Real-Time Depth Estimation from 2D Images

Jack Zhu
jackzhu@stanford.edu

Ralph Ma
ralphma@stanford.edu

1. Abstract

Single image depth prediction allows depth information to be extracted from any 2-D image database or single camera sensors. We applied transfer learning to the NYU Depth Dataset V2 and the RMRC challenge dataset of indoor images to form a model for single image depth prediction. Prediction on test images had a Root Mean Square Error of .833 which surpassed previous non-neural network methods. Through experimentation, we found that only performing backpropagation through the last five convolution layers of the VGG net used by transfer learning is better than performing backpropgation through the whole network. We also found that using upsampling/convolution instead of fully connected layers led to comparable results with 8 times less variables. Quantitative results show our predicted images contain course depth information of the objects in the input images. We attempt to explain the limitations of network by examining errors. Next, we attempted to run our model live on a Nvidia TK1 in real time. Using the Microsoft Kinect as a camera, we reached a test time classification rate of around 1 Hz. To test the model's ability to adapt to settings, we experimented with real time training for the model which improved performance in a localized setting.

2. Introduction

Depth prediction from visual images is an important problem in robotics, virtual reality, and 3D modeling of scenes. With stereoscopic images, depth can be computed from local correspondences and triangularization. However, multiple images for a scene are often not available, leading to a need for a system that can predict depth based on single images. Methods for inferring depth from a single image have involved supervised learning on segmented regions of images using features such as texture variations, texture gradients, interposition, and shading [3, 2]. Recent work using convolutional neural networks [6, 7] have shown promise in generating accurate depth maps from single 2-D images.

For our project, we propose to use transfer learning to meet depth estimation benchmarks set in [6, 7] in 2D images. We explore the differences in training on an untrained network, and on a network pre-trained to the ImageNet Classificaion task [11]. As a secondary goal, we will attempt to export our neural network architecture and perform real-time depth perception on a Nvidia TK1 to enable real-time evaluation of scene depth. We also explore real-time learning to better understand how localized data helps to improve predictions in specific indoor settings.

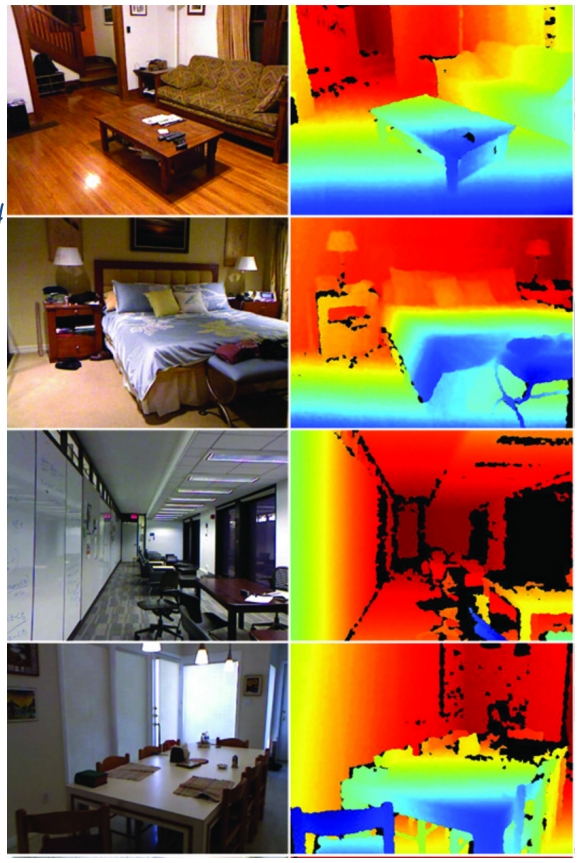

Figure 1: Depth Example

Figure 2: Examples of NYU depth V2 dataset images [10]

3. Related Work

Prior to the popularization of convolutional neural networks in early 2010s, several groups worked on single image depth prediction by pairing supervised learning and feature engineering. Make3D [2] divides each image into patches and then extracts features on texture variations, texture gradients, and haze used to represent both absolute depth and relative depth to other patches. Then the authors apply a Markov Random Fields (MRFs) to infer the depth of each path taking into account both the patch features and features relative to other patches.

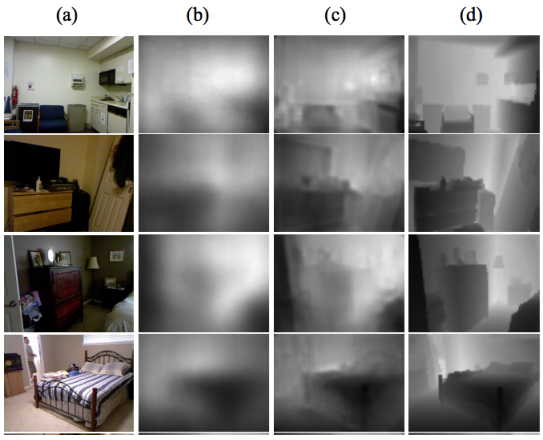

Figure 3: Example results from Eigen Et. al [6], showing (a) input, (b) output of their course network, (c) output of their multi-scale network, (d) ground truth. We aim to match the outputs of their course network (b).

In 2014, [6] released a hierarchical convolutional neural network model trained on the NYU Depth Dataset containing 407,024. The authors first feeds input data into a course model that predicts the course depths and then uses the output of the course model in a fine model to generate final prediction. With this model, the authors are able to reach a root mean square error of .871 which improves dramatically from the RMSE error of 1.21 achieved by Make 3D [2]. Figure 3 show results from the authors. As seen in the results, the course layers provides very general depth information and the fine layer on top provides the details. Interestingly, adding the fine layer increased the RMSE from .871 to .907. The authors of [6] improved on their results further in [7] using a three layer model aimed to learn depth information at three different scale. With the new model, the authors reached a RMSE of .641 on NYU Depth Dataset V2.

4. Methods

We approached this problem with the goal of creating a model that we will be able to run on the Nvidia TK1. We derived our architecture from the course model in [7]. [7] creates a course model that uses layers initialized with weights from the Oxford VGG net [12] and then uses two fully connected layers to create course depth information from input images. In [6], the authors show that while the fine-scaled models make the final result, images have greater details, adding the fine-scaled model does not have a great impact on root mean square error. Thus, we decided to only use the course model, which also takes up less memory in embedded devices.

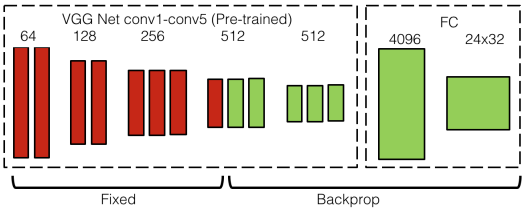

Figure 4: Our models used parts of a pre-trained VGG net [12]. We added our own fully-connected layers at the end, as well as allowed the training to back-propagate through 5 convolutional layers.

Our model is shown in figure 4. Our architecture uses the the VGG net which was originally trained on Imagenet [11] on an object classification task. The green layers represent the layers that we perform back propagation through and the red layers are the layers which we freeze the weights. All convolution layers of VGG net uses relu activation and we use relu activations for our finally fully connected layers. Between our fully connected layers, we add drop out with .5 probability of dropping out. We believe the early convolution layers learned broad feature information that can be used to better predict depth. The first box in figure 4 contains the convolution layers of the VGG net. We append the convolution layers with two fully connected layers. These fully connected layers increase the capacity of the network and allows better learning of relative depth. However, these FC layers contains a large number of parameters, thus we explore replacing the FC layers with a single up-scaling layer and a convolution layer (see Section 6.2).

For evaluation, we use the root mean square error with slight edits. The root mean square error for image T can be evaluated as

$$L = \sqrt{\frac{1}{|T|} \sum_{y \in T} ||\hat{y}_i - y_i||^2} \quad (1)$$

However, in the dataset, the depths images often have Nan areas where there are no height information. We mask those regions out of the final loss. Therefore, our root mean square error has the following equation for image T.

$$L = \sqrt{\frac{1}{|T^*|} \sum_{y \in T^*} ||\hat{y}_i - y_i||^2} \quad (2)$$

$$T^* = \{X | X \in T \text{and} X \neq \text{nan}\} \quad (3)$$

We also use L2 regularization on the last two fully connected layers of the network. Thus, if the weight of the the final two fully connected layers are W_1 and W_2, then the loss is

$$L = \sqrt{\frac{1}{|T^*|} \sum_{y \in T^*} ||\hat{y}_i - y_i||^2} + W_1^2 + W_2^2 \quad (4)$$

$$T^* = \{X | X \in T \text{and} X \neq \text{nan}\} \quad (5)$$

The root mean square error measures the difference between each pixel depth in the final predicted image $\hat{y}_i$ and ground truth depth y_i. While this loss is easy to interpret, it is does not take into account the effect of different scales of images. To perform batch gradient updates, we use Adam, a gradient-based method of optimization [9].Figure 5 shows the steps of Adam and gives explanation of the steps. m is the first moment update variable that is a linear combination of the gradient found in the back-propagation step and the previous update m. Intuitively, m will causes updates in certain directions to increase in magnitude if previous updates have also been in the same direction. v is the second moment estimate and intuitively, it will limit parameter updates in dimensions that have received large updates. The βs are decay variables. We initialized β_1 to be .9 and β_2 to be .999. ϵ provides numeric stability and we initialized it to be $1e-08$.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Figure 5: Pseudocode for performing Adam [9]

For learning, we set our batchsize to be 10 in order to fit the data into the 4095MB memory of our GPU. For our model, we first attempted to overfit the 100 data points using our model. We reached a RMSE of .22 within 500 iterations. Then we performed cross-validation on both the learning rate and regularization of the final two fully connected layers. We used a learning rate of .00001 and a regularization constant of .02.

We implement our architecture using the Keras [5] package with Theano [4] backend, and performed training on Nvidia Grid K20 provided by Amazon EC2 system. For our final model, we trained for 7 hours.

4.1. Hardware

To run our Keras models on live data, we purchased an Nvidia Tegra TK1 Development kit. The TK1 provides an NVIDIA Kepler GPU with 192 CUDA cores, and a Quad-Code ARM Cortex CPU. The board has 2GB RAM, split between GPU and CPU. We run Ubuntu 14.04 L4T, CUDA and CUDNN versions 6.5. To avoid hitting memory limitations, we added a 4GB swapfile on a high speed SD card.

To stream video, we use a Microsoft Kinect v1. We use the Libfreenect library to retrieve and align both video and calibrated depth data. This allows us to display ground-truth depths along with predicted depths. This is the same camera used to collect the RMRC dataset [1] to collect the training data we used. We built software to run and visualize the results in real time. We achieve frame rates around 1 fps. We also experimented with real-time training on captured images of scenes, by collecting data, performing preprocessing, and running training epochs in real-time.

5. Dataset and Features

For our dataset, we used the RMRC indoor depth challenge dataset consisting of RGB images of various indoor scenes and corresponding depth maps taken by Microsoft Kinect cameras. To increase our training data, we also used subsets of the NYU Depth Dataset V2. In total, we had 9900 RGB/Depth pairs which we split into 9160 training points, 320 validation points, and 320 test points.

For each rgb/depth image, we cut out 10 pixels on the border, because when the RGB and Depth images are aligned, the borders tend to have Nan values. Then we rescale each RGB image to 224x224 and each depth image to 24x24. Finally, we subtract the channels of the RGB image by (103.939, 116.779, 123.68) which are the channel means for the dataset that VGG net is trained on.

We also perform data augmentation, in order to reduce overfitting. For our dataset, we randomly select images and generate rotated version of selected images with rotation angle ranging from -5 degrees to 5 degrees. Furthermore, we randomly select images and multiply each color channel by constants ranging from .95 to 1.05. Finally, for each image, there is a 50% probability we generate the horizontally flipped image to be included in our dataset. In total, data augmentation increases our dataset size by a factor of 2.

6. Results

Some example depth estimations from our network are shown in figure 6 above. Qualitatively, it seems that the depth predictions track the general geometry of the scene well: parts of the image that are near walls have lower pre-

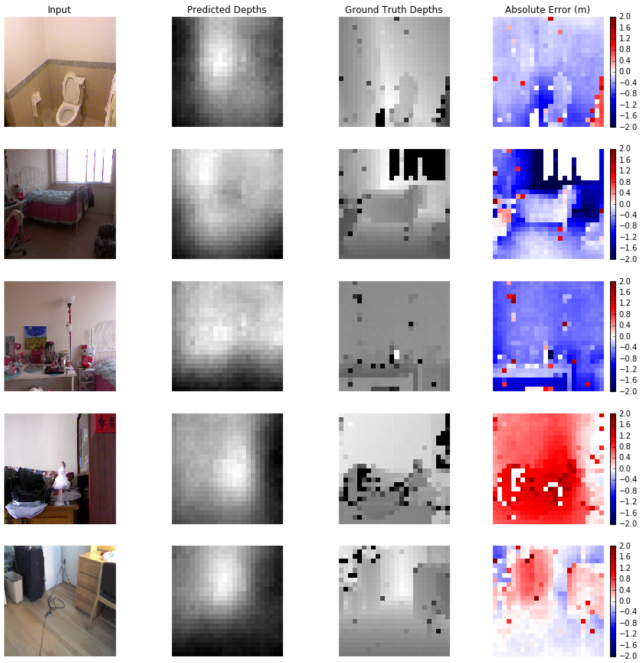

Figure 6: Results and errors of our network on our test dataset. From Left to Right: Input image, Our prediction, Ground-truth depth, Absolute error maps

dicted depth than parts of the image in the middle of the scene. Wall asymmetries are captured in the depth maps, where closer sides of the walls are correctly predicted to be closer. Large rectangular objects such as shelves are also reflected, as in example 4 and 5.

Looking at the error plots, it seems that the network predicts poorly around objects in the scene and around sharp edges and corners. On larger scenes, the model generally under-estimates, while on closer, scenes, it generally over-estimates distances. In general, these reflect the model's inability to generalize scale information from objects in the scene. It also seems that on scenes with clear floor geometries (images 1 and 5), the planes of the walls and floors have generally lower estimation errors than around corners and objects. This could be a result of the depth information captured in the geometry of the convergent lines in a scene. Overall, we observe qualitatively similar results as those in the course network of [6], in figure 3b.

Figure 7 shows our best regression results with several past methods for single-image depth estimation. We outperform both the course and the course+fine network in [6] in RMSE loss. The hierarchical model in [7], however, outperforms our model.

6.1. Transfer Learning Experiments

In our model with performed backpropagation for the last five convolution layers of the VGG net. To test the number of convolution layers that we should perform backpropagation through, we attempted to perform backpropgation through all layers. First we initialized the model with the weights found in the VGG net. Allowing the model, to backpropgate through the whole model, we reached a training error of .75 and a test error of 1.028 after 5 hours of training. By backpropagating through the entire model, we increase the capacity of the model, but we also decrease the generalization provided by the earlier layers of the VGG net, which identifies important edges and corners.

Method	RMSE Loss
Mean [6]	1.244
Make3d [6]	1.214
Course Network [6]	.871
Course + Fine Network [6]	.907
Hierarchial VGG [7]	.641
Ours	**.833**

Figure 7: Our network compared to non-CNN methods, and CNN architectures found in [6] and [7].

In our second experiment, we experimented with the importance of using the weights of the VGG net. Again, we are only performing back propagation through the last 6 layers. However, we initialize the weights of the last 6 layers of the VGG net with Javier normal initialization [8]. After 4 hours of training, we reached a training error of 1.11 and a test error of 1.18. We did not reach convergence for the model which can be accounted by the initialization.

Through these experiments we showed the power of applying Transfer Learning. Transfer learning provides layers that can better generalize to shapes important for visual tasks such as the one studied by this paper. Furthermore, the weights of models found in transfer learning provide good initializations for points.

6.2. Upsampling

```
Convolution2D (conv5_3)       (None, 512, 14, 14)          2359808
MaxPooling2D (maxpooling2d)   (None, 512, 7, 7)            0
Flatten (flatten)             (None, 25088)                0
Dropout (dropout)             (None, 25088)                0
Dense (dense)                 (None, 4096)                 102764544
Dropout (dropout)             (None, 4096)                 0
Dense (dense)                 (None, 576)                  2359872
Reshape (reshape)             (None, 24, 24)               0
_________________________________________________________________________________
Total params: 119839104
_________________________________________________________________________________
```

(a) Using FC layers

```
Convolution2D (conv5_3)       (None, 512, 14, 14)          2359808
UpSampling2D (upsampling2d)   (None, 512, 42, 42)          0
ZeroPadding2D (zeropadding2d) (None, 512, 50, 50)          0
Convolution2D (last)          (None, 1, 48, 48)            4609
MaxPooling2D (maxpooling2d)   (None, 1, 24, 24)            0
Reshape (reshape)             (None, 24, 24)               0
_________________________________________________________________________________
Total params: 14719297
_________________________________________________________________________________
```

(b) Using Convolutions

Figure 8: Different Architectures

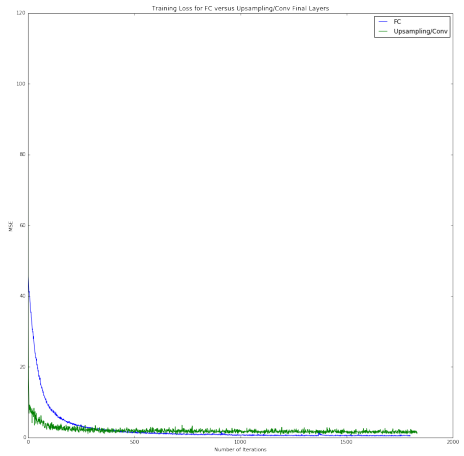

Figure 9: Training Error

In order to create a model that can fit in an embedded the system, we decided to test the importance of the final fully connected layers. With the fully connected layers, our network has 119839104 parameters with the final two parameters accounting for 87.7% of those variables. We change our architecture by replacing the the final two fully connected layers with a upsample layer that upsamples the results of the VGG network from 14 x 14 to 42 x 42. Then we we use a single convolution filter followed by a max pooling layer to create the final 24 x 24 image. Figure 8 shows 8.14 times reduction in the number of parameters.

The final-conv model has a test error of 1.348. We see from the training error graph in Figure 9, the loss converges much faster but converges higher than the model with fully connected layers. This makes sense, because the fully connected layers contains more parameters and is able to learn relationships of different parts of the image. However, the much smaller size makes it a viable candidate to be placed on an embedded device.

6.3. Embedded Hardware

The major challenge to running our models on the NVIDIA Jetson TK1 was the memory constraints set by the Tegra GPU. The GPU and CPU share 2GB of RAM, and even with a sizable swap partition, we experienced memory limitations when loading our larger networks into memory. With smaller networks (smaller fully-connected dimension), we were able to achieve real-time regression results at around 1 Hz from Kinect camera images. We also implemented performing training on real-time captured data; running 3 epochs on each batch of collected data while we moved the camera around the room. This allowed us to overfit images to the room that we were in, and resulted in more detailed and accurate real-time predictions. Figure 10 shows the real-time display, along with a sample prediction after such training on live data.

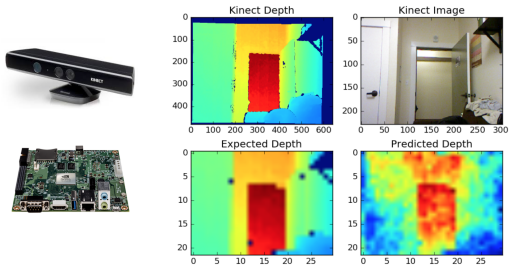

Figure 10: Left: We use an NVIDIA Jetson TK1 Development Kit and a Microsoft Kinect camera for real-time estimation. Right: Example of our interface running real-time. The predicted image is after several epochs of training on data gathered in the room.

7. Summary and Potential Improvements

We have shown that transfer learning on convolutional neural networks can be effective in learning to estimate depth from single images. However, predicting detailed depth maps remains a hard regression problem requiring many training examples. Hierarchical models such as those in [7] continue to out-perform our single-scale models.

Adding segmentation data and surface normal data, as shown in [7] may better separate objects whose depths may be obfuscated by very similar color or harsh lighting. We may have also benefited from varying the loss function; [7] mentions that they could achieve outputs that better track the input data if they regularize their loss on the gradients of the log-differences.

In hardware, we found difficulty engineering solutions to memory limitations in our hardware. To further decrease the memory representation of the fully connected layers, we could decrease the space and time needed to compute the feed-forward regression, and compute the SVD of the weigh matrix and drop the dimensions with low singular values instead.

References

[1] Rmrc: Depth from rgb dataset, 2014. 4105 RGB and Depth pairs.

[2] A. N. Ashutosh Saxena, Min Sun. Make3d: Depth perception from a single still image. *Proceeding of the Twenty-Third AAAI Conference on Artificial Intelligence*, pages 1571–1576, 2008.

[3] A. N. Ashutosh Saxena, Sun H.Chung. Learning depth from single monocular images. *NIPS*, 2005.

[4] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. J. Goodfellow, A. Bergeron, N. Bouchard, and Y. Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[5] F. Chollet. Keras deep learning library for python. convnets, recurrent neural networks, and more. runs on theano and tensorflow. *GitHub repository*, 2013.

[6] D. E. C. P. R. Fergus. Depth map prediction from a single image using a multi-scale deep network. *NIPS*, 2014.

[7] D. E. R. Fergus. Predicting depth, surface normals and semantic labels with a common multi-scale convolutional architecture. *ICCV*, 2015.

[8] Y. Glorot, Javier; Bengio. Understanding the difficulty of training deep feedforward neural networks. *13th International Conference on Artificial Intelligence and Statistics*, 2010.

[9] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[10] P. K. Nathan Silberman, Derek Hoiem and R. Fergus. Indoor Segmentation and Support Inference from RGBD Images. *ECCV*, 2012.

[11] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[12] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *ArXiv e-prints*, Sept. 2014.