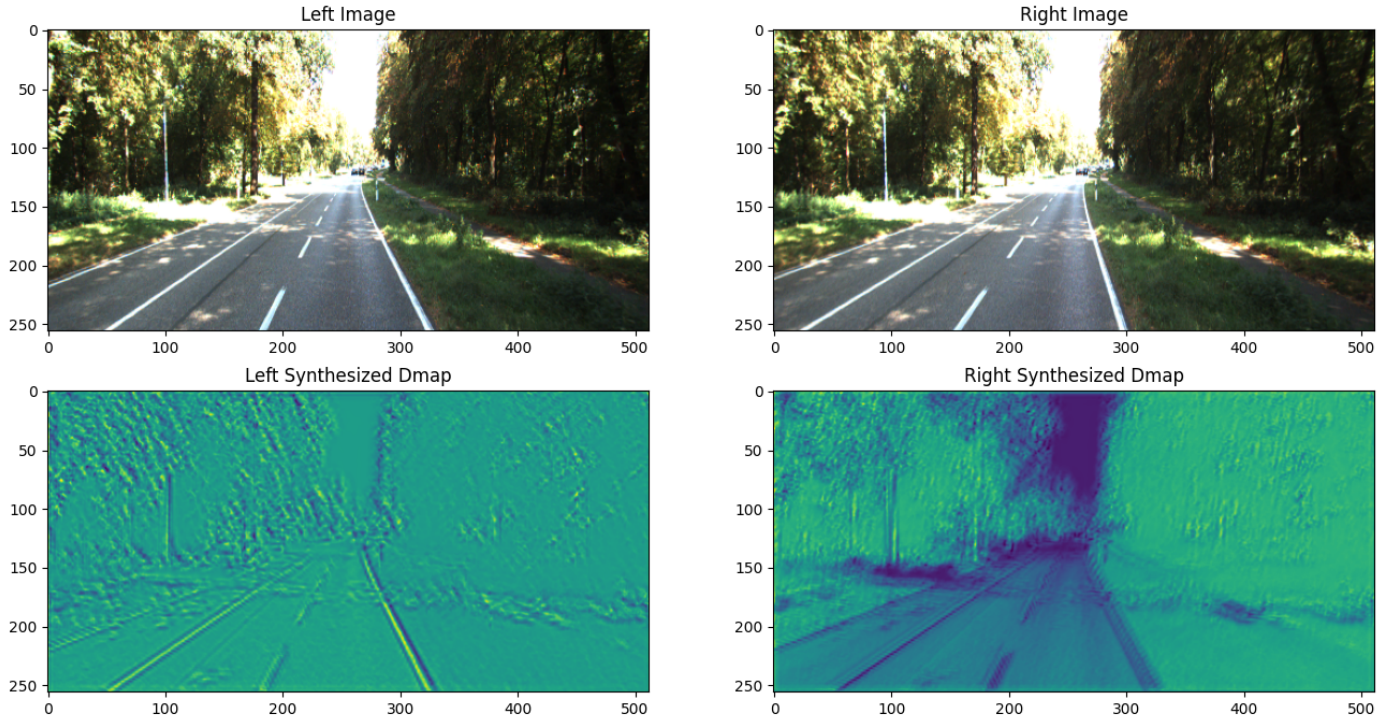
```
            disp1: disparity map
            [disp1, disp2, disp3, disp4]: disparity maps used for loss function
        """

        # Encoder
        skip1 = self.resnet.conv1(imL)
        skip1_ = self.resnet.bn1(skip1)
        skip1_ = self.resnet.relu(skip1_)
        skip2 = self.resnet.maxpool(skip1_)
        skip3 = self.resnet.layer1(skip2)
        skip4 = self.resnet.layer2(skip3)
        skip5 = self.resnet.layer3(skip4)
        skip6 = self.resnet.layer4(skip5)
        skip7 = self.layer5(skip6)
        x = self.layer6(skip7)

        # Decoder
        upconv7 = self.upconv7(x)
        iconv7 = self.iconv7(torch.cat((upconv7, skip6), 1))

        upconv6 = self.upconv6(iconv7)
        iconv6 = self.iconv6(torch.cat((upconv6, skip5), 1))

        upconv5 = self.upconv5(iconv6)
        iconv5 = self.iconv5(torch.cat((upconv5, skip4), 1))

        upconv4 = self.upconv4(iconv5)
        iconv4 = self.iconv4(torch.cat((upconv4, skip3), 1))
        disp4 = self.disp4_layer(iconv4)
        udisp4 = nn.functional.interpolate(disp4, scale_factor=1, mode='bilinear', align
        disp4 = nn.functional.interpolate(disp4, scale_factor=0.5, mode='bilinear', alig

        upconv3 = self.upconv3(iconv4)
        iconv3 = self.iconv3(torch.cat((upconv3, skip2, udisp4), 1))
        disp3 = self.disp3_layer(iconv3)
        udisp3 = nn.functional.interpolate(disp3, scale_factor=2, mode='bilinear', align

        upconv2 = self.upconv2(iconv3)
        iconv2 = self.iconv2(torch.cat((upconv2, skip1, udisp3), 1))
        disp2 = self.disp2_layer(iconv2)
        udisp2 = nn.functional.interpolate(disp2, scale_factor=2, mode='bilinear', align

        upconv1 = self.upconv1(iconv2)
        iconv1 = self.iconv1(torch.cat((upconv1, udisp2), 1))
        disp1 = self.disp1_layer(iconv1)

        return [disp1, disp2, disp3, disp4]

    def learn(self, train_loader, epochs=10, optimizer=None, loss_fcn=None):
        """
        Train the network on the dataset delivered by the dataloader train_dl.

        Args:
            train_loader: data loader with the training set
            epochs: number of epochs to train
            optimizer: PyTorch optimizer function
            loss_fcn: PyTorch or custom loss function
        """
        if optimizer is None or loss_fcn is None:
            print('Need to specify an optimizer and loss function')
            return
```

```
        for epoch in tqdm(range(epochs)):
            for i, data in enumerate(train_loader):

                imL, imR = data
                imL = imL.to(device)
                imR = imR.to(device)

                output = self(imL)
                loss = loss_fcn(output, [imL, imR])
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

            # Used for graphing the loss
            self.loss_graph.append(loss.item())
            self.ap.append(loss_fcn.lossAP.item())
            self.lr.append(loss_fcn.lossLR.item())
            self.ds.append(loss_fcn.lossDS.item())
            print("Epoch: {} Loss: {}".format(epoch, loss))
```

Visualing a Sample using the Untrained Network

In [ ]:
```
def plotDispResults(net, dataset, i, plot=True):
    """
    Plots the original left and right image. Also plots the left and right
    synthesized disparity.
    Args:
        input [disp1, disp2, disp3, disp4]
        target [left, right]
    Return:
        dmap: disparity map from the network
    """
    net.eval()
    imL, imR = dataset[i]
    dmap = net.forward(imL[None].to(device))[0].to('cpu').detach()
    dmapL, dmapR = dmap[:,0,:,:].unsqueeze(1), dmap[:,1,:,:].unsqueeze(1)

    if plot:
        fig = plt.figure(figsize=(16,8))
        ax = fig.add_subplot(2,2,1)
        plt.title('Left Image')
        ax.imshow(imL.permute(1, 2, 0))
        ax = fig.add_subplot(2,2,2)
        plt.title('Right Image')
        ax.imshow(imR.permute(1, 2, 0))
        ax = fig.add_subplot(2,2,3)
        plt.title('Left Synthesized Dmap')
        ax.imshow(dmapL[0][0])
        ax = fig.add_subplot(2,2,4)
        plt.title('Right Synthesized Dmap')
        ax.imshow(dmapR[0][0])

    return dmap
```

In [ ]:
```
# Test the untrained network on some sample images
untrained_net = MyNet().eval().to(device)
dmap = plotDispResults(untrained_net, val_dataset, 0)
```



Fitting the Network using a Single Image Pair

To ensure the correctness of the network's architecture with the custom loss function, the network will be fitted with a single left-right image pair from the test_loader.

In [ ]:
```
test_net = MyNet().to(device)
```

In [ ]:
```
# test_net.load_state_dict(torch.load(WEIGHTS_DIR+"/test_weights1"))
# test_net.eval()
```

In [ ]:
```
%%time
loss_fcn = MonocularDepthLoss(device).to(device)
optimizer = torch.optim.Adam(test_net.parameters(), lr=0.01)
print("Starting Training with #images: ", len(test_loader) * test_loader.batch_size)
test_net.learn(test_loader, epochs=1000, optimizer=optimizer, loss_fcn=loss_fcn)
```

```
Starting Training with #images:  24
  0%|          | 1/1000 [00:00<12:09,  1.37it/s]
Epoch: 0 Loss: 3.6926310062408447
  0%|          | 2/1000 [00:01<11:04,  1.50it/s]
Epoch: 1 Loss: 3.4100282192230225
  0%|          | 3/1000 [00:01<10:52,  1.53it/s]
Epoch: 2 Loss: 3.2772605419158936
  0%|          | 4/1000 [00:02<10:46,  1.54it/s]
Epoch: 3 Loss: 3.092691421508789
  0%|          | 5/1000 [00:03<10:38,  1.56it/s]
Epoch: 4 Loss: 3.0437369346618652
  1%|          | 6/1000 [00:03<10:32,  1.57it/s]
Epoch: 5 Loss: 2.963690996170044
  1%|          | 7/1000 [00:04<10:27,  1.58it/s]
Epoch: 6 Loss: 2.914308547973633
  1%|          | 8/1000 [00:05<10:19,  1.60it/s]
```

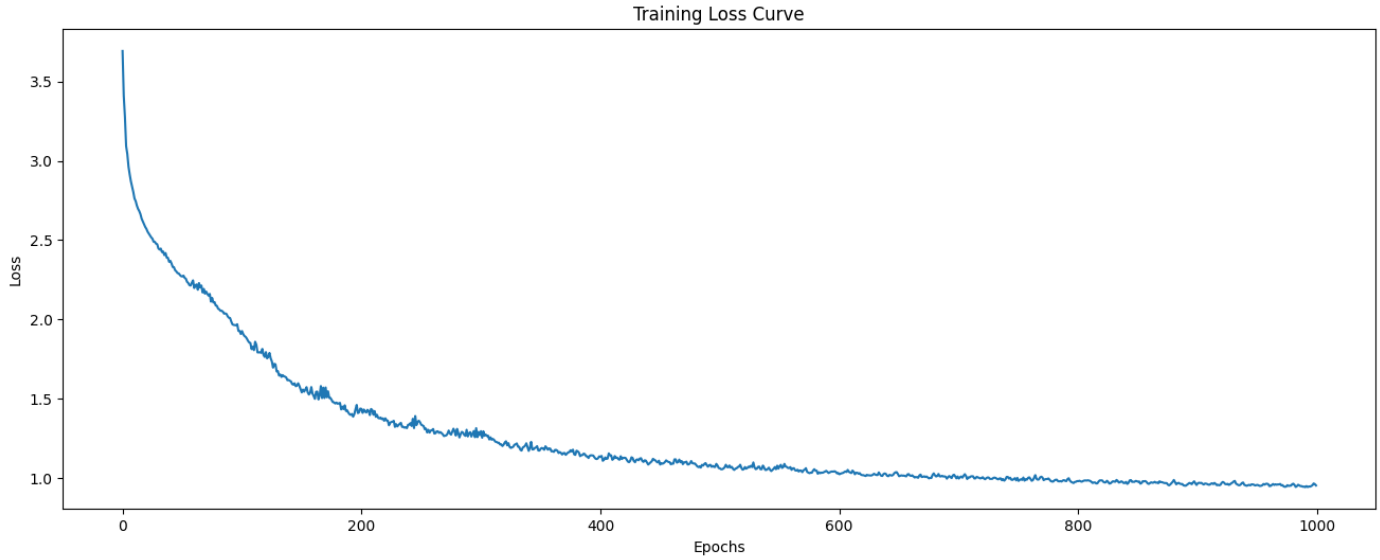
```
Epoch: 979 Loss: 0.9609482288360596
 98%|▉| 981/1000 [11:51<00:18,  1.03it/s]
Epoch: 980 Loss: 0.9641199111938477
 98%|▉| 982/1000 [11:51<00:16,  1.06it/s]
Epoch: 981 Loss: 0.9562386274337769
 98%|▉| 983/1000 [11:52<00:14,  1.16it/s]
Epoch: 982 Loss: 0.9436368346214294
 98%|▉| 984/1000 [11:53<00:12,  1.24it/s]
Epoch: 983 Loss: 0.9488251805305481
 98%|▉| 985/1000 [11:53<00:11,  1.30it/s]
Epoch: 984 Loss: 0.9575791954994202
 99%|▉| 986/1000 [11:54<00:10,  1.37it/s]
Epoch: 985 Loss: 0.9535344243049622
 99%|▉| 987/1000 [11:55<00:09,  1.40it/s]
Epoch: 986 Loss: 0.9490853548049927
 99%|▉| 988/1000 [11:55<00:08,  1.43it/s]
Epoch: 987 Loss: 0.9488379955291748
 99%|▉| 989/1000 [11:56<00:07,  1.46it/s]
Epoch: 988 Loss: 0.9452317357063293
 99%|▉| 990/1000 [11:57<00:06,  1.47it/s]
Epoch: 989 Loss: 0.9454860091209412
 99%|▉| 991/1000 [11:57<00:06,  1.48it/s]
Epoch: 990 Loss: 0.9427438378334045
 99%|▉| 992/1000 [11:58<00:05,  1.49it/s]
Epoch: 991 Loss: 0.9490416646003723
 99%|▉| 993/1000 [11:59<00:05,  1.31it/s]
Epoch: 992 Loss: 0.9431771039962769
 99%|▉| 994/1000 [12:00<00:04,  1.21it/s]
Epoch: 993 Loss: 0.9469221830368042
100%|▉| 995/1000 [12:01<00:04,  1.15it/s]
Epoch: 994 Loss: 0.9457972049713135
100%|▉| 996/1000 [12:02<00:03,  1.13it/s]
Epoch: 995 Loss: 0.9494093060493469
100%|▉| 997/1000 [12:03<00:02,  1.21it/s]
Epoch: 996 Loss: 0.9547336101531982
100%|▉| 998/1000 [12:03<00:01,  1.28it/s]
Epoch: 997 Loss: 0.9656604528427124
100%|▉| 999/1000 [12:04<00:00,  1.35it/s]
Epoch: 998 Loss: 0.9613102674484253
100%|▉| 1000/1000 [12:05<00:00,  1.38it/s]
Epoch: 999 Loss: 0.9532589912414551
CPU times: user 1min 38s, sys: 2min 58s, total: 4min 36s
Wall time: 12min 5s
```

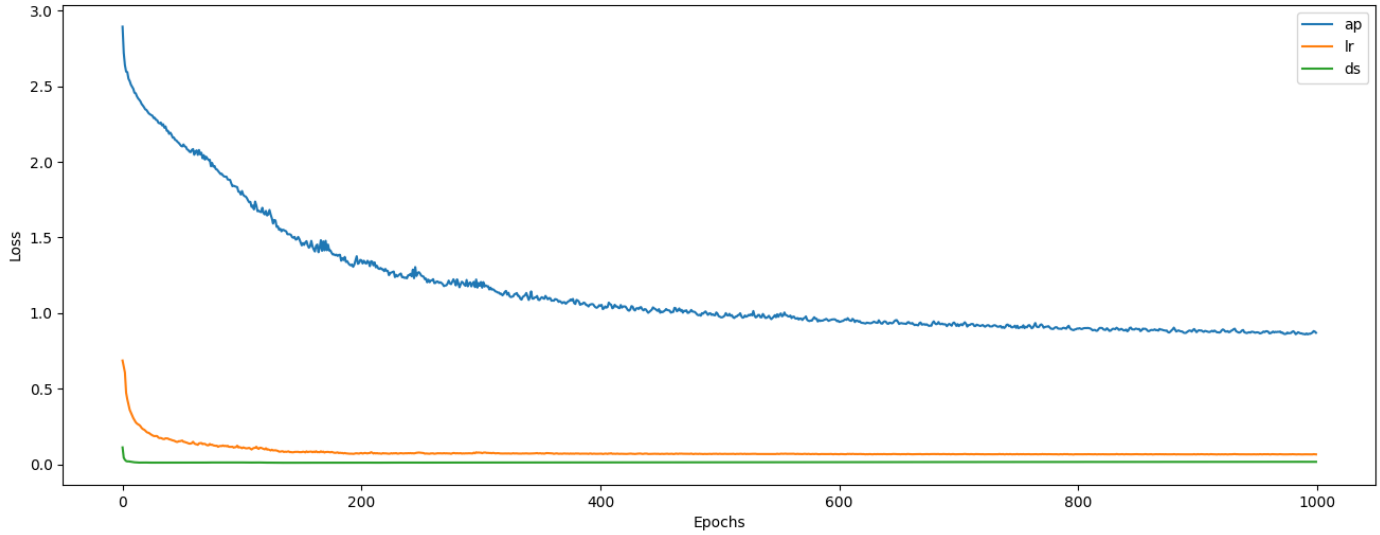
In [ ]:
```
# torch.save(test_net.state_dict(), WEIGHTS_DIR+"/test_weights1")
```

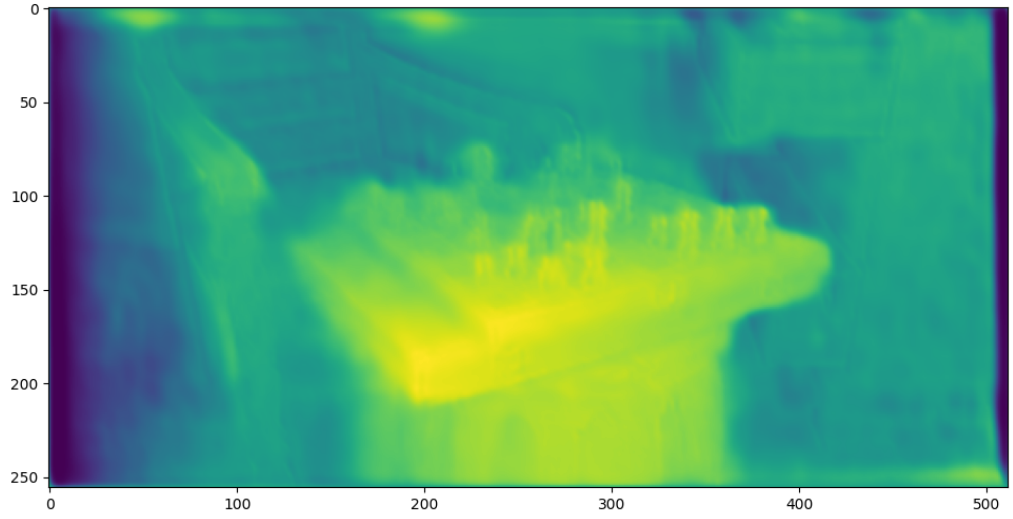
In [ ]:
```
# Plot the test results
%matplotlib inline
fig = plt.figure(figsize=(16,20))
ax = fig.add_subplot(3,1,1)
ax2 = fig.add_subplot(3,1,2)
dmap_plot = fig.add_subplot(3,1,3)

ax.set_xlabel('Epochs')
ax.set_ylabel('Loss')
```

```
ax.set_title('Training Loss Curve')
ax.plot(test_net.loss_graph)

ax2.set_xlabel('Epochs')
ax2.set_ylabel('Loss')
ax2.plot(test_net.ap, label='ap')
ax2.plot(test_net.lr, label='lr')
ax2.plot(test_net.ds, label='ds')
ax2.legend(loc='upper right')

dmap = plotDispResults(test_net, test_dataset, 0, plot=False)
dmap_plot.clear()
dmap_plot.imshow(dmap[0,0])
```

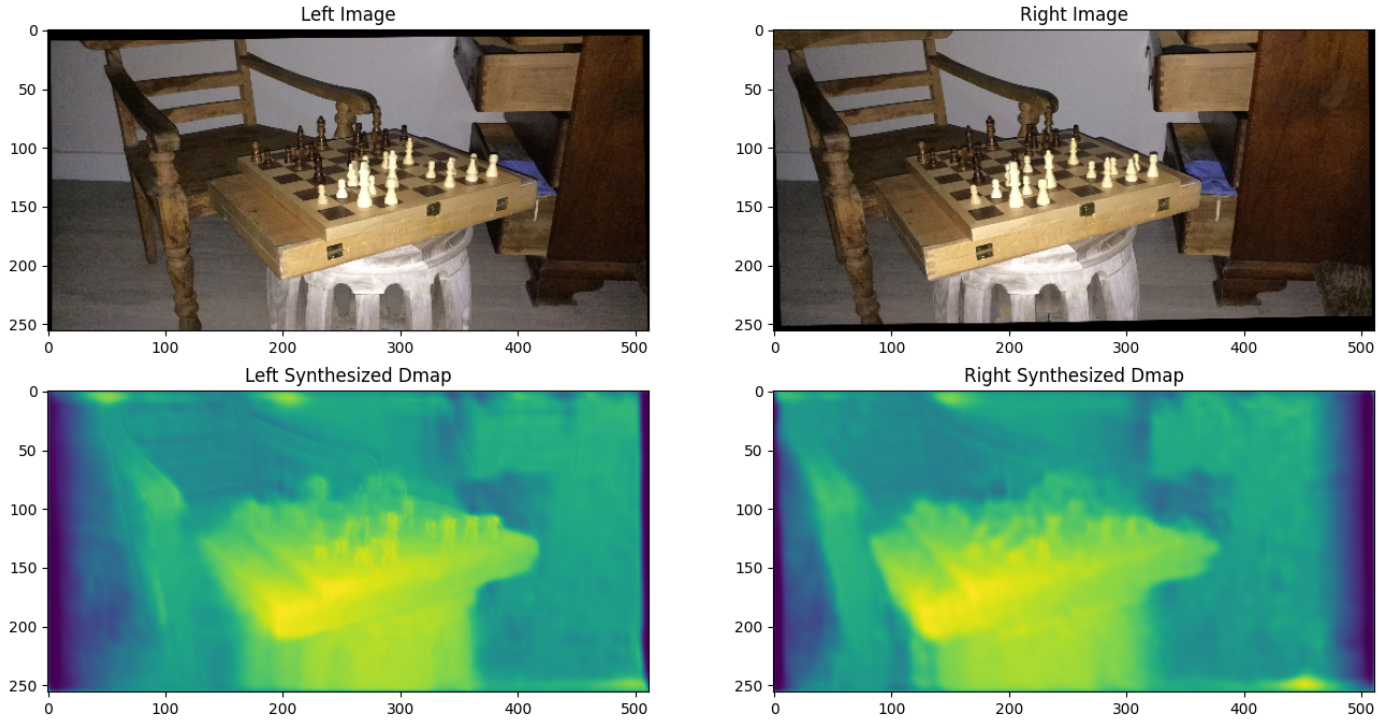
Out[ ]:
```
<matplotlib.image.AxesImage at 0x7fe9bc330eb0>
```







In [ ]:
```
dmap = plotDispResults(test_net, test_dataset, 0)
```



After training the model, the left and right disparity maps are both quite good where the chess set on the table is light green, which represents it is much closer to the camera than the darker green chair and the drawer. The background and the back of the chair are mixed between dark green and dark blue, which does not clearly show the distinction for the background's depths. The random artifacts at the top part of the chair and the bottom right corner of the disparity maps cannot be eliminated, and its cause is uncertain. One possibility is that the network was trapped in a local minimum due to the differences in the right image having a fuzzy texture in the bottom right corner and the top of the rocking chair where the left image didn't. Note: The final model was trained without using a GPU, so the training time is much longer than the fitting the single chess image for the test_net.

Training the Model

The model gets trained on the 3318 images from train_loader.

In [ ]:
```
train_net = MyNet().to(device)
# train_net.load_state_dict(torch.load(WEIGHTS_DIR+"/train_weights2"))
# train_net.eval()
```

In [ ]:
```
%%time
loss_fcn = MonocularDepthLoss(device).to(device)
optimizer = torch.optim.Adam(train_net.parameters(), lr=0.01)
print("Starting Training with #images: ", len(train_loader) * train_loader.batch_size)
train_net.learn(train_loader, epochs=5, optimizer=optimizer, loss_fcn=loss_fcn)
```

```
Starting Training with #images:  3336
 20%|██        | 1/5 [03:30<14:03, 210.91s/it]
Epoch: 0 Loss: 2.349707841873169
 40%|████      | 2/5 [06:55<10:20, 206.96s/it]
Epoch: 1 Loss: 2.268582344055176
 60%|██████    | 3/5 [10:22<06:54, 207.30s/it]
Epoch: 2 Loss: 1.6769355535507202
```

```
 80%|████████  | 4/5 [13:54<03:29, 209.13s/it]
Epoch: 3 Loss: 1.2006303071975708
100%|██████████| 5/5 [17:20<00:00, 208.16s/it]
Epoch: 4 Loss: 1.7541000843048096
CPU times: user 5min 13s, sys: 20.4 s, total: 5min 33s
Wall time: 17min 20s
```

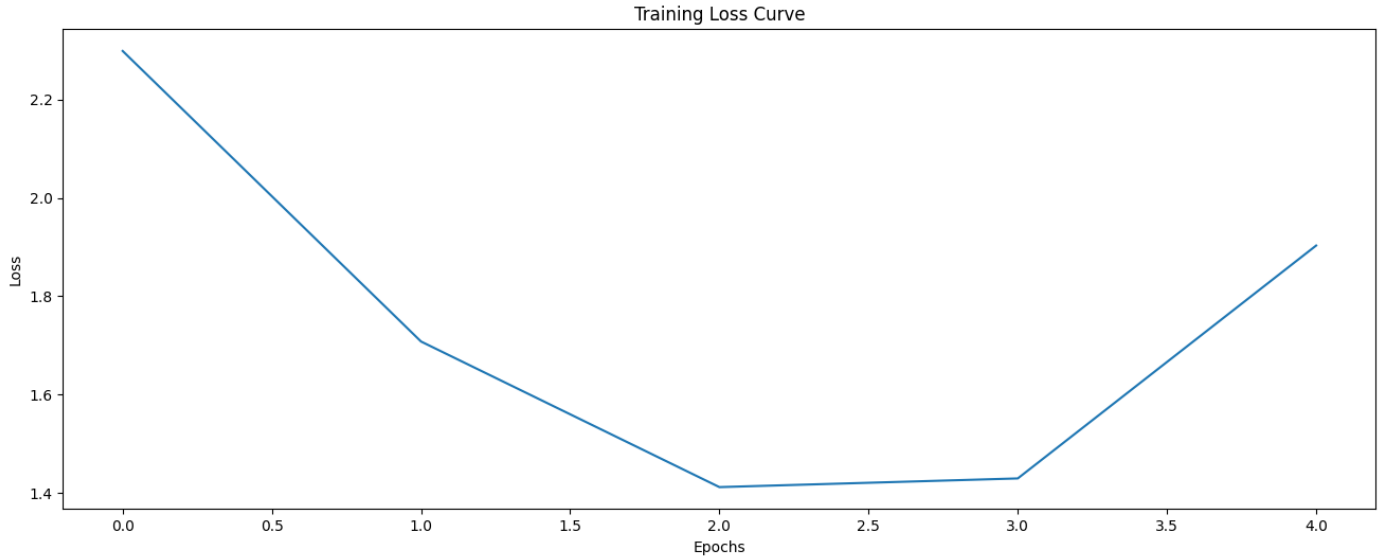
Overfit/Diverging Model

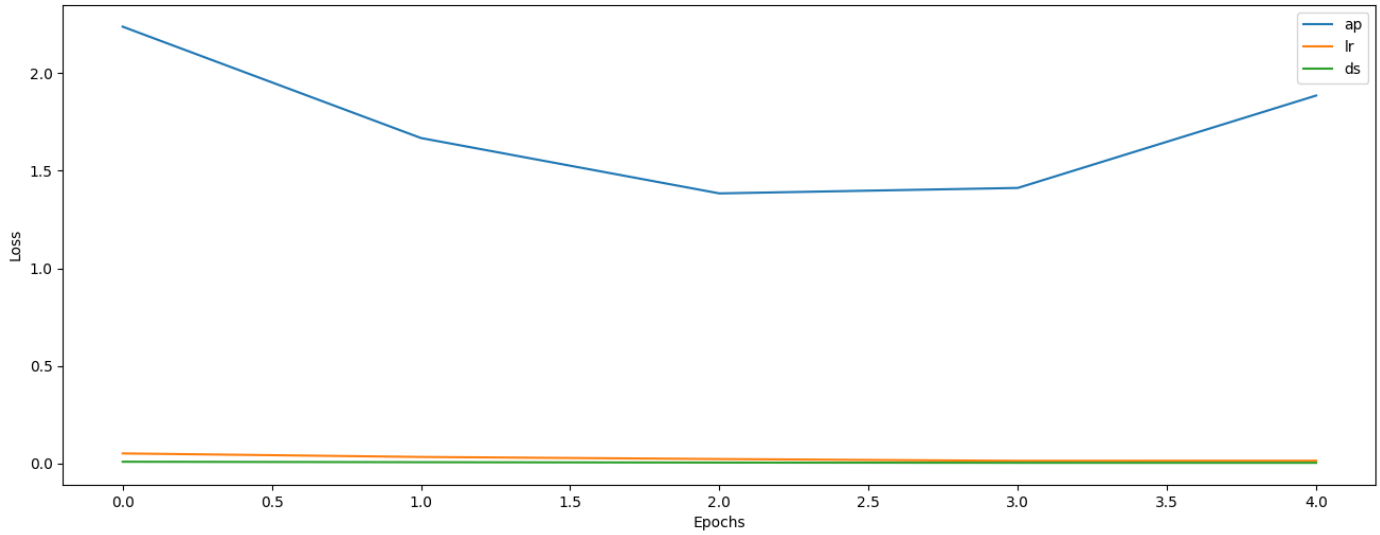
In [ ]:

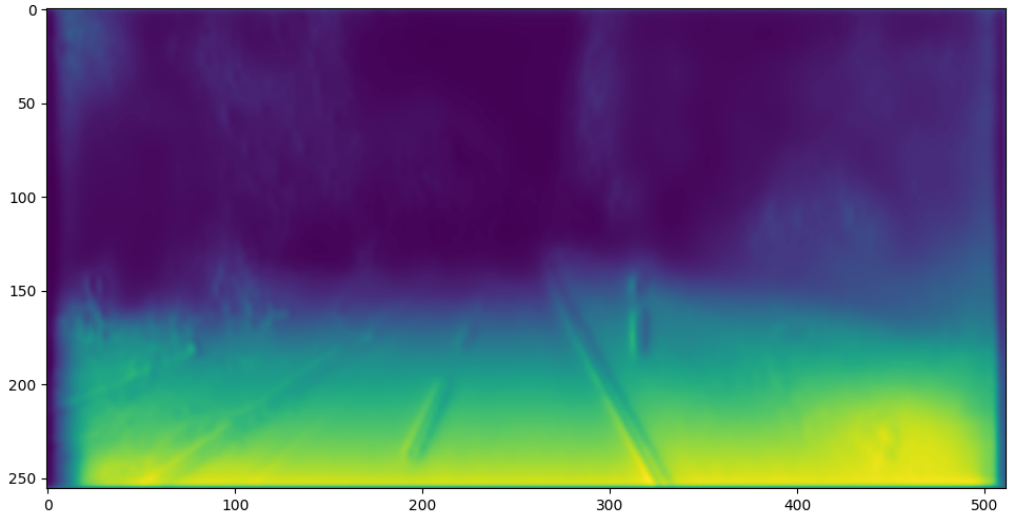
```
%matplotlib inline
fig = plt.figure(figsize=(16,20))
ax = fig.add_subplot(3,1,1)
ax2 = fig.add_subplot(3,1,2)
dmap_plot = fig.add_subplot(3,1,3)

ax.set_xlabel('Epochs')
ax.set_ylabel('Loss')
ax.set_title('Training Loss Curve')
ax.plot(train_net.loss_graph)

ax2.set_xlabel('Epochs')
ax2.set_ylabel('Loss')
ax2.plot(train_net.ap, label='ap')
ax2.plot(train_net.lr, label='lr')
ax2.plot(train_net.ds, label='ds')
ax2.legend(loc='upper right')

dmap = plotDispResults(train_net, val_dataset, 10, plot=False)
dmap_plot.clear()
dmap_plot.imshow(dmap[0,0])
```

Out[ ]:

```
<matplotlib.image.AxesImage at 0x7f12218643a0>
```







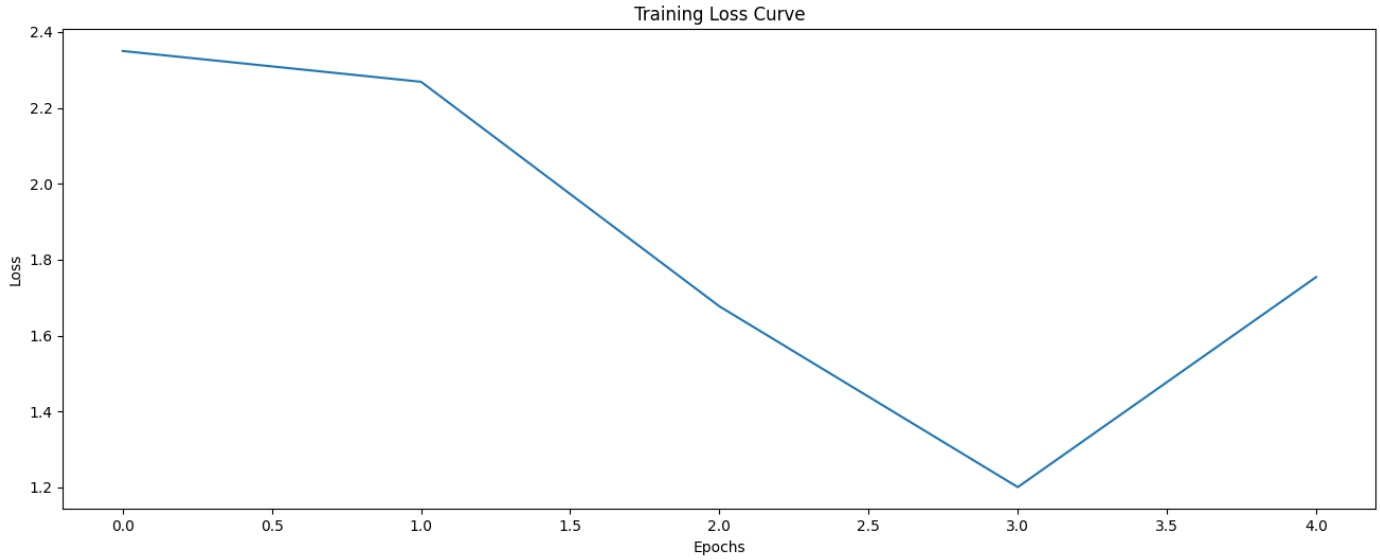
Final Model

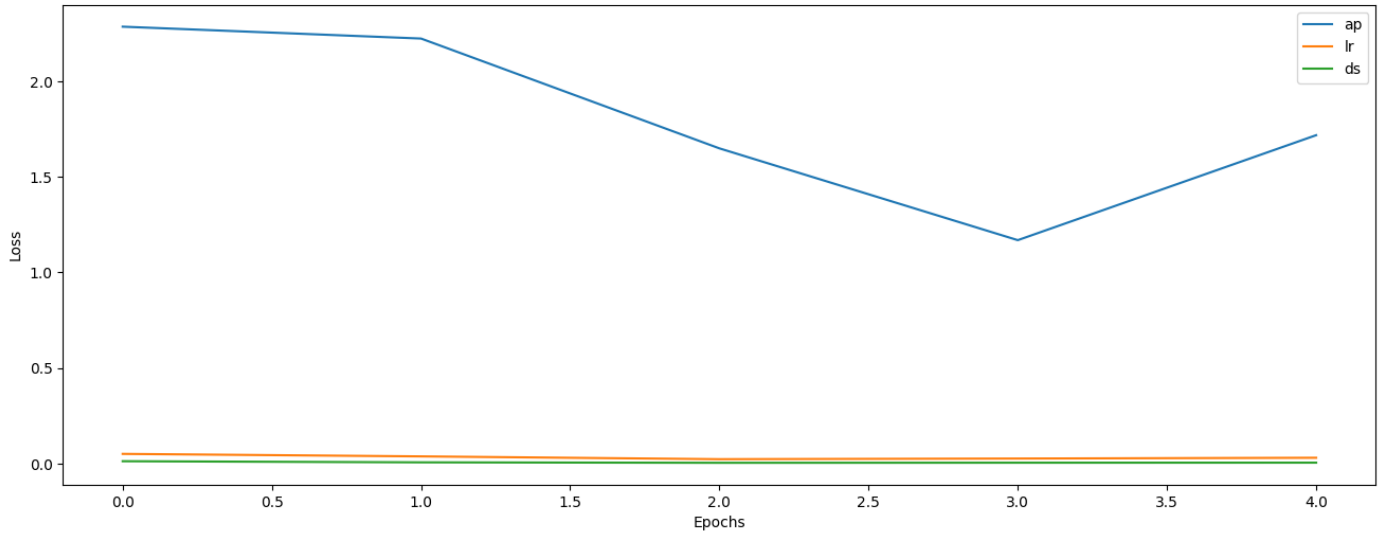
In [ ]:
```
# torch.save(train_net.state_dict(), WEIGHTS_DIR+"/train_weights2")
```

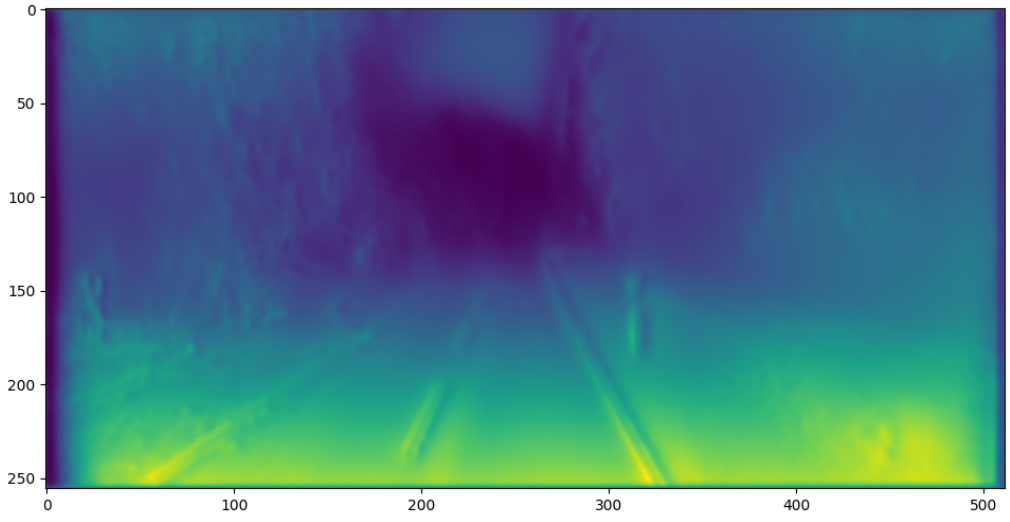
In [ ]:
```
# Plot the training results
%matplotlib inline
fig = plt.figure(figsize=(16,20))
```

```
ax = fig.add_subplot(3,1,1)
ax2 = fig.add_subplot(3,1,2)
dmap_plot = fig.add_subplot(3,1,3)

ax.set_xlabel('Epochs')
ax.set_ylabel('Loss')
ax.set_title('Training Loss Curve')
ax.plot(train_net.loss_graph)

ax2.set_xlabel('Epochs')
ax2.set_ylabel('Loss')
ax2.plot(train_net.ap, label='ap')
ax2.plot(train_net.lr, label='lr')
ax2.plot(train_net.ds, label='ds')
ax2.legend(loc='upper right')

dmap = plotDispResults(train_net, val_dataset, 10, plot=False)
dmap_plot.clear()
dmap_plot.imshow(dmap[0,0])
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7f9a08a622b0>
```







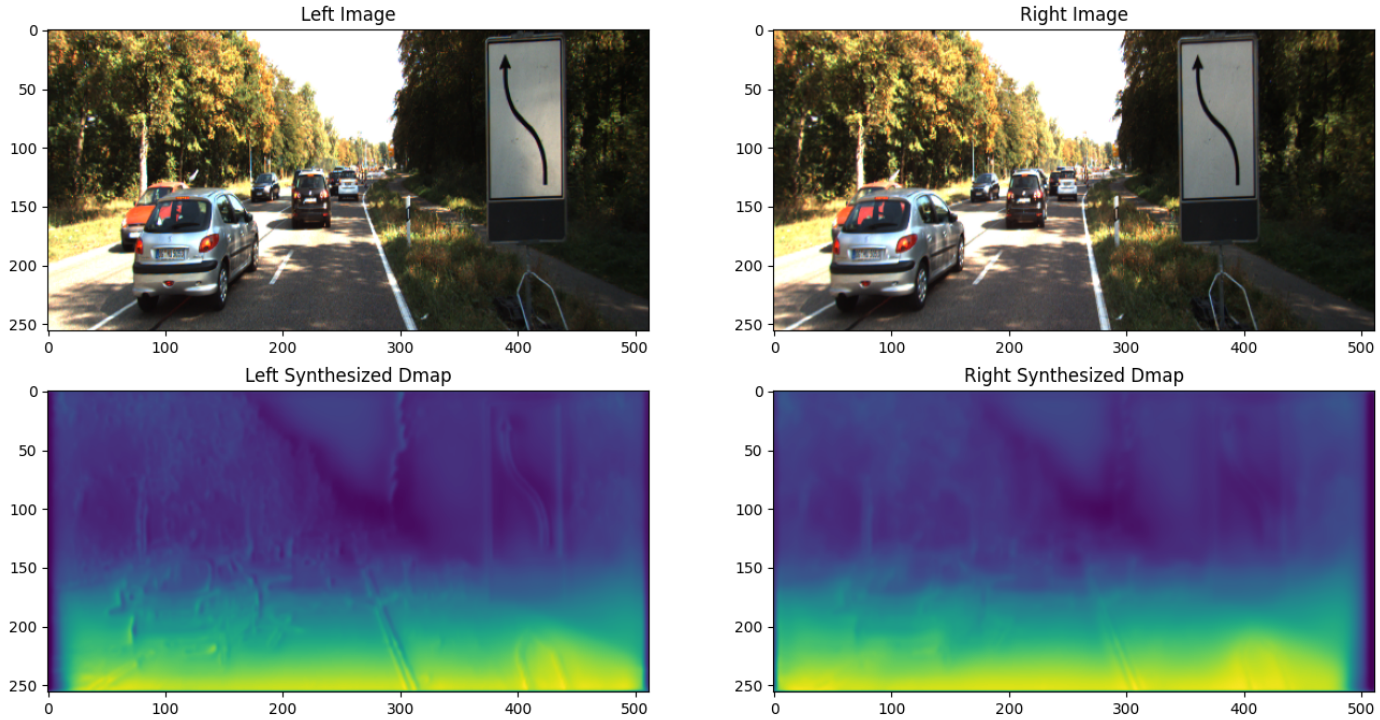
Cross Validation Loss

The network is validated using a random unseen left-right image pairs in the val_loader. Calculating the cross-validation loss shows the robustness and generalizability of a model to prevent overfitting. Note that it uses the same loss function used during training.

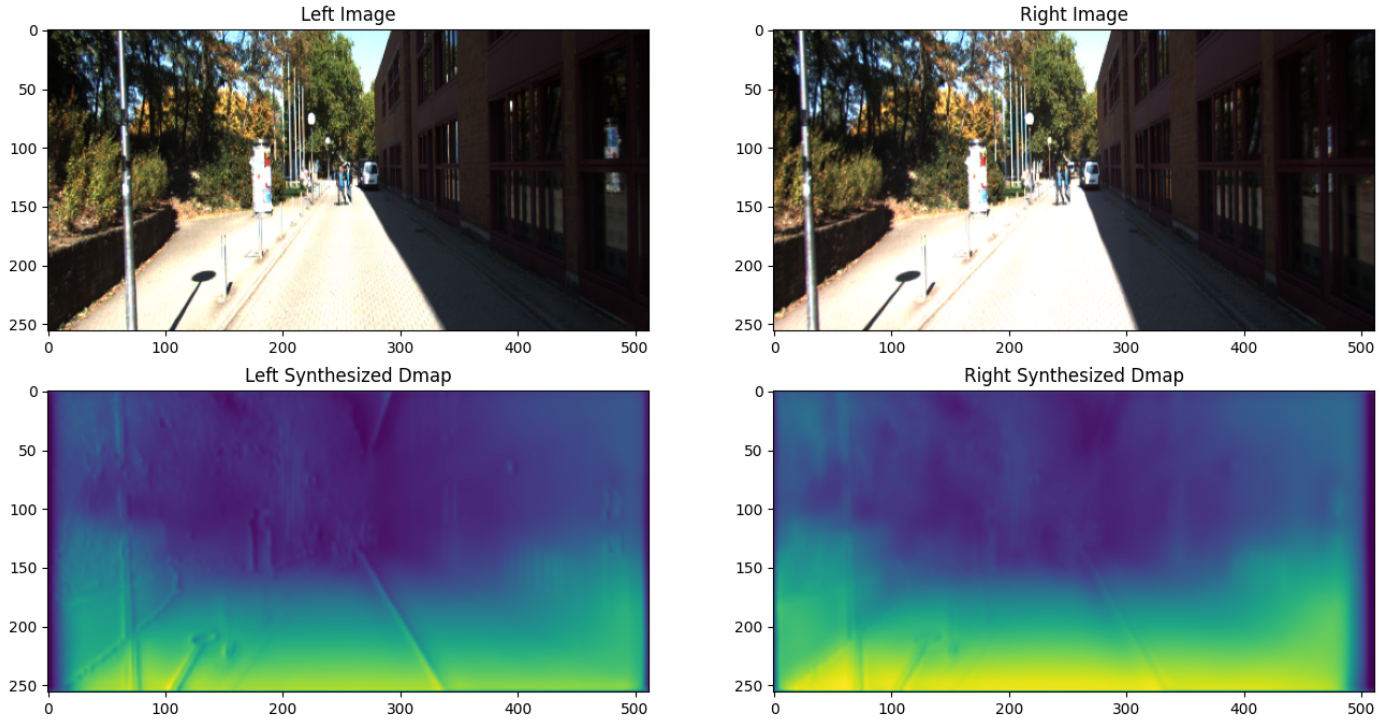
In [ ]:
```
# Set model to evaluation mode
train_net.eval()

# Iterate through the validation dataloader
total_loss = 0
with torch.no_grad():
    for i, data in enumerate(val_loader):
        imL, imR = data
        imL = imL.to(device)
        imR = imR.to(device)

        output = train_net.forward(imL.to(device))
        loss = loss_fcn(output, [imL, imR])
        print(i, loss)
        total_loss += loss.item()

    # calculate average validation loss
    avg_val_loss = total_loss / len(val_loader)
    print(avg_val_loss)
```

```
0 tensor(1.2565, device='cuda:0')
1 tensor(1.2126, device='cuda:0')
2 tensor(1.2101, device='cuda:0')
3 tensor(1.2970, device='cuda:0')
4 tensor(1.3393, device='cuda:0')
5 tensor(1.2400, device='cuda:0')
6 tensor(1.2848, device='cuda:0')
7 tensor(1.3193, device='cuda:0')
8 tensor(1.2841, device='cuda:0')
9 tensor(1.2971, device='cuda:0')
10 tensor(1.2463, device='cuda:0')
11 tensor(1.2749, device='cuda:0')
12 tensor(1.3285, device='cuda:0')
13 tensor(1.3124, device='cuda:0')
14 tensor(1.3248, device='cuda:0')
15 tensor(1.3282, device='cuda:0')
16 tensor(1.2670, device='cuda:0')
17 tensor(1.2393, device='cuda:0')
18 tensor(1.3303, device='cuda:0')
19 tensor(1.2086, device='cuda:0')
20 tensor(1.3131, device='cuda:0')
21 tensor(1.2670, device='cuda:0')
22 tensor(1.3110, device='cuda:0')
23 tensor(1.2999, device='cuda:0')
24 tensor(1.2833, device='cuda:0')
25 tensor(1.3563, device='cuda:0')
26 tensor(1.2943, device='cuda:0')
27 tensor(1.2497, device='cuda:0')
1.2848565833909171
```

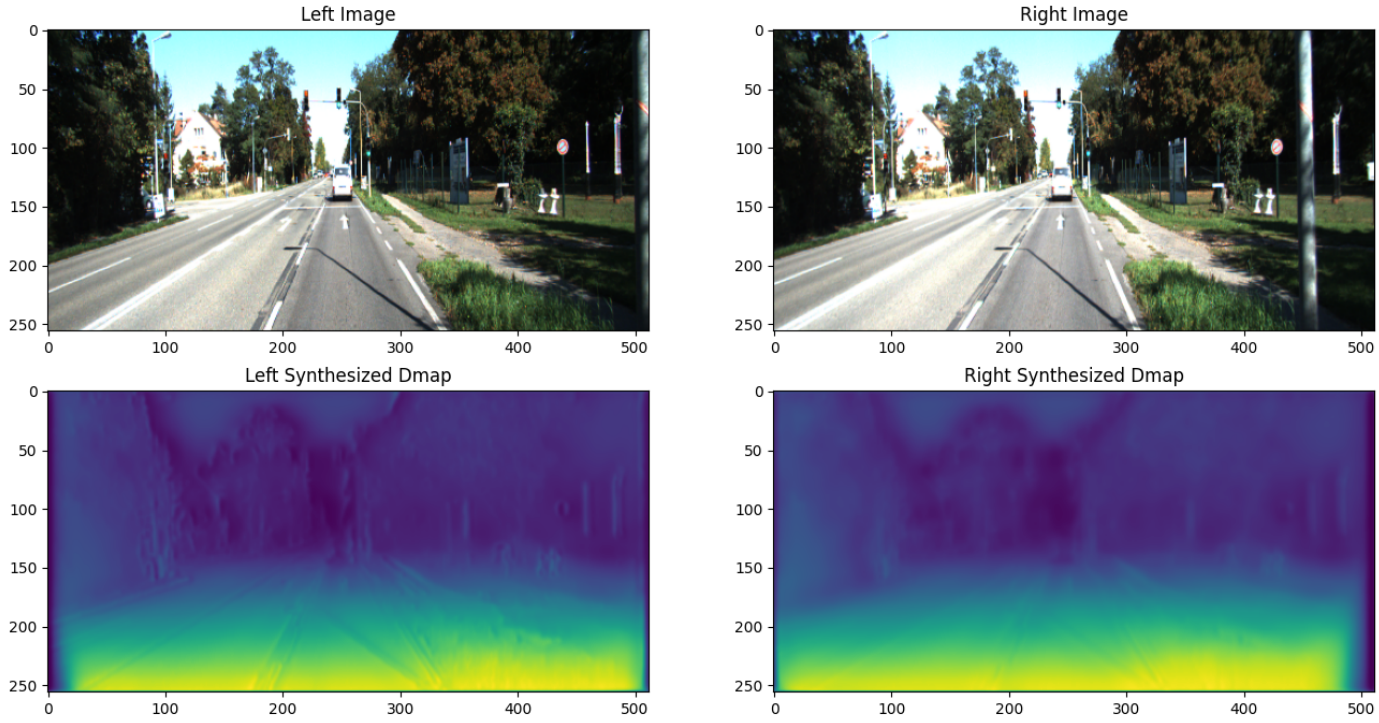
In [ ]:
```
dmap = plotDispResults(train_net, val_dataset, 300)
```



In [ ]:
```
dmap = plotDispResults(train_net, val_dataset, 350)
```



In [ ]:
```
dmap = plotDispResults(train_net, val_dataset, 600)
```



Conclusion

The model performs well on the unseen validation data with an average cross-validation loss of 1.285 despite being trained on approximately 3000 training data for five epochs. The output shows closer areas are brighter yellow and green (higher disparity value), while farther objects near vanishing points are dark blue (lower disparity values). The left-right consistency loss ensured that the left and right disparity maps were consistent, enabling the network to leverage information from both images while training. However, the left disparity map exhibited visible boundaries likely induced by the gradient smoothness loss function, while the edge boundaries were not visible on the right disparity map. The edge artifacts could imply that the gradient smoothness loss function reached two local minima, the correct right disparity map (without visible edge boundaries) and the incorrect left disparity map (with visible edge boundaries). The artifacts in the left disparity map could be removed by adding more unique dataset samples, decreasing the learning rate and reducing the regularization parameter α_{ds}. The appearance matching loss does not converged to a low value, unlike the disparity smoothness and left-right consistency losses, resulting in a blurry disparity map that is not entirely accurate. A reccomendation for improvement is to have more left-right image pairs in the train_loader. The dark blue disparity bars on the left side of the left disparity map and the right side of the right disparity map are other limitations. These areas result from regions in the left image that are not present in the right image and vice versa, making the appearance loss function unusable for these regions. The post-processing solution by Godard et al. resolved this issue by running the model on both the input image and its reflection and then combining the two images to construct a disparity map without any disparity ramps.

From training the model, the loss stops improving after approximately 3-4 epochs as the loss reaches a minimum of about 1.2. However, training the model for ten epochs results in degradation in the depth map as it produces seemingly random garbage. A possible reason for this occurence lies in potential conceptual issues with the loss function and should get explored in the future. The model should also

be training on more data. However, with limited storage of 15 GB of Google Drive space, a small learning rate, and limited computational resources of one Google Colab GPU, following the results from the paper to train the model is not practical. Other recommendations for the future include addressing challenges of the accuracy of depth estimation affected by occlusions, lighting conditions, and reflective surfaces. The estimated depth maps contained errors and artifacts, especially in areas with low texture or ambiguous features.

Overall, the self-supervised deep neural network for single image depth estimation that exploits binocular stereo data instead of aligned ground truth depth data. The model's loss function consists of photo-consistency comparison between predicted depth maps and disparity map regularization during training, leading to superior results compared to fully supervised baselines. The model can also generalize to unseen datasets and produce visually plausible depth maps. The authors suggest future work on extending the model to intake videos as the temporal consistency would improve results, investigating sparse input as an alternative training input, and predicting the full occupancy of the scene.