Monocular Depth Estimation: a Transfer Learning approach

Francesco Renis, s314803
Politecnico di Torino

Abstract

Accurate depth estimation is crucial in various computer vision tasks related to scene understanding and image reconstruction. Depth estimation models have lately shown to be powerful tools when implemented in sensor-fusion frameworks applied to contexts involving mapping of unknown environments, in applications related to mobile robotics (SLAM) or autonomous vehicles.
While the most common approaches involve processing of stereo images pairs, this project aims at developing a Convolutional Neural Network to produce estimated depth maps of scenes represented by single monocular images. The model exploits a simple encoder-decoder architecture with skip connections, where the encoder block is extracted from the backbone of a YOLOv8 pre-trained for instance segmentation tasks. This Transfer Learning approach exploits the powerful feature extraction capabilities of one of the best state-of-the-art models, allowing to obtain a small and simple architecture able to produce accurate results even if trained for short times and on a relatively small dataset such as NYUv2.

1. Introduction

Transfer Learning approaches have demonstrated to be really powerful in case of lack of computing power, since they usually exploit lighter models and short training time but are able to achieve exceptional results when compared to other strategies. As in every other engineering field, limiting energy consumption, time cost and necessary computational power is crucial. Both when surely in favour of reducing economical expenses but also when constrained by the implementation specifications of the task we are working on. This is where Transfer Learning approaches come in to play.
In particular, the approach shown in [2], which has been strongly taken as a reference for this whole work, demonstrates how a really simple Encoder can be attached to a pre-trained Encoder backbone of a DenseNet model and lead to exceptional results. The idea behind this work is then trying to shrink the model even more, by trying to use as Encoder a truncated YOLO instance segmentation model, an even lighter backbone. This model is combined with proper Data Agumentation techniques, skip connections and a custom loss function made of three different contributes and its performance are analyzed.
Experiments have been carried on with both the models.
It must be pointed out it advance that the obtained results are not so convincing and do not bear comparison with the state of the art models.

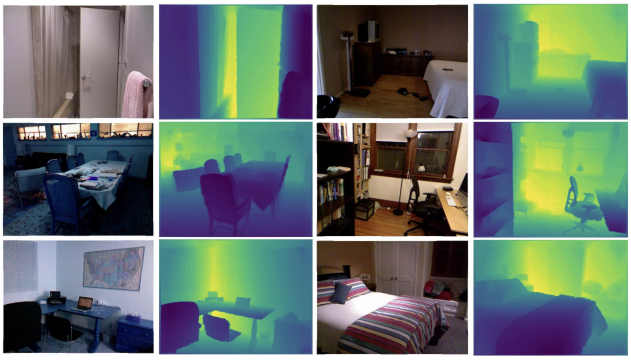

Figure 1. **NYU-Depth v2**: an example of images and relative depth maps

2. Dataset

The NYU-Depth V2 data set [1] provides images from a variety of indoor scenes as recorded by both the RGB and Depth cameras from the Microsoft Kinect at a resolution of 640x480. It is composed of a huge 428GB raw data section, containing raw rgb, depth and accelerometer data as provided by the kinect, and a smaller 4.42GB portion. This 50K samples subset contains preprocessed depth maps whose missing values (due to the inaccuracy of the sensors) have been filled in using the method described in [5]. The depth maps have an upper bound of 10 meters and are rescaled and normalized to [0,1] interval for training, which results in more accurate predictions during evaluation.

2.1. Batch Size

Deriving the ideal value for the batch size required some ”blank” training runs spent just monitoring the CPU and GPU activity. An high batch size, up to a certain limit, better exploits GPU parallel computational power and speeds up the training. On the other hand it puts high load on the CPU that has to move big chunks of data to the memory, even with all the possible speed up tweaks provided by the PyTorch library. Another influence on the ideal value is of course played by the model complexity, as heavier models require heavier workload by the GPU, limiting the maximum value of the batch size, and vice versa.
After some attempts, the best results were achieved with a batch size of 8 for the DenseNet-backbone model and a batch size of 32 for the YOLO-backbone model. On the latter of the two, that batch size value resulted as upper limit for the CPU usage, which behaved as a strong bottleneck. During the training phase the GPU rarely reached a high usage percentage, probably because also of the small size of the model, which will be further dug into later.

2.2. Training-Validation split

Training has been performed on a 90%-10% training set and validation set split and a 600 samples test set is already provided. An additional extremely tiny 1 batch portion of data has been separated from the training set to perform studies that will be analyzed later on.

2.3. Data Augmentation

Data Augmentation policies are essential to expand the set of data seen by the model and to provide additional variability to it. In this work, however, some constraints are imposed by the nature of the problem. The model we are dealing with analyzes images as a whole, and has to rely on the geometry of the real world scenes represented in order to make its predictions. This is why common image transformations like random distortion, crop or rotation are not applicable.
The techniques that have been used preserve the reciprocal relations in the elements on the image. The only two transformations implemented are random horizontal flip, with a probability of 0.5, and random color channel permutation, with a probability of 0.25.
As a side note, after some tests, it resulted that pre-loading all the dataset in RAM does not lead to significant improvement

3. Methods

As previously stated the base idea inspired from [2] is to create an extremely light model, see Table 1), that uses a truncated YOLOv8 instance segmentation model as Encoder block, connected to a custom simple Decoder. The

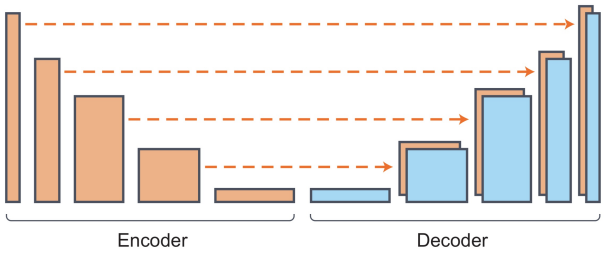

Figure 2. **Encoder-Decoder model architecture**: encoder backbone from truncated YOLO instance segmentation model connected to simple sequence of convolutional and upsampling layers with skip connections

Encoder Backbone	N.Parameters	Size
DenseNet	12,484,480	50MB
YOLO	2,142,192	5MB

Table 1. **Encoder Backbones**: visual overview of the difference between the two backbones

Encoder backbone is loaded pre-trained with unfreezed weights during the training process. A complex loss is also used to account for the complexity of the task we want to achieve.

3.1. Encoder

YOLOv8 has made significant improvements from its original version [6]. Although there is no reference paper describing accurately the work, some precise information regarding its architecture are available online [7]. Its backbone layers are kept exactly the same, the Decoder portion is attached to the last SPPF block. In particular, YOLOv8 pre-trained instance segmentation model has been used, following the intuition that it could be more accurate in analyzing object boundaries.
On the other hand, since it has been used to conduct some experiments, the DenseNet-169 backbone is as from [4] and pre-trained on ImageNet. The Decoder portion is attached its the last DenseBlock.
Finding accurate reference material, in particular regarding the YOLOv8 model, has been really tough. Even when available, it is not so straightforward to understand, sometimes for it being actually not so clear but also for the amount of cascaded knowledge required. Model definition, especially the feature extraction from the pre-defined backbone and the creation of the skip connections has been quite challenging.

3.2. Decoder

The custom attached Decoder starts with a 1x1 Convolutional layer with the same output channels as the ones of the extracted backbones and then relies on a simple series of

Layer	Output size	Function
INPUT	$480 \times 640 \times 3$	
YOLO Backbone	$8 \times 10 \times 256$	Encoder
CONV1	$8 \times 10 \times 256$	Convolution 1×1
UP1	$15 \times 20 \times 256$	Upsample 2×2
CONCAT1	$15 \times 20 \times 384$	Concatenate P5
UP1-CONVA	$15 \times 20 \times 128$	Convolution 3×3
UP1-CONVB	$15 \times 20 \times 128$	Convolution 3×3
UP2	$30 \times 40 \times 128$	Upsample 2×2
CONCAT2	$30 \times 40 \times 192$	Concatenate P3
UP2-CONVA	$30 \times 40 \times 64$	Convolution 3×3
UP2-CONVB	$30 \times 40 \times 64$	Convolution 3×3
UP3	$60 \times 80 \times 64$	Upsample 2×2
CONCAT3	$60 \times 80 \times 96$	Concatenate P2
UP3-CONVA	$60 \times 80 \times 32$	Convolution 3×3
UP3-CONVB	$60 \times 80 \times 32$	Convolution 3×3
UP4	$120 \times 160 \times 32$	Upsample 2×2
CONCAT4	$120 \times 160 \times 48$	Concatenate P1
UP4-CONVA	$120 \times 160 \times 16$	Convolution 3×3
UP4-CONVB	$120 \times 160 \times 16$	Convolution 3×3
CONV2	$120 \times 160 \times 1$	Convolution 3×3

Table 2. **Complete Architecture with YOLO backbone**: overview of the complete architecture. Each UP-CONV layer is implicitly followed by a LeakyReLU activation, as explained. Size are as used during training and testing in this work, which is half of the original size from NYU-Depth v2 Dataset

4 UpSample blocks. Each UpSample block takes as input the output from the previous layer and the output from the encoder layer with the same spatial resolution. The concatenation of the two is upsampled x2 with bilinear filtering and then passed to 2 pairs of a 3x3 convolution a Leaky ReLU activation with α equal to 0.2. This is the most basic structure needed to achieve a decent upsampling, as described in the literature. More sophisticated layers, at the cost of increased model complexity and size, do not seem to be leading to better results. In any case, the aim of this work is to develop the lightest possible model.
Table 2 shows an overview of the complete architecture with the YOLO backbone.

3.3. Loss function

In order to compensate for the simplicity of the model, more complexity has to be put inside the definition of the loss function. Due to the strong inspiration already taken and due to the extremely limited amount of computating facility which didn't allow for much trial and error opportunities, the loss terms have been implemented exactly as described in the reference article [2].

$$L(y, \hat{y}) = \lambda L_{depth}(y, \hat{y}) + L_{grad}(y, \hat{y}) + L_{SSIM}(y, \hat{y})$$

The complete loss function accounts for three terms. Without deep details, since it was not much an object of the work, it is composed as follows.
The first term is the point-wise L1 loss defined on the depth values:

$$L_{depth}(y, \hat{y}) = \frac{1}{n} \sum_{p}^{n} |y_p - \hat{y}_p|$$

The second term is the L1 loss defined over the image gradient of the depth image, where gx and gy, respectively, compute the difference in x and y components for the depth image gradients of the the ground truth and the predicted image:

$$L_{grad}(y, \hat{y}) = \frac{1}{n} \sum_{p=1}^{n} |\mathbf{g}_x(y_p, \hat{y}_p)| + |\mathbf{g}_y(y_p, \hat{y}_p)|$$

The last term uses the Structural Similarity Index Measure, which from the literature is a widely used metric for depth estimation tasks [3]. Since the SSIM has an upper bound of one, it is defined as:

$$L_{SSIM}(y, \hat{y}) = 1 - SSIM(y, \hat{y})$$

Only one weight parameter has been defined for the L_{depth} loss term, and it has been set to 0.1.

3.4. Other hyperparameters

Remaining model hyperparameters that have not been yet discussed are learning rate and epochs (training time).
The first acceptable value for the learning rate has shown to be 0.0001: higher values lead to divergence of the model (which resulted as $'nan'$ values for the computed training and validation losses), and lower values lead to really slow training epochs. Adam is chosen as optimizer with default parameters, no analyses have been carried on into it.
The number of training epochs instead depends on the specific experiment conducted, but in general its choice has been strongly influenced by the limited computing resources available, which will be analyzed later.

3.5. Performance evaluation

Model evaluations have been performed on the provided test set and compared to the state of the art results of [2]. Other than visualization, the error metrics used are:

- average relative error (rel): $\frac{1}{n} \sum_{p=1}^{n} \frac{|y_p - \hat{y}_p|}{y}$
- root mean squared error (rms): $\sqrt{\frac{1}{n} \sum_{p=1}^{n} (y_p - \hat{y}_p)^2}$
- average log error: $\frac{1}{n} \sum_{p=1}^{n} |\log_{10}(y_p) - \log_{10}(\hat{y}_p)|$

4. Implementation details

All the work has been mainly carried out using the online Kaggle Notebook platform, which offers a framework with a unknown quad-core CPU, 30GB of RAM and a NVIDIA P1000 GPU with 16GB of memory. Google Colab, which provides similar computing power, has also been minorly used.
The limited computing facilities had a heavy impact on the results of this work. This is understandable, since the utilized resources are free, but it is important to be mentioned for clarity. Limited computational power, frequent freezing and restricted amount of hours of GPU computing available per week lead to hard times in the experimental phase of this work. Moreover, across almost all the tests, even with all the possible load moved to the GPU, the CPU has been the main bottleneck of the system. It only allowed a reduced number of usable parallel workers for the Data Loaders (4 on Kaggle, 2 on Colab) which lead to moderately long training epochs. This, combined with the limited amount of hours of GPU usage available per week and the frequent unpredictable disconnections, forced to split the training phase among different small sessions, to limit the risks of losing the work and to economize on where to put trial and error efforts.

5. Experiments

With all said and clarified, the focus of this work was in the end the training of the designed model.
Because of the implementation problems and limitations described above, producing a satisfactory result has been very challenging. For a while it remained unknown whether there was an issue with the model being not appropriate for such a task or if the problems were related to insufficient training times. The answer to that probably resides in both. The main question that raises is if the model is able to learn something at all.

5.1. Overfitting test

The main test conducted was trying to lead the model to overfit to its input data, to analyze its capability to actually predict something reasonable as its output. During this experiment, both the YOLO-backbone model and the DenseNet-backbone model were trained on an extremely tiny 8 samples batch of training data for 6000 epochs, and the results are only analyzed visually.
The results in Figure 3 show that the custom designed YOLO-backbone model is indeed able to extract knowledge from the data and predict moderately accurate depth maps when compared to its reference version with DenseNet backbone, with a mild blur. Moreover, it has been trained for the same amount of iterations over the input data but for less than half the time.

Model	rel↓	rms↓	log_{10}↓
Ibraheem Alhashim et al.[2]	0.123	0.456	0.043
This model	0.471	0.602	0.095

Table 3. **Performance metrics evaluation**: this model behaves about 3 times worse than its reference state-of-the-art version.

Those results prove overall as really promising towards the idea behind this work.[1].

5.2. Best model

The best model that has been obtained has been trained on images rescaled to half their original size. The model produces images at half the input size, which are then upsampled. The model has been trained for 200 epochs, 16 hours total, with learning rate equal to 0.0001 and batch size equal to 32, as previously discussed.

The model is able to achieve quite satisfactory results, both visually and on a numerical performance perspective (Table 3).
During the experiments on the best version of the model, a peculiar issue arise: in case of shorter training time, the model could ran into producing flat depth maps in some particular scenes representing objects with small relative distances. By lengthening the training phase, as expected, this problem doesn't show anymore, and the model behaves quite well even in such critical environments (for example at row 2 in Figure 5)
Loss values in Figure 4 show a good trend of the training and validation curves. Starting from around epoch 150 the validation loss sees a small raise with respect to the training loss. The model seems to be ceasing learning and undergoing a really slight overfitting, even though it does not increase with time. This behaviour is likely related to the simplicity of the model itself, rather than to the training dataset being not adequate. Given its limited size, this model is probably not able to extract more information from the training data, which explains the trend of the loss curves and leads to having the predicted depth maps not really accurate and "sharp" from a visual point of view (Figure 5).

Given the really little size of this model, probably this issue could be compensated by adding a further loss term to account for that.

6. Conclusion

The main issues and imperfections regarding this work could be easily overcome by having a proper computing facility available, which above all would unlock the chance to invest more time in trial and error experiments.
This project shows one more time how Transfer Learn-

[1]The two models can be found respectively with names *Model_YOLO_overfit.pth* and *Model_DenseNet_overfit.pth*

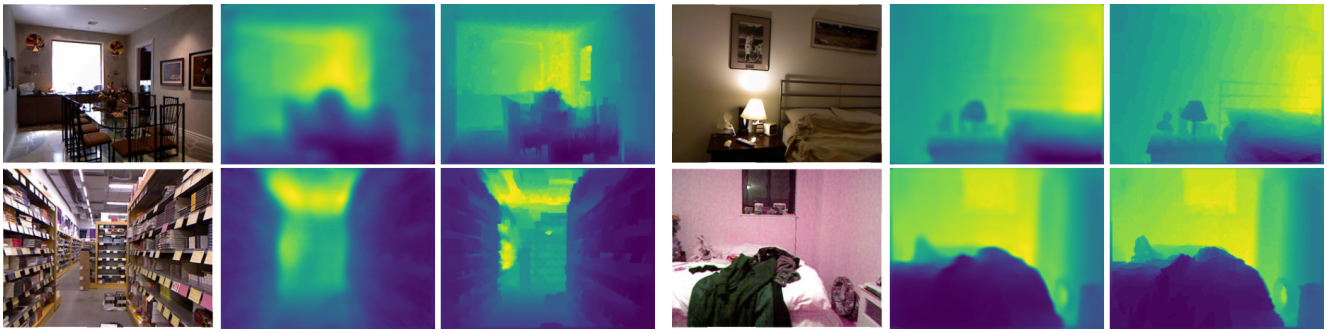

Figure 3. **Overfit tests**: Input, prediction, ground truth pairs for YOLO-backbone model and for DenseNet-backbone, respectively, to the left and to the right. Training lasted 50 minutes on the YOLO-backbone model and 80 minutes on the with DenseNet-backbone model

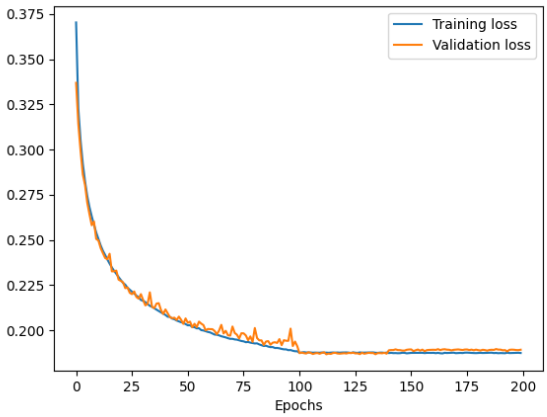

Figure 4. **Best model Training and Validation losses**

ing approaches can definetely unlock new strategies to face complex tasks, enabling the development of lighter but competitive models, that require relatively short training times when compared to similar state of the art results for the same task.

Some additional refinement work could be done. For instance, adding an extra loss term could be a strategy to compensate for the extreme simplicity of the architecture. The slight blur all over the depth map, as can also be deduced from the results of the overfitting tests (Figure 3, is probably a consequence of the structure of the model an it is likely very difficult to overcome,. An adversarial loss term, for example, could be relevant to increase the reliability of this model.

On the other hand, given the results discussed in the sections above, such a work demonstrates as really promising. The complete model weights about 20 MB (with a 9MB encoder), which is 10 times less than its equivalent state-of-the-art version with DenseNet backbone. This is its massive feature. Although the difference in size is significant, its performance is just about 3 times worse, which means that somehow it is worth the trade-off. The results are not exceptional, but in a real-world environment this model's peculiarities could be taken in advantage in applications where the depth measure does not need to be extremely accurate at any point or when exploited in a sensor-fusion framework.

References

[1] NYU Depth V2 Nathan Silberman.

[2] Ibraheem Alhashim and Peter Wonka. High quality monocular depth estimation via transfer learning.

[3] Clément Godard, Oisin Mac Aodha, and Gabriel J. Brostow. Unsupervised monocular depth estimation with left-right consistency, 2017.

[4] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks, 2018.

[5] Anat Levin, Dani Lischinski, and Yair Weiss. Colorization using optimization. 23(3):689–694.

[6] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. pages 779–788, 2016.

[7] Ultralytics. Home — docs.ultralytics.com. `https://docs.ultralytics.com/`. [Accessed 26-04-2024].

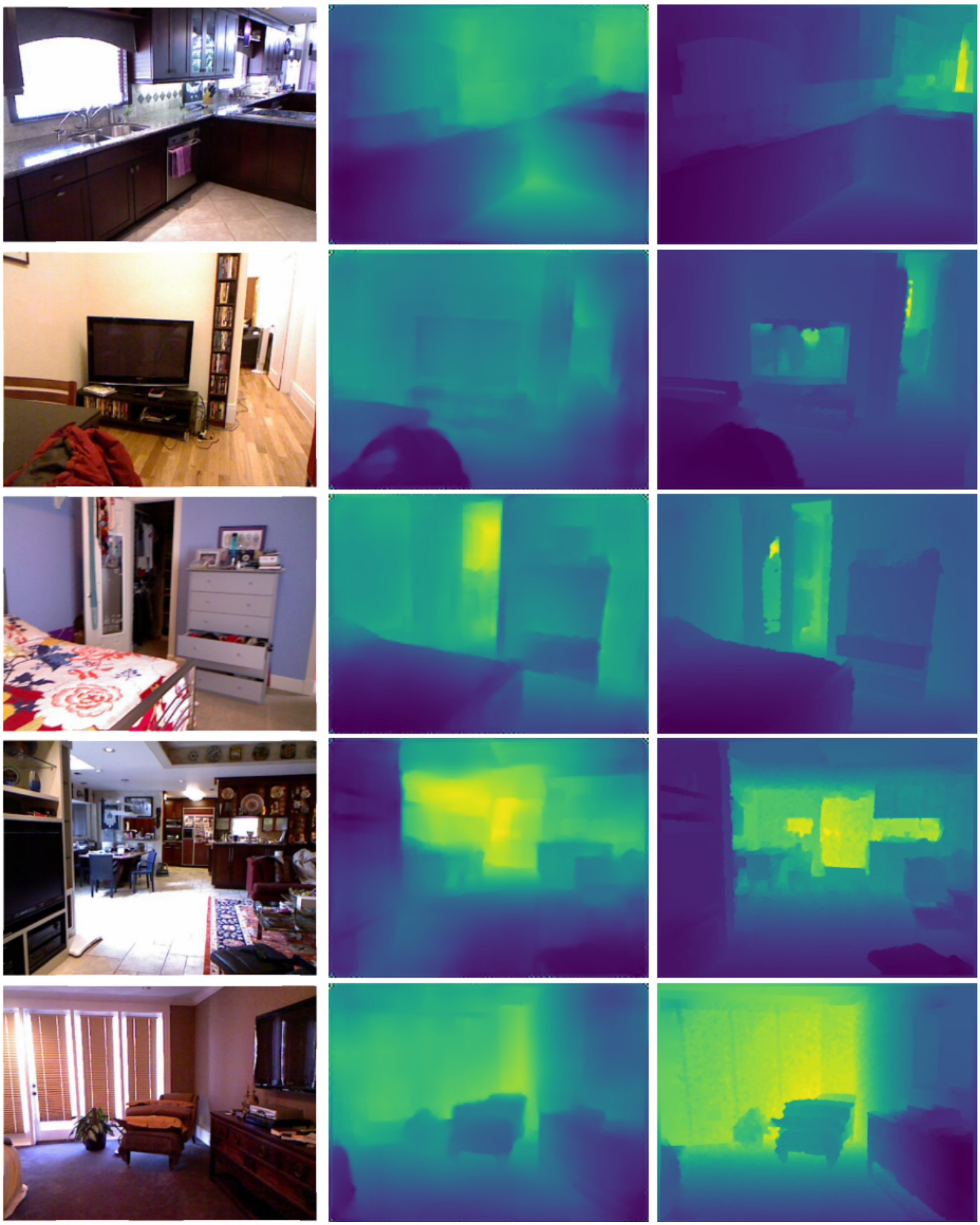

Figure 5. **Input image, predicted depth map and relative ground truth**