浙江大学

计算摄影学课程项目报告

项目名称：	单目深度估计
姓　　名：	李鹏飞 黄奇浩 李欣怡
学　　号：	3140103479 3140101616 3140103699
电子邮箱：	3140103479，3140101616, 3140103699@zju.edu.cn
联系电话：	15558120169
导　　师：	章国锋

2018 年 06 月 26 日

单目深度估计

1. 项目介绍

1.1 项目背景

深度是三维场景重建中很重要的一个信息。预测深度是理解一个场景的三维图形的一个关键的组成部分。如果有多张图像或两张图像，或者我们有相机数据，那么我们可以采用传统的三维重建算法进行计算。但是，如果只有单张图像，并且缺乏相应的相机参数，那么我们需要从各种各样的线索中找到全局信息和局部信息，并将它们结合在一起。因为这个过程含有大量的不确定性，我们只能给出确定的任务信息。这个任务的特征与深度学习方法的特征是一致的。从而，我们考虑采用神经网络进行单目深度估计。

通过查阅资料，我们觉得，这几年采用深度学习解决深度估计问题的思路大致可以分为以下部分：

a) 利用 CNN 模型来获得结果

Eigen[1][2]于 2014 和 2015 年提出，用 Multi-scale 的网络对深度进行估计。在第一篇文章中，Eigen 提出用两个尺度的网络对图像的深度进行估计。即先用全局粗略尺度的深度图整体估计，然后用对局部进行预测，再用改进后的局部深度图来矫正前者的输出。

在第二篇文章中，Eigen 对他们的工作进行了优化。即将两个尺度的网络改进为了三个尺度的网络。这样的实践是利用裁剪后不同大小的局部图进行的。我们实践了这篇文章的工作，更为详细的介绍请参考第二部分。

b) 基于 FCN 模型获得结果

在 CNN 模型中，实践的关键性一步是对模型进行预训练操作。而 J Long[3]提出了新的解决方案。不同于采用固定的全连接层大小来得到图片之间的转换，这种方法去除了全连接层，而是用于预训练网络结构一样的网络来将特征转变到与原图一样的大小。即将整个网络变成了译码-解码过程。通过这样的操作，我们可以在全连接层中获得整体网络绝大部分参数，大大提高了 GPU 的利用效率。同时，全连接层要求输入的图片尺寸一致，即要对数据进行剪裁或者缩放等处理。而这样的处理会丢掉一定的信息。而 FCN 可以直接处理几乎所有尺寸的图片。基于这个思路，I Laina[4]提升了网络层次。下图是他们的思路。

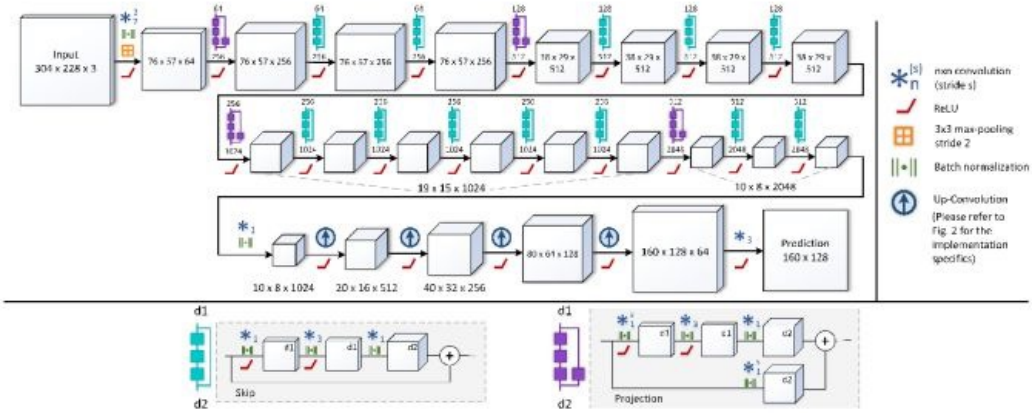


图 1 FCNR 方法的流程

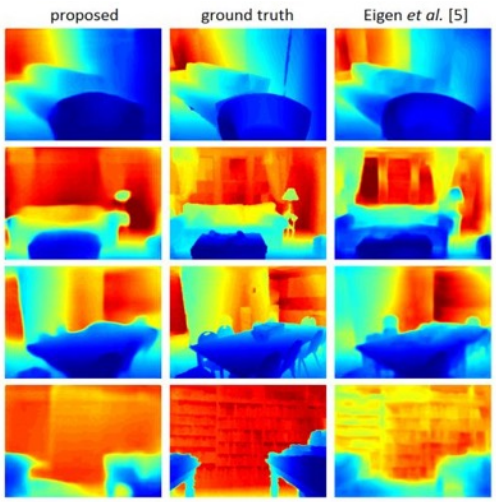


图 2 FCNR 方法与 CNN 方法结果对比

从图二可以看出，FCNR 在整体的深度预测上会有更好的表现，而 Eigen 的结果又更加丰富的细节和清晰的伦果。

c) 利用非监督学习获得深度估计

我们知道，如果利用单目相机求深度需要激光雷达，而双目相机求深度的成本相对低很多。R Gard[5]利用双目相机（或两个相机）在同一水平线上左右相距一定位置的两张图片来训练不知道真实值的图片来获得深度信息，即采用非监督学习的方式来进行深度估计。他采用的是类似 FCN 的结构，没有全连接层的影响，从而速度较快，模型较小。同时，利用 Skip-connect 保证了输出特征的相对完整性，并使用预训练网络结果作为 encoder 部分，从而使得在数据量不足的情况下也能得到不错的结果。

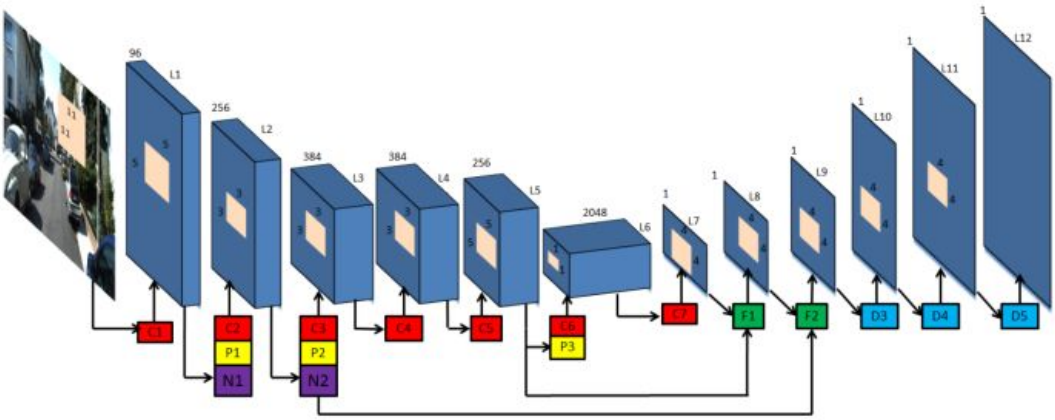


图 3 Unsupervised CNN for Single View Depth Estimation: Geometry to the Rescue 结构

在利用单目相片采用双目估计的过程中，有两种计算视差的方法。要解释这两种方法的不同，我们首先要回顾双目计算的原理。

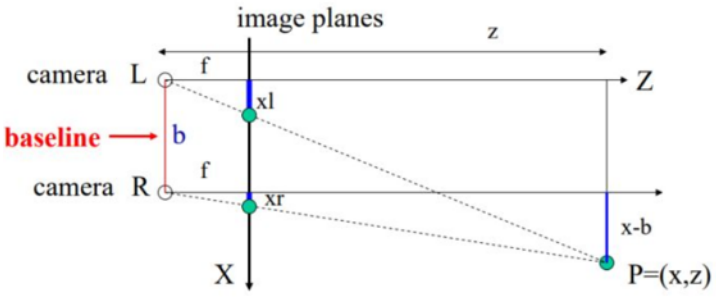


图 4 双目深度计算

$$x - b = \frac{z}{f} * x_r; x = \frac{z}{f} * x_l \Rightarrow (x_l - x_r) = \frac{f * b}{z}$$

$$d = x_l - x_r \Rightarrow x_l = d + x_r$$

$$I_r(x_r) = I_l(F(x_l) + x_l)$$

如图 4 所示，Z 为场景深度，X 为三维场景映射得到的二维平面，f 为相机焦距，b 为左右相机的位移，x1 和 x2 为同一物体在不同相机中成像的坐标。可以知道，想要计算出相机的深度信息，即要计算出 F(X1)。

从而，单目深度估计可以转换为如下形式：

$$z = \frac{d * f}{b}; d = F(I_l);$$

但是，用网络预测出来的 d 是浮点数，而像素点是整数，即存在一个转换的问题。如果之间进行转变，那么可能一些点得到了多个浮点数转换后的结果，而一些位置上的点没有 d，

因为这些点因为视差原因可能在原图中是不可见的。

从而，有两种解决方法，分别是 forward mapping 和 inverse or backward mapping。

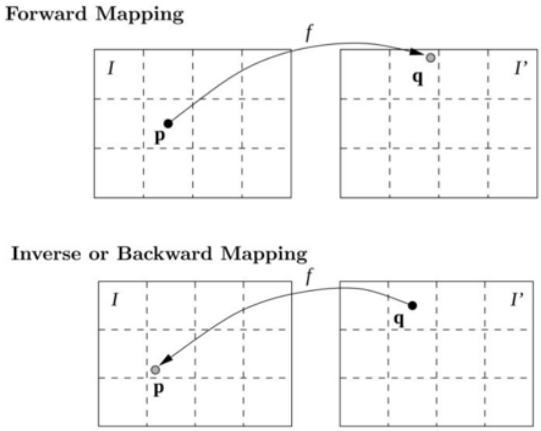


图 5 两种转换方式

这两种方法的区别是,在 forward mapping 方法中，我们会通过找到转换后最近的整数位置来将非整数位置变换过去。而在 Inverse or Backward mapping 方法中是通过对变换后的图找原图中的对应点来进行转换，这样可以保证 I’图中没有空洞，并且可以通过插值的方式得到对应非像素点的位置。

C Godard[6]基于这个思路，提出了一种结合两种转换方式的训练网络，并利用 FCN 方式进行了训练。不同的是，在 decoder 的最外 4 层，Godard 都估计了当前特征大小对应的视差值，并将它上采样后传给到 decoder 的下层，这样保证每一层都在提取视差。

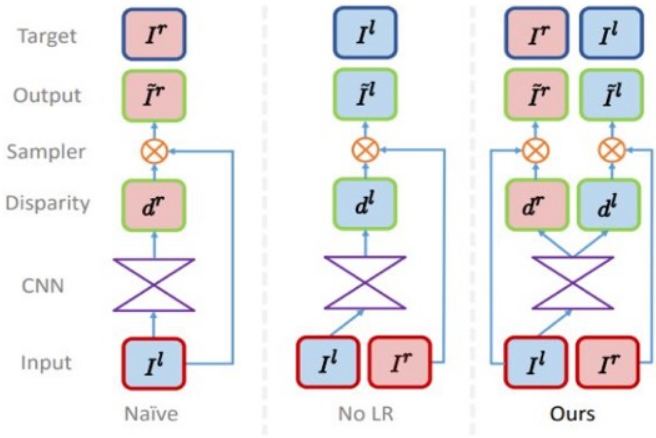


图 6 C Goadard 网络结构

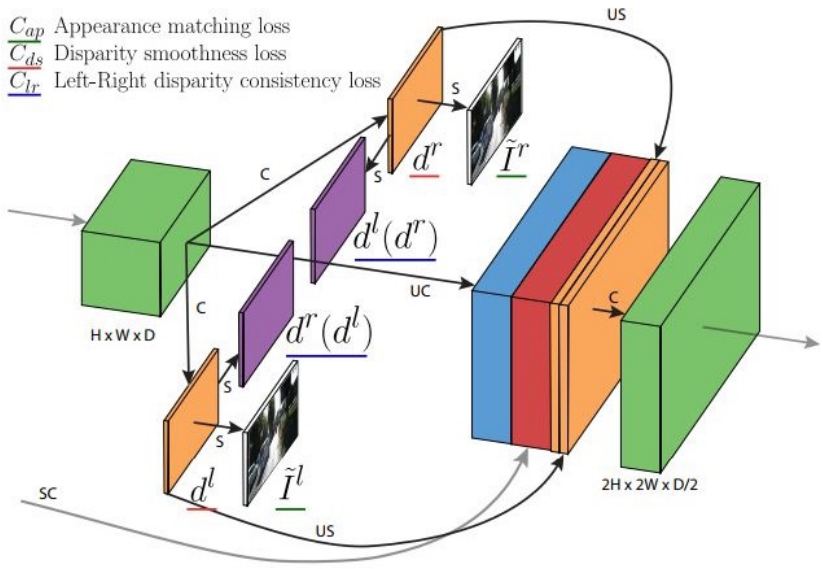


图 7 FCN 训练网络

1.2 项目的功能和要求

我们的工作可以分为两个部分。第一个部分是实践了 D Eigen[2]的方法。

这篇文章用一个多尺度的卷积网络结构解决了三个不同的计算机视觉问题。深度估计、表面法向量估计和语义分割。网络结构从输出图像直接回归输出图像。作者的方法用序列化的尺度优化估计，可以获得很多图像细节，不需要超分辨率或者低层分割。在上述三个问题的 benchmark 上作者都达到了 state-of-the-art。

第二个部分是实践了 Laina, Iro, et al[7]的方法。作者采用了 FCNR 的方法，利用更加深的网络和新的逆卷积结构和新的损失函数来得到了较好的效果。我们将比较并讨论这两种方法。具体实施细节请参考后面部分。

我们均是采用 NYUv2 数据集训练和测试深度估计模型。这个数据集包含 1449 张配有深度图的 RGB 图像，大小为 640*480。因为我们选择一次性读取 16 张图片（batch size）送入网络进行训练，因此我们的训练集划分为 1120，160 和 160,分别对应 train，valid 和 test。

2. 基本原理

2.1 深度估计模型

在学习过程中，我们首先尝试和参考了纽约大学 David Eigen 等人名为《Predicting Depth, Surface Normals and Semantic Labels with a Common Multi-Scale Convolutional Architecture》的论文。该篇论文提出了一种多层次的卷积神经网络，应用于单目图像 Depth，Normal，Labels

三种估计任务，如下图所示。

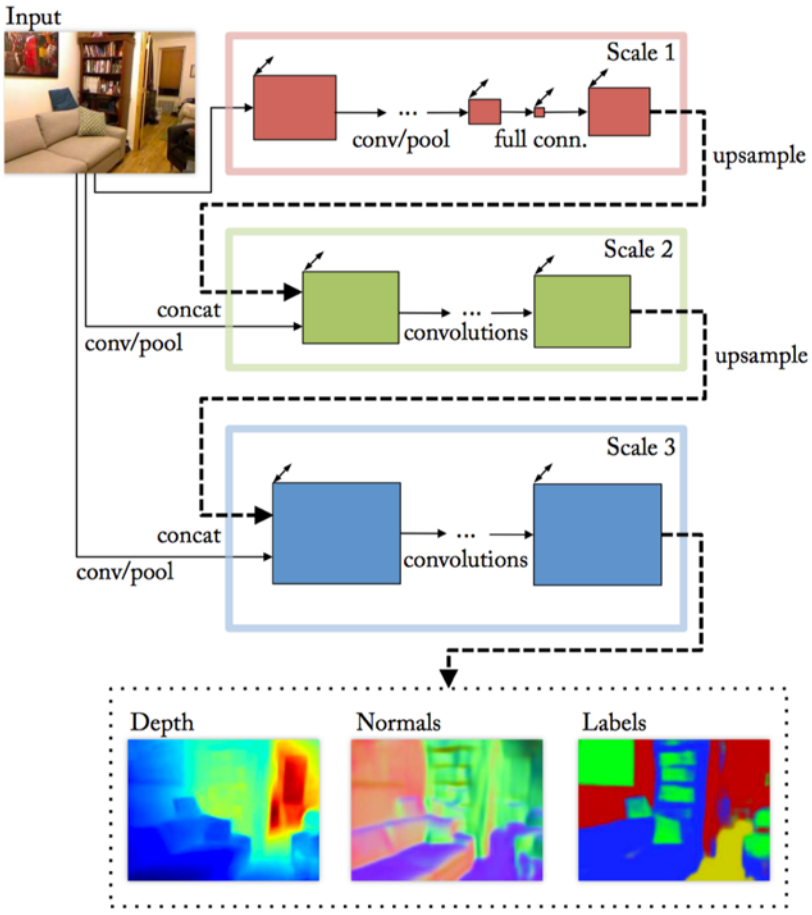


图 8 多层次网络结构

该模型通过三种不同功能的网络，将不同层级的图像特征组合起来。其中，Scale1 和 Scale2 用于预测较为粗粒度的全局图像深度信息，并输入 Scale3 中，用于优化 Scale3 中更为细致的局部图像训练过程中，得到最终的图像深度估计结果。

2.2 涉及的方法

我们采用了较为经典的卷积结构用于捕捉图像特征，采用特定损失函数用于评价网络预测深度图和训练图像所对应的深度图标签，并反向传播更新网络参数。

2.2.1 网络架构

设计和实现过程中，我们将 Scale1 和 Scale2 统一成一个网络 Net12，用于训练和测试。其中 Scale1 的基本结构采用了 VGG Net 的网络结构设计，并去除最后的两个全连接层，改为 out1 = out1.view(self.N, 64, 19, 14)的结构，即 reshape 得到 Nx64x19x14 的输出。为匹配 Scale2 的 Tensor 维度，再将其双线性 unsample 得到 Nx64x76x56 的输出，添入到 Scale2 的网络中。Scale2 经过一次卷积和池化后，合并 Scale1 的上述输入，并经过一系列卷积和池化

后得到 Nx1x74x55 的输出。

Scale3（Net3）的训练过程和 Net12 相似，同样需要将 Net12 的输出 Nx1x74x55 双线性 unsample，适当裁剪到 Nx1x147x109 的维度。图像首先经过 Net3 的卷积和池化后，与上述 Net12 的输出变换拼接，再采取若干次卷积和池化得到最终的预测输出，Nx1x147x109，N 张 147x109 的单通道深度图。

两个网络的具体参数设置可翻阅本篇报告中的附录 A 和 B。

2.2.2 损失函数

由于单目深度估计问题本身是欠约束的，这个欠约束主要体现在尺度的不确定性。例如给定图片中的一个汽车，我们并不能确定它到底是汽车模型还是真的汽车，因此我们也就没法立刻给出它和相机之间的距离。在一些三维重建算法中，我们可以加入物体的先验尺度信息，来消除这种不确定性。但在基于 CNN 的深度估计问题中，这种方法并不合适，比较简单的方法是直接忽略这种不确定性。也就是如果预测出的值与真实值均成一定倍数，那么我们认为预测结果也是正确的，误差为零。这种思想使用数学语言表达就是

$$D(y,y^*)=\frac{1}{2n}\sum_{i=1}^{n}(\log\ y_i-\log\ y_i^*+\alpha(y,y^*))^2$$

其中第三项是用来抵消尺度的影响，也就是把深度的预测值和真值都归一化到对数值平均为零的空间上。因此，$\alpha(y,y^*)=\frac{1}{n}\sum_i(\log\ y_i^*-\log\ y_i)$

将上式整理可得

$$\begin{aligned}D(y,y^*) &= \frac{1}{2n^2}\sum_{i=1,j=1}^{n}((\log\ y_i-\log\ y_j)-(\log\ y_i^*-\log\ y_j^*))^2\\ &=\frac{1}{n}\sum_i d_i^2-\frac{1}{n^2}\sum_{i,j}d_id_j=\frac{1}{n}\sum_i d_i^2-\frac{1}{n^2}(\sum_i d_i)^2\end{aligned}$$

在实际操作的过程中，除了物体边缘外，深度值通常不会发生突变。所以我们需要对输出的深度预测值加入平滑约束项，也就是计算预测图的梯度值总和。

2.2.3 数据处理

由于三个网络的输入输出和预测拼接流程不同，我们需要对数据集做训练、测试、验证集的数量分割，并对图像和相应深度图做尺度换边、大小裁剪等具体操作，最终得到如下图所示的一批图像和深度图。

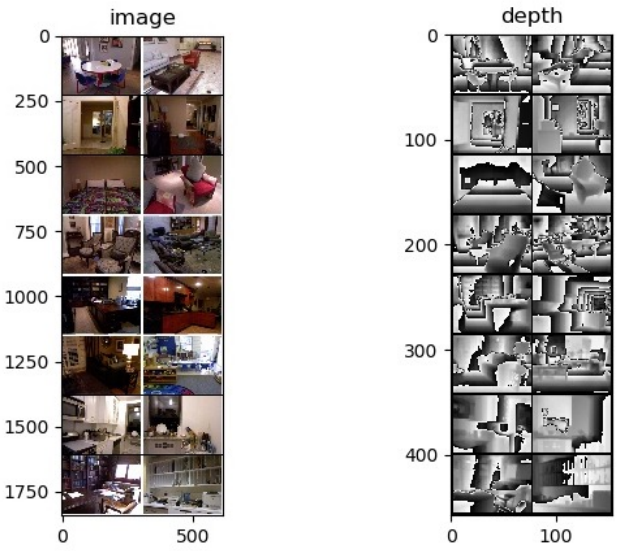


图 9 输入图像信息

2.2.4 训练、测试步骤

训练过程分两步：

第一步：训练 Net12，本地保存网络参数。得到训练和测试 loss 和准确率。

第二步：加载 Net12 和对应的网络参数，设置为仅作预测模式。训练 Net3，在网络卷积运算中加入 Net12 的预测输入。得到训练和测试 loss 和准确率。

整体**测试**步骤：加载 Net12 和 Net3 和对应的网络参数，均设置为预测模式，得到输入图像的深度图显示。

3. 项目框架

3.1 PyTorch 和 CUDA

在本实验中，PyTorch 使用最新的 0.4.0 版本，CUDA 使用 8.0 版本。

PyTorch 是在 2017 年 1 月由 Facebook 推出的。它是经典机器学习库 Torch 框架的端口（Torch 原本使用 lua 语言编程）， Torch 二进制文件包装于 GPU 加速的 Python。除了 GPU 加速和内存使用的高效外，PyTorch 受欢迎的主要因素是动态计算图的使用。已经有其他一些不太知名的深度学习框架使用动态计算图。

动态图的优点在于，计算图（computational graph）是由运行中定义（“define by run”），而不是传统的“define and run”，动态图的缺点是在模型比较大时，可能会运行到一半由于输入数据改变而出现 out of memory 问题。例如 TensorFlow 这种框架，需要先定义计算图才

能计算，不容易出现之前的问题。但 TensorFlow 在网络中间层是不能用 pdb 加设断点的，这一点对于我们这种比较依赖 pdb 进行调试的人就不是特别友好，因此我们使用了 Pytorch 框架而没有使用 TensorFlow。

最新版本的 Pytorch 有以下几个新功能，首先是 Tensors 和 Variables 已经合并，同时弃用了 volatile flag。原来 volatile flag 是用来决定网络是否存储梯度，例如网络在计算准确率时，只要有一个输入具有 volatile flag，则整个网络都不存储梯度。这样会带来一些小问题，比如误操作，因此现在改成了 torch.no_grad()这个函数来控制。还有一个比较关键的更新是添加了 dtypes，devices 和 Numpy 风格的 Tensor 创建函数，这样使得数据在 CPU 和 GPU 上的转换比较方便。

用于实验的显卡型号是 NVIDIA GTX1080Ti，在给定预训练模型的情况下，训练时间大约为两天。

3.2 代码基本结构和模块

```
├── checkpoints  // 目录
│   ├── net12.pth  // 训练后本地保存的 Net12 网络模型参数
│   └── net3.pth  // 训练后本地保存的 Net3 网络模型参数
├── data  // 目录 NYU 数据集
├── data.py
        // 实现数据集的读取和分割 以及 继承 PyTorch DataSet 类，实现我们自己的
        MyDataSet 类，实现数据变换
├── lossF.py
        // 基于 PyTorch 实现损失函数类 MyLoss 和准确率计算函数 count_acc
├── main.py   // Net12 和 Net3 的整体图像预测
├── net.py    // 基于 PyTorch 实现 Net12（Scale12）和 Net3（Scale3）
├── param_utils.py // 训练的超参数设计
├── trainNet12.py  // 训练和测试 Net12
└── trainNet3.py   // 训练和测试 Net3
```

4. 实现细节

4.1 网络结构

Scale12 继承了 PyTorch 中的 nn.Module 模块，我们的设计由两部分组成：__init__定义

网络结构，forward 中组合各个结构。

```
class Scale12(nn.Module):
    def __init__(self, N):
        super(Scale12, self).__init__()
        self.N = N
        self.layer1_1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.layer1_2 = …
        …
        self.layer1_5 = …
        self.layer1_6 = nn.Sequential(
            nn.Linear(512 * 9 * 7, 4096),
            nn.ReLU(),
            nn.Dropout()
        )
        self.layer1_7 = = nn.Sequential(
            nn.Linear(4096, 64 * 19 * 14),
            nn.ReLU(),
            nn.Dropout()
        )
        self.layer2_1 = nn.Sequential(
            nn.Conv2d(3, 96, kernel_size=9, stride=2, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2)
        )
        self.layer2_2 = …
        self.layer2_3 = …
        self.layer2_4 = …
    def forward(self, x):
        out1 = self.layer1_1(x)
        …
        out1 = out1.view(self.N, 64, 19, 14)
         # reshape Nx17024 to Nx64x19x14
         m1 = nn.Upsample(scale_factor=4, mode='bilinear')
         out1 = m1(out1)
        …
         out2 = torch.cat([out2, out1], 1)
        …
         return out2
```

上述代码中的 self.layer1_x 为 Scale1 中的网络结构。self.layer1_1 即我们定义的一个序列结构，通过 nn.Sequential()序列化地组合了卷积（被卷积的图像为 3 通道，64 个卷积核，每个卷积核的大小为 3x3，步长为 1）、ReLu、卷积、ReLu、最大池化（池化区域为 2x2，步长为 2），共 5 步操作。Scale1 采用了 VGG Net 的网络结构（即 layer1_1 到 layer1_5），为输出匹配尺寸的图像输出，我们在 layer1_6 和 layer1_7 将 Tensor 变换为 64*19*14。

其中 self.layer2_x 为 Scale2 中的网络结构。我们采用 unsample 函数和 cat 函数，实现了 Scale1 对 Scale2 的拼接输入操作。

Scale3 的网络结构定义与上述相似，具体参数和定义可以查看附录和具体代码。不同的是，由于 Net3 训练过程中，Net12 已经完成训练步骤，所以需要先实例化 Net12，并加载对应参数，作为参数输入到 Scale3 的__init__中。并在 forward 中做 unsample 操作，以匹配 Tensor 维度。

```
class Scale3(nn.Module):
    def __init__(self, net12):

    def forward(self, x):
        …
        out2 = self.net12.forward(x)
        m2 = nn.Upsample(scale_factor=2, mode='bilinear')
        out2 = m2(out2)
        …
        out3 = torch.cat([out3, out2], 1)
        …
        return out3
```

4.2 损失函数

为能实现自动求导功能，损失函数继承了 torch.nn.Module。在编写这个函数的过程中，主要遇到了以下几个坑：

a) 由于需要实现自动求导，所以最终的结果和中间的结果必须是 Torch 的张量类型，而不能是 int 这种不具有梯度功能的变量。例如求图像梯度的时候就需要使用 pad 函数，而不能直接取元素依次相减

b) 我们的目标是计算预测值与真实值的差距，但在计算过程中我们发现预测值为 0 或者预测值和真实值都为零的情况，由于 log 的运算存在，会使整个 loss 都变成 NaN。这很明显是不行的，对于零，我们的处理方式是加上一个深度的最小分辨率，这样

就避开了这个奇异点。还有预测值可能会出现负值，这肯定也是不行的，我们使用 relu 函数，直接滤掉负值部分。

```
    class MyLoss(torch.nn.Module):
        def __init__(self):
            super(MyLoss, self).__init__()

        def forward(self, estimated_depth, ground_truth):
            if estimated_depth.size() == ground_truth.size():
                x = torch.log(F.relu(estimated_depth) + depth_eps) -
torch.log(F.relu(ground_truth) + depth_eps)
                h_x = x.size()[2]
                w_x = x.size()[3]
                img_r = F.pad(x, (0, 1, 0, 0))[:, :, :, 1:]
                img_l = F.pad(x, (1, 0, 0, 0))[:, :, :, :w_x]
                img_t = F.pad(x, (0, 0, 1, 0))[:, :, :h_x, :]
                img_b = F.pad(x, (0, 0, 0, 1))[:, :, 1:, :]
                xgrad = torch.sum(torch.pow(torch.pow((img_r - img_l) * 0.5, 2) +
torch.pow((img_t - img_b) * 0.5, 2), 0.5))
                x_shape = self._tensor_size(x)
                xavag = 0.1*xgrad / x_shape
                loss_result=torch.sum(x * x) / self._tensor_shape(x) -
torch.pow(torch.sum(x) / self._tensor_shape(x), 2) / (2*pow(x_shape, 2)) + xavag


            ……
```

4.3 数据处理

数据处理主要分为四个步骤，分别是数据集划分，比例修改，随机裁剪和转换到 tensor 格式。

我们使用的 NYU 数据集含有 1449 张 RGB 和深度图像的 mat 文件。我们将数据集随机划分为 80%的训练数据，10%的测试数据和 10%的训练过程中的测试数据，即 val_data。考虑到我们后期设计成一次性读入 16 张图片，我们将每个集都划分成了 16 的倍数，即 1120，160 和 160。当我们给 dataset 不同的类型指令时，可以分别对不同的数据集进行处理。

参考图，我们的网络需要先将原始大小为 480*640 的 RGB 图像和深度图像转换为 228*304 的两种图像。接着，image 图像不再需要处理，然而 depth 图像需要转变成不同的大小，输入网络 2,3 进行计算。考虑到 depth 图像实际上是作为 image 的 label 进行计算的。如果再次进行比例变换，那么 depth 将于 image 不对应，即失去了他的意义。因此，我们需要将 depth 图像按照我们的要求进行裁剪，即裁剪到 220*296。这两个变换过程用 rescale 和

randomcrop 函数完成。

在上述处理过程中，数据都是以 ndarrays 进行处理。因此，我们将 numpy 数据转换为 tensor 数据。值得注意的是，numpy 中 image 存储是 H,W,C，而 torch 中的 image 存储是 C,H,W，所以需要进行转置操作。

在完成了要求的图像操作和归一化之后，对数据集采用 dataloader 函数读入，即可进行后续操作。

4.4 训练、测试

训练 Net12 时，我们采取 kaiming-normal 方法，初始化网络结构中权重，init.kaiming_normal(m.weight)，并用 init.uniform(m.bias)初始化 bias

```
    model = Scale12(BATCH_SIZE)
    init_weight(model.state_dict())
```

采用我们设计的 MyLoss（）函数计算损失，采用动量、随机梯度下降法优化。

```
    criterion = MyLoss()
    optimizer_ft = optim.SGD(model.parameters(), lr=LR, momentum=MOMENTUM)
    exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=STEP_SIZE,
gamma=GAMMA)

    model_final = train_model(model=model,
                              criterion=criterion,
                              optimizer=optimizer_ft,
                              scheduler=exp_lr_scheduler,
                              num_epochs=EPOCHS,
                              delta=DELTA)

    torch.save(model_final.state_dict(), path + "/checkpoints/net12.pth")
```

完成计算后，将模型参数保存在本地。

训练网络 Net3 时，大体上与训练 Net12 相似，不同的是，需要先用本地模型参数初始化 Net12，并作为 Net3 的输入。

```
    net12 = Scale12(BATCH_SIZE)
    net12_model_wts = path + "/checkpoints/net12.pth"
    net12.load_state_dict(torch.load(net12_model_wts))
    net12.eval()
    model = Scale3(net12)
    init_weight(model.state_dict())
```

为整体预测，仅需执行 main.py，其中 Net12 和 Net3 分别加载网络参数，设置为测试模式，对输入图像做深度估计，可视化地得到结果。

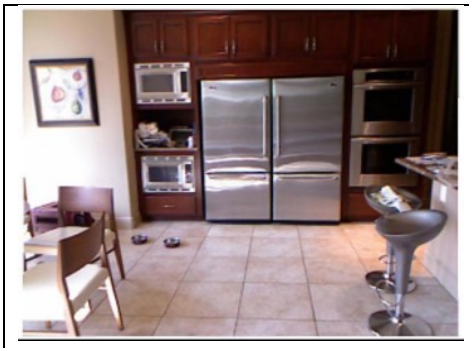
5. 实验结果与分析

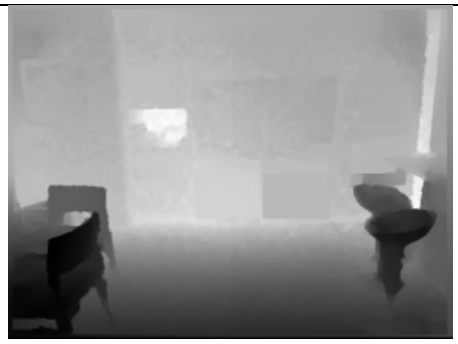
由于我们的模型缺乏良好的参数初始化方法，在一系列训练后，很难达到预测的效果。于是，我们实践了 Laina, Iro, et al[7]的方法，前面几部分已经详细地描述了其思路和方法。具体地，我们根据已有的数据（1449 张 RGB 图、深度图、标签图），重新训练了论文中的 FCRN 网络。运行时，进入 depth_fully_conv 文件路径下，执行 demo.py，该程序将加载由上述训练得到的本地网络参数文件，初始化网络，随机选取地测试集中的一张图片（RGB）作为输入数据，前馈运算，得到预测结果。

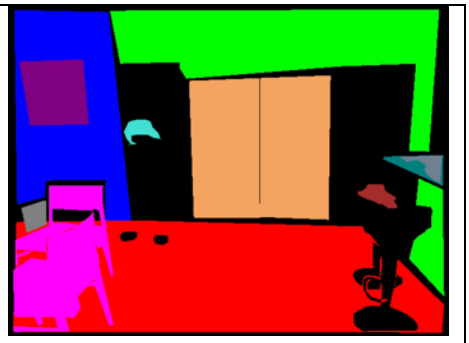
5.1 实验结果

程序运行后，将在本地路径保存 RGB 图像输入，原图所对应的深度图（depth），标签图（label），以及经过网络前馈计算输出的预测深度图（depth），预测标签图（label），如下方几张图所示。

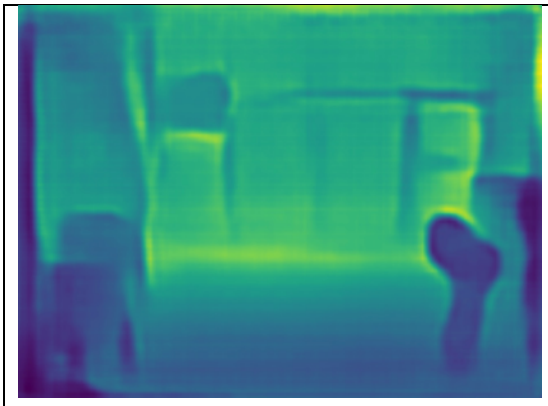
输入图像：RGB 原图（下方左一）

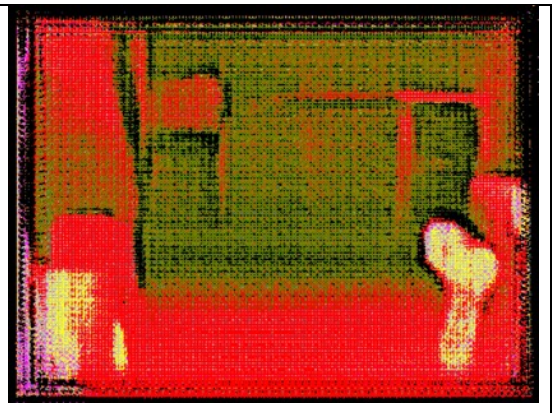






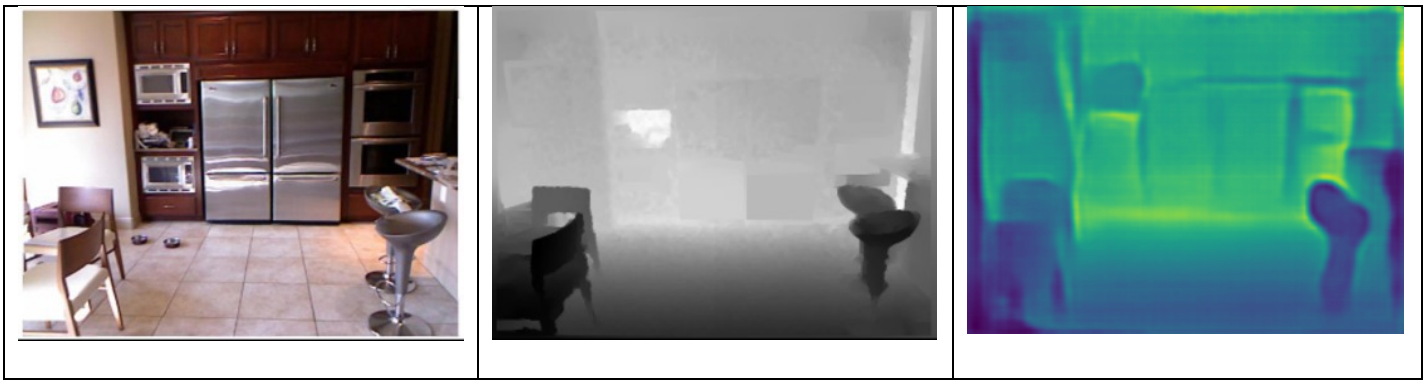
输出：深度图、标签图





5.2 比较、分析

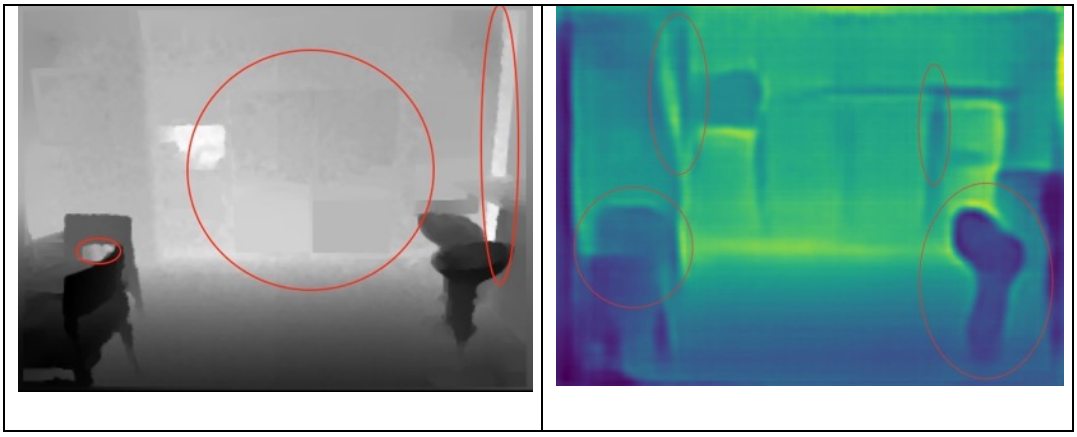
5.2.1 与输入图像对比



从左到右分别是原图，原图对应的深度图，网络预测得到的深度图。

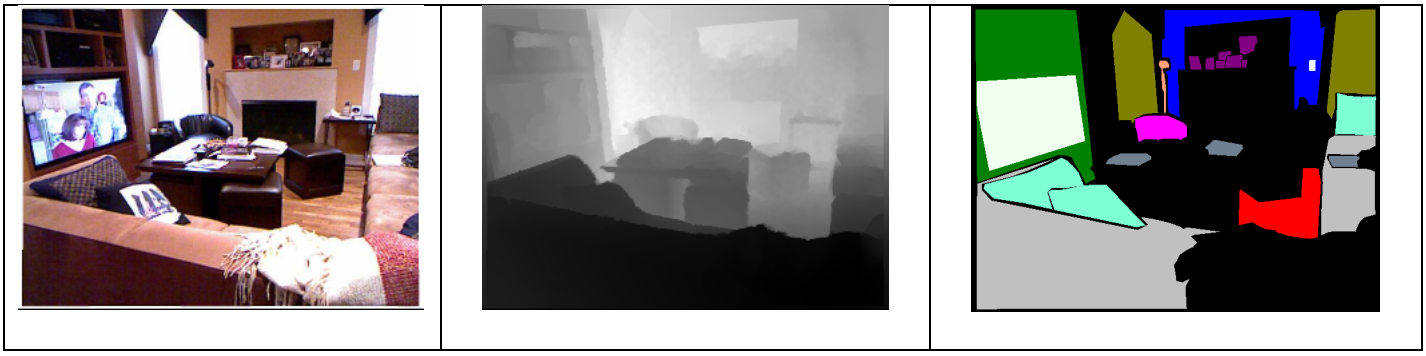
可以看到：

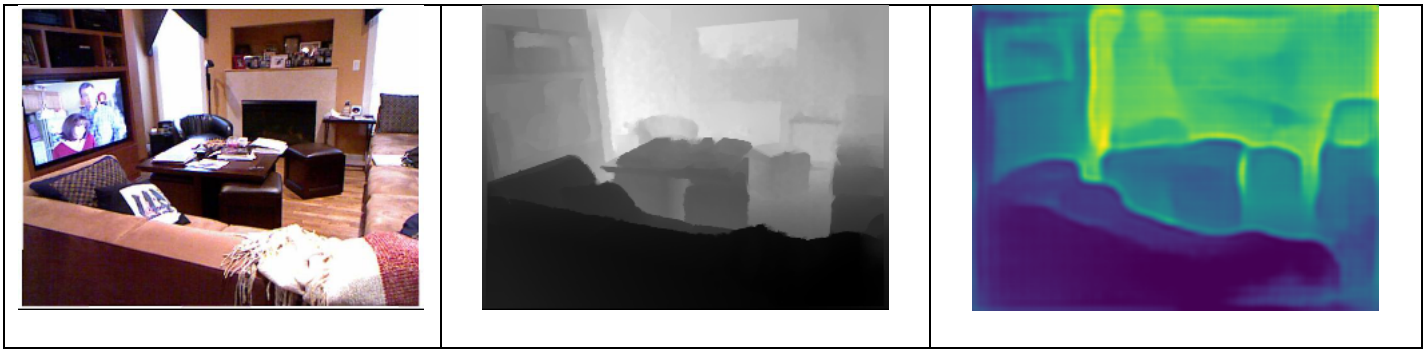
- 原始深度图很好地捕捉了图像细节信息，例如左侧的椅子中的空隙的深度信息。对比预测图像，整体细节则较为模糊，左侧的椅子的空隙的深度信息就消失了。此外，如右侧的座椅，将前后两个座椅在深度上同统合为一个深度。并且，预测图像没有很好地估计右侧的立柱的深度信息，视觉上和壁炉融合了。



- 预测深度图较好地呈现了图像前后景的深度信息和物体轮廓，例如深处的冰箱和几个烤炉的深度信息较为一致，左侧的墙壁深度和壁炉的深度也较为明显地呈现。壁炉和冰箱间的边缘深度信息也捕捉到了。

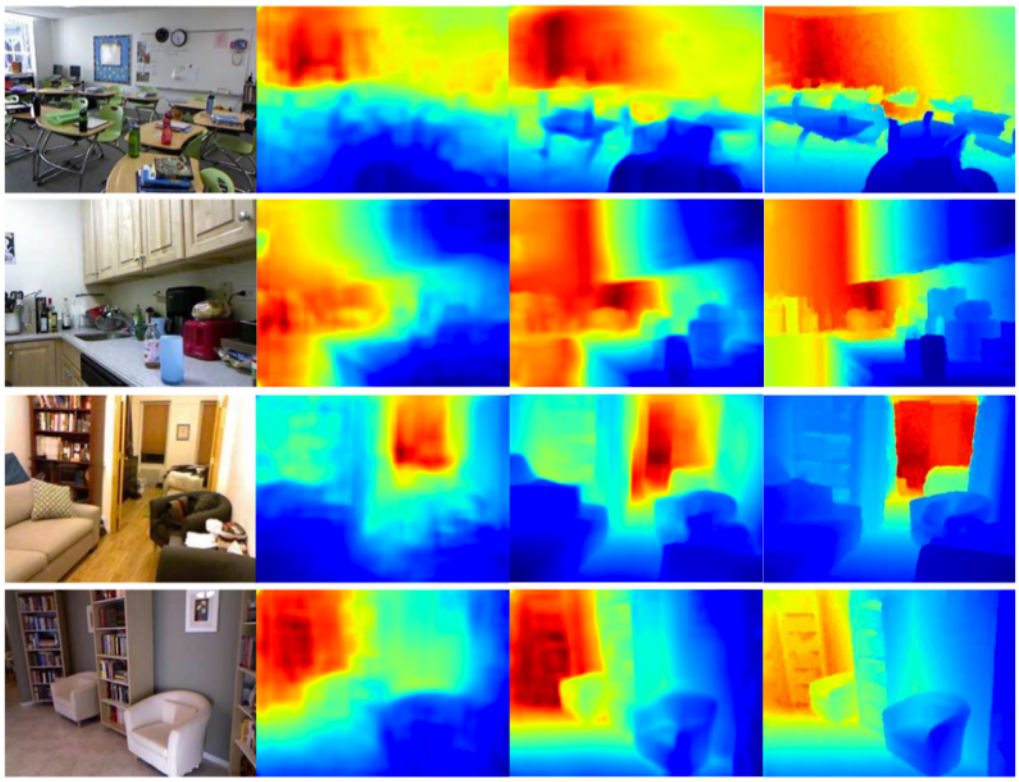
比较另一组图片：





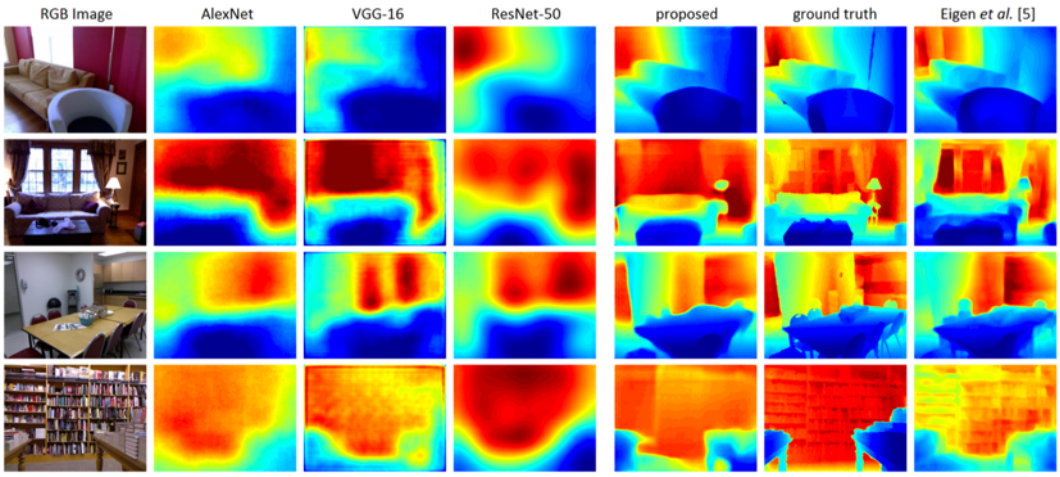
可以看到，这组图片的也印证了我们前面的差别和分析。

5.2.2 与论文结果对比



(a) (b) (c) (d)

上图为 David Eigen 等人在《Predicting Depth, Surface Normals and Semantic Labels with a Common Multi-Scale Convolutional Architecture》中得到的实验结果，（b）为他们前一篇论文《Depth Map Prediction from a Single Image using a Multi-Scale Deep Network》的结果，（c）为他们的预测结果，（d）为原始图像所对应的深度图。



上图为 Iro Laina 等人在《Deeper Depth Prediction with Fully Convolutional Residual Networks》中得到的实验结果，proposed 为 Fully-connected ResNet（FCRN）的实验效果，groud truth 为原始 RGB 对应的深度图，Eigen at al.为上一段 David Eigen 用 Multi-Scale CNN 得到的实验结果。

可以看到，FCRN 显著得提升了深度预测的结果。具体来说，我们认为 FCRN 较好的结果的原因如下：

- FCRN 去掉了全连接，适应于不同尺寸的图像，可以保持图片的尺度信息，不需要例如多层级网络中多次 crop 和 resize 的过程，损失了很多细节信息。
- VGG 全连接在生成图时，需要 reshape，得到图像张量，但无法保持图像等空间结构，FCN 可以保持这类问题的信息，尤其是图像的深度估计问题。
- 基础结构中，ResNet 残差块的特性，例如 ResNet50 比起 VGG19，我们可以加深网络，而不易产生梯度爆炸和弥散等结果。

6. 结论与体会

小组三人经过前后一个月的学习，初步地了解了计算摄影学中一个重要的问题——单目深度估计。我们首先从头复现了 Eigen 在 2015 年的 Multi-Scale 网络的工作，其次训练和实

践了 Laina 的 FCRN 估计深度的工作，得到了上述实验结果。

小组成员在此次项目中发现，数据驱动的深度估计方法的发展，与 2012 年后深度学习领域的发展有很大关系。从我们阅读和参考的论文中可以看到，一系列学者首先猜想并实验简单的 CNN 估计图像深度信息的效果，之后又尝试了 corase-to-fine 的多层级方式。深度神经网路的创新，如 VGG，ResNet，FCN 又从不同角度给予深度估计任务以新的启发，也大幅度地提升了估计效果。学者们开始关注视频序列、多角度深度估计，无监督深度估计等等领域和任务。学术上的创新，都需要我们，在掌握了基础技能后，深耕于一个领域，不断思考、猜想、证明与实践

此外，小组成员在阅读一系列论文，尝试实践中，也发现了一些问题，例如，训练网络时需要较好的初始化方法，参数调优，也包括对损失函数的设计，norm 的选择，这都与具体问题息息相关。快速复现的工程经验能够更好地迭代验证对猜想的解决方案。

小组分工

李鹏飞：撰写报告中的基本原理和实现细节，负责损失函数设计

黄奇浩：撰写报告中基本原理和实现细节，负责网络架构设计

李欣怡：撰写报告中的项目介绍和实现细节，负责数据处理

参考文献

[1] Eigen D, Fergus R. Predicting depth, surface normals and semantic labels with a common multi-scale convolutional architecture[C]//Proceedings of the IEEE International Conference on Computer Vision. 2015: 2650-2658.

[2] Eigen D, Puhrsch C, Fergus R. Depth map prediction from a single image using a multi-scale deep network[C]//Advances in neural information processing systems. 2014: 2366-2374.

[3] Long J, Shelhamer E, Darrell T. Fully convolutional networks for semantic segmentation[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2015: 3431-3440.

[4] Laina I, Rupprecht C, Belagiannis V, et al. Deeper depth prediction with fully convolutional residual networks[C]//3D Vision (3DV), 2016 Fourth International Conference on. IEEE, 2016: 239-248.

[5] Garg R, BG V K, Carneiro G, et al. Unsupervised cnn for single view depth estimation: Geometry to the rescue[C]//European Conference on Computer Vision. Springer, Cham, 2016: 740-756.

[6] Godard C, Mac Aodha O, Brostow G J. Unsupervised monocular depth estimation with left-right consistency[C]//CVPR. 2017, 2(6): 7.

[7] Laina, Iro, et al. "Deeper depth prediction with fully convolutional residual networks." *3D Vision (3DV), 2016 Fourth International Conference on*. IEEE, 2016.

附录

A. Scale1 和 Scale2 网络结构

```
Scale12(
  (layer1_1): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU()
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
  )
  (layer1_2): Sequential(
    (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
  )
  (layer1_3): Sequential(
    (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): ReLU()
    (6): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
  )
  (layer1_4): Sequential(
    (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
    (3): ReLU()
    (4): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): ReLU()
    (6): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
  )
  (layer1_5): Sequential(
    (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): ReLU()
    (6): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
  )
  (layer1_6): Sequential(
    (0): Linear(in_features=32256, out_features=4096, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.5)
  )
  (layer1_7): Sequential(
    (0): Linear(in_features=4096, out_features=17024, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.5)
  )
  (layer2_1): Sequential(
    (0): Conv2d(3, 96, kernel_size=(9, 9), stride=(2, 2), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=(3, 3), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
  )
  (layer2_2): Sequential(
    (0): Conv2d(160, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
  )
  (layer2_3): Sequential(
    (0): Conv2d(64, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
  )
  (layer2_4): Sequential(
    (0): Conv2d(64, 1, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
  )
)
```

B. Scale3 网络结构

```
Scale3(
  (layer3_1): Sequential(
    (0): Conv2d(3, 96, kernel_size=(9, 9), stride=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=(2, 2), stride=(1, 1), dilation=(1, 1), ceil_mode=False)
  )
  (layer3_2): Sequential(
    (0): Conv2d(97, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
  )
  (layer3_3): Sequential(
    (0): Conv2d(64, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
  )
  (layer3_4): Sequential(
    (0): Conv2d(64, 1, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
  )
)
```

C. 损失函数

```
    class MyLoss(torch.nn.Module):
        def __init__(self):
            super(MyLoss, self).__init__()

        def forward(self, estimated_depth, ground_truth):
            if estimated_depth.size() == ground_truth.size():
                x = torch.log(F.relu(estimated_depth) + depth_eps) -
torch.log(F.relu(ground_truth) + depth_eps)
                h_x = x.size()[2]
                w_x = x.size()[3]
                img_r = F.pad(x, (0, 1, 0, 0))[:, :, :, 1:]
                img_l = F.pad(x, (1, 0, 0, 0))[:, :, :, :w_x]
                img_t = F.pad(x, (0, 0, 1, 0))[:, :, :h_x, :]
                img_b = F.pad(x, (0, 0, 0, 1))[:, :, 1:, :]
                xgrad = torch.sum(torch.pow(torch.pow((img_r - img_l) * 0.5, 2) +
torch.pow((img_t - img_b) * 0.5, 2), 0.5))
                x_shape = self._tensor_size(x)
                xavag = 0.1*xgrad / x_shape
                loss_result=torch.sum(x * x) / self._tensor_shape(x) -
torch.pow(torch.sum(x) / self._tensor_shape(x), 2) / (2*pow(x_shape, 2)) + xavag
            else:
```

```
            print("Error: dimension of estimated_depth
and ground_truth doesn't match")
            loss_result = 0
        return loss_result

    def _tensor_size(self, t):
        return t.size()[2] * t.size()[3]
    def _tensor_shape(self, t):
        return t.size()[0] * t.size()[1] * t.size()[2] * t.size()[3]
```