3D shape classification

Riccardo Lincetto

In the following I present the project done for the 3D Augmented Reality course, explaining what I did in the "*3D-shape-classification.ipynb*" Jupyter notebook containing the relative code. All the material that will be discussed in this report is available in the following GitHub repository:

https://github.com/RicLincio/3D-shape-classification

Introduction

The notebook uses the 3DShapeNets restricted dataset to perform volumetric classification with a 3D convolutional neural network. The purpose is trying to keep the number of parameters as low as possible, while achieving a good accuracy on classification task.

Background

With the recent spreading of inexpensive 2.5D depth sensors (or RGBD sensors) it has now become crucial being able to handle volumetric shapes. To this end, 3DShapeNets (http://3dshapenets.cs.princeton.edu) proposes to represent 3D objects as probability distributions over a 3D voxel grid. Starting then from mesh data (as Object File Format (OFF) or similar formats), 3D voxel grids can be created by setting to 1 the voxels belonging to the mesh and to 0 the remaining voxels. When dealing with a 2D depth-map obtained from an RGBD sensor instead, usually there isn't enough information to completely reconstruct the scene because points in occluded space (e.g. behind the surfaces captured from the sensor) might either belong to the shape or not. To gather from the scene more information it is possible to use multiple sensors, but even with infinite of them the problem is not completely solved: it is always possible to think of an object that cannot be defined entirely because of occluded spaces, e.g. these sensors cannot see whether inside a closed box there's a cat. This approach however enables also conversion of 2D depth-maps to 3D voxel grids by setting "uncertain" points to a value between 0 and 1, representing the probability of each voxel being part of the object.

3DShapeNets provides a dataset of Object File Format samples and some code to perform a classification task on them. The dataset contains 40 different types of objects, but there is also a restricted version with only 10 different classes. The meshes of the objects in those datasets are completely defined: in their volumetric representations, obtained voxels are only set to either 0 or 1, but the provided model is also able to classify data acquired from RGBD sensors, where missing voxels are handled as previously described. Among the code provided by the authors, there are a MATLAB function already converting meshes into 3D voxel data ('3DShapeNets/util/write_input_data.m'), with different view angles, and its output (in '3DShapeNets/volumetric_data/'), which constitutes the input that the model should be fed with. In my notebook, that volumetric data will be used to perform classification directly, instead of recurring to the MATLAB function to create again the dataset. The model that is used in 3DShapeNets is a Convolutional Deep Belief Network, a network with 5 hidden layers achieving 83.5% accuracy on the restricted dataset ('ModelNet10') and 77% on the integral one ('ModelNet40').

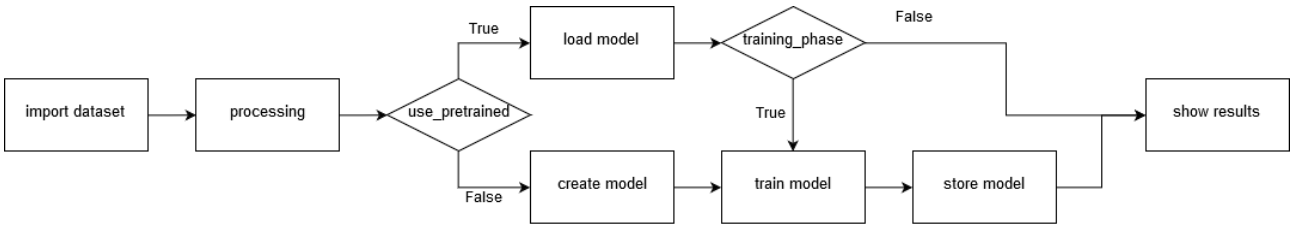
Code

Notebook setup

Here is a list of parameters that can be modified to execute properly the code:

- **path**: string containing the location where data is stored. Once the source code has been downloaded from http://3dshapenets.cs.princeton.edu and extracted to a local folder, set 'path' to the directory containing the '3DShapeNets' subfolder. Note that it doesn't need to be inside the directory where this notebook is stored because an 'absolute' path is used, not a relative one. Note: do not include the final slash and avoid using backslashes (Windows format);
- **classes**: set of classes to be used for classification. Select one of the two configurations, with 10 or 40 classes respectively, or test the code on new configurations;
- **use_pretrained**: boolean flag to indicate whether the use of a pre-trained model is preferred over the training of a new one from the beginning. Set it to 'True' to use a pre-trained model or to 'False' to train a new model.
- **model_in_name**: string representing the name of the model to be loaded, when 'use_pretrained' is set to True. The model must be stored inside "./models/";
- **training_phase**: boolean flag indicating whether there should be a training phase. This variable is used because there can be training phases both for new models and for pre-trained ones, thanks to the versatility of Keras framework. If 'use_pretrained' is set to False, then 'training_phase' is automatically set to True because a new model is required;
- **model_out_name**: string representing the name of the model to be stored. After the training phase, the model is saved in "./models/" with the name here specified: this happens independently from the value of 'use_pretrained', in fact it depends only on 'training_phase' flag.

According to the values of these parameters, the flow of the notebook is different and is represented in the following figure.



It is possible to see that there are only three possible pipelines, selected by the values of the two boolean flags: 'use_pretrained' and 'training_phase'. Since the combinations of their values is four, there's one configuration never occurring: when 'use_pretrained' is set to False in fact, the notebook takes care of automatically setting to true 'training_phase' because it doesn't make sense to run the code creating a model and without training it. Let's go now through the various stages displayed in figure.

Dataset

'3DShapeNets/volumetric_data/' folder contains 40 subfolders, each named with the corresponding class label, which are:

```
classes = ['bathtub', 'bed', 'chair', 'desk', 'dresser', 'monitor', 'night_stand', 'sofa', 'table', 'toilet',
           'airplane', 'bench', 'bookshelf', 'bottle', 'bowl', 'car', 'cone', 'cup', 'curtain', 'door',
           'flower_pot', 'glass_box', 'guitar', 'keyboard', 'lamp', 'laptop', 'mantel', 'person', 'piano', 'plant',
           'radio', 'range_hood', 'sink', 'stairs', 'stool', 'tent', 'tv_stand', 'vase', 'wardrobe', 'xbox']
```

The restricted version of the dataset contains only instances of the first ten classes. In my notebook it is possible to switch between the complete and the restricted version when setting the 'classes' parameter, as previously described, or using different configurations which should be inserted by the user.

Inside these subfolders, training and test sets are stored in different directories:

- Training set: ./30/train/
- Test set: ./30/test/

The number of instances inside each set is variable. All objects have shape 24 x 24 x 24 but, since 3 voxels of extra padding are already added in each dimension, they result having 30 x 30 x 30 voxels, as indicated by the directory where they are stored ('30').

Data is loaded into two dictionaries, 'trainSet' and 'testSet', where the keys are the class names and the values are variable length 4D numpy arrays, where the variable dimension is the first (representing the number of instances) and the remaining ones are fixed to 30: the size of each of them is then [number_of_instances, 30, 30, 30]. This is done using the function 'load_mat', which can be found inside 'utils.py' in the repository.

Processing

Since the notebook directly imports volumetric data, the only processing required is transforming the dictionary into a suitable format. Since the CNN model will be defined with Keras and the convolution will be 3-dimensional, the input for the model must be a 5D numpy array where:

- The first dimension is the number of instances in the training or test set;
- The second, third and fourth dimensions are equal to 30;
- The fifth dimension is the number of channels for each voxel, which in this case is 1, since we are dealing with black and white shapes.

This is done for both training and test set by first creating a zero-array with the right shape and then stacking along the first dimension all the instances of all the classes in the dictionary. Since this procedure requires knowing the indexes of the portion of array on which the data is copied, together with these arrays it is possible to create also 1D vectors containing the correspondent labels, for both training and test sets. This processing results in creating then 4 vectors, 'X_train', 'Y_train', 'X_test', 'Y_test', where instances belonging to the same class are all grouped together and classes are imported sequentially, thus maintaining their order. Nonetheless, a data shuffling is not required in this case because Keras will take care of it.

Model

The model from 3DShapeNets is a 5-layer Convolutional Deep Belief Network, where the 3 convolutional layers have respectively 48, 160, 512 filters and the two fully connected layers have 1200 and 4000 units. The training of this network requires a lot of computational power and for this reason I decided to train a network

with a lower number of parameters, trying to keep an acceptable accuracy (around 90% for the 10-class classification). Network layers are reported in the following table.

Layer type	Output shape	Number of parameters
Conv3D	(None, 12, 12, 12, 20)	6880
LeakyReLU	(None, 12, 12, 12, 20)	0
Conv3D	(None, 6, 6, 6, 50)	8050
LeakyReLU	(None, 6, 6, 6, 50)	0
MaxPooling3	(None, 3, 3, 3, 50)	0
Flatten	(None, 1350)	0
Dropout	(None, 1350)	0
Dense	(None, 10)	13510
Activation	(None, 10)	0
Total:		28440

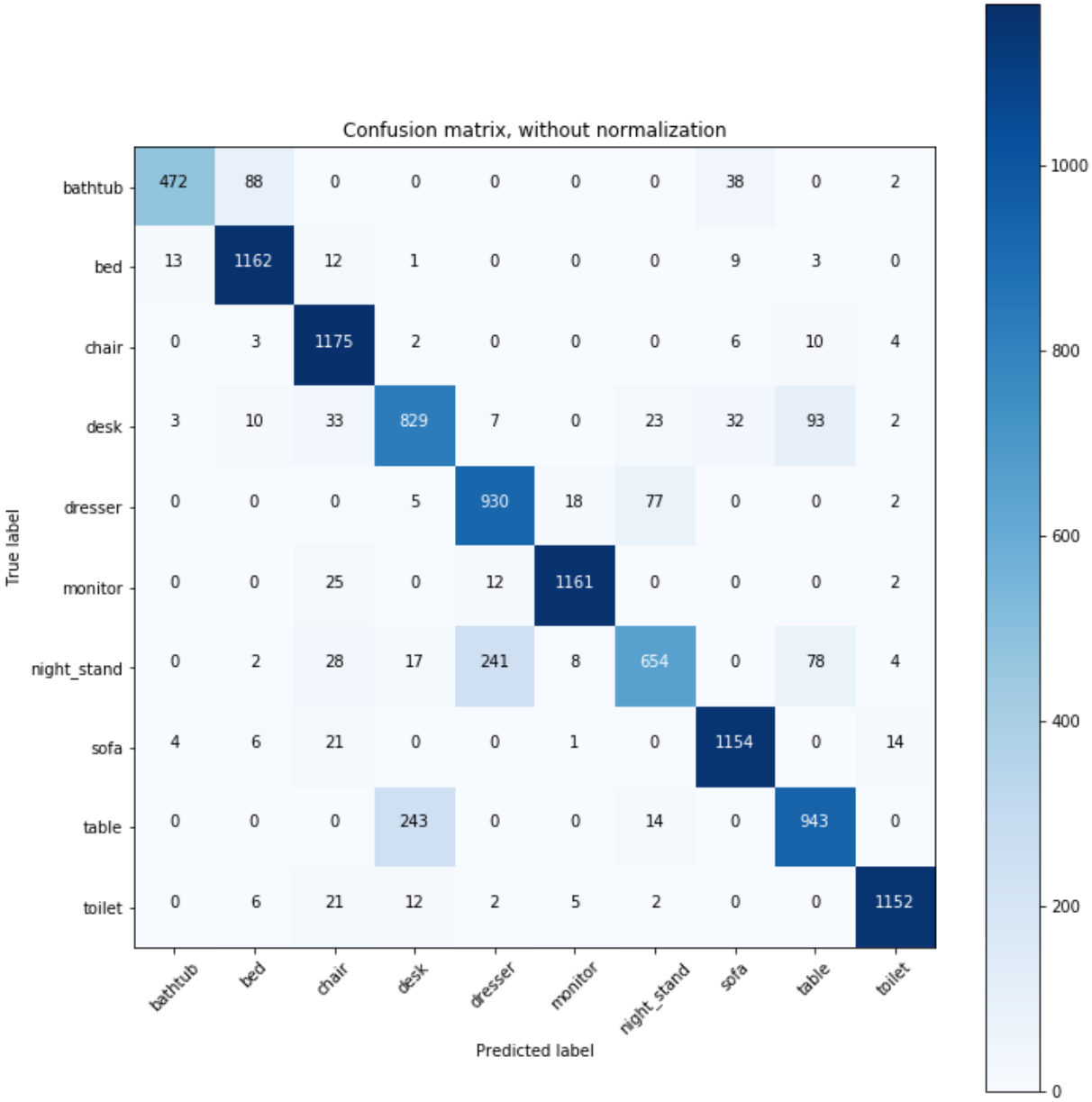
This model has only two convolutional layers and one fully connected layer, resulting in a network with only 28440 trainable parameters. Training this network will be much faster than training the one proposed in 3DShapeNets project. It needs to be noticed though that the model is built according to the number of classes involved: the number of filters for the two layers are 'n_classes * 2' and 'n_classes * 5' and the fully connected layer has 'n_classes' units, where n_classes is the number of classes. The network described in the above table is then referred to a 10-class classification.

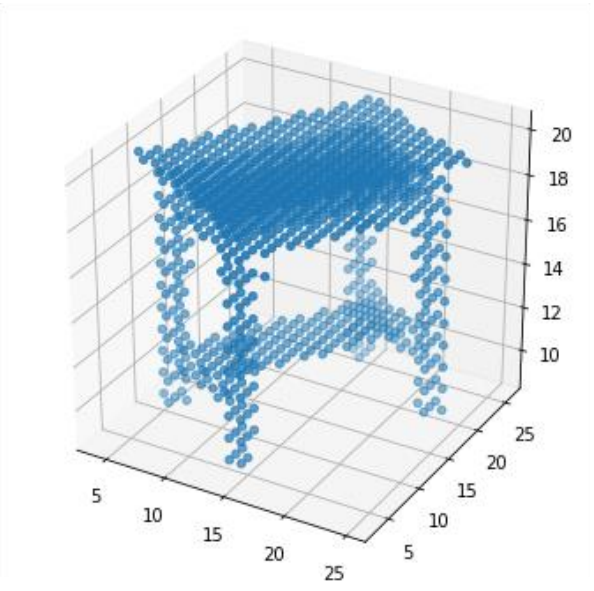
The parameters used for training phase, chosen after several trials, are:

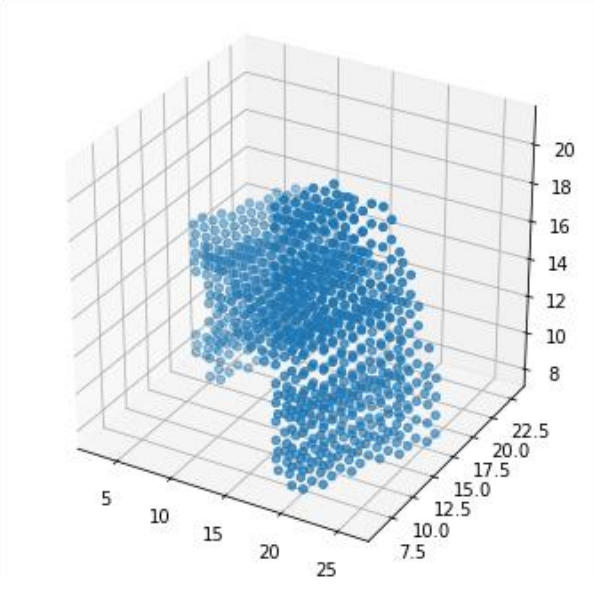
- Batch size: 128
- Learning rate: 0.001
- Learning rate decay: 0.0001
- Epochs: 20

Results

Executing several times the notebook with the restricted dataset, i.e. 10 classes, the trained model results in having an average accuracy on the test set of 88.5%, with peaks during the training phase of 90%. It could be useful then to use a CheckPointer object from Keras to store the best model. Plotting the confusion matrix, shown in the following page, it can be seen that the most common source of errors is confusing tables with desks or night stands with dressers (and vice versa). This seems justifiable by the fact that these couples of classes usually differ by their sizes, which aren't accounted for in this dataset: objects are in fact stored within the same number of voxels with a normalized scale, thus neglecting the information on the original size of the object. This is immediately noticed when looking at the two examples shown in the next page: on the left a sample from the class 'table' is plotted, while on the right there's a 'desk'.


Confusion matrix, without normalization
bathtub 472 88 0 0 0 0 0 38 0 2
bed 13 1162 12 1 0 0 0 9 3 0
chair 0 3 1175 2 0 0 0 6 10 4
desk 3 10 33 829 7 0 23 32 93 2
dresser 0 0 0 5 930 18 77 0 0 2
monitor 0 0 25 0 12 1161 0 0 0 2
night_stand 0 2 28 17 241 8 654 0 78 4
sofa 4 6 21 0 0 1 0 1154 0 14
table 0 0 0 243 0 0 14 0 943 0
toilet 0 6 21 12 2 5 2 0 0 1152
bathtub bed chair desk dresser monitor night_stand sofa table toilet
True label
Predicted label
1000 800 600 400 200 0


20 18 16 14 12 10
5 10 15 20 25
5 10 15 20 25


20 18 16 14 12 10 8
5 10 15 20 25
7.5 10.0 12.5 15.0 17.5 20.0 22.5

Comments

This simple network seems to perform better than the model proposed in 3DShapeNets project, at least on the restricted dataset, with a test accuracy of 88.5% against 83.5%. A possible explanation for this result is that a larger network, as the one with 5 layers, might overfit the data, losing its generalization capabilities.

Despite the possibility of choosing the complete dataset in the notebook, running the code on it could be unsuccessful because of memory constraints. My code in fact was conceived to run on the restricted version and thus the data-loading and processing part is not efficient for a that large amount of data. A better version could import the data already in a 5D array instead of storing it first in a dictionary: this would spare more free memory, that might be used to keep in the main memory the entire final structure. Alternatives to this approach could also prefer to leave the data in the mass storage, instead of loading it into the main memory, and use a batching function to pass only the required data to the processing unit.

As future work it would be also interesting to perform a classification task on incomplete objects, as if they were captured by RGBD sensors: this could shed light on which type of data is preferable to train the model. My project in fact assumes that all volumes are entirely defined both in training and test sets, but a desirable property of the model is its capability of recognizing also uncomplete shapes. Using then the same model to recognize both complete and incomplete objects would be very convenient, instead of training two different models on the two types of data.