Cross-domain Deep Encoding for 3D Voxels and 2D Images

Jingwei Ji
Stanford University
jingweij@stanford.edu

Danyang Wang
Stanford University
danyangw@stanford.edu

1. Introduction

3D reconstruction is one of the hardest problems in computer vision. An ideal application is the generation of a 3D scene from one single image. However, the amount of information contained in one isolated image is not enough for reasonable reconstruction of scene with rich information, so the direct back-projection from 2D to 3D is a dead end. Instead, we need to extract the common semantic information implicitly contained in both 2D image and 3D shape, so that a machine could recognize an object regardless whether it is originally shown in a 2D or 3D form. The semantic information is in a highly abstracted form, i.e. an encoded feature. Such an encoded feature will be concise but representative for object in various forms. Therefore, the first step of a solid 3D reconstruction is to find a way to jointly encode both 2D and 3D forms of information.

Hand-designing a powerful joint encoder would be difficult. Fortunately, deep learning approach serves as a feasible way to accomplish this task. With the help of massive data for training, a proper designed deep neural network (DNN) architecture could be used for cross-domain encoding. The design principle is to establish two encoding network channels for 3D shapes and 2D images, respectively. Each channel is consist of units including convolutional, ReLU, pooling and fully connected layers. Both channels would output encoded features of the same size. To guarantee that the encoded feature is representative of object in different forms, the metric of the embedding space should be learned well. In other words, the embedding space should have a metric such that similar objects are clustered together, and different objects are separated.

The key of metric learning is the choice of loss function, used as a learning guidance. In this work, we experiment with the triplet loss function proposed and examined in [11], [6]. In each iteration, the triplet loss will be computed after the forward-propagation computation of encoded features, then the gradients of weights in each layer are computed in back-propagation to update the network parameters. After enough iterations of updating the network parameters, this deep encoder should be trained well enough to reasonably encode each input 3D shape and 2D image, and reach a low loss in the embedding space.

This work is a foundation step of a larger 3D reconstruction project in progress, thus the design of the encoding network could not be too complex due to the limitation of GPU memory. To design a smaller network instead of the pretrained heavy weighed ones, we decide to design and train from scratch. We experiment on several combination of layers and examine them on learning performance. Another challenge is the implementation of 3D layers. In our work, 3D shapes are presented in the form of voxels. Although deep learning library including Theano [9] and TensorFlow [1] have supported 3D voxel layers, some modifications are needed to fit them into our skeleton. Finally, we show the t-SNE embedding [10] as a visualization of the resulting embedding space.

2. Related Works

2.1. Deep learning on volumetric data

Deep learning has been a popular approach to solve computer vision problems since AlexNet [4]'s winning on image classification in 2012. Intensive studies have been carried out on object recognition, however mostly on 2D images. Recently deep learning approach has also been applied on volumetric modality of data to target on 3D reconstruction, such as [3]. We start from the skeleton codes from [3] and build up our own architecture design.

2.2. Triplet loss function

Metric learning is crucial in establishing the embedding space. A good embedding space should be general to different categories of objects, and also distinct enough for different objects. To well learn the metric, the loss function should be properly designed. We attempted the triplet loss used in [11][6] in this work. More details on the loss function would be shown in section 3.3.

3. Technical Details

3.1. Dataset and data processing

Our data are from the car category in ShapeNet [2]. In total, there are 7497 CAD models of different kinds of cars, including sedan, coupe, sports car, etc. 80% of them are used as training set, and 20% as testing set. During the training, the network will not use information from testing set for learning.

The 3D volumetric data are generated through voxelization on mesh data provided by ShapeNet. The voxelization tool is given by [5]. We generate $96 \times 96 \times 96$ voxel for all the CAD models in the car category. The voxel size is comparably larger than previous works on 3D voxel deep learning, which is for our future research on high resolution 3D reconstruction. In this work, we still target on dealing with data with lower resolution, so we set a pooling layer right behind the input layer.

The 2D images are rendered from ShapeNet CAD models of cars. The rendering images offer a direct labeling without further manual annotation. The size of images are 486×486. Similar as the voxels, we also use a pooling layer to lower the resolution. Instead of rendering the image from uniform distribution of viewpoints, we follow the distribution of real image statistics to make the rendered images more real. In detail, we count the viewpoints data from the manual annotations in a new dataset ObjectNet3D (will be released soon), then sample the viewpoints with a method of KDE [8]. The viewpoints distribution of the real images in ObjectNet3D is shown in Fig 2. To make the rendered images more diversified, we also add random color background as noise to avoid overfitting.

The triplet loss requires the data to be trained in a triplet way. Each 3D volumetric shape is paired with one positive image and one negative image, as shown in Fig 1. The positive image contains the same object as in the 3D voxel, while the negative image contains a different object. More details would be illustrated in section 3.3.

3.2. Deep encoding network architecture

The whole deep encoding network architecture is shown in Fig 3. The detail setting of parameters is illustrated in Table 1. In general, for both channels, we use two conv-relu-conv-relu-pool units followed by two fully connected layers. This architecture is chosen from the several candidates according to their learning performance, which is illustrated in section 4.1.

The implementation is completed on Theano, which is very supportive to 3D DNN layers. Still, we modified several layers ourselves to fit our skeleton codes.

Figure 1. The triplet grouped data. Every triplet contains a 3D voxel with positive and negative images. Green: positive relation; red: negative relation.

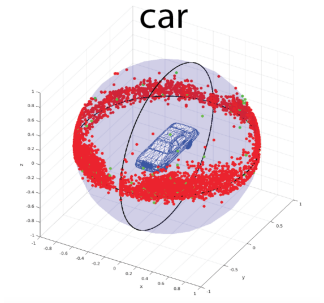

Figure 2. Viewpoints distribution of the real images of car in ObjectNet3D. The camera positions are visualized as points on the unit sphere (red: in-plane rotation $< 15^\circ$; green: in-plane rotation $\geqslant 15^\circ$). Clearly, most viewpoints have a small elevation angle, which is very different from a uniform elevation distribution.

3.3. Loss function

After the deep encoding, the encoded features of voxel, positive and negative images are used to compute the triplet loss. Triplet loss is trained on the triplet data p_i, v_i, n_i, denoting the i-th positive image, voxel and negative image. In the triplet data, p_i and v_i have the same label, presenting the same object, while n_i and v_i are different. The loss function is as following:

$$L = \frac{1}{m} \sum_{i}^{m} \max(D^2_{pi,vi} - D^2_{ni,vi} + \alpha, 0), \qquad (1)$$

where $D_{x,y} = ||f(x) - f(y)||$ is the Euclidean distance between two encoded features, and α is the margin parameter, m is the size of mini-batch. During the training, the positive pair would be pulled closer and negative pair pushed away from each other in the embedding space. Ideally, the triplet

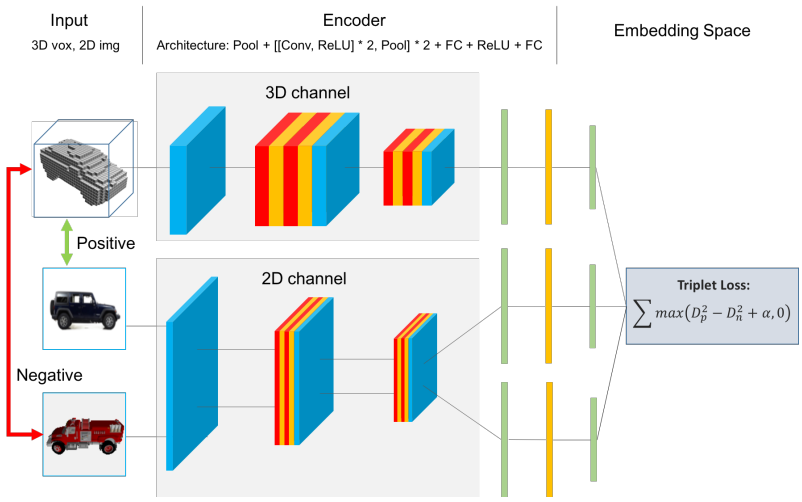

Figure 3. The sketch of the cross-domain deep encoding architecture. Red: convolutional layer; yellow: leaky ReLU layer; blue: pooling layer.

loss would help form many clusters in this way.

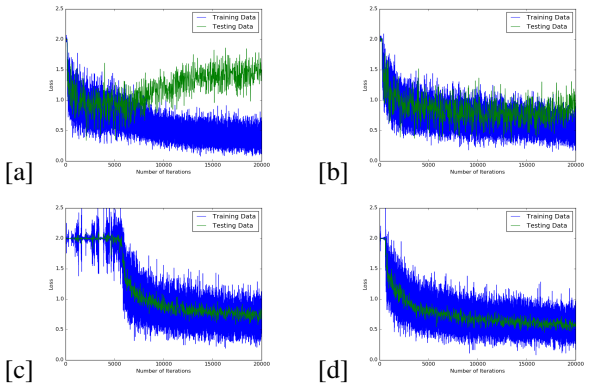

Figure 4. The learning curve for (a) 1-conv-1-fc, (b) 2-conv-1-fc, (c) 3-conv-1-fc, (d) 2-conv-2-fc. We use MSRA filler for weights initialization, Adam update policy with default learning rates, margin $\alpha = 1$. Test loss is calculated on one mini-batch from testing dataset portion every 20 iterations.

4. Experiments and Results

In the experiments, we first observe the learning behaviors of several network architecture candidates. Then we select the best network out of them and trained on the full training dataset (80% portion), and visualize the final encoded feature space using t-SNE embedding.

4.1. Learning behavior of different network

We build up the networks from scratch, so we experiment with the architecture candidates in an add-on style. Denote a network with m conv-relu-conv-relu-pool units followed by n fc layers by m-conv-n-fc, we have built four candidates: 1-conv-1-fc, 2-conv-1-fc, 3-conv-1-fc, 2-conv-2-fc. The learning curve of the first 20000 iterations are shown in Fig 4. The training dataset has around 6000 CAD models, and we fetch a mini-batch of 24 triplet into the network, so every 250 iterations is an epoch. From the learning curves, we can see that the shallowest 1-conv-1-fc would cause overfitting easily after 7000 iterations. In 2-conv-1-fc, overfitting begins around 20000 iterations. As for 3-conv-1-fc, the network is not easy to train, so in the first 5000 iterations the loss is not decreasing. Sometimes we need more 20000 iterations just for the loss to begin lowering down. The 2-conv-2-fc is better than the other three, due to its more weights in fc layers and shallower convolutional structure than 3-conv-1-fc. So we use 2-conv-2-fc as our final choice. The details of this architecture is shown in Table 1.

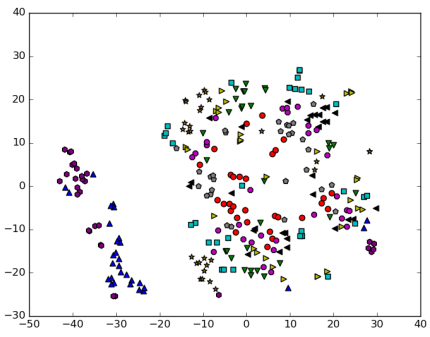

Figure 5. The t-SNE embedding of 300 testing images of 10 objects. Each marker denotes one object.

channel	label	size-in	size-out	memory	params	weights
voxel	pool0	$96 \times 96 \times 96 \times 2$	$32 \times 32 \times 32 \times 2$	256 KB	$p = 3$	0
	conv1a	$32 \times 32 \times 32 \times 2$	$32 \times 32 \times 32 \times 16$	2MB	$3 \times 3 \times 3 \times 16$	1.7KB
	rect1a	$32 \times 32 \times 32 \times 16$	$32 \times 32 \times 32 \times 16$	2MB		0
	conv1b	$32 \times 32 \times 32 \times 16$	$32 \times 32 \times 32 \times 32$	2MB	$3 \times 3 \times 3 \times 32$	3.4KB
	rect1b	$32 \times 32 \times 32 \times 32$	$32 \times 32 \times 32 \times 32$	4MB		0
	pool1	$32 \times 32 \times 32 \times 32$	$16 \times 16 \times 16 \times 32$	512KB	$p = 2$	0
	conv2a	$16 \times 16 \times 16 \times 32$	$16 \times 16 \times 16 \times 64$	1MB	$3 \times 3 \times 3 \times 64$	6.8KB
	rect2a	$16 \times 16 \times 16 \times 64$	$16 \times 16 \times 16 \times 64$	1MB		0
	conv2b	$16 \times 16 \times 16 \times 64$	$16 \times 16 \times 16 \times 128$	2MB	$3 \times 3 \times 3 \times 128$	13.5KB
	rect2b	$16 \times 16 \times 16 \times 128$	$16 \times 16 \times 16 \times 128$	2MB		0
	pool2	$16 \times 16 \times 16 \times 128$	$4 \times 4 \times 4 \times 128$	32KB	$p = 4$	0
	flat2	$4 \times 4 \times 4 \times 128$	$4 * 4 * 4 * 128 = 8192$	32KB		0
	fc3	8192	4096	16KB	8192×4096	128MB
	rect3	4096	4096	16KB		0
	fc4	4096	128	0.5KB	4096×128	2MB
image	pool0	$486 \times 486 \times 3$	$162 \times 162 \times 3$	307.5KB	$p = 3$	0
	conv1a	$162 \times 162 \times 3$	$162 \times 162 \times 16$	1.6MB	$3 \times 3 \times 16$	576B
	rect1a	$162 \times 162 \times 16$	$162 \times 162 \times 16$	1.6MB		0
	conv1b	$162 \times 162 \times 16$	$162 \times 162 \times 32$	3.2MB	$3 \times 3 \times 32$	1.1KB
	rect1b	$162 \times 162 \times 32$	$162 \times 162 \times 32$	3.2MB		0
	pool1	$162 \times 162 \times 32$	$54 \times 54 \times 32$	364.5KB	$p = 3$	0
	conv2a	$54 \times 54 \times 32$	$54 \times 54 \times 64$	729KB	$3 \times 3 \times 64$	2.3KB
	rect2a	$54 \times 54 \times 64$	$54 \times 54 \times 64$	729KB		0
	conv2b	$54 \times 54 \times 64$	$54 \times 54 \times 128$	1.4MB	$3 \times 3 \times 128$	4.5KB
	rect2b	$54 \times 54 \times 128$	$54 \times 54 \times 128$	1.4MB		0
	pool2	$54 \times 54 \times 128$	$6 \times 6 \times 128$	18KB	$p = 9$	0
	flat2	$6 \times 6 \times 128$	$6 * 6 * 128 = 4608$	18KB		0
	fc3	4608	4096	16KB	4608×4096	72MB
	rect3	4096	4096	16KB		0
	fc4	4096	128	0.5KB	4096×128	2MB
total				48MB		204MB

Table 1. Details of the architecture of the cross-domain deep encoder. p is the pooling parameter. All sliding windows in conv layers use stride of 1. Flat layer flattens an n-dimension array into a vector. We use leaky ReLU for the rectified layer. The number type is float 32, taking 4 bytes each. The output memory and for images are doubled in the calculation of total amount, since there are both positive and negative images paired with each voxel.

4.2. Embedding space visualization

To validate the quality of our embedding space, we visualize the space using t-SNE [10] method. The technique is for dimensionality reduction, and has been widely used for approximate visualization of high dimensional space. The result of the images embedding is shown in Fig 5. However, the voxels are not embedded well in this space. They are embedded quite far away from images clusters. We believe that one reason is the use of triplet loss. Our setting of triplet takes into consideration the difference between images, but no difference between voxels. In other words, the setting for voxel and image is not symmetric. One way to solve this is to use a recently proposed lifted structure loss function [7]. This loss function would consider the relation between every two datums in one mini-batch, so the similarity of voxels would also be take into account. The implementation of lifted structure embedding space would be finished in the future work.

5. Conclusion

In this work, we proposed a cross-domain deep encoding approach to bridge 3D voxel and 2D image data. We experiment on several DNN architecture to establish the deep encoder, and tested them on learning behaviours and embedding space visualization. We have also argued that lifted structure loss could improve the joint encoding quality.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] A. X. Chang, T. A. Funkhouser, L. J. Guibas, P. Hanrahan, Q. Huang, Z. Li, S. Savarese, M. Savva, S. Song, H. Su, J. Xiao, L. Yi, and F. Yu. Shapenet: An information-rich 3d model repository. *CoRR*, abs/1512.03012, 2015.

[3] C. B. Choy, D. Xu, J. Gwak, K. Chen, and S. Savarese. 3d-r2n2: A unified approach for single and multi-view 3d object reconstruction. *CoRR*, abs/1604.00449, 2016.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[5] F. S. Nooruddin and G. Turk. Simplification and repair of polygonal models using volumetric techniques. *Visualization and Computer Graphics, IEEE Transactions on*, 9(2):191–205, 2003.

[6] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[7] H. O. Song, Y. Xiang, S. Jegelka, and S. Savarese. Deep metric learning via lifted structured feature embedding. *arXiv preprint arXiv:1511.06452*, 2015.

[8] H. Su, C. R. Qi, Y. Li, and L. J. Guibas. Render for cnn: Viewpoint estimation in images using cnns trained with rendered 3d model views. In *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[9] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[10] L. Van Der Maaten. Accelerating t-sne using tree-based algorithms. *The Journal of Machine Learning Research*, 15(1):3221–3245, 2014.

[11] K. Q. Weinberger and L. K. Saul. Distance metric learning for large margin nearest neighbor classification. *The Journal of Machine Learning Research*, 10:207–244, 2009.