Project 7: Self-supervised single-image depth-estimation using neural-networks

Authors:

- **Sharhad Bashar**
- **Lizhe Chen**
- **Genséric Ghiro**
- **Futian Zhang**

1) Abstract: highlights of the main points of the project.

As the applications of Computer Vision evolve, the necessity for accurate depth estimation is growing exponentially. Neural networks are a convenient way to obtain such map; however, supervised learning is hard to achieve for this particular purpose due to the difficulty of retrieving enough ground truths and their lack of precision. Instead, self-supervised learning comes up as a good alternative and shows promising results. It exploits disparity maps and their relation with depth, a well-known concept from epipolar geometry. Therefore, the goal here is to train a network (without using ground truths) that can output an image's dense depth map. For this, two views of the same image (right and left) will be required during the training phase. They will be used to compute a disparity map, from which depth can be retrieved. Once this first step is completed, only the left image (i.e. one view) will be required to get a depth estimation from the network.

2) Introduction: review of topic and selected methodology/approach, data source, and related technical ideas/algorithms & their motivation

○ The depth estimation of an image can be achieved with neural networks. While a supervised network might be the first idea that comes in mind, it is not the best solution as in practice the ground truth for the dense depth map of an image is really hard to obtain, and supervised learning would require a few thousands of those. Indeed, expensive equipment is required (laser, scans, etc.) and the results are not even that precise. However, this can be overcomed by using disparities (self-supervised network). Therefore, no ground truth will be needed at all during training (it can be used later on for validation purposes only). This technique involves setting up two cameras on the same horizontal line and predicting the disparity map for the image, using its left and right views. Then, the actual depth can be calculated with the camera parameters. For that purpose, a disparity prediction network can be trained. It takes the left image as input, predict the disparity maps for both left and right views, reconstructs the right image using disparity maps and uses the latter with the real right image to compute a loss function, needed to backpropagate the network.

Example of ground truth dense depth map:

○ To conduct our project, we used the well-known KITTI dataset. The main reasons behind that decision are that it contains both the left and right images for each scene, it proposes a large depth range (close distances to infinity depth) and it is commonly used by SOTA algorithms. In addition, it also has the LiDAR ground truth, which allows us to evaluate our self-supervised network. For obvious reasons (lack of resources, mainly GPUs), we only used a small portion of that dataset. The original training set had 93K pairs, but only 5882 pairs were selected for our training. For validation, 500 pairs were manually selected. This validation dataset serves mainly as a benchmark for depth prediction and depth completion (i.e. sparse to dense). As for data augmentation, we are using Random Flip. However, since binocular image pairs are used to train a monocular depth predictor, this transform requires a particular handling. Indeed, if a pair of images is flipped, then it also needs to be swapped (i.e. the right image becomes the left image and vice versa) to make sure that the relative position is still left and right. Information on how to download and prepare the dataset can be found in the folder "dataset" included with this report (see README.txt).

Left, right image pair sample from KITTI dataset:

○ In [5], a 14-layer encoder was designed to extract the features of the input. Then, a symmetric decoder with skip connections from the encoder was used to produce a disparity pyramid for loss calculation. Our implementation takes a part of resnet18 and a part of vgg16 as encoders and adapts the decoder from [5]. The use of two different encoders is primarily to investigate whether or not this decoder can be used with other pretrained networks and to compare the accuracy of the results. For more advanced taks, such as disparity predictions, a deep neural network was needed to extract the features. However, with the deepening of the network, the accuracy of the validation set may decrease. We avoided this (overfitting) by calculating the validation error at each epoch and choosing the lowest one (instead of choosing the lowest training error). Finally, a general hypothesis is that the gradient descent signals backpropagated from the last layers to the first layers are attenuated when they pass through the layers. To overcome this, we concatenated the input and output of certain layers as an input for later layers. It also allows us to fit a residual function and build short paths between early and later layers, that can serve to update the top parameters properly. We use ResNet as our main backbone, so the skip connection feature will be used in the encoder as well. Also, ResNet was used as a starter in our assignment 4 and decent results were obtained. We think that segmentation task can be treated as the combination of classification and depth detection (to separate the objects from background), thus why we believe that ResNet could have good performance for this project as well.

Original network (from which we derived our network, see [5])

○ Finally, it can be noticed that there are unavailable disparities. This is mainly caused by the stereo disocclusion effect on the left (or right) side of the left (or right) input image (i.e. the pixels out of the edge cannot be shifted). For the left (or right) output (disparity map), we notice disparity ramps on the left (or right) side. In order to reduce this effect, [5] introduces a post-processing method applied on the output. For that, we also need to compute the flipped disparity of the flipped input image (dL”) at test time. While it (dL”) is similar to the original output (dL), the disparity ramp is located on the right side instead. Therefore, we combine both by using a weight of 10% for dL (where its left margin is at) and a weight of 10% for dL'' (where its right margin is at) to form the final disparity map. The central part of the final disparity map is an average of both dL and dL''. The details of the PP implementation can be found in 5e).

Effect of post-processing on disparity map

3) Contribution section:

Sharhad Bashar:

- Ground Truth data set creator
- Literature review
- Calculating the Rotation, Transformation and Projection matricies
- Evaluation and Validation class, and debugging

Lizhe Chen:

- Implemented the single image disparity prediction model using ResNet as encoder
- Dataset research and paper research. Prepare the KITTI dataset loading pipeline for the PyTorch framework
- Designed & implemented the python classes and member functions as the tools for the network validation and training
- Implemented the post processing function to reduce the effect of stereo disocclusions
- Report: Dataset Description, ResNet Encoder Introduction

Gensëric Ghiro:

- Research papers reading
- Helped in the design of the network
- Construction and redaction of the final report
- Implemented the trainer, and debugging

Futian Zhang:

- Implemented data loader to load the dataset
- Implemented the data augmentation
- Implemented VGG model for training
- Helped integrate the project together

4) Outline section: overall structure of the report

• 5aa) Imports: Loading librairies required to run the code

• 5a) Dataloader: Data augmentation

• 5b) Network: Two implementations, one with vgg16 as the encoder, and the other one with resnet18 as the encoder

• 5c) Validator: typical validation loop of neural networks

• 5d) Trainer: typical training loop of neural networks

• 5e) Post-processing: Slightly enhances the smoothness of the image (as done in [5], see references below)

• 5f) Evaluation: Comparison of results with those from other sources

• 5g) Training (the 2 models)

• 6) Results display: Graph of the validation error, sample of outputs for both networks with and without post-processing applied and comparison with references

• 7) Conclusion: Discussing results

• 8) References

5aa) Imports

In [1]:
```
%matplotlib inline
```

In [2]:
```
# Before running this, make sure that loss.py is in the same folder as t
his file
# We did not include it here as part of the notebook, since it has been
 taken from [3] (see references) without modification
# and thus, it is not our own work
from loss import MonodepthLoss
```

In [3]:
```
from __future__ import absolute_import, division, print_function

import scipy
import skimage
from scipy.sparse.linalg import spsolve

import os
import glob
import random
import numpy as np
import copy
from PIL import Image
import torch
import time
from matplotlib import pyplot as plt

#5a)
import torch.utils.data as data
from torchvision import transforms
import torchvision.transforms.functional as tF

#5b)
import torch.nn as nn
import torch.nn.functional as F
import importlib
import torchvision.models as models

#5c)
import torch.optim as optim
from torch.utils.data import DataLoader

#5d)
import pickle

#5e)
from torch.utils.data.dataloader import *
import pandas as pd
```

5a) Dataloader

In [4]:
```
def pil_loader(path):
    # open path as file to avoid ResourceWarning
    # (https://github.com/python-pillow/Pillow/issues/835)
    with open(path, 'rb') as f:
        with Image.open(f) as img:
            return img.convert('RGB')


class JointRandomFlip(object):
    def __call__(self, L, R):
        if np.random.random_sample()>0.5:
            return (tF.hflip(R),tF.hflip(L))
        return (L,R)

class JointRandomColorAug(object):

    def __init__(self,gamma=(0.8,1.2),brightness=(0.5,2.0),color_shift=(
0.8,1.2)):
        self.gamma = gamma
        self.brightness = brightness
        self.color_shift = color_shift

    def __call__(self, L, R):
        if  np.random.random_sample()>0.5:

            random_gamma = np.random.uniform(*self.gamma)
            L_aug = L ** random_gamma
            R_aug = R ** random_gamma

            random_brightness = np.random.uniform(*self.brightness)
            L_aug = L_aug * random_brightness
            R_aug = R_aug * random_brightness

            random_colors = np.random.uniform(self.color_shift[0],self.c
olor_shift[1], 3)
            for i in range(3):
                L_aug[i, :, :] *= random_colors[i]
                R_aug[i, :, :] *= random_colors[i]

            # saturate
            L_aug = torch.clamp(L_aug, 0, 1)
            R_aug = torch.clamp(R_aug, 0, 1)

            return L_aug, R_aug

        else:
            return L, R

class JointToTensor(object):
    def __call__(self, L, R):
        return tF.to_tensor(L),tF.to_tensor(R)

class JointToImage(object):
    def __call__(self, L, R):
        return transforms.ToPILImage()(L),transforms.ToPILImage()(R)
```

```
class JointCompose(object):
    def __init__(self, transforms):
        """
        params:
           transforms (list) : list of transforms
        """
        self.transforms = transforms

    # We override the __call__ function such that this class can be
    # called as a function i.e. JointCompose(transforms)(img, target)
    # Such classes are known as "functors"
    def __call__(self, img, target):
        """
        params:
            img (PIL.Image)    : input image
            target (PIL.Image) : ground truth label
        """
        assert img.size == target.size
        for t in self.transforms:
            img, target = t(img, target)
        return img, target


class TwoViewDataset(data.Dataset):

    def __init__(self,
                 data_path,
                 resize_shape=(512,256),
                 is_train=False,
                 transforms=None,
                 sanity_check=None,
                 color='RGB'):
        super(TwoViewDataset, self).__init__()
        self.data_path = data_path

        self.interp = Image.ANTIALIAS
        self.resize_shape = resize_shape
        self.is_train = is_train
        self.transforms=transforms
        self.color = color

        if is_train:
            self.imgR_folder = os.path.join(data_path, "train", "image_r
ight")
            self.imgL_folder = os.path.join(data_path, "train", "image_l
eft")
        else:
            self.imgR_folder = os.path.join(data_path, "val", "image_rig
ht")
            self.imgL_folder = os.path.join(data_path, "val", "image_lef
t")


        self.imgR=[os.path.join(self.imgR_folder, x) for x in os.listdir
(self.imgR_folder)]
        self.imgL=[os.path.join(self.imgL_folder, x) for x in os.listdir
```

```
(self.imgL_folder)]


    def __len__(self):
        return len(list(glob.glob1(self.imgL_folder, "*.jpg")))

    def __getitem__(self, index):
        #print(np.array(Image.open(self.imgR[index]).convert('RGB')).sha
pe)
        colorR=Image.open(self.imgR[index]).convert(self.color).resize(s
elf.resize_shape)
        colorL=Image.open(self.imgL[index]).convert(self.color).resize(s
elf.resize_shape)
        #print(np.array(colorR).shape)

        if self.transforms is not None:
            colorR, colorL = self.transforms(colorR, colorL)
        return colorL, colorR
```

5b) Network 1 (with Resnet18 as encoder) and Network 2 (with VGG16 as encoder)

In [5]:
```
class get_disp(nn.Module):
    def __init__(self, num_in_channels):
        super(get_disp, self).__init__()
        self.p2d = (1, 1, 1, 1)
        self.disp = nn.Sequential(nn.Conv2d(num_in_channels, 2, kernel_s
ize=3, stride=1),
                                  nn.BatchNorm2d(2),
                                  torch.nn.Sigmoid())

    def forward(self, x):
        x = self.disp(F.pad(x, self.p2d))
        return 0.3 * x


class iconv(nn.Module):
    def __init__(self, num_in_channels, num_out_channels, kernel_size, s
tride):
        super(iconv, self).__init__()
        p = int(np.floor((kernel_size - 1) / 2))
        self.p2d = p2d = (p, p, p, p)

        self.iconv = nn.Sequential(nn.Conv2d(num_in_channels, num_out_ch
annels, kernel_size=kernel_size, stride=stride),
                                  nn.BatchNorm2d(num_out_channels))

    def forward(self, x):
        x = self.iconv(F.pad(x, self.p2d))
        return F.elu(x, inplace=True)


class iconv_dilate(nn.Module):
    def __init__(self, num_in_channels, num_out_channels, kernel_size, s
tride, dilation):
        super(iconv, self).__init__()
        p = int(np.floor((kernel_size - 1) / 2))
        self.p2d = p2d = (p, p, p, p)

        self.iconv = nn.Sequential(nn.Conv2d(num_in_channels, num_out_ch
annels, kernel_size=kernel_size, stride=stride, dilation=dilation),
                                  nn.BatchNorm2d(num_out_channels))

    def forward(self, x):
        x = self.iconv(F.pad(x, self.p2d))
        return F.elu(x, inplace=True)

class upconv(nn.Module):
    def __init__(self, num_in_channels, num_out_channels, kernel_size, s
cale):
        super(upconv, self).__init__()
        self.scale = scale
        self.conv1 = iconv(num_in_channels, num_out_channels, kernel_siz
e, 1)

    def forward(self, x):
        x = nn.functional.interpolate(x, scale_factor=self.scale, mode=
'bilinear', align_corners=True)
```

```
        return self.conv1(x)

class upconv_dilate(nn.Module):
    def __init__(self, num_in_channels, num_out_channels, kernel_size, s
cale, dilation):
        super(upconv, self).__init__()
        self.scale = scale
        self.conv1 = iconv(num_in_channels, num_out_channels, kernel_siz
e, 1, dilation)

    def forward(self, x):
        x = nn.functional.interpolate(x, scale_factor=self.scale, mode=
'bilinear', align_corners=True)
        return self.conv1(x)


###################### NETWORK 1 ######################
class ResnetDispModel(nn.Module):

    def __init__(self, num_input_channel=3, encoder='resnet18', pretrain
ed=True, dilate=False):
        super(ResnetDispModel, self).__init__()
        self.num_input_channel = num_input_channel


        assert encoder in ['resnet18', 'resnet34', 'resnet50', \
                           'resnet101', 'resnet152'], \
            "Incorrect encoder type"
        if encoder in ['resnet18', 'resnet34']:
            filters = [64, 128, 256, 512]
        else:
            filters = [256, 512, 1024, 2048]



        resnet = getattr(importlib.import_module("torchvision.models"),
encoder)(pretrained=pretrained)
        resnet_pool1 = list(resnet.children())[1:4]

        self.conv1 = resnet.conv1
        self.maxpool = nn.Sequential(*resnet_pool1)

        self.layer1 = resnet.layer1
        self.layer2 = resnet.layer2
        self.layer3 = resnet.layer3
        self.layer4 = resnet.layer4

        if dilate:
            self.upconv6 = upconv(filters[3], 512, 3, 2, 2)
            self.iconv6 = iconv_dilate(filters[2] + 512, 512, 3, 1, 2)

            self.upconv5 = upconv(512, 256, 3, 2, 3)
            self.iconv5 = iconv_dilate(filters[1] + 256, 256, 3, 1, 4)
        else:
            self.upconv6 = upconv(filters[3], 512, 3, 2)
            self.iconv6 = iconv(filters[2] + 512, 512, 3, 1)
```

```
            self.upconv5 = upconv(512, 256, 3, 2)
            self.iconv5 = iconv(filters[1] + 256, 256, 3, 1)

        self.upconv4 = upconv(256, 128, 3, 2)
        self.iconv4 = iconv(filters[0] + 128, 128, 3, 1)
        self.disp4_layer = get_disp(128)

        self.upconv3 = upconv(128, 64, 3, 1) #
        self.iconv3 = iconv(64 + 64 + 2, 64, 3, 1)
        self.disp3_layer = get_disp(64)

        self.upconv2 = upconv(64, 32, 3, 2)
        self.iconv2 = iconv(64 + 32 + 2, 32, 3, 1)
        self.disp2_layer = get_disp(32)

        self.upconv1 = upconv(32, 16, 3, 2)
        self.iconv1 = iconv(16 + 2, 16, 3, 1)
        self.disp1_layer = get_disp(16)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.xavier_uniform_(m.weight)

    def forward(self, x):
        # encoder
        x_conv1 = self.conv1(x)
        x_pool1 = self.maxpool(x_conv1)
        x1 = self.layer1(x_pool1)
        x2 = self.layer2(x1)
        x3 = self.layer3(x2)
        x4 = self.layer4(x3)
        # skips
        skip1 = x_conv1
        skip2 = x_pool1
        skip3 = x1
        skip4 = x2
        skip5 = x3

        # decoder
        upconv6 = self.upconv6(x4)
        concat6 = torch.cat((upconv6, skip5), 1)
        iconv6 = self.iconv6(concat6)

        upconv5 = self.upconv5(iconv6)
        concat5 = torch.cat((upconv5, skip4), 1)
        iconv5 = self.iconv5(concat5)

        upconv4 = self.upconv4(iconv5)
        concat4 = torch.cat((upconv4, skip3), 1)
        iconv4 = self.iconv4(concat4)
        self.disp4 = self.disp4_layer(iconv4)
        self.udisp4 = nn.functional.interpolate(self.disp4, scale_factor
=1, mode='bilinear', align_corners=True)
        self.disp4 = nn.functional.interpolate(self.disp4, scale_factor=
0.5, mode='bilinear', align_corners=True)

        upconv3 = self.upconv3(iconv4)
```

```
        concat3 = torch.cat((upconv3, skip2, self.udisp4), 1)
        iconv3 = self.iconv3(concat3)
        self.disp3 = self.disp3_layer(iconv3)
        self.udisp3 = nn.functional.interpolate(self.disp3, scale_factor
=2, mode='bilinear', align_corners=True)

        upconv2 = self.upconv2(iconv3)
        concat2 = torch.cat((upconv2, skip1, self.udisp3), 1)
        iconv2 = self.iconv2(concat2)
        self.disp2 = self.disp2_layer(iconv2)
        self.udisp2 = nn.functional.interpolate(self.disp2, scale_factor
=2, mode='bilinear', align_corners=True)

        upconv1 = self.upconv1(iconv2)
        concat1 = torch.cat((upconv1, self.udisp2), 1)
        iconv1 = self.iconv1(concat1)
        self.disp1 = self.disp1_layer(iconv1)

        return self.disp1, self.disp2, self.disp3, self.disp4

###################### NETWORK 2 ######################
class VGGDispModel(nn.Module):
    def __init__(self, num_input_channel = 3, encoder='vgg16', pretraine
d=True):
        super(VGGDispModel, self).__init__()
        self.num_input_channel = num_input_channel

        vgg16_bn = models.vgg16_bn(pretrained=True)
        # pretrained_dict = vgg16.state_dict()
        # model_dict = vgg16.state_dict()
        #
        # pretrained_dict = {k: v for k, v in pretrained_dict.items() if
k in model_dict}
        # model_dict.update(pretrained_dict)
        # vgg16.load_state_dict(model_dict)

        filters = [64, 128, 256, 512, 512]
        layers = list(vgg16_bn.children())[0]
        self.layer0 = nn.Sequential(*layers[0:6])
        self.layer1 = nn.Sequential(*layers[6:13])
        self.layer2 = nn.Sequential(*layers[13:23])
        self.layer3 = nn.Sequential(*layers[23:33])
        self.layer4 = nn.Sequential(*layers[33:43])

        self.upconv5 = upconv(filters[4], 256, 3, 2)
        self.iconv5 = iconv(filters[3] + 256, 256, 3, 1)

        self.upconv4 = upconv(256, 128, 3, 2)
        self.iconv4 = iconv(filters[2] + 128, 128, 3, 1)
        self.disp4_layer = get_disp(128)

        self.upconv3 = upconv(128, 64, 3, 2) #
        self.iconv3 = iconv(filters[1] + 64 + 2, 64, 3, 1)
        self.disp3_layer = get_disp(64)

        self.upconv2 = upconv(64, 32, 3, 2)
        self.iconv2 = iconv(filters[0] + 32 + 2, 32, 3, 1)
```

```
        self.disp2_layer = get_disp(32)

        self.upconv1 = upconv(32, 16, 3, 2)
        self.iconv1 = iconv(16 + 2, 16, 3, 1)
        self.disp1_layer = get_disp(16)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.xavier_uniform_(m.weight)

    def forward(self, x):
        # encoder
        x0 = self.layer0(x)
        x1 = self.layer1(x0)
        x2 = self.layer2(x1)
        x3 = self.layer3(x2)
        x4 = self.layer4(x3)
        # skips
        skip1 = x0
        skip2 = x1
        skip3 = x2
        skip4 = x3

        # decoder

        upconv5 = self.upconv5(x4)
        concat5 = torch.cat((upconv5, skip4), 1) # 512
        iconv5 = self.iconv5(concat5)

        upconv4 = self.upconv4(iconv5)
        concat4 = torch.cat((upconv4, skip3), 1) # 256
        iconv4 = self.iconv4(concat4)
        self.disp4 = self.disp4_layer(iconv4)
        self.udisp4 = nn.functional.interpolate(self.disp4, scale_factor
=2, mode='bilinear', align_corners=True)
        self.disp4 = nn.functional.interpolate(self.disp4, scale_factor=
0.5, mode='bilinear', align_corners=True)

        upconv3 = self.upconv3(iconv4)

        concat3 = torch.cat((upconv3, skip2, self.udisp4), 1) #128
        iconv3 = self.iconv3(concat3)
        self.disp3 = self.disp3_layer(iconv3)
        self.udisp3 = nn.functional.interpolate(self.disp3, scale_factor
=2, mode='bilinear', align_corners=True)
        self.disp3 = nn.functional.interpolate(self.disp3, scale_factor=
0.5, mode='bilinear', align_corners=True)

        upconv2 = self.upconv2(iconv3)
        concat2 = torch.cat((upconv2, skip1, self.udisp3), 1) #64
        iconv2 = self.iconv2(concat2)
        self.disp2 = self.disp2_layer(iconv2)
        self.udisp2 = nn.functional.interpolate(self.disp2, scale_factor
=2, mode='bilinear', align_corners=True)
        self.disp2 = nn.functional.interpolate(self.disp2, scale_factor=
0.5, mode='bilinear', align_corners=True)
```

```
        upconv1 = self.upconv1(iconv2)
        concat1 = torch.cat((upconv1, self.udisp2), 1)
        iconv1 = self.iconv1(concat1)
        self.disp1 = self.disp1_layer(iconv1)
        self.disp1 = nn.functional.interpolate(self.disp1, scale_factor=
0.5, mode='bilinear', align_corners=True)



        return self.disp1, self.disp2, self.disp3, self.disp4
```

5c) Validator

In [6]:

```
class Validator:
    def __init__(self, val_loader, batch_size, params_file=None, use_gpu
=False):
        self.use_gpu = use_gpu
        self.params_file = params_file
        self.val_loader = val_loader
        if use_gpu :
            self.device = "cuda:0"
        else:
            self.device = "cpu"
        self.loss = MonodepthLoss(
            n=4,
            SSIM_w=0.85,
            disp_gradient_w=0.1, lr_w=1).to(self.device)
        self.val_losses = []
        self.batch_size = batch_size


    def validate(self, network):

        network.eval()

        total_loss = 0
        counter = 0
        for i, data in enumerate(self.val_loader):
            left, right = data

            if self.use_gpu:
                left = left.cuda()
                network = network.cuda()
                right = right.cuda()

            model_outputs = network(left)

            loss = self.loss(model_outputs, [left, right])
            self.val_losses.append(loss.item())
            total_loss += loss.item()
            counter += 1

        total_loss /= counter

        return total_loss
```

5d) Trainer

In [7]:
```
class Trainer:
    def __init__(self, network, train_loader, optimizer, batch_size, mod
el, params_file=None, use_gpu=False):
        self.net = network
        self.use_gpu = use_gpu
        self.optimizer = optimizer
        self.validator = None
        self.history = {"Train": [], "Val": []}
        self.params_file = params_file
        self.train_loader = train_loader
        self.batch_size = batch_size
        self.model = model
        if use_gpu :
            self.device = "cuda:0"
        else:
            self.device = "cpu"

        self.loss_function = MonodepthLoss(
            n=4,
            SSIM_w=0.85,
            disp_gradient_w=0.1, lr_w=1).to(self.device)


    def setValidator(self, validator):
        self.validator = validator

    def setOptimizer(self, opt):
        self.optimizer = opt

    def saveParams(self, path):
        torch.save(self.net.state_dict(), path)

    def loadModel(self, path):
        self.net.load_state_dict(torch.load(path))

    def train(self):
        total_loss = 0.0

        self.net.train()
        counter = 0
        for i, data in enumerate(self.train_loader):
            left, right = data
            if self.use_gpu:
                left = left.cuda()
                self.net = self.net.cuda()
                right = right.cuda()


            self.optimizer.zero_grad()
            disps = self.net(left)
            loss = self.loss_function(disps, [left, right])
            loss.backward()
            self.optimizer.step()
            total_loss += loss.item()
            counter += 1
```

```
        main_loss = total_loss / counter

        return main_loss

    def run_train(self, epoch):
        if self.params_file:
            self.loadModel(self.params_file)
        save_name = "params" + self.model + ".pkl"
        prev_score = np.inf
        if self.validator:
            prev_score = self.validator.validate(self.net)

        for e in range(epoch):

            loss = self.train()
            print("Epoch: {} Loss: {}".format(e, loss))
            self.history["Train"].append(loss)

            if self.validator:
                val_score = self.validator.validate(self.net)
                self.history["Val"].append(val_score)
                if val_score < prev_score:
                    print("update model file with prev_score {} and curr
ent score {}".format(prev_score, val_score))
                    self.saveParams(save_name)
                    prev_score = val_score

            with open("train_history" + self.model + ".pickle", 'wb') as
handle:
                pickle.dump(self.history, handle, protocol=pickle.HIGHES
T_PROTOCOL)

    def copyNetwork(self):
        return copy.deepcopy(self.net)
```

5e) Post-processing

In [8]:
```
def postprocess(image, network): #input is PIL image

    spF = tF.to_tensor(tF.hflip(image)).unsqueeze(0)
    sp = tF.to_tensor(image).unsqueeze(0)
    disp = network(sp)[0][0][0].detach().numpy()
    f_disp = network(spF)[0][0][0].detach().numpy()
    width = disp.shape[-1]
    height = disp.shape[-2]
    dl = disp
    d_l = np.fliplr(f_disp)
    wl = np.zeros((height, width), dtype=np.float)
    for i in range(height):
        for j in range(width):
            if (j / width) <= 0.1:
                wl[i, j] = 1.0
            elif (j / width) > 0.2:
                wl[i, j] = 0.5
            else:
                wl[i, j] = 5 * (0.2 - (j / width)) + 0.5

    w_l = np.fliplr(wl)

    return dl * wl + d_l * w_l
```

5f) Evaluation

In [9]:
```
#### Adaptation from [6] (original can be found under  monodepth/utils/e
valuation_utils.py)

class Evaluation():
    def __init__(self):
        self.monodepthLoss = MonodepthLoss()

    def __title__(self):
        return ['Abs Rel', 'Sq Rel', 'RMSE', 'RMSE log', '\u03B4 < 1.25'
, '\u03B4 < \$1.25^2$', '\u03B4 < \$1.25^3$']

    def absRelCalc(self, original, prediction):
        absRel = np.mean(np.abs(original - prediction) / (original + 1))
        return absRel

    def sqRelCalc(self, original, prediction):
        sqRel = np.mean(((original - prediction)**2) / (original + 1))
        return sqRel

    def rmseCalc(self, original, prediction):
        rmse = (original - prediction) ** 2
        rmse = np.sqrt(rmse.mean())
        return rmse * 100

    def rmseLogCalc(self, original, prediction):
        rmseLog = (np.log(original + 1) - np.log(prediction + 1)) ** 2
        rmseLog = np.sqrt(rmseLog.mean())
        return rmseLog

    def deltaCalc(self, original, prediction):
        thresh = np.maximum((original / prediction), (prediction / origi
nal))
        delta1 = (thresh < 1.25   ).mean()
        delta2 = (thresh < 1.25 ** 2).mean()
        delta3 = (thresh < 1.25 ** 3).mean()
        return delta1, delta2, delta3

    def computeErrors(self, original, prediction):
        imLeft = self.monodepthLoss.generate_image_right(original, predi
ction.cpu())
        original = original.detach().numpy()
        prediction = imLeft.detach().numpy()

        absRel = self.absRelCalc(original, prediction)
        sqlRel = self.sqRelCalc(original, prediction)
        rmse = self.rmseCalc(original, prediction)
        rmseLog = self.rmseLogCalc(original, prediction)
        delta1, delta2, delta3 = self.deltaCalc(original, prediction)
        return absRel, sqlRel, rmse, rmseLog, delta1, delta2, delta3
```

5g) Training Network with resnet18 encoder and with vgg16 encoder

In [10]:
```
use_gpu = True
use_pretrained_params = True # if set to true, make sure that you have p
aramsRes.pickle and paramsVGG.pickle (those
                              # parameters were trained with our model an
d saved)
                              # otherwise, if set to False, the training
 for both models will be done again (takes around
                              # 10 hours for both models on a GTX 1080 TI
for 50 epochs)
epochs = 50

#Change path ("dataset/dataset/") to where data is loaded
val_dataset = TwoViewDataset("dataset/dataset/", is_train=False, transfo
rms=JointToTensor())
trn_dataset = TwoViewDataset("dataset/dataset/", is_train=True, transfor
ms=JointToTensor())
```

In [11]:
```
## Resnet18
num_workers = 8 # NUM_WORKERS must be 0 for Windows (can be anything els
e otherwise)
val_loader = data.DataLoader(val_dataset, batch_size=8, num_workers=num_
workers, shuffle=False)
trn_loader = data.DataLoader(trn_dataset, batch_size=8, num_workers=num_
workers, shuffle=False)

networkRes = ResnetDispModel(3)

if use_pretrained_params: #make sure to have paramsRes.pkl in the folder
    networkRes.load_state_dict(torch.load("paramsRes.pkl", map_location=
torch.device('cpu')))
    networkRes.eval()
else:
    val_Res = Validator(val_loader = val_loader, batch_size = 1, use_gpu
=use_gpu)
    opt_Res = torch.optim.SGD(networkRes.parameters(), lr=1e-2, weight_d
ecay=1e-6,momentum=0.5, nesterov=False)
    trn_Res = Trainer(network = networkRes, train_loader = trn_loader,
                      optimizer = opt_Res, batch_size = 8, model = "Res"
, use_gpu=use_gpu)
    trn_Res.setValidator(validator = val_Res)
    trn_Res.run_train(epoch = epochs)
    trained_net_res = networkRes
```

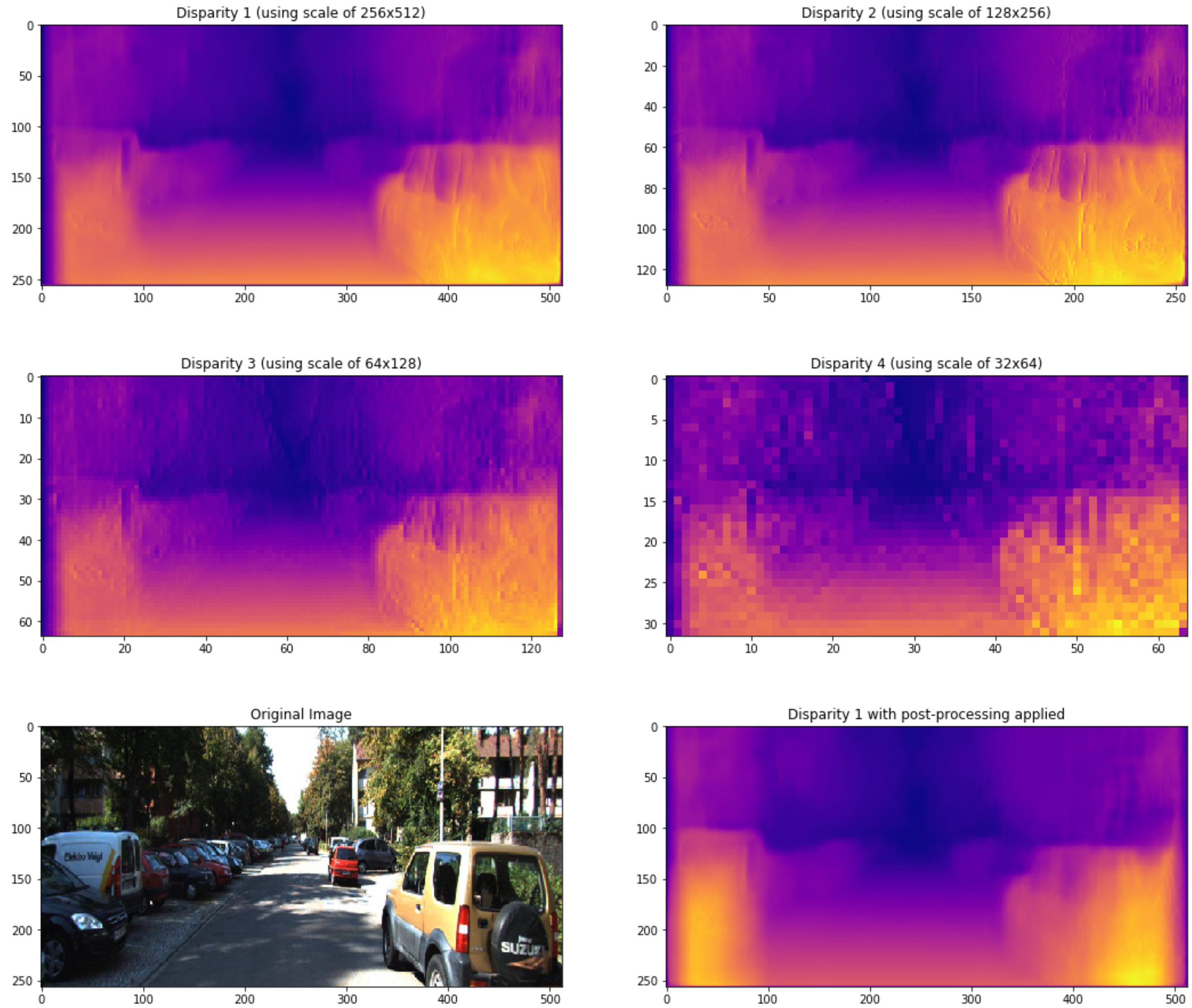
In [12]:
```
## VGG16
num_workers = 2 # NUM_WORKERS must be 0 for Windows (can be anything els
e otherwise)
val_loader = data.DataLoader(val_dataset, batch_size=2, num_workers=num_
workers, shuffle=False)
trn_loader = data.DataLoader(trn_dataset, batch_size=2, num_workers=num_
workers, shuffle=False)

networkVGG = VGGDispModel(3)

if use_pretrained_params: #make sure to have paramsVGG.pkl in the folder
    networkVGG.load_state_dict(torch.load("paramsVGG.pkl", map_location=
torch.device('cpu')))
    networkVGG.eval()
else:
    val_VGG = Validator(val_loader = val_loader, batch_size = 1, use_gpu
=use_gpu)
    opt_VGG = torch.optim.SGD(networkVGG.parameters(), lr=1e-2, weight_d
ecay=1e-6,momentum=0.5, nesterov=False)
    trn_VGG = Trainer(network = networkVGG, train_loader = trn_loader,
                      optimizer = opt_VGG, batch_size = 1, model = "VGG"
, use_gpu=use_gpu)
    trn_VGG.setValidator(validator = val_VGG)
    trn_VGG.run_train(epoch = epochs)
    trained_net_VGG = networkVGG
```

6) Results

In [13]:
```
sample_postprocessed = TwoViewDataset("dataset/dataset/", is_train=False
)[105][0]
sample = TwoViewDataset("dataset/dataset/", is_train=False, transforms=J
ointToTensor())[105][0]
sample = sample.unsqueeze(0)

if (not use_pretrained_params):
    sample = sample.cuda()
    sample_postprocessed = sample_postprocessed.cuda()

disp1_Res, disp2_Res, disp3_Res, disp4_Res = networkRes(sample)
disp_postprocessed_Res = postprocess(sample_postprocessed, networkRes)
disp1_VGG, disp2_VGG, disp3_VGG, disp4_VGG = networkVGG(sample)
disp_postprocessed_VGG = postprocess(sample_postprocessed, networkVGG)
```

In [14]:
```
if (not use_pretrained_params):
    sample = sample.cpu()
    sample_postprocessed = sample_postprocessed.cpu()

plt.figure(2, figsize=(18, 16))
plt.suptitle("Disparity maps when using Resnet18 as encoder", fontsize=3
0)
plt.subplot(321)
plt.imshow(np.squeeze(np.array(disp1_Res.cpu().detach().numpy()))[0], cm
ap="plasma")
plt.title("Disparity 1 (using scale of 256x512)")
plt.subplot(322)
plt.imshow(np.squeeze(np.array(disp2_Res.cpu().detach().numpy()))[0], cm
ap="plasma")
plt.title("Disparity 2 (using scale of 128x256)")
plt.subplot(323)
plt.imshow(np.squeeze(np.array(disp3_Res.cpu().detach().numpy()))[0], cm
ap="plasma")
plt.title("Disparity 3 (using scale of 64x128)")
plt.subplot(324)
plt.imshow(np.squeeze(np.array(disp4_Res.cpu().detach().numpy()))[0], cm
ap="plasma")
plt.title("Disparity 4 (using scale of 32x64)")
plt.subplot(325)
plt.imshow(np.transpose(np.squeeze(np.array(sample)), axes=(1,2,0)))
plt.title("Original Image")
plt.subplot(326)
plt.imshow(disp_postprocessed_Res, cmap = "plasma")
plt.title("Disparity 1 with post-processing applied")
plt.show()
```

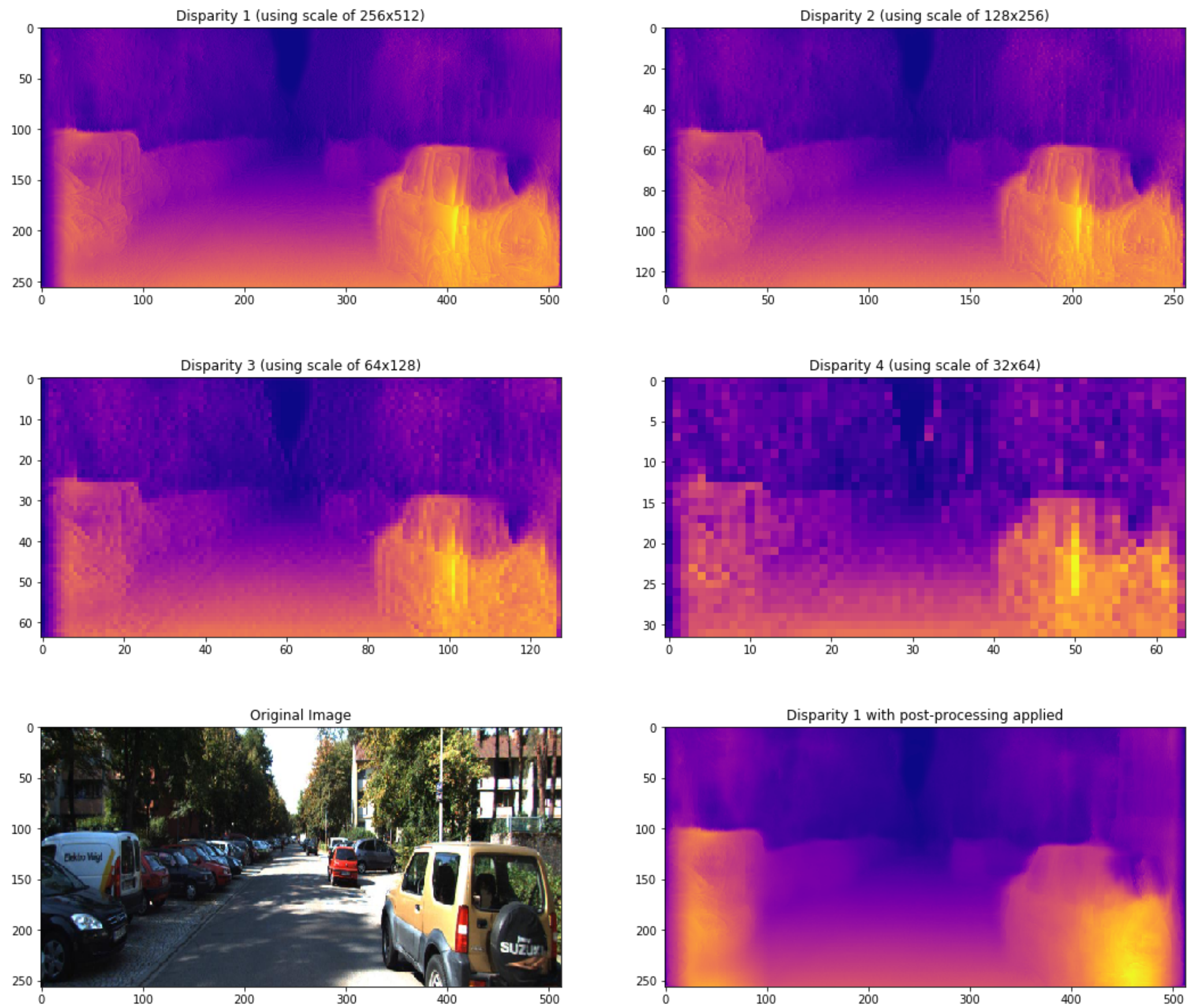
Disparity maps when using Resnet18 as encoder



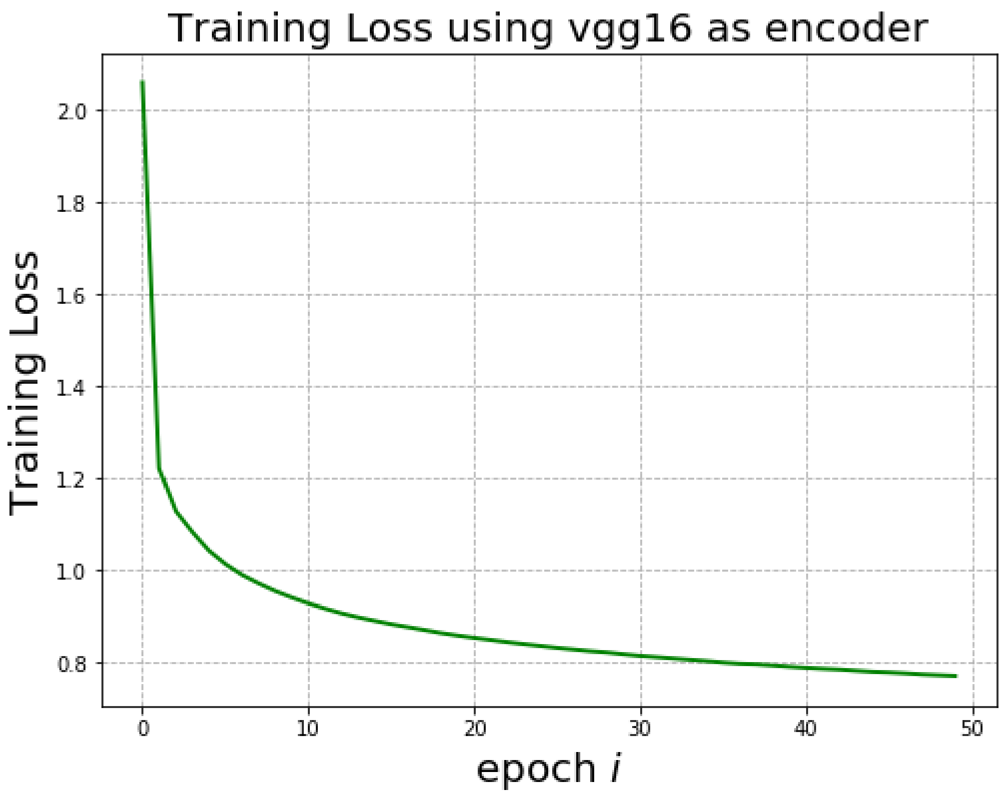
In [15]:
```
with open('train_historyRes.pickle', 'rb') as handle:
    data_Res = pickle.load(handle)
epoch = np.arange(len(data_Res["Train"]))
fig = plt.figure(3, figsize=(8, 6))
plt.plot(epoch, data_Res["Train"], linewidth=2.0, color ="green")
plt.title("Training Loss using resnet18 as encoder", size = 20)
plt.xlabel("epoch $i$", fontsize=20)
plt.ylabel("Training Loss", fontsize=20)
plt.grid(linestyle='dashed')
plt.show()
```

In [16]:
```
if (not use_pretrained_params):
    sample = sample.cpu()
    sample_postprocessed = sample_postprocessed.cpu()

plt.figure(4, figsize=(18, 16))
plt.suptitle("Disparity maps when using VGG16 as encoder", fontsize=30)
plt.subplot(321)
plt.imshow(np.squeeze(np.array(disp1_VGG.cpu().detach().numpy()))[0], cm
ap="plasma")
plt.title("Disparity 1 (using scale of 256x512)")
plt.subplot(322)
plt.imshow(np.squeeze(np.array(disp2_VGG.cpu().detach().numpy()))[0], cm
ap="plasma")
plt.title("Disparity 2 (using scale of 128x256)")
plt.subplot(323)
plt.imshow(np.squeeze(np.array(disp3_VGG.cpu().detach().numpy()))[0], cm
ap="plasma")
plt.title("Disparity 3 (using scale of 64x128)")
plt.subplot(324)
plt.imshow(np.squeeze(np.array(disp4_VGG.cpu().detach().numpy()))[0], cm
ap="plasma")
plt.title("Disparity 4 (using scale of 32x64)")
plt.subplot(325)
plt.imshow(np.transpose(np.squeeze(np.array(sample)), axes=(1,2,0)))
plt.title("Original Image")
plt.subplot(326)
plt.imshow(disp_postprocessed_VGG, cmap = "plasma")
plt.title("Disparity 1 with post-processing applied")
plt.show()
```

Disparity maps when using VGG16 as encoder



In [17]:
```
with open('train_historyVGG.pickle', 'rb') as handle:
    data_VGG = pickle.load(handle)
epoch = np.arange(len(data_VGG["Train"]))
fig = plt.figure(5, figsize=(8, 6))
plt.plot(epoch, data_VGG["Train"], linewidth=2.0, color ="green")
plt.title("Training Loss using vgg16 as encoder", size = 20)
plt.xlabel("epoch $i$", fontsize=20)
plt.ylabel("Training Loss", fontsize=20)
plt.grid(linestyle='dashed')
plt.show()
```



In [18]:

```
np.seterr(divide='ignore', invalid='ignore')
monodepthLoss = MonodepthLoss()
im_left = monodepthLoss.generate_image_right(sample, disp1_VGG.cpu())

gt = sample * 255
pred = im_left * 255
rmse = (gt - pred) ** 2
rmse = rmse.detach().numpy()
rmse = np.sqrt(rmse.mean())

pred = np.transpose(np.squeeze(im_left.detach().numpy()), axes=(1,2,0))
original = np.transpose(np.squeeze(sample.detach().numpy()), axes=(1,2,0
))
rmse = np.sqrt(np.sum((pred[pred> 0.0] - original[pred>0.0])**2)/np.sum(
pred>0.0))


evalutation = Evaluation()
absRel, sqlRel, rmse, rmseLog, delta1, delta2, delta3 = evalutation.comp
uteErrors(sample, disp1_Res)

## data was taken from [5] and is presented here only for comparison pur
poses
data = [['Monodepth with 3D', 0.412, 16.37, 13.693, 0.512, 0.690, 0.833,
0.891],
        ['Monodepth with 3Ds', 0.151, 1.312, 6.344, 0.239, 0.781, 0.931,
0.976],
        ['Monodepth with no LR', 0.123, 1.417, 6.315, 0.220, 0.841, 0.93
7, 0.973],
        ['Monodepth', 0.124, 1.388, 6.125, 0.217, 0.841, 0.936, 0.975],
        ['Monodepth pp', 0.100, 0.934, 5.141, 0.178, 0.878, 0.961, 0.986
],
        ['Monodepth resnet pp', 0.097, 0.896, 5.093, 0.176, 0.879, 0.962
, 0.986],
        ['Ours resnet pp', 0.805, 0.319, 21.336, 0.149, 0.516, 0.686, 0.
774],
        ['Ours VGG', 0.771, 0.245, 18.892, 0.132, 0.460, 0.647, 0.753]
       ]

df = pd.DataFrame(data, columns = ['Method', 'Abs Rel', 'Sq Rel', 'RMSE'
, 'RMSE log', '\u03B4 < 1.25', '\u03B4 < $1.25^2$', '\u03B4 < $1.25^3$'
])
df
```

```
c:\program files (x86)\python\lib\site-packages\torch\nn\functional.py:
2693: UserWarning: Default grid_sample and affine_grid behavior will be
changed to align_corners=False from 1.4.0. See the documentation of gri
d_sample for details.
  warnings.warn("Default grid_sample and affine_grid behavior will be c
hanged "
```

Out[18]:

	Method	Abs Rel	Sq Rel	RMSE	RMSE log	δ < 1.25	δ < 1.25^2	δ < 1.25^3
0	Monodepth with 3D	0.412	16.370	13.693	0.512	0.690	0.833	0.891
1	Monodepth with 3Ds	0.151	1.312	6.344	0.239	0.781	0.931	0.976
2	Monodepth with no LR	0.123	1.417	6.315	0.220	0.841	0.937	0.973
3	Monodepth	0.124	1.388	6.125	0.217	0.841	0.936	0.975
4	Monodepth pp	0.100	0.934	5.141	0.178	0.878	0.961	0.986
5	Monodepth resnet pp	0.097	0.896	5.093	0.176	0.879	0.962	0.986
6	Ours resnet pp	0.805	0.319	21.336	0.149	0.516	0.686	0.774
7	Ours VGG	0.771	0.245	18.892	0.132	0.460	0.647	0.753

7) Conclusion: summary of observations & results, and limitations & future possible improvements

○ First of all, we get a typical training loss curve (i.e. monotonically decreasing) for both encoders, which means that our optimizer was able to correctly minimize the error. The disparity maps (1-4) for both encoders look decently accurate; in other words, while it might not be as precise as the state-of-art monodepth ([1]), there is a good distinction between depths. Obviously, disparity maps 2 to 4 do not look as good as the first one, but this is only because their scale is reduced and we plotted them as if it were the same size as the first one. Although the computation time is doubled, the final post-processing leads to both better accuracy and less visual artifacts, as intended. Finally, while there are almost no differences between the two encoders, we can still notice that vgg16 "squeezed" the width of the image, and as a result, the output does not look as similar as the original image than it does with resnet18.

○ As previously metioned, our results are not as good as the state-of-art monodepth ([1]). This is something expected, as our implementation had many flaws. For starters, Resnet class can support hundreds or more convolutional layers. However, in our implementation, only the first five layers of resnet18 were used as the encoder, which considerably limits resnet's potential to generalize to unseen data, due to an increased simplicity of the network. Also, we are only using a small portion of the KITTI dataset as we don't have the material (CPUs) to run our network over the entire dataset. This, again, reduces our network's potential to learn over unseen data, as it does not have enough data to generalize. Third, we know that neural networks are typically applied to transformed images. Unfortunetaly, we did not succeed in implementing color augmentation. Therefore, we only used Random Flip, which might not be enough. Maybe more or different transforms would lead to better results (by a better data augmentation).

○ For future works, it would be interesting to see how the size of the dataset affects the performance of the network. Training with larger dataset will definitely improve the results, but it could also be relevant to study how to get better results with a small dataset, while avoiding overfitting. One possible way is to improve our data augmentation step with a larger variety of methods, such as color augmentation or random crop. A better augmentation strategy would improve the generalization of the network even with small training dataset. We should also test and improve the network’s transferring abilities. For example, we should test our network with different input resolutions to see whether it is scale invariant or not, since in some circumstances or applications the input may be captured by low-resolution cameras (e.g. cell phone camera). However, it probably would be harder to extract the features from low-resolution images. Also, we should test the network with different scene types. All images from the KITTI dataset are outdoor scenes, but it would be interesting to find out if our network could be transferred to indoor scenes instead, such as NYU depth. Since the light distribution and the depth range will be very different, we expect that our current network won’t have good performance on such dataset and modifications will likely be required.

8) References

[1] Godard, C., Mac Aodha, O., Firman, M. and Gabriel, G. (2019). "Digging Into Self-Supervised Monocular Depth Estimation". Retrieved 13 December 2019, from https://arxiv.org/pdf/1806.01260 (https://arxiv.org/pdf/1806.01260).

[2] Zhan, H., Weerasekera, C., Garg, R. and Reid, I. (2019). "Self-supervised Learning for Single View Depth and Surface Normal Estimation". Retrieved 13 December 2019, from https://arxiv.org/abs/1903.00112 (https://arxiv.org/abs/1903.00112).

[3] OniroAI, MonoDepth-PyTorch, (2018), GitHub repository, https://github.com/OniroAI/MonoDepth-PyTorch (https://github.com/OniroAI/MonoDepth-PyTorch)

[4] Pillai, S., Ambrus, R. and Gaidon, A. (2018). SuperDepth: Self-Supervised, Super-Resolved Monocular Depth Estimation. Retrieved 13 December 2019, from https://arxiv.org/abs/1810.01849 (https://arxiv.org/abs/1810.01849).

[5] Godard, C., Mac Aodha, O., & Brostow, G. (2017). "Unsupervised Monocular Depth Estimation with Left-Right Consistency". Retrieved 13 December 2019, from https://arxiv.org/abs/1609.03677 (https://arxiv.org/abs/1609.03677)

[6] mrharicot/monodepth, (2017), GitHub repository, https://github.com/mrharicot/monodepth (https://github.com/mrharicot/monodepth)

[7] Garg, R., Kumar, V., Carneiro, G. and Reid, I. (2016). "Unsupervised CNN for Single View Depth Estimation: Geometry to the Rescue". Retrieved 13 December 2019, from https://arxiv.org/pdf/1603.04992 (https://arxiv.org/pdf/1603.04992).