CS230

Visual Scene Embedding with Neural Networks

Colin Schultz
Stanford University
colinrs@stanford.edu

Abstract

Humans are very adept at understanding a lot of information about their environment using a small number of primarily visual observations. We can quickly determine our relative position and plan a path through a cluttered room from just a single viewpoint. However, artificial systems such as robots still struggle to accurately and concisely extract relevant data from visual inputs. We propose a model which uses an arbitrary number of visual observations to produces a dense embedding of the environment, which can then be used to aid in important tasks for robots, namely mapping and localization.

1 Introduction

One important requirement to allow a robot to perform useful tasks is the ability for it to perceive and store information about its environment. We focus on visual inputs as they provide a very large amount of information but can be difficult to effectively process. To solve the perception and understanding problem we use a Representation network, which takes an arbitrary number of observations and outputs a dense embedded representation of the scene. Each observation consists of both an RGB image and the pose from which it was observed.

The merit of this generated embedding is determined by how well it can be used to solve useful tasks. In this paper, we used mapping and localization as the tasks which the system has to perform based on the scene embedding. These are two important tasks which allow a robot to plan paths and maintain a sense of its position for navigation purposes. To solve these two tasks, we use two more neural network models: a Mapping network which uses just the scene embedding and generates a 2-dimensional binary map of the scene, and a Localization network which takes the embedding and an image as input, and outputs the pose from which that image was taken. The binary map produced by the Mapping network represents whether or not each pixel of the map is safe to traverse or is occupied by an obstacle.

2 Related work

The specific tasks we approach in this paper (robot mapping and localization) have been studied extensively. These problems are traditionally solved using SLAM (Simultaneous Mapping And Localization) techniques. Different SLAM methods involve the use of Lidar, stereo vision, or simple cameras, but the popular methods are not performed using artificial neural networks [1]. Traditional SLAM approaches can suffer from large memory costs due to storing an increasing number of observations and key points [2]. Additionally, maps generated by visual SLAM techniques are generally made up of landmarks that do not necessarily show the presence or absence of obstacles in a given location [3].

While we could not find examples of neural networks being applied to the exact tasks we approach, there are somewhat similar tasks to which neural networks have been applied. The idea of a scene embedding was inspired by "Neural Scene Representation and Rendering" [4]. This paper generated representations of a scene for the purpose of rendering novel views of the scene. Another task which is somewhat similar to the mapping task is 3D reconstruction. Some methods for this task use a convolutional encoder to produce a dense embedding of a visual observation [5]. This embedding is then used to find what areas are occupied by the object. In this way, our mapping task could be analogously called "2D reconstruction."

CS230: Deep Learning, Spring 2020, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

3 Dataset and Features

Our dataset consisted of a large number of synthetic scenes, where a scene s comprises a tuple $(m, \mathbf{x}, \mathbf{p})$. m is a 64x64 binary map, representing whether each pixel's area is safe or occupied. $\mathbf{x}$ is a list of 64 128x128 RGB images of the scene. $\mathbf{p}$ is a list of 64 poses which correspond to the images in $\mathbf{x}$. Specifically $\mathbf{p}_i$ represents the position of the camera when $\mathbf{x}_i$ was rendered. Each pose vector $\mathbf{p}_i$ is a 7-dimensional vector where the first 3 components represent the (x, y, z) position of the camera, the next 3 components compose a unit vector in $\mathbb{R}^3$ that represents the direction of the camera, and the final component represents the roll of the camera around that unit vector.

We generated these scenes using PyBullet, a physics simulator with synthetic rendering capabilities [6]. The scenes are 8x8 meters, with walls that enclose an axis-aligned rectangular space within the bounds of the scene. The scene is also populated with randomly placed furniture models which were obtained from the IKEA Dataset [7]. The colors of the walls, floor, and furniture were all randomized. In addition, the bounds of the walls within the 8x8 meter scene were also randomized.

The map is 64x64, so each pixel represents a square of 12.5cm on each side. The map is generated by placing a cuboid bounding box over each pixel and checking for collisions with the furniture. If a collision is detected, the given pixel is set to 0, meaning "unsafe" or "obstructed", otherwise it is set to 1 to signify "safe". The bounding box is 0.3 meters tall, which was chosen to approximately represent the height of Stanford Student Robotics' "Pupper" robot [8]. This means that locations which are underneath an obstacle such as a table but still have adequate vertical clearance are considered safe in the map.

The images are rendered from random poses in unobstructed locations. The (x, y) location and yaw are chosen randomly, and the z location, pitch, and roll are also varied uniformly within bounds designed to roughly approximate the unsteady motion of a quadruped robot.

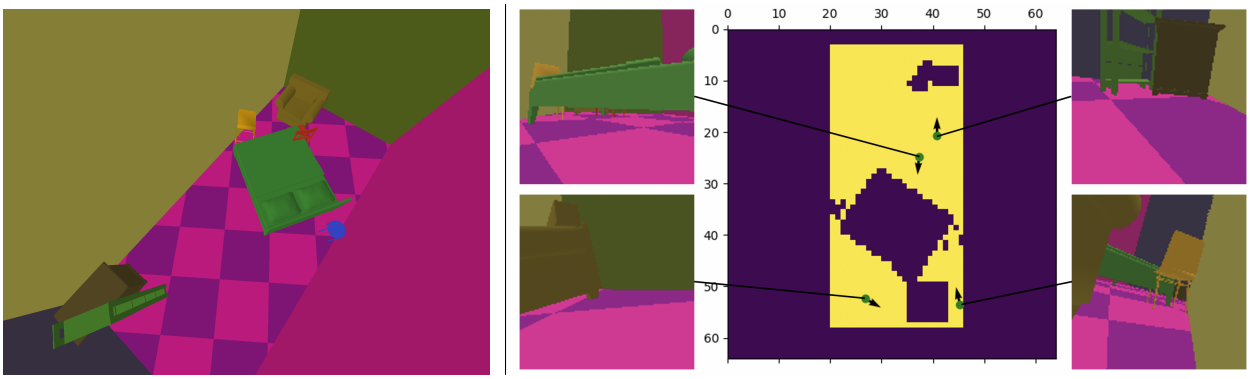

Figure 1: (left) A birds-eye view of a generated scene. (right) The corresponding map for the scene and 4 rendered observations with their poses on the map. The map visual was made with matplotlib [10]. The darker pixels are labelled as unsafe while the lighter pixels are labelled safe.

The original dataset had approximately 40,000 of such scenes (with 64 images each) for training data, and 600 scenes each for a validation and test set. In later experiments, the training dataset was expanded to include approximately 60,000 scenes.

4 Methods

Models were built using NumPy [11] and Tensorflow 2 [12].

4.1 Tile Convolutional Encoder

In the implementation of both the Representation network and Localization network we face the problem of combining both image inputs and dense vector inputs. In the case of the Representation network, we input both images and pose vectors, and for the Localization network we input an image and the scene embedding vector. We solve this issue in both cases using what we will call a Tile Convolutional Encoder. We use a MobileNetV2 [13] convolutional architecture with weights pretrained on ImageNet as the base of the encoder. We chose this model

because of the availability of pretrained weights and the fact that it is optimized for use on mobile systems such as robots.

To incorporate the dense inputs, we take the vector input and repeat it 16 times over two axes so it becomes a 16x16xN tensor where N is the original dimensionality of the dense input. For example, after tiling the 7-dimensional pose vector we would have a 16x16x7 tensor. This tiled tensor is then concatenated with the output of the 5th block of the convolutional architecture, along the filters axis. We then apply a 1x1 convolution to that output to decrease the number of channels so that it can be fed back into the remainder of the convolutional encoder.

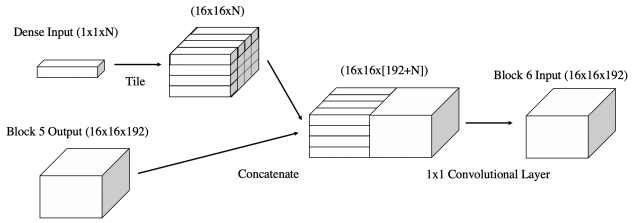

Figure 2: Tiling is used to incorporate dense inputs into the convolutional architecture.

The idea of using tiling on dense inputs in an intermediate layer was inspired by open-source implementations [14][15] of "Neural Scene Representation and Rendering" [4].

The output of the encoder is calculated using a global average pooling over the output of the last convolutional block. Then this output is passed through a dense linear layer with M units to achieve the desired output dimensionality.

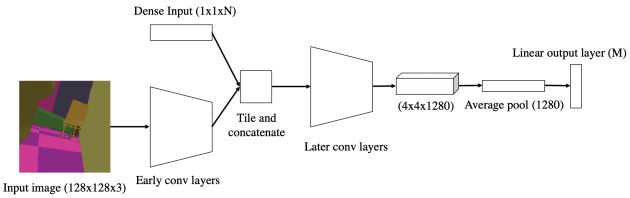

Figure 3: The complete Tile Convolutional Encoder

For more specifics on the base convolutional architecture, we recommend referring to "Mobilenetv2: Inverted residuals and linear bottlenecks" [13].

4.2 Representation Network

The Representation network is tasked with using a series of observations to generate an embedded representation. The model we developed uses a Tile Convolutional Encoder with $N = 7$ (the size of a pose vector) and $M = 512$ (the size of the embedding vector). We then pass the output of the encoder into a GRU with 512 units to output the final embedding. We chose the GRU because it has no extra hidden states besides its previous output. This means that the system only needs to store the current embedding in between observations. This embedding it both used for the actual mapping and localization tasks as well as creating the next embedding after a new observation.

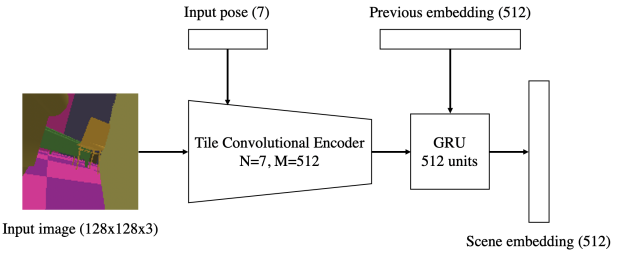

Figure 4: The Representation network architecture

4.3 Localization Network

The Localization network takes an image as input as well as the current scene embedding, and outputs an estimate of the pose from which the image was taken. The architecture is simply a Tile Convolutional Encoder with $N = 512$ (the size of the embedding) and $M = 7$ (the size of a pose vector). The only processing applied to the 7 linear output units of the encoder is that the 3 components which represent the orientation in the pose are normalized to make a unit vector in $\mathbb{R}^3$.

4.4 Mapping Network

The Mapping network takes just the scene embedding and outputs a binary 2-dimensional map of the scene. We considered different architectures for the mapping network including a deconvolutional architecture and one based on Occupancy Networks [9]. Ultimately we chose a simple feed-forward, fully connected network. Because the output is only 64x64 (4096 total pixels), it was feasible to use a fully-connected output and we did not need to rely on the parameter sharing provided by deconvolutions. We also decided against an Occupancy Network-like approach because our map labels are of a fixed resolution so we would gain nothing from the arbitrary resolution advantage of Occupancy Networks.

The final architecture for the mapping net consists of 5 dense layers with the following number of units: 1024, 2048, 2048, 2048, and 4096 in order from input to output. Remember that the input to this network is the 512-dimensional embedding vector. All of the hidden layers in this network used ReLU activation and the final output layer used a sigmoid activation to produce the binary map. The 4096-dimensional output is reshaped to a 64x64 map.

5 Training

5.1 End-to-end Model

It would be impossible to train the networks separately, as there are no ground truth labels for scene embedding vectors. Instead, a scene embedding is considered "good" if it can be effectively utilized for the other tasks.

Therefore, we trained the models in an end-to-end fashion, optimizing the entire process of input images to map and pose estimates. We calculate a loss function for the map and pose estimates, and the gradients are propagated back through the Mapping and Localization networks, then through the scene embedding and Representation network.

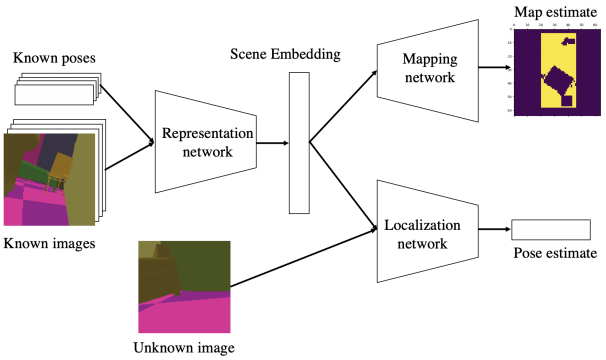

Figure 5: The end-to-end model

5.2 Loss Functions

For the mapping network we used a weighted version of per-pixel binary cross entropy wherein pixels that are on the edges of obstacles are weighted more highly. We chose this method to discourage the model from finding a local minimum where each pixel just shows the average probability of being "safe" over all training samples. This was inspired by [16].

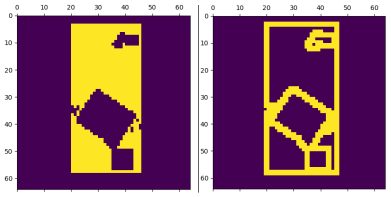

Figure 6: (right) A mask used to weight the loss function for the map label (left). Lighter pixels in the mask are weighted higher.

The loss used for the Localization network was a combination of mean squared error and cosine similarity. We used MSE for the position and roll components, and cosine similarity for the orientation unit vector. Different weights were given to the components to prioritize good position estimates over good orientation estimates.

5.3 Preprocessing

The image inputs are preprocessed by normalizing the values between -1 and 1. For the pose inputs, the position and roll components are normalized between -1 and 1 based on their minimum and maximum possible values. The orientation component is already a unit vector and need not be normalized.

5.4 Training

We trained the end-to-end model over our training dataset of 60,000 scenes. Each time a scene was selected in a batch, 16 images would be randomly chosen as inputs to the representation network and 8 other images would be chosen as inputs to the Localization network. We used an Adam optimizer with a learning rate of 10^{-4} and a batch size of 16 scenes. We trained the model for around 300 epochs over about a week. The exact number is unclear as some adjustments were made over the course of training. The high length of training time was prohibitive for carrying out multiple independent experiments.

6 Results and Discussion

Below are some results from the test set, which was not optimized against in any way during training.

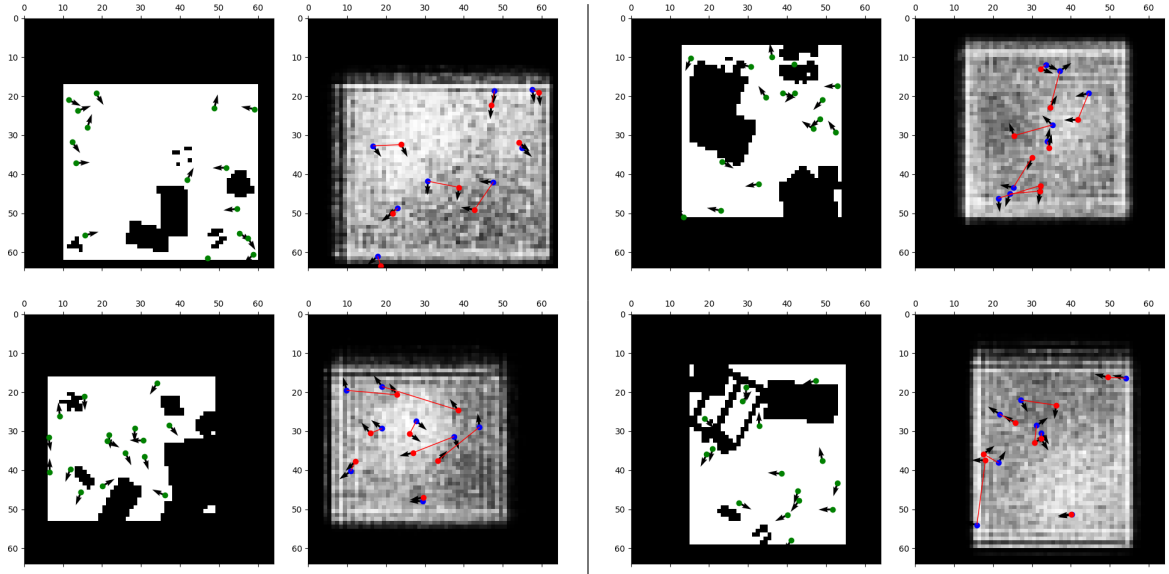

Table 1: Results from the test set. The right image in each pair shows the ground truth map as well as the given observation points. The left images are the generated maps and localization results. Blue points are the true locations and red points are the estimated locations of novel observations.

The results show that the generated representations are quite successful in capturing the dimensions of the room. The generated maps also show a noisy interpretation of which areas of the room had more obstacles. The localization results are inconsistent, with some estimates being very close to the ground truth while others show a larger error. Some of the poorly estimated locations might be due to useless images that show just a flat wall. Because there is no texturing in the scenes, it would be impossible to determine a precise location based on a flat color.

The quantitative metric that we chose for human interpretation is mean Euclidean distance in the (x, y) plane for localization estimates. Over the test set, we found a mean Euclidean distance of about 3.52 meters. This may not be a very accurate metric to determine viability in robotics applications, as this evaluation uses random observation and estimation locations which is very unrealistic for a robot.

7 Conclusions and Future Work

Neural scene embeddings show promise for use in robotics applications. The embeddings have a very small memory footprint (only 512 values) and are capable of capturing geometric and visual aspects of a novel scene in ways that are useful for mapping and localization tasks. While the performance in these tasks stands to be improved, it was sufficient to show the power of neural scene embeddings.

There are still challenges to overcome before our methods could be applied to actual robots. Real environments would have much more complexity and variability than our generated scenes. It is possible that a model could take advantage of the increased visual complexity by inferring depth from textures and direction from lighting, but processing these complexities would be a much more difficult task. Additionally, collecting enough labeled data from real environments would be very expensive.

8 Code

Code for this project can be found here: `https://github.com/colin-r-schultz/CS230-Project`

References

[1] Sumikura, Shinya, Mikiya Shibuya, and Ken Sakurada. "OpenVSLAM: A Versatile Visual SLAM Framework." Proceedings of the 27th ACM International Conference on Multimedia. 2019.

[2] Li, Fu, et al. "Towards Visual SLAM with Memory Management for Large-Scale Environments." Pacific Rim Conference on Multimedia. Springer, Cham, 2017.

[3] Estler, Daniel. "Path Planning and Optimization on SLAM-Based Maps."

[4] Eslami, SM Ali, et al. "Neural scene representation and rendering." Science 360.6394 (2018): 1204-1210.

[5] M. Tatarchenko, A. Dosovitskiy, and T. Brox. "Octree generating networks: Efficient convolutional architectures for high-resolution 3D outputs." InProc. of the IEEE Interna-tional Conf. on Computer Vision (ICCV), 2017.

[6] PyBullet. `https://pybullet.org/`

[7] Joseph J. Lim, Hamed Pirsiavash, and Antonio Torralba. "Parsing IKEA Objects: Fine Pose Estimation." ICCV 2013.

[8] Stanford Pupper. `https://stanfordstudentrobotics.org/pupper`

[9] Mescheder, Lars, et al. "Occupancy networks: Learning 3d reconstruction in function space." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2019.

[10] matplotlib. `https://matplotlib.org`

[11] NumPy. `https://numpy.org`

[12] Tensorflow. `https://www.tensorflow.org`

[13] Sandler, Mark, et al. "Mobilenetv2: Inverted residuals and linear bottlenecks." Proceedings of the IEEE conference on computer vision and pattern recognition. 2018.

[14] Shivam Saboo. "Neural-Scene-Representation-and-Rendering." `https://github.com/shivamsaboo17/Neural-Scene-Representation-and-Rendering`

[15] Oliver Groth. "tf-gqn." `https://github.com/ogroth/tf-gqn`

[16] Liu, Shuo, et al. "ERN: edge loss reinforced semantic segmentation network for remote sensing images." Remote Sensing 10.9 (2018): 1339.