实验报告

实验要求：图像视差匹配，通过立体匹配得到两张图像的视差图，需要详细的实验过程和结果分析。

1. **概念介绍**

1.立体匹配

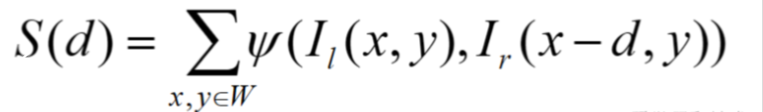
立体匹配的基本概念:

匹配目标: 在立体匹配中，主要目标是确定左图像中像素的右图像中的对应像素。这个对应像素通常位于相同的行。

视差（Disparity）: 视差 d 是右图像中对应像素 xr 和左图像中像素 xl 之间的水平位置差。视差是深度信息的关键指标。

匹配方法:方法涉及在左图像中以某个像素为中心取一个窗口 W，然后将这个窗口沿水平方向平移视差 d，并将其放置在右图像中。接着比较左图像中窗口 W 和右图像中平移后窗口 W 内的像素强度值。

比较度量:比较度量公式通常有如下形式：



这里 S(d) 是某个特定视差 d 的匹配得分。

I\_l(x, y) 和 I\_r(x+d, y) 分别是左图像和右图像中相应位置的像素强度值。

求和是对窗口 W 中所有像素进行的，意味着比较的是整个窗口的像素强度差的总和或平均值。

分析:这种方法基于的假设是，左图像中的窗口和右图像中相对应的窗口在视觉内容上应该是相似的。通过调整视差 d 并计算每个可能的 d 值的匹配得分 S(d)，可以确定最佳匹配视差，即得分最低（或最高，取决于具体实现）的那个 d。这种方法有效但计算量较大，尤其是在处理大尺寸图像和大窗口时。

总的来说，这个立体匹配方法是通过比较左图像中的一个窗口和右图像中相应平移的窗口来寻找最佳匹配视差的一种经典方法。它在众多应用中被广泛使用，如3D重建、机器人导航等。

1. **HSM**

1. 研究问题

立体视觉对远场深度感知不可靠，因为基于三角测量的立体系统的深度误差随深度成二次方增加，这意味着立体匹配将提供不稳定的远场深度估计。而高分辨率是解决远场深度估计不稳定的一个有前景的解决方法，这也给自动驾驶提供了立体视觉的一个解决方案，但SOTA无法有效处理高分辨率图像，原因有两点：

网络不够高效

缺乏高分辨率训练数据

2. 研究方法

（1）提出由粗到精的分层立体匹配网络HSMNet，处理高分辨率图像，并且能够按需输出视差图，达到性能和速度的权衡，能够用于自动驾驶中。

（2）合成一个来自城市模拟器的高分辨率的立体数据集用作训练，并且收集了一个来自自动驾驶汽车的真实高分辨率数据集用作测试。

（3）提出一种非对称的数据增强策略，以提高模型对标定误差、曝光变化和相机遮挡的鲁棒性。

2.1 分层立体匹配 (HSM) 网络

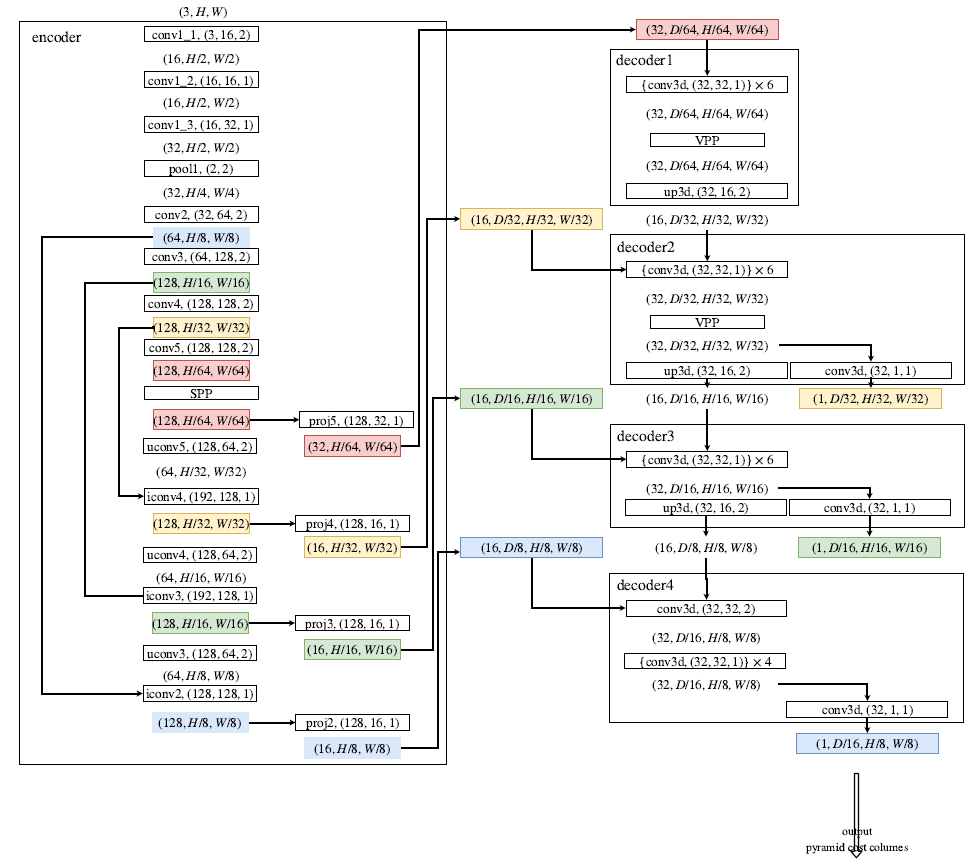
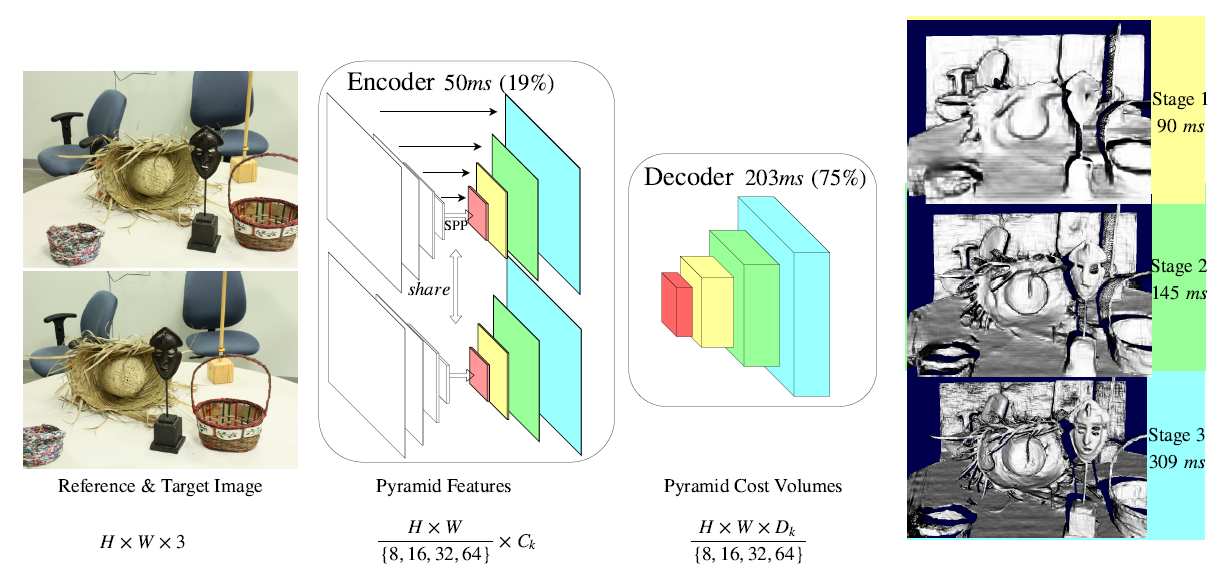
设计分层从粗到细网络的核心思想是首先通过网络积极地对高分辨率图像进行下采样，同时提取多尺度特征，然后使用潜在的对应关系逐渐建立一个分辨率增加的成本量金字塔。

2.1.1 设计原则

采用空间金字塔池化 (SPP)，增加感受野，提取多尺度特征。SPP原文中，将多尺度特征上采样到原始分辨率，为了减少内存，将池化特征保持在原始粗分辨率。

在视差维度上跨越的 3D 卷积，使能够有效地处理高分辨率成本量。

采用多尺度损失函数。



2.1.2 特征金字塔编码器

使用特征金字塔编码器来提取描述符以进行粗到细匹配。

为了在保持粗尺度信息的同时有效地提取具有不同细节级别的特征，采用了具有跳过连接的编码器-解码器架构。的特征编码器由带有4个残差块的自定义 resnet 主干组成，然后是4个SPP 层（同样，以有限的计算和内存增加感受野）。通过串联融合 SPP 之前和之后的特征，并应用 up-convs（上采样后的卷积），逐渐增加特征图的分辨率。

2.1.3 特征体积

为了控制特征体积的大小，将特征投影到较低维的子空间，然后通过考虑沿水平扫描线的潜在匹配描述符对之间的视差来构建4D特征体积。构建了一个由4个体积组成的金字塔，每个体积都具有增加的空间分辨率和增加的视差分辨率。

2.1.4 特征体积解码器

通过 4 个解码器块从粗到细过滤特征量。第一个和第二个块包含一个 VPP 层来聚合上下文信息。还在解码器块的末尾应用 3D 上卷积（三线性上采样后的卷积），其输出与更精细的原始特征量融合，以逐渐增加空间和视差维度分辨率。最后，在每个尺度上进行成本量预测，并通过可微 soft argmin 计算​​差异。

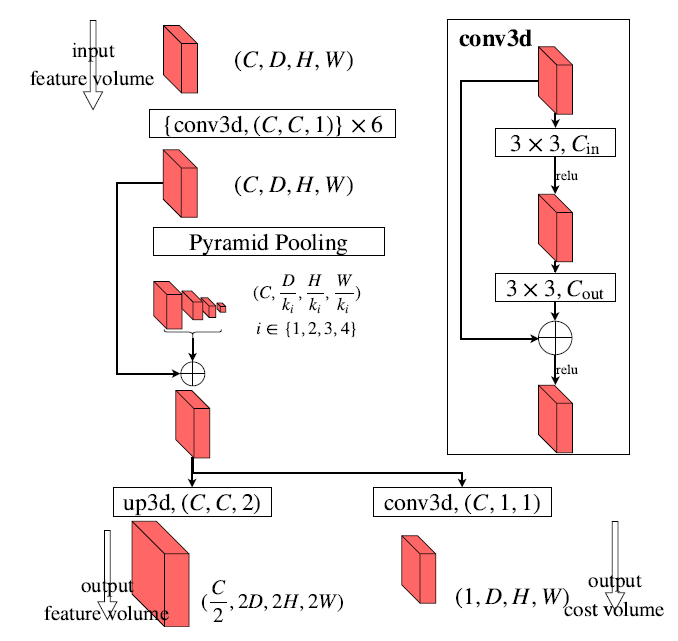
图 4 显示了每个特征体积的解码或过滤。让首先将 conv3D块 定义为两个具有残差连接的 3D 卷积。

特征量由 6 个conv3D块过滤。

与特征提取的情况一样，然后应用体积金字塔池化（将 SPP 扩展到特征量）来生成特征，这些特征可以为高分辨率输入捕获足够的全局上下文。

输出被三线性上采样到更高的空间（和视差）分辨率，以便它可以与金字塔中的下一个 4D 特征体积融合。

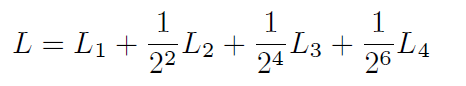
为了报告根据当前比例计算的按需差异，输出用另一个 conv3D 块处理以生成 3D 输出成本量。在计算金字塔下游的后续特征量之前，此成本量可以直接报告差异。



2.1.5 多尺度损失

训练网络在训练阶段进行不同尺度的预测，这允许在任何金字塔级别按需输出视差，并且还用于正则化整个网络：

其中损失被缩放以考虑在每个金字塔级别增加的视差分辨率。L1 代表最精细级别的损失，L4 是最粗级别的损失。自然损失是当前金字塔级别的候选视差上的 softmax 分布。



2.2 立体数据增强

提出了 3 种非对称增强技术，以针对基于学习的方法相应地解决标定误差、曝光变化和相机遮挡这些问题。

2.2.1 y-视差增强

大多数立体系统假设相机经过完美校准并且对应关系位于同一水平扫描线上。 然而，很难完美地校准高分辨率图像对，尤其是在温度变化大和振动大的情况下 [9]。 此类错误会导致具有 y 分量的地面实况视差匹配（例如，匹配到不同的水平扫描线）。

分级匹配通过在粗尺度上偏置匹配来部分缓解这个问题，其中校准误差不那么严重。另一种方法是强制网络在训练时间内学习对此类错误的鲁棒性。 请注意，相机校准中的误差可以由单应性 H2 R3 3表示，导致 Iimperfect(x) = Iperfect(!(x;Hx)) 其中 x 是图像坐标。为了模拟真实世界的校准误差，根据校准误差矩阵扭曲目标视图图像。为了进一步限制扭曲图像的空间，将 H 限制为刚性 2D 变换。

2.2.2 非对称色度增强

立体相机在不同的光照和曝光条件下不可避免，例如一个相机在阴影下。因此，出于安全考虑，使算法对这种成像不对称性具有鲁棒性至关重要。通过对参考图像和目标图像应用不同的色度增强来实现这一目标，希望的特征编码网络可以学习到对此类成像变体鲁棒的表示。

2.2.3 非对称掩蔽

大多数立体匹配算法假设目标视图中始终存在对应关系。但是，当发生遮挡或难以找到对应关系时，此假设不成立。另一方面，发现单眼线索（例如形状和连续性）以及上下文信息有助于估计差异。为了迫使模型更多地依赖上下文线索，应用了非对称掩码，它用整个图像的平均 RGB 值随机替换目标视图中的矩形区域。

1. **实验步骤**

1、核心网络代码：

# 导入所需的库

import argparse # 用于解析命令行参数

import cv2 # OpenCV库，用于图像处理

import math # 数学库

from models import hsm # 导入自定义的模型

import numpy as np # NumPy库，用于数值计算

import os # 用于操作系统相关的操作，如创建目录、读写文件等

import pdb # Python的调试库

import skimage.io # 用于图像的输入输出

import torch # PyTorch库，用于深度学习

import torch.nn as nn # PyTorch神经网络库

import torch.backends.cudnn as cudnn # 用于设置和运行CUDA操作的库

from torch.autograd import Variable # 用于自动求导

import time # 用于计算时间

from models.submodule import \* # 导入模型的子模块

from utils.eval import mkdir\_p, save\_pfm # 导入工具函数，用于创建目录和保存PFM文件

from utils.preprocess import get\_transform # 导入预处理函数，用于获取图像变换

# 设置CUDA的参数

cudnn.benchmark = False # 如果网络的输入数据维度或类型上变化不大，设置True可以增加运行效率

# 设置命令行参数

parser = argparse.ArgumentParser(description='HSM') # 创建解析器

parser.add\_argument('--datapath', default='./data-mbtest/', help='test data path') # 设置数据路径参数

parser.add\_argument('--loadmodel', default=None, help='model path') # 设置模型路径参数

parser.add\_argument('--outdir', default='output', help='output dir') # 设置输出目录参数

parser.add\_argument('--clean', type=float, default=-1, help='clean up output using entropy estimation') # 设置清理输出的参数

parser.add\_argument('--testres', type=float, default=0.5, help='test time resolution ratio 0-x') # 设置测试时的分辨率比例参数

parser.add\_argument('--max\_disp', type=float, default=-1, help='maximum disparity to search for') # 设置搜索的最大视差参数

parser.add\_argument('--level', type=int, default=1, help='output level of output, default is level 1 (stage 3), can also use level 2 (stage 2) or level 3 (stage 1)') # 设置输出级别参数

args = parser.parse\_args() # 解析命令行参数

# 数据加载

from dataloader import listfiles as DA # 导入数据加载器

test\_left\_img, test\_right\_img, \_, \_ = DA.dataloader(args.datapath) # 加载数据

# 构建模型

model = hsm(128,args.clean,level=args.level) # 创建模型实例

model = nn.DataParallel(model, device\_ids=[0]) # 使用DataParallel包装模型，使其可以在多个GPU上运行

model.cuda() # 将模型移动到GPU上

# 加载预训练模型

if args.loadmodel is not None: # 如果提供了预训练模型的路径

pretrained\_dict = torch.load(args.loadmodel) # 加载预训练模型

pretrained\_dict['state\_dict'] = {k:v for k,v in pretrained\_dict['state\_dict'].items() if 'disp' not in k} # 过滤掉预训练模型中与视差相关的参数

model.load\_state\_dict(pretrained\_dict['state\_dict'],strict=False) # 将过滤后的预训练模型参数加载到当前模型中

else:

print('run with random init') # 如果没有提供预训练模型的路径，将使用随机初始化的模型

print('Number of model parameters: {}'.format(sum([p.data.nelement() for p in model.parameters()]))) # 打印模型的参数数量

# 进行一次空运行，以便于热启动CUDA

multip = 48 # 设置倍数

imgL = np.zeros((1,3,24\*multip,32\*multip)) # 创建左图像的占位符

imgR = np.zeros((1,3,24\*multip,32\*multip)) # 创建右图像的占位符

imgL = Variable(torch.FloatTensor(imgL).cuda()) # 将左图像的占位符转换为CUDA张量

imgR = Variable(torch.FloatTensor(imgR).cuda()) # 将右图像的占位符转换为CUDA张量

with torch.no\_grad(): # 在不需要计算梯度的情况下运行模型

model.eval() # 将模型设置为评估模式

pred\_disp,entropy = model(imgL,imgR) # 运行模型，得到预测的视差和熵

# 定义主函数

def main():

processed = get\_transform() # 获取图像变换

model.eval() # 将模型设置为评估模式

for inx in range(len(test\_left\_img)): # 遍历所有的测试图像

print(test\_left\_img[inx]) # 打印当前处理的左图像的路径

imgL\_o = (skimage.io.imread(test\_left\_img[inx]).astype('float32'))[:,:,:3] # 读取左图像

imgR\_o = (skimage.io.imread(test\_right\_img[inx]).astype('float32'))[:,:,:3] # 读取右图像

imgsize = imgL\_o.shape[:2] # 获取图像的尺寸

if args.max\_disp>0: # 如果设置了最大视差

if args.max\_disp % 16 != 0: # 如果最大视差不是16的倍数

args.max\_disp = 16 \* math.floor(args.max\_disp/16) # 将最大视差设置为小于等于它的最大的16的倍数

max\_disp = int(args.max\_disp) # 将最大视差转换为整数

else: # 如果没有设置最大视差

with open(test\_left\_img[inx].replace('im0.png','calib.txt')) as f: # 打开对应的标定文件

lines = f.readlines() # 读取标定文件的所有行

max\_disp = int(int(lines[6].split('=')[-1])) # 从标定文件中获取最大视差

## change max disp

tmpdisp = int(max\_disp\*args.testres//64\*64) # 计算临时的最

2、立体匹配代码：

# 构建模型

model = hsm(128,args.clean,level=args.level) # 创建一个hsm模型对象

model = nn.DataParallel(model, device\_ids=[0]) # 使用DataParallel包装模型，使模型可以在多个GPU上并行运算

model.cuda() # 将模型移动到GPU上

# 加载预训练模型

if args.loadmodel is not None: # 如果指定了预训练模型的路径

pretrained\_dict = torch.load(args.loadmodel) # 加载预训练模型

pretrained\_dict['state\_dict'] = {k:v for k,v in pretrained\_dict['state\_dict'].items() if 'disp' not in k} # 删除预训练模型中所有与视差相关的参数

model.load\_state\_dict(pretrained\_dict['state\_dict'],strict=False) # 将预训练模型的参数加载到当前模型中

else:

print('run with random init') # 如果没有指定预训练模型的路径，那么就使用随机初始化的参数运行模型

print('Number of model parameters: {}'.format(sum([p.data.nelement() for p in model.parameters()]))) # 打印模型的参数数量

# 进行一次空运行，以便于PyTorch能够自动选择最优的cudnn算法

multip = 48

imgL = np.zeros((1,3,24\*multip,32\*multip)) # 创建一个全零的左图像

imgR = np.zeros((1,3,24\*multip,32\*multip)) # 创建一个全零的右图像

imgL = Variable(torch.FloatTensor(imgL).cuda()) # 将左图像转换为Variable，并移动到GPU上

imgR = Variable(torch.FloatTensor(imgR).cuda()) # 将右图像转换为Variable，并移动到GPU上

with torch.no\_grad(): # 关闭自动求导

model.eval() # 将模型切换到评估模式

pred\_disp,entropy = model(imgL,imgR) # 对全零图像进行一次前向传播

# 定义主函数

def main():

processed = get\_transform() # 获取预处理函数

model.eval() # 将模型切换到评估模式

for inx in range(len(test\_left\_img)): # 遍历所有的测试图像

print(test\_left\_img[inx]) # 打印当前处理的左图像的路径

imgL\_o = (skimage.io.imread(test\_left\_img[inx]).astype('float32'))[:,:,:3] # 读取左图像，并将其转换为float32类型

imgR\_o = (skimage.io.imread(test\_right\_img[inx]).astype('float32'))[:,:,:3] # 读取右图像，并将其转换为float32类型

imgsize = imgL\_o.shape[:2] # 获取图像的尺寸

if args.max\_disp>0: # 如果指定了最大视差

if args.max\_disp % 16 != 0:

args.max\_disp = 16 \* math.floor(args.max\_disp/16) # 将最大视差调整为16的倍数

max\_disp = int(args.max\_disp) # 将最大视差转换为整数

else: # 如果没有指定最大视差

with open(test\_left\_img[inx].replace('im0.png','calib.txt')) as f: # 打开对应的标定文件

lines = f.readlines() # 读取标定文件的所有行

max\_disp = int(int(lines[6].split('=')[-1])) # 从标定文件中获取最大视差

# 调整最大视差

tmpdisp = int(max\_disp\*args.testres//64\*64) # 计算临时的最大视差

if (max\_disp\*args.testres/64\*64) > tmpdisp: # 如果临时的最大视差小于真实的最大视差

model.module.maxdisp = tmpdisp + 64 # 那么将模型的最大视差设置为临时的最大视差加64

else:

model.module.maxdisp = tmpdisp # 否则，将模型的最大视差设置为临时的最大视差

if model.module.maxdisp ==64: model.module.maxdisp=128 # 如果模型的最大视差为64，那么将其设置为128

model.module.disp\_reg8 = disparityregression(model.module.maxdisp,16).cuda() # 创建一个视差回归对象，用于计算8倍下采样的视差图

model.module.disp\_reg16 = disparityregression(model.module.maxdisp,16).cuda() # 创建一个视差回归对象，用于计算16倍下采样的视差图。

3、结果展示：

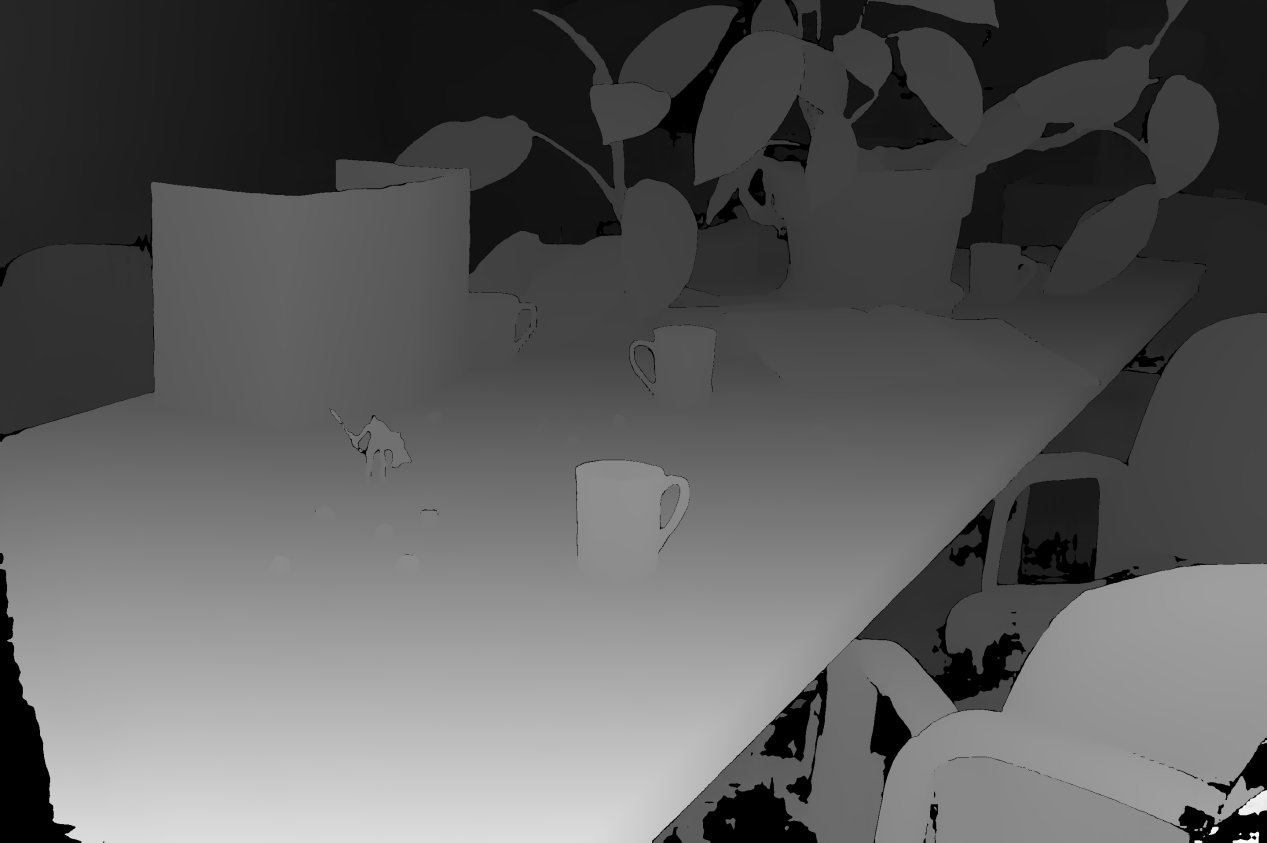
Input one



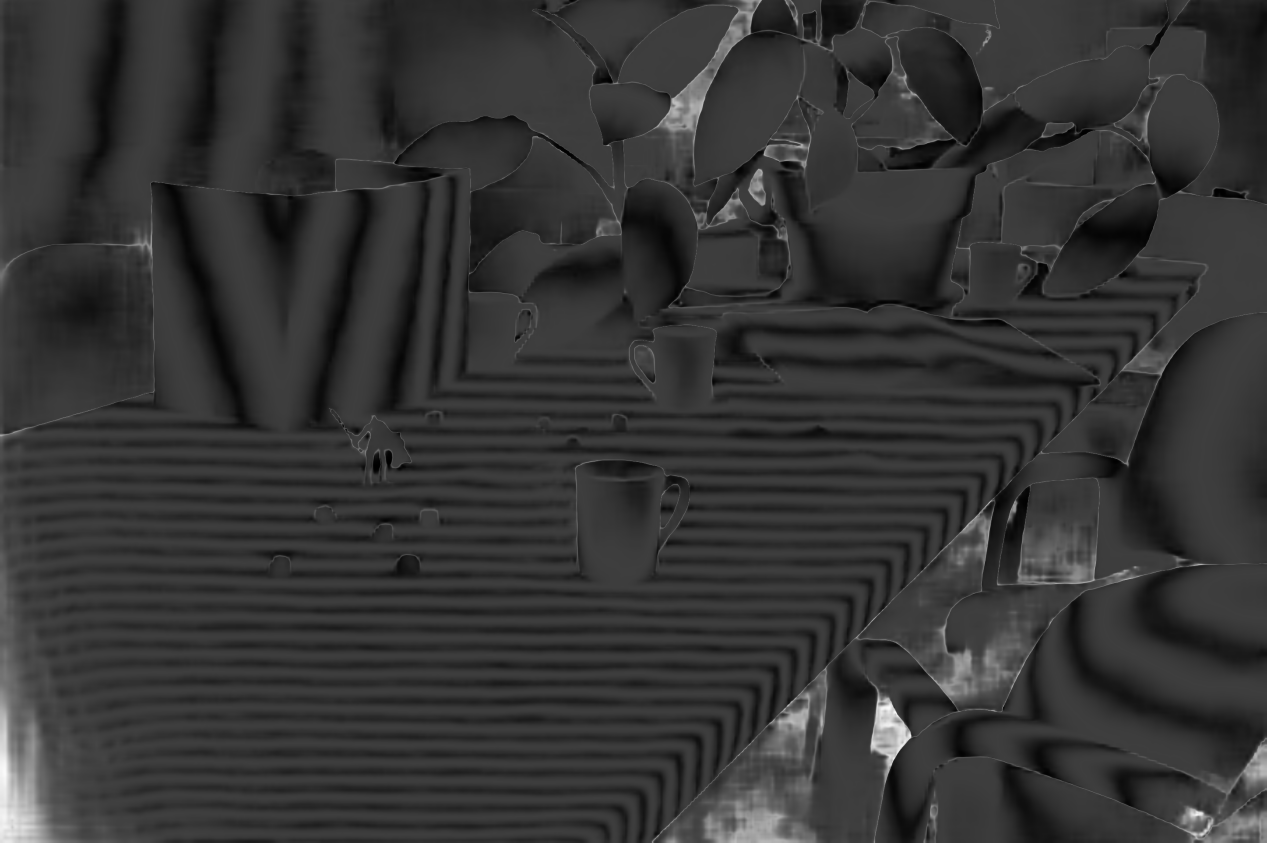
Input two



视差图



视差图可以帮助理解物体在空间中的位置

熵图

熵图可以帮理解视差图的可靠性。

4、结果分析：

在分析视差图时，通常会关注以下几个方面：

连续性：在物体表面上，期望视差值是连续变化的。如果视差图中存在许多尖锐的视差变化，那么可能意味着匹配错误。

边界保持：在物体的边界处，视差值应该有明显的变化。如果视差图中的物体边界模糊不清，那么可能意味着边界保持能力较弱。

纹理保持：在具有丰富纹理的区域，视差图应该能够保持原始图像的纹理信息。如果视差图中的纹理信息丢失，那么可能意味着纹理保持能力较弱。

深度感：视差图应该能够反映出场景的深度信息。在视差图中，视差值较大的区域对应于离相机较近的物体，视差值较小的区域对应于离相机较远的物体。

对于得到的这张视差图，它以灰度色调呈现了一个场景的不同深度和距离。在视差图中，颜色的深浅通常用来表示深度，颜色越深表示物体离相机越近，颜色越浅表示物体离相机越远。

从这张视差图中，可以看到以下几点：

杯子：图中有两个杯子，一个在前景，一个在图像的中心附近。这两个杯子的颜色较深，说明它们相对于其他物体更接近相机。

纸张：在桌子的左侧，有一张平放的纸张或者其他平面物体。它的颜色比杯子浅，说明它离相机的距离比杯子远。

植物：从右侧伸入画面的是一株植物，它在场景中形成了一种悬挂的效果。植物的颜色比杯子和纸张都要浅，说明它离相机的距离最远。

背景：背景的颜色非常浅，几乎是白色，这说明背景离相机的距离最远。

阴影和高光：通过有效地使用阴影和高光，这张视差图创建了一种深度和维度感，使得场景看起来更加立体。

总的来说，这张视差图展示了一个丰富的深度场景，各个物体的位置和深度都被清晰地展现出来。这对于许多计算机视觉任务，如物体检测、场景理解和导航等，都是非常重要的。