Modeling 3D Shapes by Reinforcement Learning Supplementary Material

Cheng Lin[1,2], Tingxiang Fan[1], Wenping Wang[1], and Matthias Nießner[2]

[1] The University of Hong Kong
[2] Technical University of Munich

1 Network Architecture

Fig. 1 and Fig. 2 show the detailed architecture of the Prim-Agent and the Mesh-Agent respectively. We also indicate the shape of the tensor output from each layer.

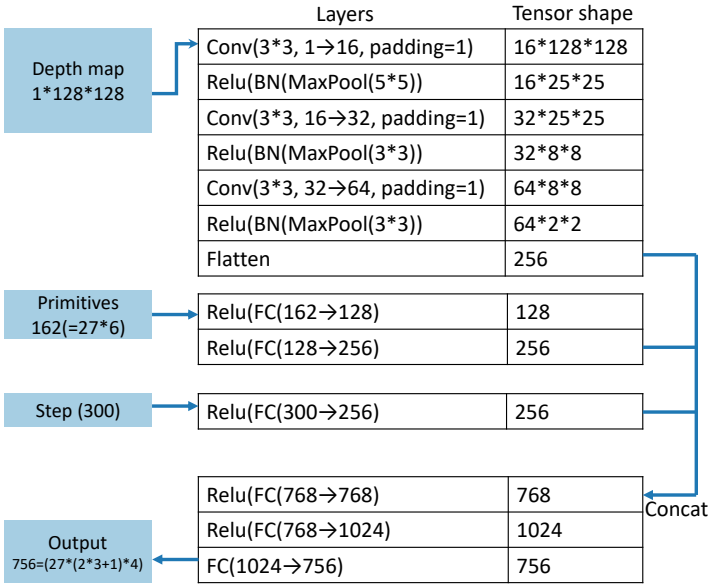

Fig. 1. The detailed network architecture of the Prim-Agent. BN: Batch Normalization Layer; FC: Fully Connected Layer.

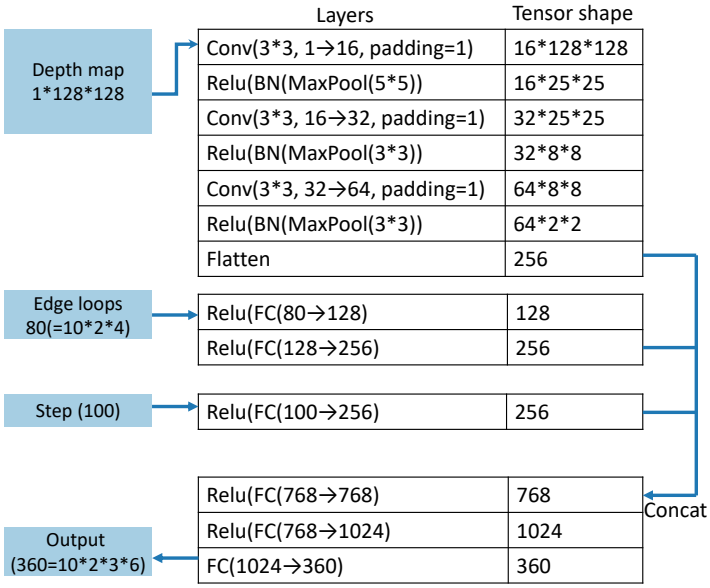

Fig. 2. The detailed network architecture of the Mesh-Agent. BN: Batch Normalization Layer; FC: Fully Connected Layer. The feature dimension of a loop point is 4, i.e., (x_l, y_l, z_l, a) where $a \in \{0, 1, 2\}$ additionally indicates the axis the loop plane is vertical to.

2 Illustration of the Choices of Method Design

2.1 Solution Space Reduction

We should note that it is not trivial for an RL agent to learn to model 3D shapes. The biggest challenge is that the action space has enormous modeling operations, and many of them are irrelevant. In the paper, there are in total 1116 different actions and the network will be unrolled for 400 steps, which leads to a huge solution space of 1116^{400}. Therefore, the exploration to find good policies will be extremely difficult. Here we summarize the key ideas to make this task feasible.

Divide the solution space Inspired by the hierarchical understanding of human modelers, we divide these operations into two categories, i.e., primitive-based operations and mesh-based operations, to reinforce more connections between different actions. Therefore, we propose two sperate agents, i.e., Prim-Agent and Mesh-Agent. The solution space is split down into 756^{300} and 360^{100} respectively for each step and the difficulty of learning is reduced as well.

Learn an initial policy As described in the paper, the agents are first trained to imitate the demonstrations generated by heuristics. Second, with the learned initial policy, the agents then learn in an RL paradigm by collecting the rewards. Since most of the actions in the huge solution space produce very poor

performance which is meaningless, the initial policy can significantly reduce the number of exploration of poor performance.

Restrict the actions in each step The strategies mentioned above can already help the agents learn reasonable policies, but the training efficiency is still fairly low. Also, we observe sometimes the agents are stuck at a state and output repetitive actions; therefore, at each step, we force the agents to edit a different primitive or loop from the last step.

To overcome these two issues, the strategy we adopt is that, at the k^{th} step, we force the agents to only choose the actions that can operate the i^{th} primitive (or loop), where $i = k \bmod m$ and m is the number of the primitives (or loops). The action space is further narrowed down in each step, and the agents will not be stuck at a repetitive action.

2.2 Local IoU Reward

The local IoU reward encourages the Prim-Agent to make each primitive cover more valid parts of a target shape, which will make the primitives overlap first. Therefore, deleting an overlapped primitive will gain high sparsity reward without losing much accuracy. Without the local IoU reward, since simplicity conflicts with accuracy, the agents cannot be motivated to balance the parsimony and the accuracy to give structurally meaningful and simple representations.

2.3 Double Replay Buffers for IL

If we only use one buffer, the expert's new demonstrations are mixed together with the old ones. This may lead to inadequate learning of the new experiences, given that the old and new data are sampled together but the old ones are sufficiently learned in previous iterations. Therefore, we propose to use two buffers: the short-term replay buffer $\mathcal{D}^{demo}_{short}$ is for learning the newest demonstrations, while the long-term one $\mathcal{D}^{demo}_{long}$ is for reviewing the histories. This is shown to be more effective.

3 Virtual Expert Algorithm

We give the detailed algorithm of the virtual expert for the Prim-Agent in Algorithm 1 with pseudo-code. We iteratively visit each primitive, test all the potential actions for the primitive and execute the one which can obtain the best reward. Note the selection of actions is divided into two stages: (1) during the first half of the process, we do not consider any delete operations but only edit the corners; (2) in the second half, deleting a primitive is allowed.

For the Mesh-Agent, we iteratively visit each edge loop, test all the potential actions, and execute the one which can obtain the best reward. Note there is only one stage for the "expert" of mesh editing.

Algorithm 1: Virtual Expert for Primitive-based Shape Abstraction

Input: m cuboid primitives $\mathcal{P} = \{P_1, P_2, ..., P_m\}$; target shape O; maximal step N_{max}
Output: a sequence of actions $\mathcal{A} = \{a_1, a_2, ..., a_N\}$
repeat
  for *each* $P_i \in P$ **do**
    $Step$++
    if $Step \leq 0.5 * N_{max}$ **then**
      find the action a which has the highest reward to tweak a cuboid corner
    else
      find the action a which has the highest reward to tweak a cuboid corner or delete a cuboid
    execute and output the action a
    update the state s
until $Step = N_{max}$;

4 Primitive Merging

Even though we have introduced a parsimony term in the reward function, the output of the Prim-Agent may still have some small or redundant primitives. We design a simple algorithm to merge these primitives as follows.

We define a graph G for the output primitives. In this graph, each node represents a primitive P_i. The merging of $P_i(V_i, V_i')$ and $P_j(V_j, V_j')$ will lead to a new primitive $P_{ij}(\min\{V_i, V_j\}, \max\{V_i', V_j'\})$. Two nodes P_i and P_j will be connected by an edge if $IoU(P_i \bigcup P_j, P_{ij}) \geq \epsilon$.

We compute the connected components for the graph G and then merge all the primitives in the same connected components into a single primitive. The merging process is performed for two iterations, while ϵ is set to 0.85 and 0.90 respectively in each iteration.

5 Edge Loop Assignment

Given M' primitives and N edge loops, we assign the edge loops onto the longest axis of each primitive while the loops are uniformly distributed in that direction. The number of loops $E(P_k)$ assigned to a primitive P_k is determined by

$$E(P_k) = \max\{\lceil N \frac{V(P_k)}{\sum_i V(P_i)} + 0.5 \rceil, 2\}, \tag{1}$$

where $V(P_i)$ is the volume for the primitive P_i and $i \in \{1, 2, ..., M'\}$. It can be seen that the number of loops assigned to a cuboid is proportional to its volume; thus a larger cuboid will be assigned more loops. Each cuboid is assigned at least two loops on the boundaries. When dealing with the last primitive $P_{M'}$, we directly assign all the remaining unallocated loops on to $P_{M'}$.