Department of Computer Science, University of Leicester

CO7201 Individual Project
Object Identification Within an Industrial Plant Setting Using Deep Learning

Final Report

By

Jian Shi (js825@student.le.ac.uk)

12th Jan 2018

University Id: 159046193

Project Supervisor	Raman Rajeev (rr29@leicester.ac.uk)
Second Marker	Fer-Jan De Vries (fdv1@leicester.ac.uk)
Industrial Supervisor	Andrew Pike (Andrew.pike1@ge.com)

DECLARATION

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s). Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s). I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name: Jian Shi

Date: 12th Jan 2018

Abstract

In this work, we propose a 3D point cloud geometry data recognition system using Convolutional Neural Networks, especially for dealing with the limited data circumstances. For fulfilling this task, we dived into the theories and explored the prior works of Convolutional Neural Networks. Then did a series of data processing on point clouds for a decent representation to feed the neural network. This work is divided into two steps, object classification that identifies single object and object localization that localizes objects in a scene. Due to the limit number of data, we introduced some work arounds to mitigate the impact of the small datasets. We used data augmentation to generate more data, transfer learning to transfer knowledges among neural networks and clustering algorithms to avoid training another neural network for segmentation. This application shows a robust performance on object classification while the object localization is not stable enough for production. We believe using a training algorithm for object localization would lead to a better performance in future works.

Table of Contents

Table of Figures

1 Introduction

This project will explore the potential of deep learning methods for the automatic identification of objects and their locations within industrial plant digitized scans. This is needed so that objects within the point cloud are correctly translated into a CAD library object (e.g. pipe, valve, turbine, etc.) with specific dimensions and a relative location to other objects in the scene. Subsequently engineers can create customized plant retrofit solutions like to replace the pipework, valve arrangement or turbomachinery. Manually processing point cloud geometry data is not straight-forward. Not as intuitive as 2D images, visualizing point cloud data mostly requires professional software like Cyclone, Computer-Aided Design (CAD), etc. Users need to put more effort on training themselves in order to use those professional applications properly. Even though, traversing through a point cloud to locate a group of points is still difficult and time-consuming. If this automated localization could be done reliably then the manual engineering effort could be greatly reduced, yielding significant cost savings. We aim to develop an algorithm model achieving object detection and localization within a point cloud, which can be easily extended when new geometries come in.

In this thesis, we will dive into the topic of 3-dimensional (3D) object recognition from both theoretical and practical perspectives. The main objective is to implement an application to recognize objects within a 3D scene. We aim to use this application to recognize the objects from industrial power plants. This

application is based on the study of Convolutional Neural Networks (CNN), volumetric representation of 3D point cloud data and cluster algorithms.

In the context of computer vision, object classification and localization are the two primary requirements for a machine to understand a complex environment. The former would focus on recognizing the class label of any single input object while the latter would detect and localize those objects within a scene. With the development of autonomous robots like self-driving cars, traditional 2D approaches were superseded by 3D ones [1]. Sensors such as LiDAR and RGBD cameras are widely equipped on modern robot, which can provide a rich source of 3D information [2]. Those sensors can produce Point Cloud data and RGBD data respectively. Furthermore, point cloud data is a vector of X, Y, Z coordinates that measures the external surface of an object [3] while RGBD data is basically a 3D pixel array like a stack of images [4]. The industrial power plant data we would use is the point cloud data generated by multiple LiDAR cameras.

The main challenge of this project is to set up a 3D CNN model for point cloud data and extend this model for a general usage. In the fields of Virtual Reality and Augmented Reality, 2D CNN can provide reliable results for object recognition and detection [5]. While there are less researches on processing 3D point cloud geometry data since it is irregular, not grid-aligned, and in places very sparse structure, with strongly varying point density [6]. Due to the additional dimension, more various features would be detected for each geometry that could make it harder to distinguish from each other. Compare to

2D CNN, learning those massive feature patterns would require more training data, along with more computation power.

Another challenge of the project is to improve the performance of the network. Neural network is a data-oriented programming technique, so the performance of neural network would be case-by-cased. Using suitable hyperparameter values and proper techniques like normalization, regularization could result in a better performance. For neural networks, we want to feed as much data as possible since the performance will normally be better as the amount of data goes up. However, in real cases, we may either not be able to obtain a large dataset or not be allowed to spend massive time on training. Hence, we will trade off among various methodologies regarding the efficiency and performance.

In this work, we propose a 3D point cloud geometry data recognition system using Convolutional Neural Networks, especially for dealing with the limited data circumstances. This thesis is structured as follows, section 2 reviews the architecture of Convolutional Neural Networks. Section 3 illustrates the context of this work from a technical perspective. Under the configured environment, the point cloud data is processed and fed into the neural networks in section 4 and 5. In section 0, some relevant experiments are carried out along with the corresponding results and discussion. At last, section 7 draws some conclusions and classifies some future works for improvement.

2 Convolutional Neural Networks

Deep learning has been revived recently due to the large dataset and the big leap of computation power [7]. It has been widely used in the area of computer vision and speech recognition. Deep Neural Network (DNN) is an architecture of deep learning, which consists of multiple hidden layers between the input and output layers [8]. The characteristics of Neural Network are important to intelligent applications and robotics, such as parallel processing ability, fault tolerance ability and learning ability [9]. Convolutional Neural Network (CNN) is a type of deep neural network which is strong at pattern recognition, as well as feature extraction, hence it is mainly used to deal with computer vision problems, like image recognition, object detection and localization [10]. CNN is now the state-of-art methodology for image recognition.

CNN is a type of Artificial Neural Networks (ANN) which could mimic how human brain works by connecting massive neurons together. There are two types of structures of ANN, feed forward and feedback networks [9]. Feed forward neural network computes the output linearly by transmitting the signal towards the end. The latter is also known as recurrent neural network that allows signal travelling on both directions, which means every neuron could refer to the computed values from previous neurons. CNN is a feed forward neural network.

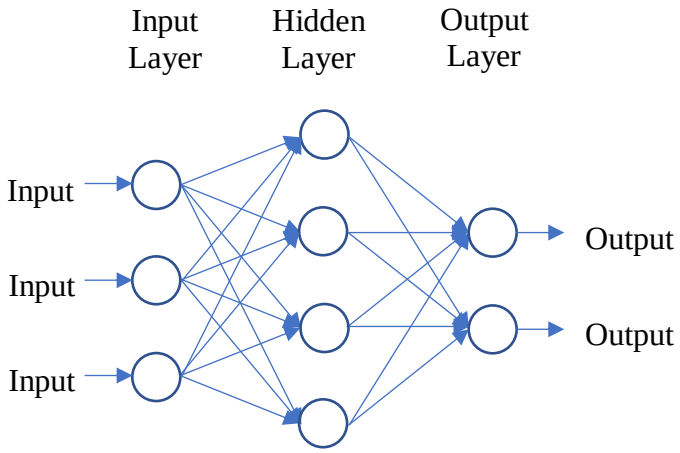


Figure 2.1A shallow feed forward neural network that contains only one hidden layer

Figure 2.1 shows a three-layer neural network contains one hidden layer which receives inputs from 3 input neurons and gone through the 4-neuron hidden layer to compute the output for 2 output neurons. Neural networks are based on a number of trainable neurons while those neurons are distributed in different layers.

2.1 CNN Architecture

CNN normally consists multiple hidden layers on where the convolutional computations happened. Convolution operations are the major operations for CNN. Apart from that, there are many other operations would be completed in a CNN model. This section introduces the major operations in CNN and demonstrates how does a CNN model handle 2D images and how to extend it for processing 3D point clouds.

2.1.1 Convolution Layer

Convolution operation is the core concept in CNN. It would happen on every convolution layer which will be mainly used for pattern recognition, also can help with noise reduction [11]. Take an example of convolving on 2D images, a convolutional kernel would be overlapped on the image while the convolutional

kernel can be an any shaped matrix. Then the kernel will slide over the whole image producing an element-wise product on each stay. Meanwhile, an assumption is made here that if a feature can be detected at position (x_1, y_1), it also could appear at a different position (x_2, y_2). This technique is call shared weights [12]. Notably, applying this can massively reduce the computable parameters by reusing the kernel weights of each neuron. The matrix of all products represents the result of this convolutional operation. As demonstrated in Figure 2.2 [13], a 2 x 2 convolutional kernel is used to convolve through a 3 x 4 image. It turns out a 2 x 3 output matrix which is called a feature map generated by the convolutional kernel. In CNN, we normally use rectified feature map rather than this input feature map which will be explained in section 2.1.2.

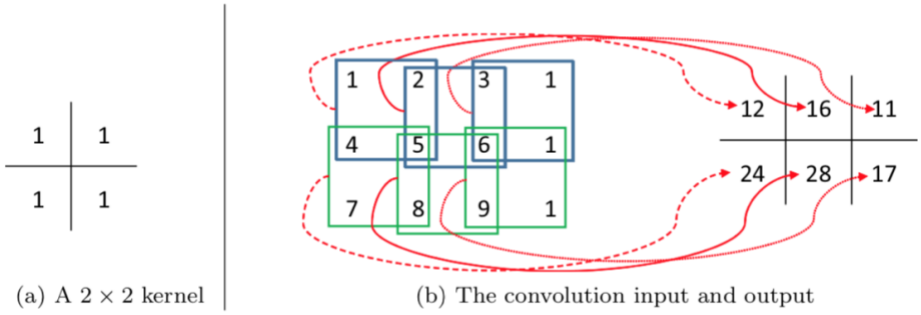


Figure 2.2Convolution process with 2 x 2 kernel, stride 1 x 1, no padding

The convolutional operation can be controlled by stride and padding. Stride is the number of pixels would be skipped while sliding the kernel filter window through the image. The stride size we were using in Figure 2.2 is 1 x 1 which means to skip 1 pixel horizontally and vertically while sliding. Padding is an operation that would add extra boarders to the image before convolution. CNN normally uses zero-padding to add extra boarders with all zeros to the image.

After we decided which stride and padding to use, we can compute the size of output feature map by using this equation:

$$n_{out} = \left\lfloor \frac{n_{in} + 2p - k}{s} \right\rfloor + 1$$

where the n_{in} and n_{out} are the width or height of the input and output, *k* is the size of convolution kernel, *p* is the padding size and *s* is the stride.

There are two padding strategies defined in Tensorflow framework (section 3.1), same padding and valid padding. With same padding, it would guarantee that the size of output feature map is as same as input if the stride is set to 1. The size of the output feature map with same padding would be:

$$n_{out} = \left\lceil \frac{n_{in}}{s} \right\rceil$$

where the n_{in} and n_{out} are the width or height of the input and output, *s* is the stride. Same padding would try to evenly pad on both side, which means top and bottom, left and right. If the total padding pixels is an odd number, the bottom and right would pad with an extra column. Here is the algorithm used in Tensorflow:

$$hw = \lceil (n_{out} - 1) * s + k \rceil$$

$$tr = \left\lfloor \frac{hw}{2} \right\rfloor$$

$$bl = hw - tr$$

where the *hw* is the total padding along height or width, the *tr, bl* are the padding size on top or right, bottom or left respectively, the n_{out} is the width or height of the output, *k* is the size of convolution kernel and *s* is the stride.

While valid padding basically means no padding in Tensorflow. It would ignore left-most and bottom-most columns if the convolution kernel does not fit. When

stride is 1, the size of output feature map would be shrunk by k – 1. The equation is:

$$n_{out} = \left\lfloor \frac{n_{in} - k + 1}{s} \right\rfloor$$

where the n_{in} and n_{out} are the width or height of the input and output, *k* is the size of convolution kernel and *s* is the stride. In CNN, multiple convolution kernels will be applied on an image at each convolutional layer which could extract various features. For example, kernel (a) could be used for vertical edge detection while kernel (b) could be used for horizontal edge detection.

$$\begin{pmatrix} 1 & -1 \\ 1 & -1 \end{pmatrix} \qquad \begin{pmatrix} 1 & 1 \\ -1 & -1 \end{pmatrix}$$

Kernel (a) Kernel (b)

An example of vertical and horizontal edge detection is provided in Figure 2.3 [13].

Figure 2.3Edge detection. Left – Original image. Middle – Horizontal edge detection. Right – Vertical edge detection

Common 2D pixel-based images are grey scale images and RGB images. Gray scale images are black and white image that is normally represented by 2D arrays while the RGB images consist red, green and blue colour channels that is normally represented by 3D arrays in which the 3rd dimension (depth) is fixed to be 3 that represents RGB colours respectively. Obviously, the input image in Figure 2.3 is an RGB image. Hence, the image need to be convolved using 'RGB

kernels' which means there are RGB channels (depth is 3) in each kernel as well. CNNs use multiple kernels for feature detection at each layer. The output would be a stack of feature maps that is a 3D array while the depth of it is the number of applied kernels. Thus, the hidden layer in a CNN looks like blocks rather than a sequence of neurons.

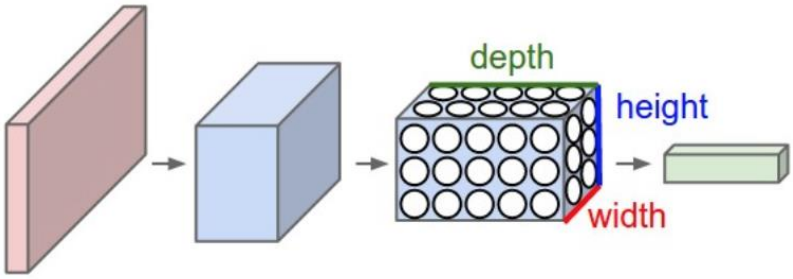


Figure 2.4 Visualization of hidden layers in CNN [14].

2.1.2 Activation Layer

Effectively, convolution operation would compute the following equation at each neuron:

$$y = \sum Wx + b$$

Where the output is y, W is the shared weights matrix of each neuron and b is the bias. The output y is clearly a linear function which can be negative infinite to positive infinite. While we want to translate it into a more understandable number like a score or a class label by introducing non-linearity. Activation functions is used for this purpose. Different activation functions have their own advantages and drawbacks which is essential to different neural networks. For example, sigmoid function is always used to squash the outputs into a range between 0 and 1, for every input x:

$$\mathrm{f(x)} = \frac{1}{1 + \mathrm{e}^{-\mathrm{x}}}$$

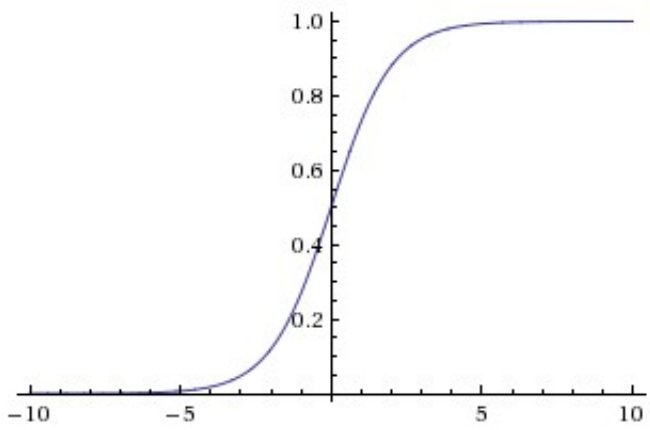


Figure 2.5Sigmoid non-linearity function.

Obviously, the gradient of sigmoid would approach to zero when the output is extremely low or high. This may slow down the learning speed of neural networks. Learning process of neural networks will be discussed in section 2.2.

In CNN, the preferred activation function is Rectified Linear Unit (ReLU) rather than Sigmoid. ReLU has been proved having a faster learning speed than Sigmoid due to its linearity and non-saturated form [15]. For every input *x*:

$$f(x) = \max(0, x)$$

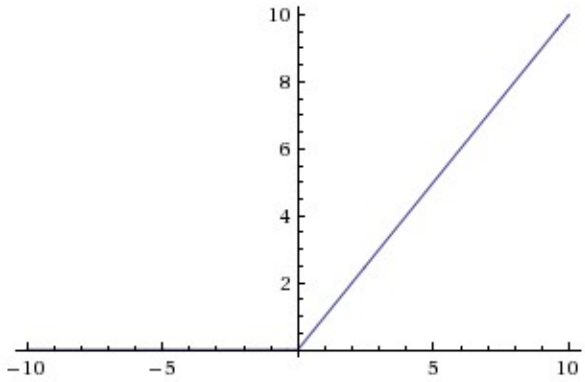


Figure 2.6ReLU activation function.

Besides, ReLU does not have exponential operations that can highly reduce the computation cost. The gradient of ReLU is always one as long as the output is

positive. As mentioned in section 2.1.1, the output from convolution operation normally would go through ReLU activation function. It could convert the output to a non-negative representation. Figure 2.7 is a sample output from ReLU activation function [16].

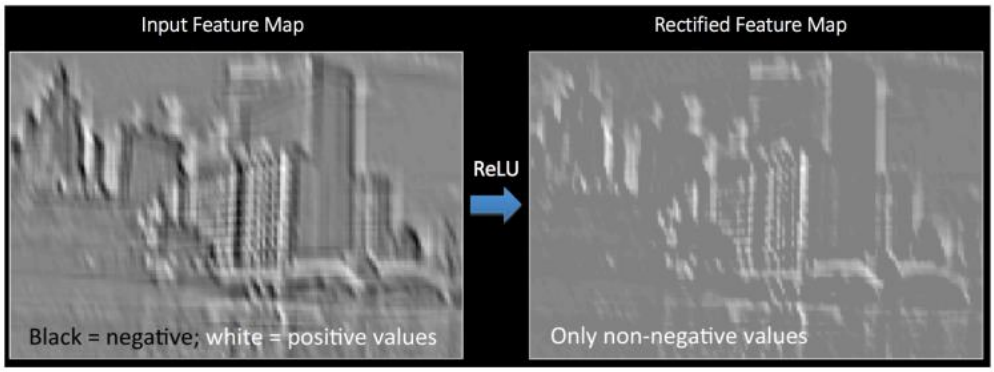


Figure 2.7 Left – Output from convolution operation. Right – Output from ReLU activation function

2.1.3 Pooling Layer

Pooling is a down-sampling operation that normally followed after convolution and activation layers. The output feature maps from convolution and activation layers can be simplified as per our need. Pooling operation is used to summarize the output of neighbouring groups of neurons in the same kernel map [15]. It can reduce the number of parameters and computations, hence can also reduce overfitting. As same as convolutional operations, pooling operations can also be controlled by filter size, stride and padding. Traditionally, we would try to avoid overlapping pooling that means the stride of pooling is as same as the filter size [15]. There are many different pooling operations, such as max pooling, average pooling, etc. In CNN, the most commonly used pooling operation is max pooling.

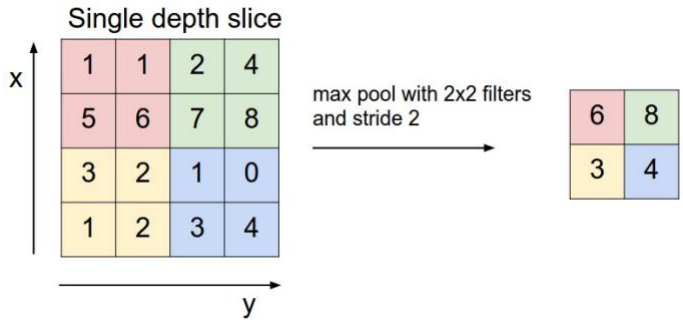


Figure 2.8An example of max pooling with 2 x 2 filters and stride 2 [16]. Max pooling would pick the maximum number from each 2 x 2 filtering area. Worth mentioning that the average pooling would calculate the average number in each 2 x 2 area.

Additionally, Springenberg et al argue that all pooling layers can be replaced with stride-increased convolutional layers without losing accuracy [17]. They introduced a full-conv net which consists only convolutional layers. Mathematically, convolution operations are similar to pooling operations since it is also a down-sampling operation that produces one value for each stay. In this thesis, we would follow the traditional way that uses pooling layers.

2.1.4 Fully Connected Layer

Normally, CNN would use at least one fully connected layer at the end. The output from all convolution, activation and pooling layers is a stack of feature maps. Fully connected layer is used to flatten a stack of feature maps into a one-dimensional feature space then connecting all flattened neurons for summarizing those feature maps. The output layer of CNN is a fully connected layer that computes the score for each class which is called logits.

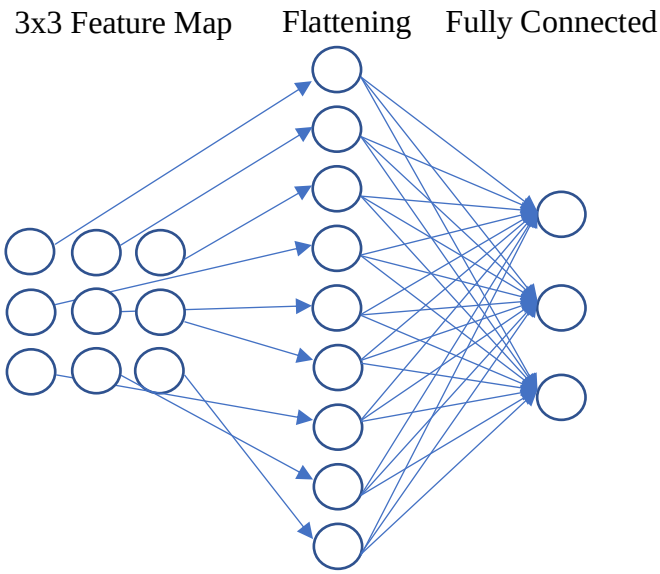


Figure 2.9 Demonstration of fully connected layer. A 3x3 feature map is flattened to 9x1 feature space then connected each other to a 3 neurons layer.

2.1.5 Softmax with Cross-Entropy

Fully connected layer outputs a logits array while we want to translate the logits into its corresponding probabilities. A softmax classifier is normally used for addressing this problem. Softmax function is defined as:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \text{ for } j = 1, \ldots, K.$$

where z is a K sized logits array. Softmax will calculate the nature exponent for each element in vector z then outputting a vector of probabilities by the corresponding nature exponential weight of z_j.

In classification problem, softmax is normally used with cross-entropy cost function. The output of cross-entropy cost function denotes the distance between prediction and ground-truth label. Neural networks use a softmax classifier to compute a vector of probabilities then calculating the cost. Cross-entropy function is:

$$H(p, q) = -\sum_{x=1}^{t} p(x) \log q(x)$$

where p, q denotes the t sized vector of predictions and ground-truth labels respectively while x represents the current index. Note that softmax function can guarantee none of the probabilities reaches to zero, hence the input of cross-entropy is guaranteed as a valid input. Cross-entropy will compute the cost of the prediction as per the ground-truth labels. As Pavel Golik says, with random initialized weights, cross-entropy would converge faster to find the local minimum comparing to squared error whereas squared error may lead to a better performance if there is a good initialization [18]. The output from cross-entropy cost function would be used for training neural network.

2.2 CNN Training

Normally, all the weights and biases in a neural network would be randomly initialized which is obviously not able to provide a robust result for prediction. In this section, we would introduce how to train a neural network and how to optimize the training process and results. This section will address parameters and hyperparameters, backpropagation and overfitting problem.

2.2.1 Parameters and Hyperparameters

Generally, the weights and biases of each neuron are the major parameters in a CNN model. Training is basically a process to update the parameters of all the neurons to improve the prediction accuracy. Assume that the neural network is a black box while the parameters would be inside the black box and the hyperparameters would be at the outside that are used to control the process of the black box. Effectively, the results of a neural network are unpredictable since

it is highly dependent on the input data, that is where hyperparameters come in to make it more controllable. Different neural networks have different hyperparameters. There are also some common hyperparameters like the number of neurons and layers, the activation functions for each layer, convolution kernel, etc. Theoretically, all hyperparameters need to be fine-tuned for the best performance of a neural network model.

2.2.2 Backpropagation

In a neural network, the input data would go through every layer and neuron to compute the cost from the prediction to the ground-truth label. Backpropagation is a method to compute the cost contribution for each neuron after forward passing the input data [19]. Frankly, training is a process to minimize the cost by updating the parameters accordingly. In terms of training neural network models, backpropagation is normally used by the gradient descent algorithm. Hence, we would introduce the gradient descent algorithm and its optimization algorithms.

2.2.2.1 Gradient Descent

Gradient descent is the technique used to update the parameters of a neural network by calculating the gradient of the cost function. There are three variants of gradient descent algorithm, batch gradient descent, stochastic gradient descent and mini-batch gradient descent.

2.2.2.1.1 Batch Gradient Descent

Batch gradient descent computes the gradient of the cost function for the entire dataset. In machine learning, gradient descent is by far the most common way to optimize neural networks [20]. Optimize a neural network means to minimize

the cost function in order to find the global minimal of the cost function.

Assume $J(\theta)$ is cost function and θ is all the parameters:

$$\theta := \theta - \eta \frac{dJ(\theta)}{d\theta}$$

where η is learning rate that is used to control the learning speed. Gradient descent would repeat this process until $\frac{dJ(\theta)}{d\theta}$ reaches to zero. Figure 2.1shows gradient descent process on a quadratic cost function.

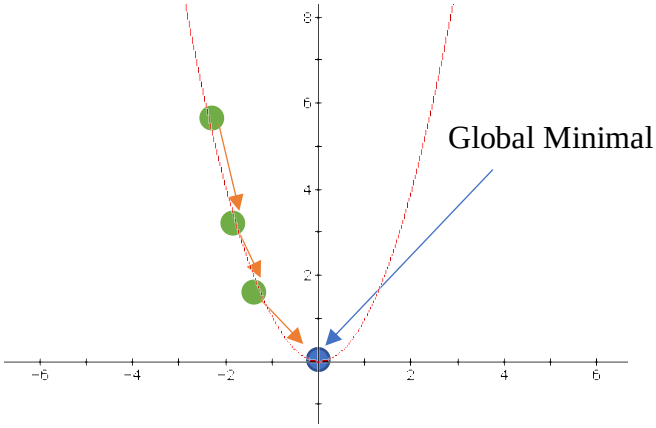


Figure 2.10 Demonstration of Gradient Descent.

In CNN, at least weights and biases need to be optimized, if using w, b to represent weights and biases and $J(w, b)$ is the cost function. To optimize w and b:

$$w := w - \eta \frac{\partial J(w, b)}{\partial w}$$

$$b := b - \eta \frac{\partial J(w, b)}{\partial b}$$

where η is the learning rate. Batch gradient descent computes the gradient of the entire training dataset then giving one updating. In machine learning, iterating through a whole dataset refers to complete an epoch. Hence, many epochs may be taken for minimizing the gradient of cost function to zero. Batch gradient descent is a slow process and may cause 'out of memory' problem if the training

dataset is too big. To address this issue, stochastic gradient descent and mini-batch gradient descent were introduced.

2.2.2.1.2 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) computes the gradient of one training data and update the parameters accordingly, for each training data x^i and ground-truth label y^i:

$$\theta := \theta - \eta \frac{dJ(\theta, x^i, y^i)}{d\theta}$$

For a large dataset, despite of the memory fitting problem, batch gradient descent may waste the computation power on computing the gradients of similar examples. However, batch gradient descent could converge to the exact minimum while SGD is normally fluctuating around the exact minimum, because SGD updates the parameters frequently with a high variance which would result in an overshooting problem [20]. Apart from that, it is not time efficient to perform a backpropagation immediately after computing the gradient for one training data.

2.2.2.1.3 Mini-Batch Gradient Descent

Mini-batch gradient descent updates the parameters after computing the gradient of a batch of input data. With a vector of training data *x* and label *y*, for every mini batch *n*:

$$\theta := \theta - \eta \frac{dJ\left(\theta, x^{(i;i+n)}, y^{(i;i+n)}\right)}{d\theta}$$

Mini-batch gradient descent reduces the variance of the parameter updates which could result in a stable convergence [20]. Meanwhile, performing a backpropagation at one shot using a matrix of computed gradient is more

efficient. Mini-batch gradient descent is used more commonly than the other two. This thesis is using mini-batch gradient descent to train the CNN model.

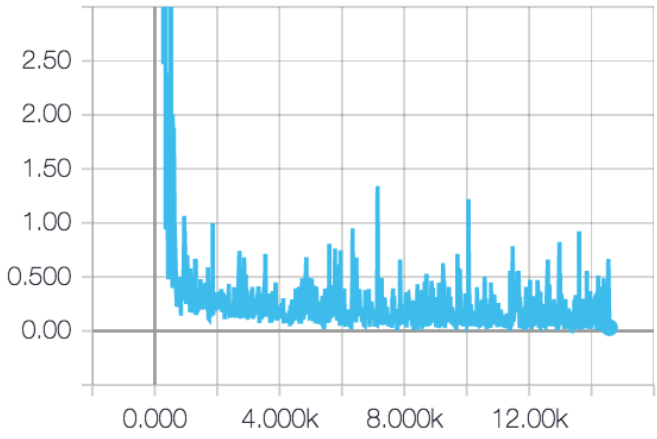


Figure 2.11 Minimizing cross-entropy cost function with mini-batch gradient descent. It would trend down to zero but it would be noisier.

2.2.2.2 Optimizer

For gradient descent, $\theta := \theta - \eta \frac{dJ(\theta)}{d\theta}$ is the kernel algorithm to update the parameters. In order to accelerate the speed of gradient descent, there are several optimization algorithms which is called optimizers for updating the parameters. Optimizers aim to achieve a faster convergence to the global minimal of the cost function.

2.2.2.2.1 Momentum

Momentum is the term takes from the classical physics which gives the gradient descent an initial velocity before every updating. It is like to throw a ball from a hill that gathers momentum and its velocity keeps increasing [21]. In momentum, updating the parameters θ_t in current iteration t will take the previous steps into account, it would be updated as follows:

$$\theta_t := \gamma\theta_{t-1} - \eta \frac{dJ(\theta_t)}{d\theta_t}$$

where γ is the momentum, η is the learning rate and θ_{t-1} is the parameters from last step. By using momentum, the convergence speed is accelerated towards the same convergence direction otherwise, it reduces oscillation around the minimal point. Besides, it could give a push to the gradient of the cost function when it stuck in a local minimal.

2.2.2.2.2 RMSProp

Root Mean Square prop (RMSProp) is an adaptive learning rate method that adjusts the learning rate by the computed gradient accordingly [22]. It uses a move average of squared gradient to adapt the learning rate for different parameters to reduce the oscillation while converging. The learning rate would be adapted as follows:

$$MS(\theta_t) = \delta MS(\theta_{t-1}) + (1-\delta)(\frac{dJ(\theta_t)}{d\theta_t})^2$$

$$\theta := \theta - \frac{\eta}{\sqrt{MS(\theta_t)} + \epsilon} \times \frac{dJ(\theta)}{d\theta}$$

where the function $MS(\theta_t)$ would compute the exponentially decaying average of gradients with the decay rate δ then adapting the learning rate using $\frac{\eta}{\sqrt{MS(\theta_t)}+\epsilon}$. Note that ϵ is a small constant value to make sure the denominator will not be zero. In practice, δ is normally suggested using 0.9 [22]. RMSProp could zoom in the learning rate when got a small gradient and zoom out the learning rate when got a big gradient by applying square root on it. It would be helpful when facing a high dimensional parameters problem.

2.2.2.2.3 Adam

Adaptive Moment Estimation (Adam) [23] is another algorithm for optimizing the learning rate for each parameter. It is like a combination of Momentum of RMSProp which would consider both exponentially decaying of past gradient *m* and exponentially decaying of past squared gradient *v*:

$$m_t = \delta_1 m_{t-1} + (1-\delta_1)\frac{dJ(\theta_t)}{d\theta_t}$$

$$v_t = \delta_2 v_{t-1} + (1-\delta_2)(\frac{dJ(\theta_t)}{d\theta_t})^2$$

where δ_1 and δ_2 are the decay rate for *m* and *v*. Notably, *m* and *v* would estimate the mean and variance of the gradients respectively while they are biased towards zero in the initial steps since they are initialized as 0. Hence, to correct the bias:

$$\hat{m}_t = \frac{m_t}{1 - \delta_1}$$

$$\hat{v}_t = \frac{v_t}{1 - \delta_2}$$

Thus, the parameters would be updated as follows:

$$\theta := \theta - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \times \hat{m}_t$$

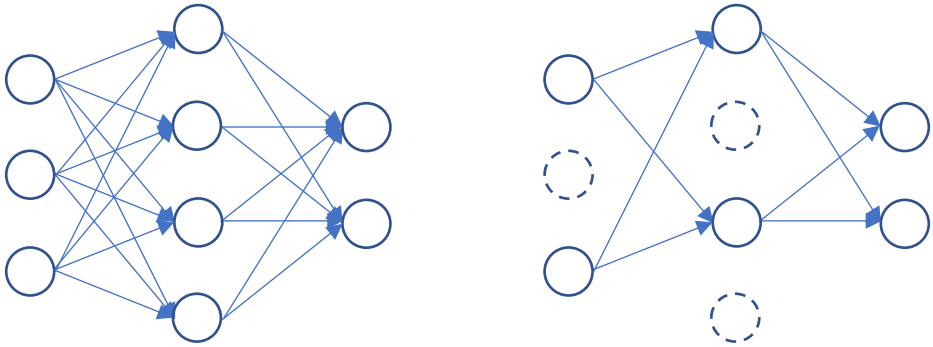
Adam optimizer shows a robust and stable performance in logistic regression, multi-layer neural networks and convolutional neural networks [23]. In this thesis, Adam optimizer would be used as default.

2.3 Overfitting

As aforementioned, neural networks are data-driven algorithm that the output is unpredictable. Tuning hyperparameter could add more controls on the process in the black box while some errors may be introduced by the data itself. Overfitting is a common issue caused by the unseen data, since the whole dataset would normally be separated into training set and testing set where the algorithm may fit the training data well whereas does not fit the testing data. This problem can be addressed by serval regularization technique.

2.3.1 Dropout Layer

Dropout [24] is a technique that will randomly shut down some neurons between fully connected layers. The key idea of it is to reduce the impact of the influential features. In CNN, a feature would have a significant weight if a feature is heavily used for some classes. Dropout layers would randomly close some neurons to reduce their impact. In the end, dropout regularization can significantly reduce overfitting.



(a) Original neural network (b) After dropout

Figure 2.12The results of original neural network and randomly drop out some neurons in that neural network.

2.3.2 Batch Normalization

While training with mini-batch gradient descent, batch normalization [25] is a technique to normalize the input data for each training mini-batch within the architecture of neural networks. In a neural network, all parameters in all layers would change frequently that slows down the convergence, that refers to the internal covariate shift. Internal covariate shift is defined as the change in the distribution of network activations due to the change in network parameters during training [25]. To reduce the impact of internal covariate shift, batch normalization would transform the input data for each layer according to the mean and variance of each activation. This is normally used before activations and after fully connected layers. Assuming to input a mini-batch of data *x*, Batch normalization is defined as follows:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^i$$

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} (x^i - \mu)^2$$

$$\hat{x}^i = \frac{x^i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

where it computes the mean μ and the variance σ^2 of the mini-batch x then fit all data into the standard Gaussian distribution ($\mu = 0, \sigma^2 = 1$). Hence, all the data comes from the same distribution which mitigates the internal covariate shift problem. Here is the scaling and shifting process in the end:

$$y^i = \gamma \hat{x}^i + \beta$$

where the γ and β are the learnable parameters for scaling and shifting the transformed input data. This technique accelerates the convergence speed and improves the performance of a neural network model significantly. In some cases, dropout layers can be eliminated by batch normalization [25]. Batch normalization can also help for reducing overfitting.

3 Technical Background

In this section, we will look into the Tensorflow framework for machine learning and states the working environment. Meanwhile, we will introduce the data source and gives the architecture of CNN model we would use in this thesis.

3.1 Intro to Tensorflow framework

There are many machine learning frameworks, like Torch, Theano, Caffe, Keras and Tensorflow. Most of those frameworks are written in highly optimized C++ code and provide APIs for other languages like Python, Java, C++, GO, etc. Support for GPU-acceleration is a standard technology for those well-known frameworks which can highly speed up the training of CNN to save our time [26]. In this project, we will mainly use Tensorflow with Python environment to build our neural network. In Tensorflow, mathematical computations are described as Stateful Dataflow Graphs where the operators are represented as nodes and their relationships are represented as edges [27]. It will help us construct and visualize our neural network. Besides, Tensorflow is built by Google and is getting more and more popular in the community that would help us solve our problems efficiently.

3.2 Data source

Point cloud data may have various representations in the real-world scanning. The basic point cloud format is *.xyz in which contains only x, y, z coordinates. There are also more advanced data format like *.pts, *.ptx etc., in which normally contains other properties like reflection and RGB channels. Notably, some of the data format like *.ply is binary files which cannot be easily processing. The target industrial data is the laser scans of a power plant provided

by GE Power. We cropped some interested objects from a 25-gigabyte scanning, including 10 pipes, 7 valves and 4 warning signs. All of those objects would be converted into *.xyz format while the number of points are variant from each other, ranged from 12,592 to 100,675.

Due to the small size of industrial data, we used another open sourced point clouds as a supplementary for training. In this thesis, we will train our model with the dataset from ShapeNet [28], which contains 31963 pre-labelled 3D point cloud models in 16 categories. Since all the *.pts data in this dataset does not contain further information apart from the x, y, z coordinates, it can be treated as *.xyz data. Worth to mention that each point cloud object contains approximately 8000 points.

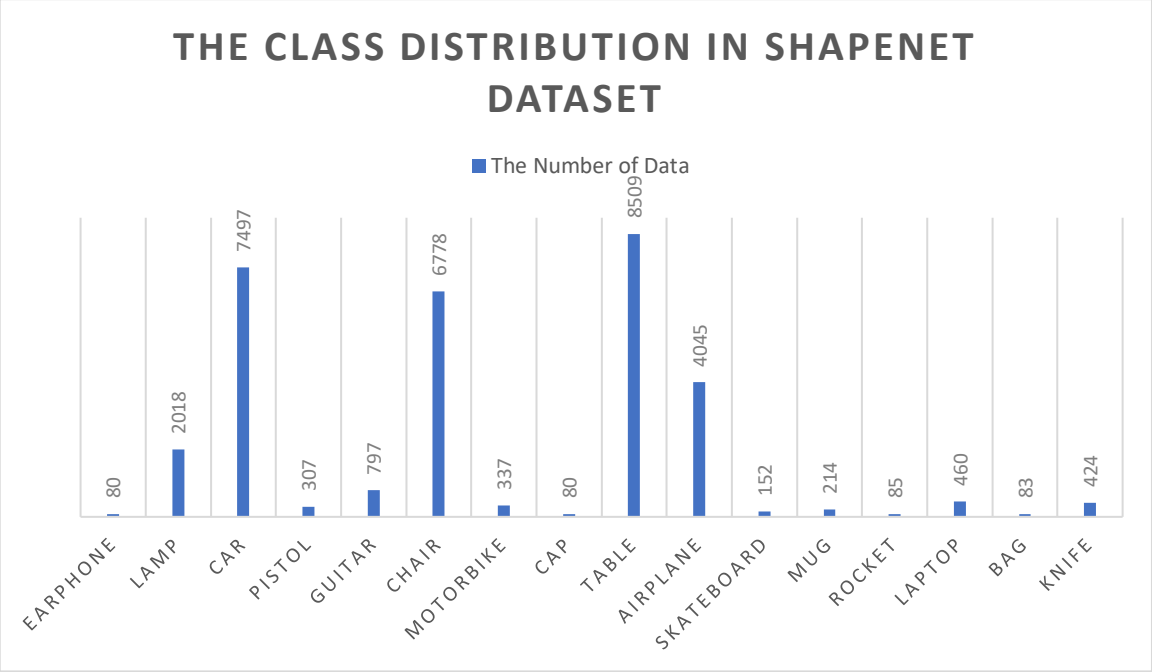


Figure 3.1The Class Distribution in ShapeNet Dataset

All of the data would be split into 80% training set and 20% testing set. Having 20% testing set helps to see the true performance of our CNN model.

3.3 Environment

All the works and experiments were carried out under the same environment. For hardware, we would use Intel Xeon E7-4850 with 32GB memory and NVIDIA Tesla K20. For software, we were using Python 3.6, Tensoflow 1.4.0, Tensorflow-GPU 1.4.0, CUDA 8.0, cuDNN 6.1. The latest version of Python and Tensorflow are used for building up CNN while CUDA 8.0 and cuDNN 6.1 are the required software to enable NVIDIA GPU training.

3.4 CNN architecture

In the field of 2D image recognition, VGG Net has been proven to have a good performance compare to other architecture [29]. In this thesis, a VGG-like architecture is used in which a pooling operation would be followed after 2 convolution operations. The convolution layer would be fixed to use 3 x 3 x 3 kernel filters, stride 1 and same padding. The pooling operation is performed over 2 x 2 filters with stride 2. Then a bath normalization layer would be applied and followed by a fully connected layer which outputs 1024 classes. After going through drop out regularization, the output would be obtained from final layer. All input data would be converted to 32 x 32 x 32 voxel grids without RGB colour channels which refers to section 6.1.1.

Table 3.1CNN configurations

Layers	Settings	Output Array Shape
Input	None	(32, 32, 32, 1)
Convolution	Kernel Size (3, 3, 3), Stride 1, Same padding, Filters 16	(32, 32, 32, 16)
ReLU	None	(32, 32, 32, 16)
Convolution	Kernel Size (3, 3, 3), Stride 1, Same padding, Filters 32	(32, 32, 32, 32)
ReLU	None	(32, 32, 32, 32)
Max Pooling	Kernel Size (2, 2, 2), Stride 2, Same padding	(16, 16, 16, 32)
Convolution	Kernel Size (3, 3, 3), Stride 1, Same padding, Filters 64	(16, 16, 16, 64)
ReLU Layer	None	(16, 16, 16, 64)

Convolution	Kernel Size (3, 3, 3), Stride 1, Same padding, Filters 128	(16, 16, 16, 128)
ReLU	None	(16, 16, 16, 128)
Max Pooling	Kernel Size (2, 2, 2), Stride 2, Same padding	(8, 8, 8, 128)
Convolution	Kernel Size (3, 3, 3), Stride 1, Same padding, Filters 256	(8, 8, 8, 256)
ReLU Layer	None	(8, 8, 8, 256)
Convolution	Kernel Size (3, 3, 3), Stride 1, Same padding, Filters 512	(8, 8, 8, 512)
ReLU	None	(8, 8, 8, 512)
Max Pooling	Kernel Size (2, 2, 2), Stride 2, Same padding	(4, 4, 4, 512)
Batch Norm	None	(4, 4, 4, 512)
Fully Con	Number of neurons 1024	(1024)
ReLU	None	(1024)
Drop Out	Keep rate 0.5	(512)
Output	Number of classes	(number of classes)
Softmax	None	(number of classes)

4 Data Processing

Point cloud data need to be pre-processed in order to feed into neural networks. In this section, we implemented a series of data processing methods, including processing point clouds and labels, data augmentation and data persistence. Besides, due to the limited number of data we have, we would augment our industrial dataset and apply transfer learning technique (section 5.1.2) as a work around. Hence, we would separate our data into two datasets, one for training a trained model and another one for applying transfer learning. The first dataset contains all geometries who has at least 2000 available objects. All the rest would be included into the second dataset while each geometry would contain at least 100 objects by applying data augmentation. The web front end of data pre-processing is under appendix I.ii.

4.1 Point Cloud Normalization

For classification, we normalized point clouds to the same scale since the shape of an object is more important than its size. Hence, all point clouds would be measured by the same measurement which may ease the future works. Assume that pX, pY, pZ represent a point in a point cloud and pMinX, pMinY, pMinZ, pMaxX, pMaxY, pMaxZ to represent the minimum and maximum value in a point cloud along each axis. Firstly, the initial value of pMinX, pMinY, pMinZ, pMaxX, pMaxY, pMaxZ would be assigned to the first point in the point cloud:

$$pMinX = pMaxX = pX$$
$$pMinY = pMaxY = pY$$
$$pMinZ = pMaxZ = pZ$$

, for every point:

$$pMinX = \min(pX, pMinZ)$$
$$pMinY = \min(pY, pMinY)$$
$$pMinZ = \min(pZ, pMinZ)$$
$$pMaxX = \max(pX, pMaxX)$$
$$pMaxY = \max(pY, pMaxY)$$
$$pMaxZ = \max(pZ, pMaxZ)$$

Assume pMax is the biggest value along all axes:

$$pMax = \max(pMaxX - pMinX, pMaxY - pMinY, pMaxZ - pMinZ)$$

Then we would move all points into the first quadrant and divide all coordinates by the maximum value, for every point:

$$pX = (pX - pMinX)/pMax$$
$$pY = (pY - pMinY)/pMax$$
$$pZ = (pZ - pMinZ)/pMax$$

The pseudo implementation for normalizing a point cloud:

```
NormalizePointCloud (pointCloud) {
	minimumValue = [pointCloud[0].X, pointCloud[0].Y, pointCloud[0].Z]
	maximumValue = pointCloud[0].X

	for each point in pointCloud {
		if (point[0] < minimumValue[0]) {
			minimumValue[0] = point.X
		}
		if (point[1] < minimumValue[1]) {
			minimumValue[1] = point.Y
		}
		if (point[2] < minimumValue[2]) {
			minimumValue[2] = point.Z
		}
	}
	# move to first quadrant
	newCloud = new list
	for each index, pair in pointCloud {
		newPoint = new list
		newPoint[0] = point[0] - minimumValue[0]
		newPoint[1] = point[1] - minimumValue[1]
		newPoint[2] = point[2] - minimumValue[2]
		newCloud[index] = newPoint
		for each value in newPoint {
			if (value > maximumValue) then {
				maximumValue = value
			}
		}
	}
	normalized = newCloud / maximumValue
	return normalized
}
```

Figure 4.1Implementation of point cloud normalization.

4.2 Volumetric Representation

Volumetric representation is a transformation aims to fit a point cloud into a voxel grid.

By far, the original point cloud has been normalized and ready to be fit into voxel grids. If the size of voxel grids we are converting to is m x m x m, the voxel grids can be initialized as

$$V_{ijk} = 0 \ (i, j, k \ \in \mathbb{Z}, 0 \leq i < m, 0 \leq j < m, 0 \leq k < m)$$

Thus, we can populate the voxel with a normalized point cloud by locating every point as

$$i = \left\lfloor \frac{pX + \varepsilon}{m} \right\rfloor, \varepsilon \to 0$$

$$j = \left\lfloor \frac{pY + \varepsilon}{m} \right\rfloor, \varepsilon \to 0$$

$$k = \left\lfloor \frac{pZ + \varepsilon}{m} \right\rfloor, \varepsilon \to 0$$

where the ε is a small value used to make sure *i, j, k* would not equal to m. There are three types of voxel grids:

- **Binary Occupancy Grid**.

  In this representation, each voxel has only binary value, occupied or unoccupied. For every point:

  $$V_{ijk} = 1 \ (i = \left\lfloor \frac{pX}{m} \right\rfloor, i = \left\lfloor \frac{pY}{m} \right\rfloor, i = \left\lfloor \frac{pZ}{m} \right\rfloor)$$

  The implementation in pseudo code:

```
BinaryGrid (normalizedPointCloud, dimension) {
        gridLengthOfX = 1 / dimension[0]
        gridLengthOfY = 1 / dimension[1]
        gridLengthOfZ = 1 / dimension[2]
        output = new listOfZeros
        epsilon = 0.000000000001
        for each point in normalizedPointCloud {
                x_loc = Integer(point[0] / (gridLengthOfX + epsilon))
                y_loc = Integer(point[1] / (gridLengthOfY + epsilon))
                z_loc = Integer(point[2] / (gridLengthOfZ + epsilon))
                output[x_loc, y_loc, z_loc] = 1
        }
        return output
}
```

Figure 4.2 Implementation of Binary Grid converter.

- Density Occupancy Grid.

In this representation, each voxel would represent the relative ratio of the number of points in each voxel regarding the total number of points. For every point:

$$V_{ijk} = V_{ijk} + 1 \ (i = \left\lfloor \frac{pX}{m} \right\rfloor, i = \left\lfloor \frac{pY}{m} \right\rfloor, i = \left\lfloor \frac{pZ}{m} \right\rfloor)$$

After all the iterations, we would divide every voxel by the total number of points t:

$$V_{ijk} = \frac{V_{ijk}}{t} \ (i, j, k \ \in \mathbb{Z}, 0 \leq i < m, 0 \leq j < m, 0 \leq k < m)$$

The implementation in pseudo code:

```
DensityGrid (normalizedPointCloud, dimension) {
        gridLengthOfX = 1 / dimension[0]
        gridLengthOfY = 1 / dimension[1]
        gridLengthOfZ = 1 / dimension[2]
        output = new listOfZeros
        epsilon = 0.000000000001
        max_volume_size = 0
        for each point in normalizedPointCloud {
                x_loc = Integer(point[0] / (gridLengthOfX + epsilon))
                y_loc = Integer(point[1] / (gridLengthOfY + epsilon))
                z_loc = Integer(point[2] / (gridLengthOfZ + epsilon))
                output[x_loc, y_loc, z_loc] += 1
                if (output[x_loc, y_loc, z_loc] > max_volume_size) then {
                        max_volume_size = output[x_loc, y_loc, z_loc]
                }
        }
        return output / max_volume_size
}
```

Figure 4.3 Implementation of Density Grid converter.

- Hit Occupancy Grid.

This representation only considers hits in each voxel. For every point:

$$V_{ijk} = V_{ijk} + 1 \ (i = \left\lfloor \frac{pX}{m} \right\rfloor, i = \left\lfloor \frac{pY}{m} \right\rfloor, i = \left\lfloor \frac{pZ}{m} \right\rfloor)$$

The implementation in pseudo code:

```
HitGrid (normalizedPointCloud, dimension) {
        gridLengthOfX = 1 / dimension[0]
        gridLengthOfY = 1 / dimension[1]
        gridLengthOfZ = 1 / dimension[2]
        output = new listOfZeros
        epsilon = 0.000000000001

        for each point in normalizedPointCloud {
                x_loc = Integer(point[0] / (gridLengthOfX + epsilon))
                y_loc = Integer(point[1] / (gridLengthOfY + epsilon))
                z_loc = Integer(point[2] / (gridLengthOfZ + epsilon))
                output[x_loc, y_loc, z_loc] += 1
        }
        return output
}
```

Figure 4.4 Implementation of Hit Grid converter.

From the experiments of those representations in section 6.1, it turns out that the 32 x 32 x 32 density occupancy grids have the best performance in our experiment.

(a) Point Cloud (b) Occupancy Grid

Figure 4.5Point cloud (left) and converted occupancy grid.

4.3 Labelling of Categorical Data

One-hot encoding is a technique to convert the real-world labels into machine understandable labels. Categorical data is the data has been labelled as categorical label, like 'dog', 'cat', etc. Generally, categorical data could be processed by two steps: Integer Encoding and One-hot encoding. Integer encoding would convert the categorical data into numeric representations. For instance, 'dog' is 1, 'cat' is 2. Integer encoding is not sufficient for machine

learning since the nature ordering may result in a poor performance or unexpected results [30]. Hence, we should have a step further to convert the integer encoded values into one-hot encoded, like 'dog' is [1, 0], 'cat' is [0, 1]. In this thesis, all input data is categorical data and will be one-hot encoded.

```
IntegerEncoding(labels) {
        cates = new list
        for each label in labels {
                if label not in cates then {
                        cates.add(label)
                }
       }
       return cates
}

OneHotEncoding(labels) {
        cates = IntegerEncoding(labels)
        onehot = new list
        for each label in labels{
                encoded = new listOfZeros()
                encoded[cates.indexOf(label)] = 1.0
                onehot.add(encoded)
        }
        return onehot
}
```

Figure 4.6 Implementation of one-hot encoding.

4.4 Data Augmentation

Training a neural network needs a big volume of data since it is a data-hungry algorithm. The small industrial dataset is clearly not enough for training. Data Augmentation (DA) is a general method for solving unbalanced data and overfitting problems by transforming input images to generate more training data. For 2D images, traditional transformation methods consist shifting, zooming, rotating, etc. while there are some fancy ways like using Generative Adversarial Nets (GANs) [31], colour jittering and Fancy Principle Component Analysis (PCA) [32]. Some of the fancy ways like using GANs are not easy to apply on point cloud data. Hence, we are going to augment our point cloud with some traditional approaches, like squeezing, rotating and noising.

4.4.1 Squeeze

Point cloud data is a vector of point coordinates which means we can even transform the whole point cloud to another geometry by modifying the points. Squeezing is a relatively simple transformation that could be achieved by multiplying a fixed ratio along an axis. Assuming the squeezing ratio is v:

a) Squeeze by x:

$$X = x * \mathrm{v}$$
$$Y = \mathrm{y}$$
$$Z = z$$

b) Squeeze by y:

$$X = x$$
$$Y = \mathrm{y} * \mathrm{v}$$
$$Z = z$$

c) Squeeze by z:

$$X = x$$
$$Y = \mathrm{y}$$
$$Z = z * v$$

The implementation of point cloud squeeze by x-axis:

```
SqueezeByX (pointCloud, percentage) {
        output = new list
        for each point in pointCloud {
                newPoint = point
                newPoint[0] = newPoint[0]*v
        }
        return output
}
```

Figure 4.7 Implementation of point cloud squeeze by x-axis.

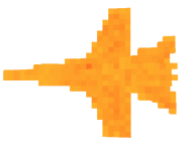
Squeeze by x-axis Percentage	**0%**	**15%**	**30%**	**45%**
Point cloud				
Voxel grid				

Figure 4.8Visualization of squeezing an airplane by x-axis. From left to right are original point cloud, 15%, 30% and 45% respectively.

4.4.2 Rotate

Rotation a 3D object is not as intuitive as rotating a 2D image. It would help CNN learn to recognize the same object in different perspectives. Assume the required rotation angle is θ, every coordinate in the point cloud needs to be renewed as:

a) Rotate around x-axis:

$$X = x$$
$$Y = y * \cos(\theta) - z * \sin(\theta)$$
$$Z = y * \sin(\theta) + z * \cos(\theta)$$

b) Rotate around y-axis:

$$X = z * \sin(\theta) + x * \cos(\theta)$$
$$Y = y$$
$$Z = z * \cos(\theta) - x * \sin(\theta)$$

c) Rotate around z-axis:

$$X = x * \cos(\theta) - y * \sin(\theta)$$
$$Y = x * \sin(\theta) + y * \cos(\theta)$$
$$Z = z$$

Here is the implementation of point cloud rotation by x-axis:

```
RotateByX (pointCloud, degree){
       res = new list
       for each point in pointCloud {
              newPoint = new list
              newPoint[0] = point[0]
              newPoint[1] = point[1]*cos(degree) - point[2]*sin(degree)
              newPoint[2] = point[1]*sin(degree) + point[2]*cos(degree)
              res[index] = new_point
      }
      return res
}
```

Figure 4.9 Implementation of point cloud rotation by x-axis.

Rotation by x-axis Percentage	**0%**	**60%**	**120%**	**180%**

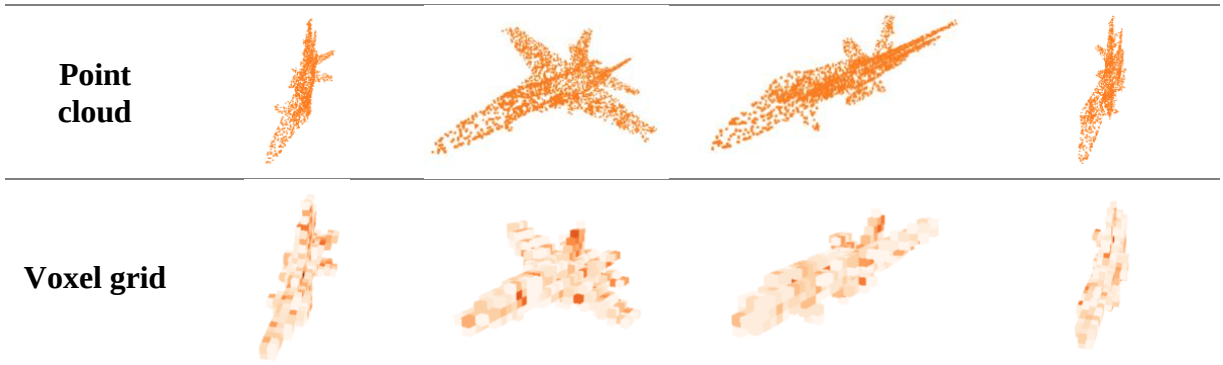


Figure 4.10Visualization of point cloud rotation by x-axis. From Left to Right are the original point cloud, 60-degree, 120-degree and 180-degree rotation respectively.

4.4.3 Gaussian Noise

In real-world, point clouds normally come with different ratio of noises. Hence, a good way of data augmentation is to mimic the real-world case to add some noise onto the cloud. Gaussian Noise is known as a white noise which can be generated by Gaussian distribution:

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

in which μ means the mean value while σ means the standard deviation. In this thesis, we would use λ to denote the ratio of noise would be added on regarding the number of points t in the normalized point cloud. Meanwhile, σ is set to 1 and μ is set to 0 in our case. We would simply generate $\lfloor \lambda * t \rfloor$ number of noisy points and append the generated points after the normalized point cloud. For a random picked point (p_x, p_y, p_z) and random generated real number m, a noise point could be generated by:

$$n_x = p_x + \frac{1}{\sqrt{2\pi}} e^{-\frac{m^2}{2}} (pMaxX - pMinX), m \in \mathbb{R}$$

$$n_y = p_y + \frac{1}{\sqrt{2\pi}} e^{-\frac{m^2}{2}} (pMaxY - pMinY), m \in \mathbb{R}$$

$$n_z = p_z + \frac{1}{\sqrt{2\pi}} e^{-\frac{m^2}{2}} (pMaxZ - pMinZ), m \in \mathbb{R}$$

where $pMaxX$, $pMaxY$, $pMaxZ$, $pMinX$, $pMinY$, $pMinZ$ are the maximum and minimum values along x, y, z axes. The implementation in pseudo code is:

```
        gaussianPoints(numberOfPoints) {
                points = new list
                for each index in range(numberOfPoints) {
                        x = randomNumber()
                        point = 1/root(2*pi) * power(e, -x*x/2)
                        points.add(point)
                }
                return points
        }

GaussianNoise (pointCloud, percentage) {
        sizeOfNoises = Integer(percentage*lengthOf(pointCloud))
        max_x = min_x = pointCloud[0][0]
        max_y = min_y = pointCloud[0][1]
        max_z = min_z = pointCloud[0][2]
        for each point in pts {
                if (max_x > point[0]) then { max_x = point[0] }
                if (max_y > point[1]) then { max_y = point[1] }
                if (max_z > point[2]) then { max_z = point[2] }
                if (min_x < point[0]) then { min_x = point[0] }
                if (min_y < point[1]) then { min_y = point[1] }
                if (min_y < point[2]) then { min_y = point[2] }
        }
        noise_x = gaussianPoints(noise_size) * (max_x - min_x)
        noise_y = gaussianPoints(noise_size) * (max_y - min_y)
        noise_z = gaussianPoints(noise_size) * (max_z - min_z)
        noises = new list
        for each index in range(noise_size) {
                pos = randomInteger(lengthOf(pointCloud))
                noises[index][0] = pointCloud[pos][0] + noise_x[index]
                noises[index][1] = pointCloud[pos][1] + noise_y[index]
                noises[index][2] = pointCloud[pos][2] + noise_z[index]
        }
        pointCloud.add(noises)
        return pointCloud
}
```

Figure 4.11Implemenetation of adding Gaussian Noise.

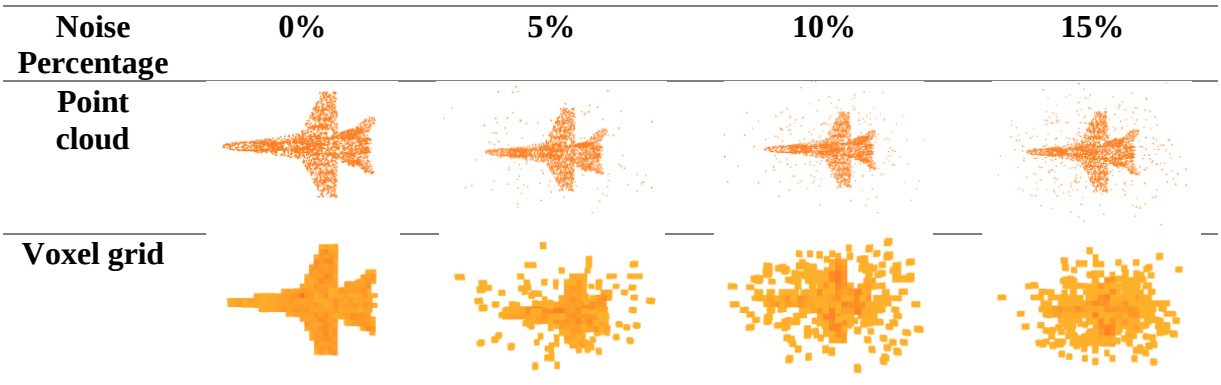


Figure 4.12Visualization of different ratio of Gaussian Noises. From left to right are original point cloud, 5%, 10% and 15% noise points respectively.

4.5 Dataset Persistence

Data shuffling is a normal technique to enhance the training process. If the input data is ordered one class after another like A-Z, neural network training would be biased to one after another. Another issue is that cross validation would not work if we keep the order since there may want to predict an untrained geometry. Hence, the upcoming problem is that how to keep the order of shuffled data to reduce the variance for next training. In order to reuse a dataset without any slight change, we would capsulate the whole dataset into Hierarchical Data Format 5 (HDF5). HDF5 is designed to store and organize large amounts of data [33]. Therefore, as aforementioned, we would build two HDF5 dataset to store those two datasets respectively. Each HDF5 dataset contains three fields, training data, their corresponding one-hot encoded labels and the real labels.

5 Object Recognition

This application takes a point cloud of multiple objects then figuring out the proper labels of each point cloud object. Besl & Jain proposed a general object recognition system [34] which can be translated into two steps. The first step is to model a descriptor for every geometry. Then to localize objects by comparing every region in a scene with every geometry description. Therefore, we would follow this guidance to divide this work into two steps, classification and localization. The web frontend of training and predicting is under appendix I.ii and I.iii respectively.

5.1 3D Object Classification

We hope our 3D CNN can be fed with point cloud data. As mentioned in section 0, training on point clouds is not easy due to its characteristics. Recent works like FusionNet [35] and VoxNet [2] introduced V-CNN and VoxNet respectively. Those works trained their CNN classifier on the volumetric representation of point clouds rather than on the point clouds themselves. This volumetric representation is named voxel grids that allows us to efficiently measure the point cloud as well as storing and manipulating the point cloud in a simple and efficient data structure [2]. Theoretically, voxel grid is sufficient for CNN to detect the shape of point clouds, although we will lose some information from this down-sampling behaviour. In this thesis, all experiments are based on this volumetric representation. Object classification could be achieved in two steps, obtaining a pre-trained model and applying transfer learning.

5.1.1 Training with Hyperparameter Search

The proper suite of hyperparameters is essential to train the CNN model. The value of hyperparameters could control the speed of learning and determine the performance of an algorithm. Generally, grid search and random search are widely used methods for finding suitable hyperparameters. As James & Yoshua said, random search is more efficient in high dimensions (5 dimensions in the paper) while grid search has similar performance as random search in low dimensional problems (3 dimensions in the paper) [36]. In this case, we would perform a grid search since there are only 3 hyperparameters we are going to optimize, which are batch size, drop-out rate and learning rate.

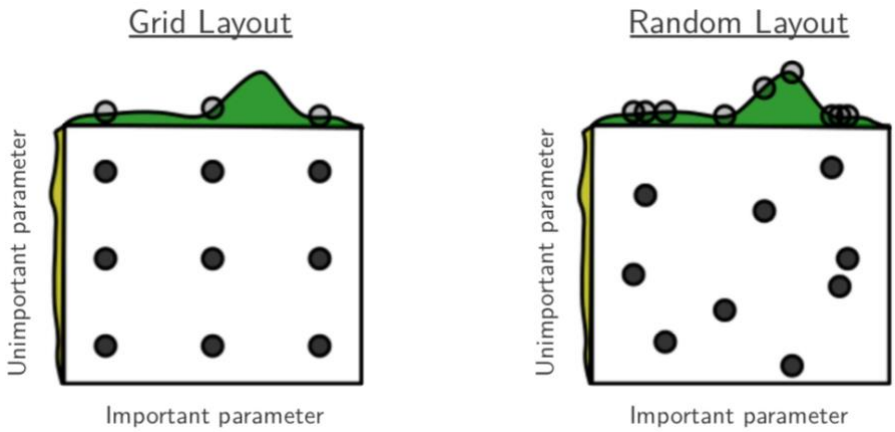


Figure 5.1Nine trials on grid search and random search [36]. It turns out that random search explores nine distinct values for each parameter.

Grid search is one of the classic ways of choosing hyperparameters in machine learning. Yoshua proposed a general guidance for exploring hyperparameters that we should consider the best value on the border, the scale of values and the computational cost [37]. In this regard, a manual search would be done at first to obtain an idea of the possible range of each hyperparameter. Then we would perform a grid search in those ranges to find the best hyperparameter

combination. The manual search and grid search would be performed on 10% of the whole dataset and trained for 5 epochs for each suite of hyperparameters. Thus, we would implement the early stop technique to terminate the training if the prediction accuracy converges to be stable.

```
grid_search(learning_rate_list, batch_size_list, keep_rate_list) {
      best_param = empty
      best_accuracy = empty
      for each lr in learning_rate_list {
            for each bs in batch_size_list {
                  for each kr in keep_rate_list {
                        model = training_with_parameter(lr, bs, kr)
                        accuracy = prediction(model)
                        if accuracy > best_accuracy then {
                              best_param = [lr, bs, kr]
                              best_accuracy = empty
                        }
                  }
            }
      }
      return best_param
}
```

Figure 5.2 Pseudo Code of Grid Search. This will search for 3 hyperparameters: learning rate, batch size and keep rate.

We would perform a grid search on three hyperparameters which are learning rate, batch size and keep rate. The value grid for grid search would be defined as:

Keep rate: [0.5, 0.7]

Batch Size: [16, 32, 64]

Learning rate: [0.01, 0.005, 0.001, 0.0005]

Grid search would iterate every possible combination of those hyperparameters and test out which combination could result in a better result.

Table 5.1Grid Search with Early Stop. The empty slot means it has been converged that the improvement of accuracy is less than 0.001.

Keep Rate	Batch Size	Learning Rate	Accuracy of Epoch				
			1	2	3	4	5
0.7	**16**	**0.01**	0.949	0.975	0.952	0.923	0.832
0.5	**16**	**0.01**	0.875	0.943	0.966	0.893	0.858
0.7	**32**	**0.01**	0.965	0.964	0.967	0.957	0.858
0.5	**32**	**0.01**	0.958	0.963	0.947	0.86	0.89
0.7	**64**	**0.01**	0.895	0.931	0.932		
0.5	**64**	**0.01**	0.96	0.964	0.968	0.972	0.912
0.7	**16**	**0.005**	0.963	0.972	0.959	0.93	0.961
0.5	**16**	**0.005**	0.968	0.952	0.963	0.92	0.966
0.7	**32**	**0.005**	0.957	0.946	0.963	0.96	0.962
0.5	**32**	**0.005**	0.949	0.959	0.972	0.972	
0.7	**64**	**0.005**	0.951	0.962	0.969	0.977	0.974
0.5	**64**	**0.005**	0.949	0.971	0.975	0.969	0.97
0.7	**16**	**0.001**	0.968	0.978	0.981	0.98	0.98
0.5	**16**	**0.001**	0.974	0.973	0.983	0.982	
0.7	**32**	**0.001**	0.981	0.972	0.985	0.984	0.98
0.5	**32**	**0.001**	0.983	0.983			
0.7	**64**	**0.001**	0.983	0.98	0.985	0.982	0.983
0.5	**64**	**0.001**	0.981	0.978	0.98	0.984	0.985
0.7	**16**	**0.0005**	0.981	0.974	0.985	0.984	
0.5	**16**	**0.0005**	0.982	0.969	0.989	0.989	
0.7	**32**	**0.0005**	0.978	0.977	0.984	0.987	0.988
0.5	**32**	**0.0005**	0.978	0.982	0.984	0.987	0.987
0.7	**64**	**0.0005**	0.985	0.985			
0.5	**64**	**0.0005**	0.983	0.988	0.988		

From Table 5.1, it is clear that the most important hyperparameter is learning rate since the accuracy would not converge if a big learning rate were used. While the batch size and drop-out rate would not have a big influence on the results when the learning rate is small enough. Notably, 0.5 drop-out rate, 32 batch size, 0.001 learning rate and 0.7 drop-out rate, 64 batch size and 0.0005 learning rate have the most outstanding convergence speed. However, we prefer to have a better prediction accuracy than saving more time on training. Hence,

we trained a model with 0.5 drop-out rate, 16 batch size and 0.0005 learning rate:

	Car	Lamp	Table	Airplane	Chair
Car	1756	0	0	2	0
Lamp	4	435	29	4	3
Table	1	0	1545	3	6
Airplane	2	0	2	626	0
Chair	4	3	27	1	1376

Figure 5.3Confusion matrix of the pre-trained model.

5.1.2 Transfer Learning

Transfer learning is a technique for dealing with limited training data problem. While in industrial environment, it can hardly meet the requirement of obtaining massive training data since thousands of data need to be labelled manually. For this concern, applying transfer learning technique allows us to train an effective model without a big amount of data which could reduce the workload dramatically. Since transfer learning is a technique to reuse a trained model, it can transfer the knowledge from a related task that has already been learned for learning a new task [38]. The motivation of transfer learning is to prevent the neural network from learning low level features like lines and curves. Since low level features could highly potentially be reused for all input data. In this case, the input data could be used for learning the data-specific features. In section 6.3, we have visualized the extracted feature map and output from each layer.

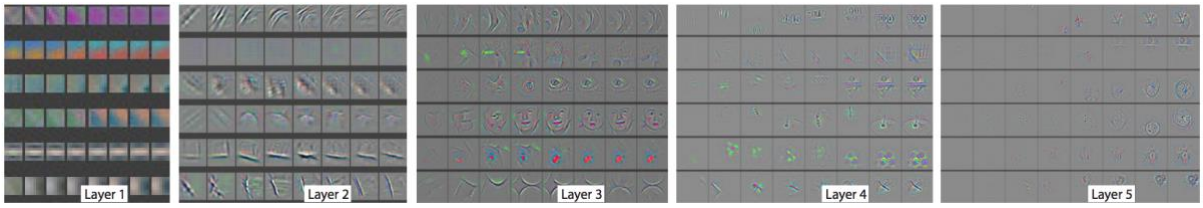


Figure 5.4What features have been learnt in different layers in 2D CNN [39].

This technique would reuse the trained weights and biases from the frontal layers of a pre-trained CNN model while the posterior layers may be reconstructed, then training on a new dataset. In section 6.2, we explored how much data is sufficient for transfer learning and on which layer would be the best to start training. It turns out that we shall train our model in first four layers and prepare at least 70 training objects for each class that could reach 96% prediction accuracy.

	Knife	Motorbike	Bag	Pipes	Rocket	Guitar	Laptop	Mug	Pistol	Cap	Warning Sign	Valve	Skateboard	Earphone
Knife	20	0	0	0	0	0	0	0	0	0	0	0	0	0
Motorbike	0	19	0	0	0	0	1	0	0	0	0	0	0	0
Bag	0	0	20	0	0	0	0	0	0	0	0	0	0	0
Pipes	0	0	0	20	0	0	0	0	0	0	0	0	0	0
Rocket	0	0	0	0	20	0	0	0	0	0	0	0	0	0
Guitar	0	0	0	0	1	19	0	0	0	0	0	0	0	0
Laptop	0	0	0	0	0	0	19	0	0	0	0	0	0	1
Mug	0	0	0	0	0	0	1	19	0	0	0	0	0	0
Pistol	0	0	0	0	0	0	0	0	20	0	0	0	0	0
Cap	0	0	0	0	0	0	0	0	0	20	0	0	0	0
Warning Sign	0	0	0	0	0	0	0	0	0	0	20	0	0	0
Valve	0	0	0	0	2	0	1	0	0	0	1	16	0	0
Skateboard	0	0	0	0	0	0	0	0	0	0	0	0	20	0
Earphone	0	0	0	0	0	0	2	1	0	0	0	0	0	17

Figure 5.5Confusion matrix of the test set results of transfer learning. Training 14 classes using pre-trained model. Trained with 70 data from each class and started training at the 3rd layer.

5.2 3D Object Detection

The next step is to detect objects in a scene. Not like pixel based RGB images or 3D RGB-D data. Point cloud is effectively a vector of x, y, z coordinates that the geometry would be changed even if the whole vector has been reordered. This

makes it hard to learn a feature that all related points may be randomly scattered in the file. Some researchers are using Recurrent Neural Network (RNN) [40], K-Nearest Neighbours (KNN) [41] to explore the relationship among points. However, those methods need another bunch of data for training and point clouds shall be semantically labelled [6]. Due to the limit number of data, we would prefer mathematic cluster approaches to avoid collecting and labelling more data. In this thesis, we will be using spatial clustering as the region proposal method for addressing object detection and localization.

5.2.1 Cluster for Region Proposal

The methodologies of 2D object detection can be divided to region proposal based and non-region-proposal based. Region proposal based works like R-CNN [42], Fast R-CNN [43] and Faster R-CNN [44] are classic methods in this area. Besides, Ian & Derek proposed a category-independent region proposal method which helps reducing additional training process for 2D image region proposals [45]. You Only Look Once (YOLO) is a bounding box based algorithm which has been proved to be faster than R-CNN and Fast R-CNN [46]. Theoretically, those algorithms could be fine-tuned for voxel grids since they are used for pixel-based 2D images. However, point clouds can represent any sized data which makes it hard to decide what size of voxel grids should be used.

In the area of 3D object segmentation, prior works like PointNet [40] and Semantic3D [6] have successfully trained their neural network model to segment a point cloud semantically into small pieces of point clouds with billions of labelled points. Those methods use supervised learning to figure out the

relation among points while we prefer an unsupervised approach to minimize human effort on labelling. Cluster algorithm is used to determine which points belong together. Here, we would introduce three cluster algorithms, K-Means, MeanShift and DBSCAN.

5.2.1.1 K-Means

K-Means clustering algorithm aims to divide an object into k clusters where a centroid could be found for each cluster [47]. The algorithm would minimize the Within-Cluster Sum of Squares (WCSS) between all the points and its centroid. The objective is to minimize the variance of S_i:

$$\arg\min_{S} \sum_{i=1}^{k} \sum_{x \in s_i} \|x - \mu_i\|^2 = \arg\min_{S} \sum_{i=1}^{k} |S_i| \, Var \, S_i$$

where x is a vector of input points and S is a set that contains k clusters. The downside of k-means algorithm is that we need to define the specific number of clusters before clustering while we want an algorithm to be intelligent enough to figure it out automatically.

5.2.1.2 Mean Shift

The mean shift algorithm is similar to k-means that is called likelihood mean shift [48]. For mean shift, a kernel function $K(x_i - x)$ would be used and there are many different kernels like Epanechnikov Kernel, Uniform Kernel and Normal Kernel. Mean shift assumes that all input data is sampled from the Probability Density Function (PDF) of the given kernel while the algorithm would estimate the local maxima. Assume that initial estimate of local maxima is x, the next mean shift would be:

$$m(x) = \frac{\sum_{x_i \in N(x)} K(x_i - x)\, x_i}{\sum_{x_i \in N(x)} K(x_i - x)} - x$$

In order to find the local maxima, mean shift would be performed recursively and it would stop if the gradient is closed to zero. The most import parameter in mean shift algorithm is the bandwidth on which the whole mean shift procedure relies. It can be set as a fixed value or using other algorithms like balloon and sample point estimators [49]. Take an example of a density estimator of fixed bandwidth *h*:

$$f(x) = \sum_i K(x - x_i) = \sum_i k\left(\frac{\|x - x_i\|^2}{h^2}\right)$$

where x_i are the input points and *k(r)* is the kernel function.

Mean shift clustering algorithm is used in computer vision and image processing. It is an efficient algorithm for dealing with 2D computer vision problems [50] which is used to segment color images by seeking for the modes and peeks of the pixel intensities. Meanwhile, it fixes the drawback of K-Means that the centroids of input data could be calculated instead of setting the number of clusters.

5.2.1.3 DBSCAN

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is a data clustering algorithm that groups the nearby points together [51]. There are two important parameters in DBSCAN algorithm: ϵ and the minimum number of points for a dense region [52]. ϵ is the threshold to determine if two points belong to the same cluster or not. It will compare the Euclidean Distance between points with ϵ and mark the visited points as cluster or noise which

means it is not a deterministic algorithm, since the results may differ from each other if input the data with different order. Hence, it would also be a good algorithm to remove the outliers from the input point cloud. The pseudocode of DBSCAN as follows [52]:

```
DBSCAN(DB, dist, eps, minPts) {
   C = 0
   for each point P in database DB {
      if label(P) ≠ undefined then continue
      Neighbors N = RangeQuery(DB, dist, P, eps)
      if |N| < minPts then {
         label(P) = Noise
         continue
      }
      C = C + 1
      label(P) = C
      Seed set S = N \ {P}
      for each point Q in S {
         if label(Q) = Noise then label(Q) = C
         if label(Q) ≠ undefined then continue
         label(Q) = C
         Neighbors N = RangeQuery(DB, dist, Q, eps)
         if |N| ≥ minPts then {
            S = S ∪ N
         }
      }
   }
}

RangeQuery(DB, dist, Q, eps) {
   Neighbors = empty list
   for each point P in database DB {
      if dist(Q, P) ≤ eps then {
         Neighbors = Neighbors ∪ {P}
      }
   }
   return Neighbors
}
```

5.2.1.4 Comparison

A quick comparison would be carried out to show the performance of each clustering algorithm. Those three algorithms are tested on two different point clouds, the first one contains 5 separate objects without overlapping and the second one has two jointed objects. All algorithms are using the same settings apart from the number of clusters in K-Means. Parameter settings as follows:

a) DBSCAN is using 0.02 ϵ and 10 min points.

b) Mean Shift is using K-nearest neighbour to auto-estimate the bandwidth for 300 iterations.

c) K-Means need to set the number of clusters to 5 and 2 separately as per the point cloud and would run for 50 iterations.

Testing cluster algorithms on point cloud (a):

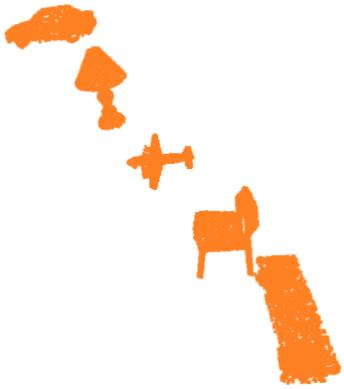
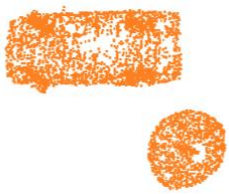


Figure 5.6 Origin point cloud (a) for clustering. This point cloud contains 5 separated objects. Top to bottom, left to right are car, lamp, airplane, chair, table respectively.

DBSCAN	Mean Shift	K-Means
		
		
		

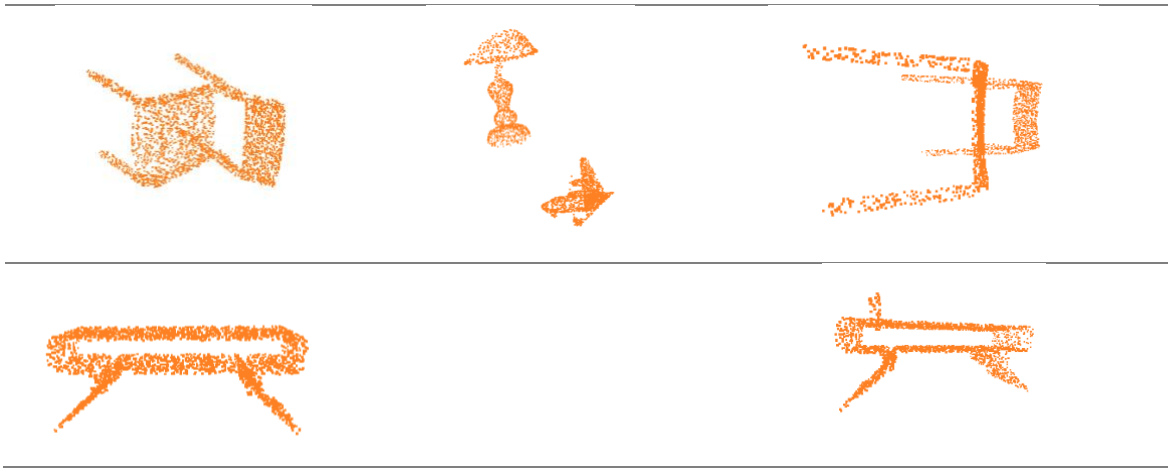


Figure 5.7 Results of clustering Figure 5.6. DBSCAN and K-Means have the best performance that fully separated 5 objects while the mean shift found the wrong centroids within the point cloud.

Testing cluster algorithms on point cloud (b):

Figure 5.8Origin point cloud (b) for clustering. This point cloud contains two jointed objects. A lamp on a table.

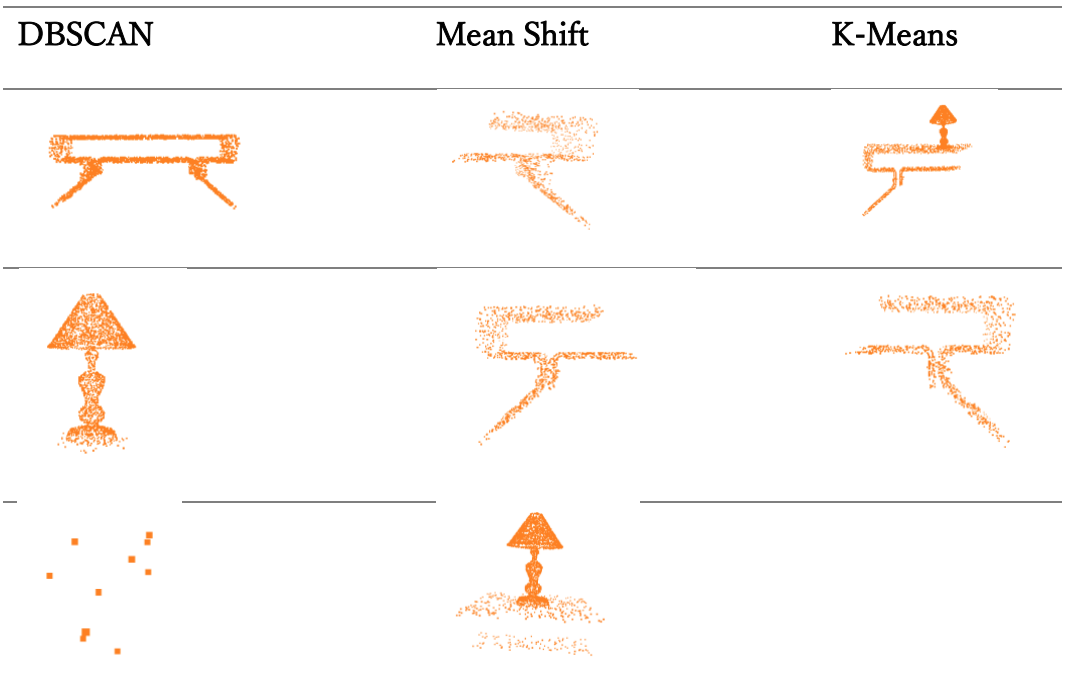


Figure 5.9 Result of clustering. DBSCAN has the best performance that separates the table and lamp, but another noisy point cloud has been separated out as well. Mean shift separates 3 point clouds, two half tables and a noisy lamp. K-Means finds a lamp on a half table and another half table.

Overall, for the provided examples, DBSCAN algorithm has the best performance while the others may be working well in other cases. Since it is unsupervised learning algorithms that could not guarantee the results and performance. Meanwhile, all parameters in those algorithms need to be fine-tuned and tested for different cases due to there is hardly to obtain a suite of parameters for all the point clouds. Hence, the segmented objects could be fed back to the trained classification model to obtain the proper label.

6 Experiments

6.1 Occupancy Grid

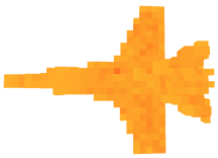
Occupancy grid is a representation of point cloud in a specific sized space. The prior work of VoxNet [2] shows the different representations of occupancy grids have similar performances on both point cloud data and RGBD data while the data comes from Sydney Urban Objects dataset [53] and NYUv2 [54] respectively. Their result shows that density grid performs slightly better than binary grid while binary grid performs slightly better than hit grid. We will compare their performance with ShapeNet [28] data and their noise tolerance abilities among those aforementioned occupancy grids.

In this experiment, we were using selected geometries whose data size is above 2000. This turns out to obtain 5 classes, including chair, airplane, lamp, table, car. Then all of the selected data would be transformed into density grids, binary grids and hit grids for 3 different data representations. The data would be split to 80% training data and 20% testing data. We would train 3 different models using those transformed data with same hyperparameter settings, which is 0.0005 learning rate, 0.5 drop-out rate, 16 batch-size and 5 epochs. For every training, the weights and bias would be initialized from the same seed value which is 1234 in our case.

6.1.1 Voxel Grid Size

In this experiment, we randomly picked 5 geometry point clouds and converted them into 16 x 16 x 16, 32 x 32 x 32 and 48 x 48 x 48 occupancy grids to select the best grid size for our data.

Table 6.1Comparison among different grid size of voxel grid

Label	Point Cloud	16 x 16 x 16	32 x 32 x 32	48 x 48 x 48
AIR-PLANE				
CAR				
TABLE				
LAMP				
CHAIR				

From Table 6.1, all geometries could be represented well by any sized grid apart from car. The car can hardly be recognized with 16 x 16 x 16 grid. There are many holes on the surface if using 48 x 48 x 48 grid. Recall that all of our open sourced point clouds have merely 8000 points, hence higher grid size may perform better for a bigger point cloud. Therefore, we would use 32 x 32 x 32 grid for training our model.

6.1.2 Performance

In this experiment, we trained 3 CNN models under exact same settings with different occupancy grid representations. Those occupancy grids are generated from the same point clouds while the order may various from each other because of the data shuffling. Each dataset contains 28,847 objects and the last 20% (5,829 objects) are used for testing. All results below are obtained using testing data only.

Table 6.2 Performance of different occupancy grid representations

OCCUPANCY GRID	ACCURACY
DENSITY GRID	98.4%
BINARY GRID	98.1%
HID GRID	98.1%

Table 6.3 Confusion Matrix after training on Density Occupancy Grid

	Car	Lamp	Table	Airplane	Chair
Car	1756	0	0	2	0
Lamp	4	435	29	4	3
Table	1	0	1545	3	6
Airplane	2	0	2	626	0
Chair	4	3	27	1	1376

Table 6.4 Confusion Matrix after training on Binary Occupancy Grid

	Car	Lamp	Table	Airplane	Chair
Car	1749	1	0	6	2
Lamp	2	440	24	5	4
Table	0	1	1547	2	5
Airplane	0	2	2	626	0
Chair	3	4	44	0	1360

Table 6.5 Confusion Matrix after training on Hit Occupancy Grid

	Car	Lamp	Table	Airplane	Chair
Car	1730	4	14	11	1
Lamp	3	374	15	1	3
Table	0	2	1393	0	38
Airplane	1	5	1	872	1
Chair	0	4	14	1	1353

From Table 6.2, we can see that prediction accuracy of density grid has a slightly better performance compare to others. From [Table 6.3, Table 6.4, Table 6.5], CNNs are more inclined to predict a lamp and a chair as a table. In some cases, the similarities of chairs and tables may confuse the neuron network. This is more understandable to lamps since it has less training data (2,018 objects) compare to tables (8,509 objects). Therefore, a balanced dataset may lead to a better performance.

6.1.3 Noise Tolerance

In this experiment, we would test the noises tolerance ability of different occupancy grids on trained models from section 6.1.2. Meanwhile, we produced 20 noisy datasets by adding Gaussian Noise on each individual point cloud with 0.5 variance. This turns out 20 datasets with from 1% to 20% noise ratio. We are expecting to specify which type of occupancy grid has the best performance under different noise ratio.

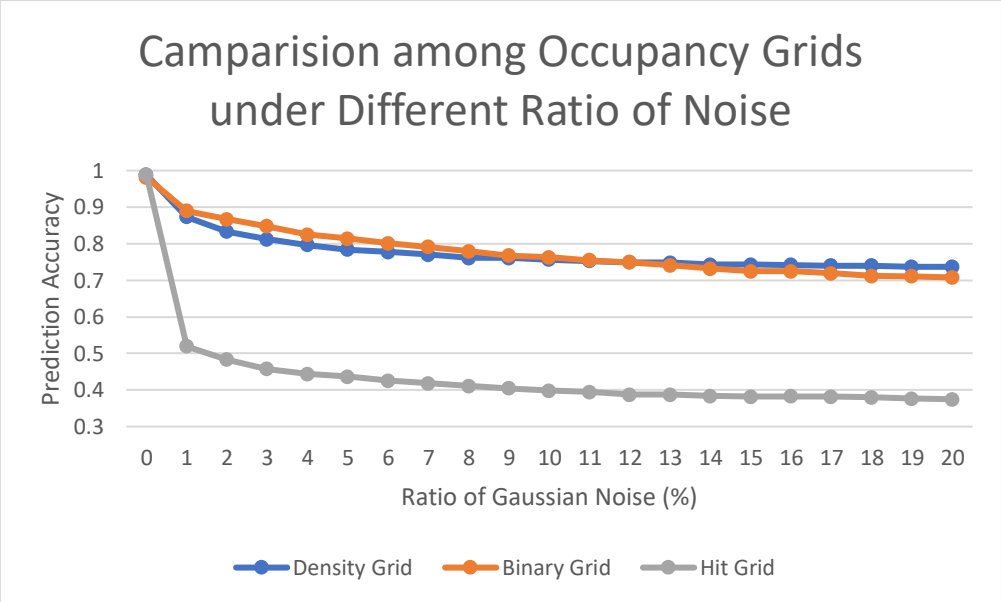


Figure 6.1Camparision among Occupancy Grids under Different Ratio of Noise

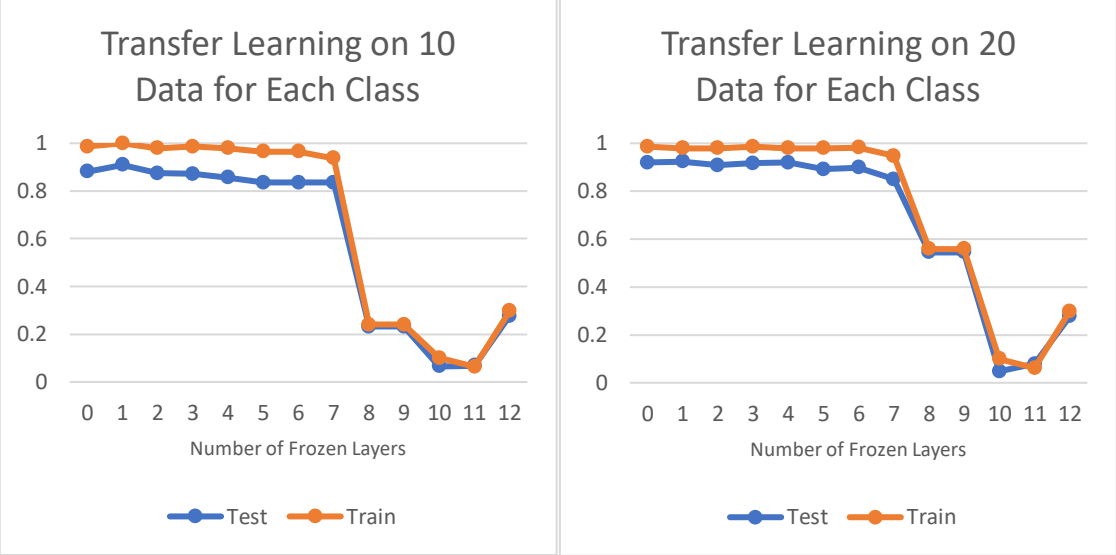
From Figure 6.1, all of those representations have similar performances on the clean dataset that reaches around 98% accuracy while it starts decrease its accuracy dramatically when the testing data comes to be noised. Hit grid has the worst noise tolerance since it drops straight away from 1% noised data, while the other two have a smoother curve. Binary grid shows its noise tolerance ability until 12% noise ratio, while the density grid takes over when the noise ratio goes up from 12%. Since binary grid would only measure the shape of point cloud, slight noised data would not affect their coarse shapes in a lower noise ratio whereas noises are not distinguishable in a higher ratio since all occupied cells are equally to one. Oppositely, density grid measures the relative ratio of points from voxel to voxel where Gaussian Noise would affect it more than binary grid since it would change the whole ratio distribution of voxel grid. However, the slight performance differences between binary grids and density grids due to the noise reduction nature of convolution and max pooling operations have different impact on those representations. Binary grids perform better in a lower noise ratio while density grids are better in a high noise ratio circumstance.

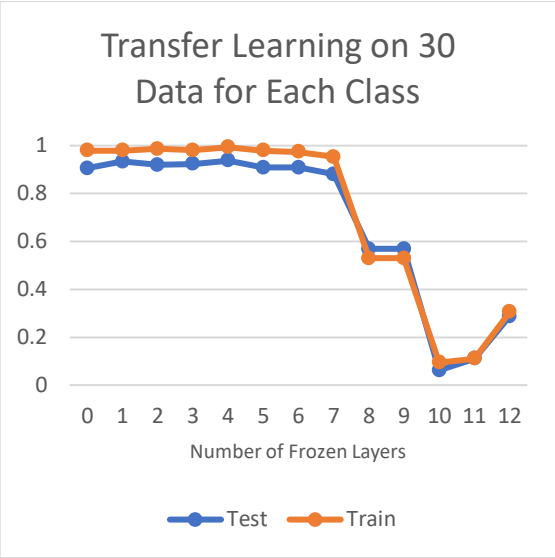
6.1.4 Summary

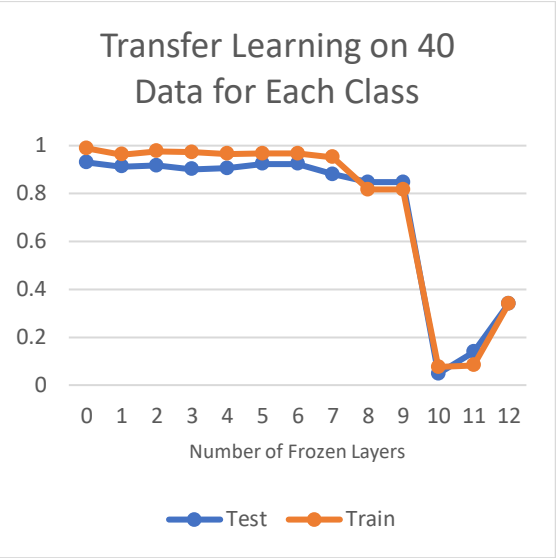
Some point clouds may be taken carefully that can be guaranteed to have a clean and full-scaled representation. In that case, binary grids are sufficient for neural network to learning the geometry from clean shapes. While clean and full-scaled representations are normally hard to obtain in our case, we may be not allowed or able to reach some places in a power plant to have a comprehensive scanning. 32 x 32 x 32 density grids may be the best under that circumstances.

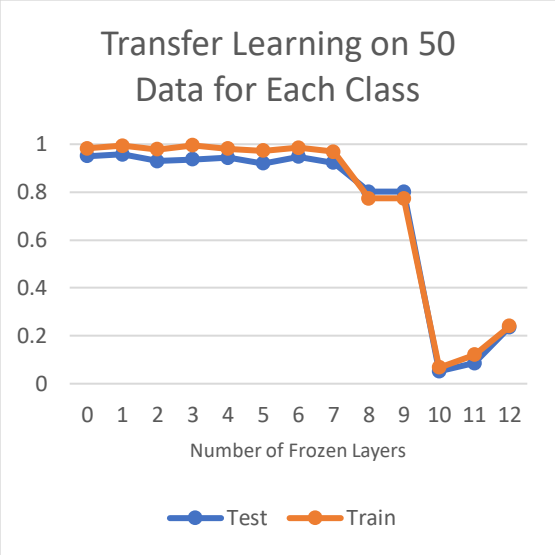
6.2 Transfer Learning

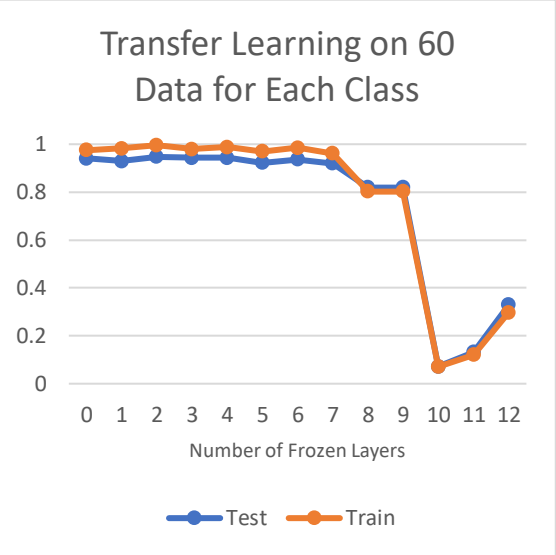
In this experiment, we reused a pre-trained model trained with 5 classes and all point clouds are represented by density occupancy grid representation. We aimed to train a new model for classifying an object in 14 classes by applying transfer learning technique. We would make 10 datasets containing all 14 classes with different data size. The size of data in each dataset would be increased by 10. A test dataset would be separated out with 20 point clouds in each class. In terms of applying transfer learning, we would overwrite the last layer of the pre-trained model for classifying 14 classes instead of 5. All hyperparameters were defined as same as the result from grid search (section 5.1.1). Then the frontal layers of the pre-trained model would be frozen one after one. Therefore, we have 280 test point clouds and we would train our model by feeding 140, 280, 420, ⋯, 1400 point clouds as input for each number of frozen layers.

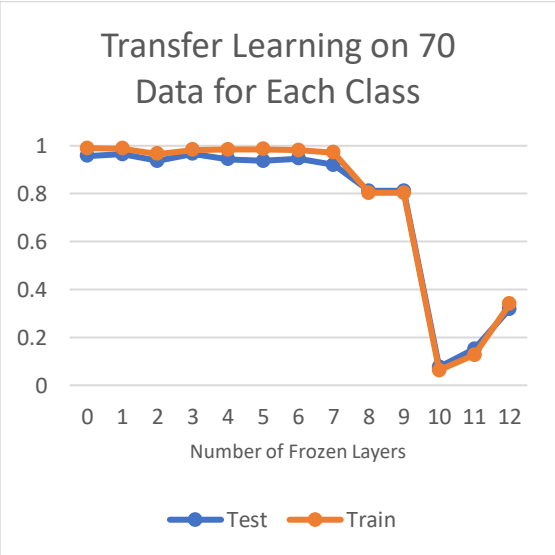


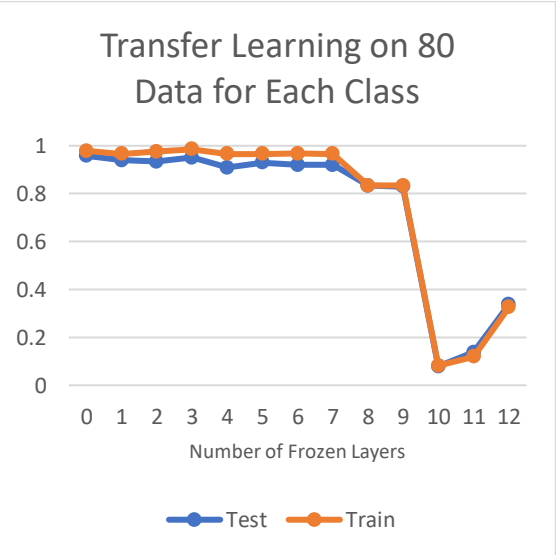

Transfer Learning on 30
Data for Each Class
1
0.8
0.6
0.4
0.2
0
0 1 2 3 4 5 6 7 8 9 10 11 12
Number of Frozen Layers
Test
Train

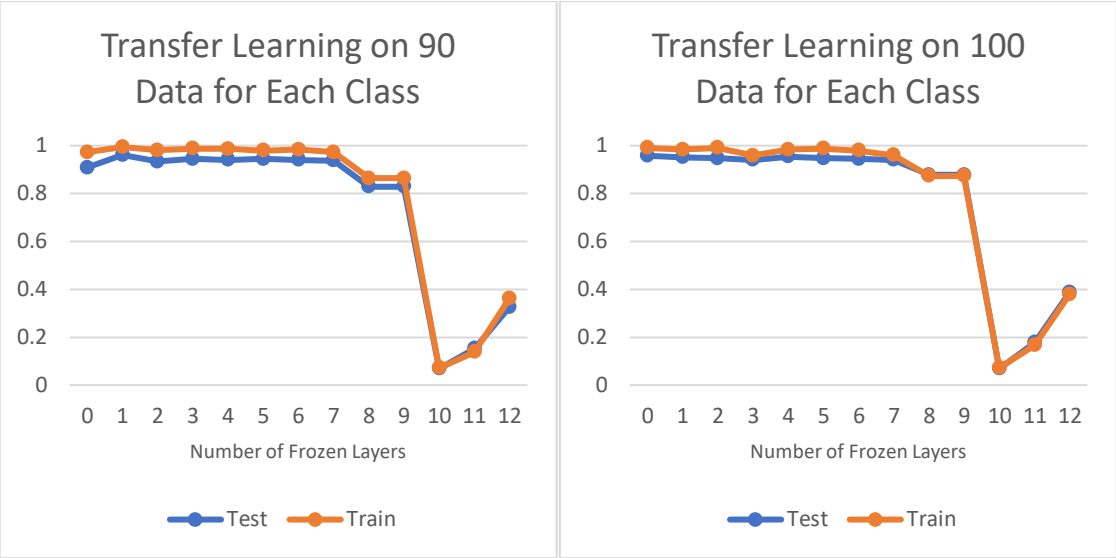

Transfer Learning on 40
Data for Each Class
1
0.8
0.6
0.4
0.2
0
0 1 2 3 4 5 6 7 8 9 10 11 12
Number of Frozen Layers
Test
Train


Transfer Learning on 50
Data for Each Class
1
0.8
0.6
0.4
0.2
0
0 1 2 3 4 5 6 7 8 9 10 11 12
Number of Frozen Layers
Test
Train


Transfer Learning on 60
Data for Each Class
1
0.8
0.6
0.4
0.2
0
0 1 2 3 4 5 6 7 8 9 10 11 12
Number of Frozen Layers
Test
Train


Transfer Learning on 70
Data for Each Class
1
0.8
0.6
0.4
0.2
0
0 1 2 3 4 5 6 7 8 9 10 11 12
Number of Frozen Layers
Test
Train


Transfer Learning on 80
Data for Each Class
1
0.8
0.6
0.4
0.2
0
0 1 2 3 4 5 6 7 8 9 10 11 12
Number of Frozen Layers
Test
Train



Here is the corresponding operation for each layer number:

0 – Input Layer

1 – Convolution Layer

2 – Convolution Layer

3 – Max Pooling

4 – Convolution Layer

5 – Convolution Layer

6 – Max Pooling

7 – Convolution Layer

8 – Convolution Layer

9 – Max Pooling

10 – Batch Normalization

11 – Fully Connected Layer

12 – Dropout Layer

13 – Output Layer

Here are the major findings from this experiment:

- **The result of training at 0 frozen layers is surprisingly good in this experiment.** Training at 0 frozen layers means to train a new model without reusing any pre-trained weight and bias. It can reach 88.1% of accuracy even with only 10 point clouds for each class for training. It may because of the normalized density voxel grids of those point clouds are strongly various from each other which can accelerate the convergence speed of training. Meanwhile, those 3D point clouds do not have any

background like 2D images, which means the neural network does not need to put any effort on distinguishing the useful features from background noise.

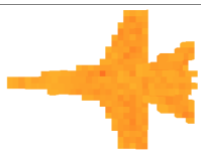
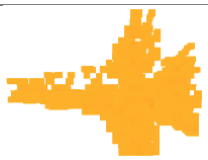
- **Frozen the first four layers are the best for transfer learning**. No matter how much input data has been fed, the highest accuracy would always appear on one of the first four layers. Mostly, the more input data we have, the higher accuracy would be more potentially appeared on training from later layers. The accuracy is quite stable at around 94% for 100 point clouds for each class if we start training on any layer before the seventh layer, whereas the accuracy starts to drop after frozen the first layer for 10 point clouds for each class.
- **We need at least 70 point clouds for each class for transfer learning**. There is an overfitting problem if we trained our model with less than 70 data for each class. Training accuracy and testing accuracy would be narrowed down to 2% if we train our model with more than 70 point clouds from each class. The highest prediction accuracy is 96.6% if we start training on the 3rd layer with 70 point clouds from each class.
- **Only training on the last three layers may not help that much**. From the figures, we can see that the accuracy of only training on the last output layer performs is better than training with drop-out layer, fully connected layer and batch normalization layer. Also, refers to the discussion of internal covariant shift in section 2.3.2 that drop out layer may be useless

if we have batch normalization. Therefore, the drop out layer may be not helping for small dataset.

6.3 Layer Output

In this experiment, we took the point cloud geometry of an airplane as an example to feed into the trained CNN model. We would pick one kernel filter from each convolution layer and to visualize the process of handling 3D volumetric data in CNN.

Table 6.6 The extracted output voxel grids from each layer.

Layer	Kernel	Output Front	Output Side	Output Size
				32x32x32
Conv_1				32x32x32
ReLU				32x32x32
Conv_2				32x32x32
ReLU				32x32x32

Pool_3				16x16x16
Conv_4				16x16x16
ReLU				16x16x16
Conv_5				16x16x16
ReLU				16x16x16
Pool_6				8x8x8
Conv_7				8x8x8
ReLU				8x8x8

Conv_8		8x8x8
ReLU		8x8x8
Pool_9		4x4x4

In our CNN model (section 3.4), there are 512 output feature maps would be obtained as the output in pooling layer 9. Hence, it will eventual be transformed into 32,768 features detected then being fully connected to calculate the score for each class.

7 Conclusion

The major objective of this thesis is to develop an application for object recognition within a 3D point cloud. First, we have reviewed the state-of-art technologies then demonstrated the used hardware and software. After obtained data, we looked through the class distribution of the data sources and gave some solutions of data augmentation to balance the distribution. All the data has been normalized and encapsulated to HDF5 files. Next, we trained our CNN model and applied transfer learning technique that completed the object classification task. Then we used an unsupervised clustering approach to segment the objects and fed back to the classification model performing the prediction. Hence, the basic requirement has been achieved while there are still some future works to do:

- **Using more data augmentation strategy.** In this work, a few data augmentation method were used for obtaining more data in a cheaper way. While in 2D image augmentation, some works use Generative Adversarial Nets (GANs) [31] which uses style transformation to generate new images. More data augmentation methods should be tried if it is hard to obtain more data, since it helps to reduce overfitting.
- **Get more training proper labelled data.** As the research of PointNet [40], point clouds segmentation could be carried out in a supervised learning method which is using Recurrent Neural Network (RNN) to figure out the relationship among points. All points in point clouds need to be labelled for training which requires a high workload. This supervised

learning approach achieved 83.9% of segmentation accuracy which is better than the unsupervised approach.

- **Consider the colours in point clouds**. In this thesis, all experiments are based on only the coordinates of the point clouds. The reason of not taking colours into account is that every voxel can have only one suite of RGB colours, which will lose more information if there are various coloured points in a voxel in volumetric representation. Besides, for clustering, taking colours into account may be able to improve the performance of the clustering algorithm. For example, mean shift algorithm is used to segment a 2D image by its colour space [50].
- **Measure the performances of clustering algorithms**. For supervised learning, every point would be labelled to a specific class which makes it easy to measure the performance of the segmentation algorithm. While it is not intuitive to evaluate the results for unsupervised clustering algorithms. With a robust clustering performance measurement method, techniques like grid search and random search could be used to find the best parameters for clustering.

Bibliography

[1] A. Garcia-Carcia, F. Gomez-Donoso, J. Garcia-Rodriguez, S. Orts-Escolano, M. Cazorla and J. Azorin-Lopez, "PointNet: A 3D Convolutional Neural Network for Real-Time Object Class Recognition," in *Neural Networks (IJCNN), 2016 International Joint Conference on.*, Vancouver, BC, Canada, 2016.

[2] M. Daniel and S. Sebastian, "VoxNet- A 3D Convolutional Neural Network for Real-Time Object Recognition," in *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, Hamburg, Germany, 2015.

[3] Wikipedia, "Point Cloud," [Online]. Available: https://en.wikipedia.org/wiki/Point_cloud. [Accessed 17 11 2017].

[4] "What is RGB-D Camera (Depth Sensor)," IGI Global, [Online]. Available: https://www.igi-global.com/dictionary/rgb-d-camera-depth-sensor/50427. [Accessed 27 11 2017].

[5] M. Rastegari, V. Ordonez, J. Redmon and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," *arXiv preprint arXiv:1603.05279* , 2016.

[6] T. Hackel, N. Savinov, L. Ladicky, J. D. Wegner, K. Schindler and M. Pollefeys, "Semantic3D.net: A new Large-scale Point Cloud Classification Benchmark," *arXiv preprint arXiv:1704.03847,* 2017.

[7] A. Canziani, A. Paszke and E. Culurciello, "An Analysis of Deep Neural Network Models for Practical Applications," *arXiv preprint arXiv:1605.07678,* 2016.

[8] Wikipedia, "Deep Neural Networks," [Online]. Available: https://en.wikipedia.org/wiki/Deep_learning#Deep_neural_networks . [Accessed 27 11 2017].

[9] V. Sharma, S. Rai and A. Dev, "A Comprehensive Study of Artificial Neural Networks," *International Journal of Advanced Research in Computer Science and Software Engineering,* vol. 2, no. 10, pp. 278-284, 10 2012.

[10] A. Krizhevsky, I. Sutskever and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in Neural Information Processing Systems 25,* p. 1106–1114, 2012.

[11] Wikipedia, "Gaussian Noise," Wikipedia, 28 August 2017. [Online]. Available: https://en.wikipedia.org/wiki/Gaussian_noise. [Accessed 14 12 2017].

[12] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard and L. D. Jackel, "Backpropagation Applied to Handwritten Zip Code Recognition," *Neural Computation,* vol. 1, no. 4, pp. 541-551, 1989.

[13] J. Wu, "Introduction to Convolutional Neural Networks," LAMDA Group, Nanjing, China, 2017.

[14] CS231n, "CS231n Convolutional Neural Networks for Visual Recognition," Stranford University, 28 11 2017. [Online]. Available: http://cs231n.github.io/convolutional-networks/. [Accessed 24 12 2017].

[15] A. Krizhevsky, I. Sutskever and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Communications of the ACM* , vol. 60, no. 6, pp. 84-90, 2017.

[16] R. Fergus, "Neural Networks MLSS 2015 Summer School," Facebook AI Research , 2015. [Online]. Available:

http://mlss.tuebingen.mpg.de/2015/slides/fergus/Fergus_1.pdf. [Accessed 24 12 2017].

[17] J. T. Springenberg, A. Dosovitskiy, T. Brox and M. Riedmiller, "Striving For Simplicity: The All Convolutional Net," ICLR, Freiburg, Germany, 2015.

[18] P. Golik, P. Doetsch and H. Ney, "Cross-Entropy vs. Squared Error Training: A Theoretical and Experimental Comparison.," *INTERSPEECH,* pp. 1756-1760, 2013.

[19] Wikipedia, "Backpropagation," Wikipedia, 20 12 2017. [Online]. Available: https://en.wikipedia.org/wiki/Backpropagation. [Accessed 29 12 2017].

[20] S. Ruder, "An overview of gradient descent optimization algorithms," arXiv:1609.04747v2, 2017.

[21] A. S. Walia, "Types of Optimization Algorithms used in Neural Networks and Ways to Optimize Gradient Descent," Medium, 10 6 2017. [Online]. Available: https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f. [Accessed 30 12 2017].

[22] a. G. H. T. Tieleman, "RMSProp: Divide the gradient by a running average of its recent magnitude.," COURSERA: Neural Networks for Machine Learning, 2012.

[23] D. P. Kingma and J. L. Ba, "ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION," in *ICLR,* 2015.

[24] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research,* no. 15, pp. 1929-1958, 2014.

[25] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training b y Reducing Internal Covariate Shift," arXiv:1502.03167v3, 2015.

[26] "What Is GPU-Accelerated computing?," Nvidia Corporation, [Online]. Available: http://www.nvidia.com/object/what-is-gpu-computing.html. [Accessed 27 11 2017].

[27] Wikipedia, "TensorFlow," [Online]. Available: https://en.wikipedia.org/wiki/TensorFlow. [Accessed 27 11 2017].

[28] A. X. Chang, T. Funkhouser, L. 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," arXiv:1512.03012, 2015.

[29] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks For Large-Scale Image Recognition," in *ICLR,* 2015.

[30] J. Brownlee, "Why One-Hot Encode Data in Machine Learning?," Machine Learning Process, 28 7 2017. [Online]. Available: https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/. [Accessed 28 12 2017].

[31] J. Wang and L. Perez, "The Effectiveness of Data Augmentation in Image Classification using Deep Learning," Stanford University, 2017.

[32] L. Taylor and G. Nitschke, "Improving Deep Learning using Generic Data Augmentation," arXiv:1708.06020v1, 2017.

[33] Wikipedia, "Hierarchical Data Format," Wikipedia, 12 12 2017. [Online]. Available: https://en.wikipedia.org/wiki/Hierarchical_Data_Format. [Accessed 25 12 2017].

[34] P. J. Besl and R. C. Jain, “Three-dimensional object recognition,” *ACM Computing Surveys (CSUR) - Annals of discrete mathematics, 24,* vol. 17, no. 1, pp. 75-145, March 1985.

[35] V. Hegde and R. Zadeh, “FusionNet: 3D Object Classification Using Multiple Data Representations,” arXiv:1607.05695, 2016.

[36] J. Bergstra and Y. Bengio, “Random Search for Hyper-Parameter Optimization,” *Journal of Machine Learning Research,* vol. 13, pp. 281-305, 2012.

[37] Y. Bengio, “Practical Recommendations for Gradient-Based Training of Deep Architectures,” arXiv:1206.5533v2, 2012.

[38] L. Torrey and J. Shavlik, “Transfer Learning,” in *Handbook of Research on Machine Learning Applications*, 2009.

[39] M. D. Zeiler and R. Fergus, “Visualizing and Understanding Convolutional Networks,” in *CoRR,* USA, 2014.

[40] C. R. Qi, H. Su, K. Mo and L. J. Guibas, “PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation,” arXiv:1612.00593v2, 2017.

[41] A. Savchenkov, “Generalized Convolutional Neural Networks for Point Cloud Data,” arXiv:1707.06719v1 , 2017.

[42] R. G. J. D. T. D. J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation Tech report,” arXiv:1311.2524v5, 2014.

[43] R. Girshick, “Fast R-CNN,” arXiv:1504.08083, 2015.

[44] S. Ren, K. He, R. Girshick and J. Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”.

[45] I. Endres and D. Hoiem, “Category Independent Object Proposals,” ECCV, 2010.

[46] J. Redmon and A. Farhadi, “YOLO9000: Better, Faster, Stronger,” arXiv:1612.08242, 2016.

[47] WikiPedia, “k-means clustering,” WikiPedia, 27 12 2017. [Online]. Available: https://en.wikipedia.org/wiki/K-means_clustering. [Accessed 28 12 2017].

[48] M. A. Little and N. S. Jones, “Generalized Methods and Solvers for Piecewise Constant Signals: Part I,” *Proceedings of the Royal Society A,* vol. 467, no. 2135, p. 3088–3114, 2011.

[49] “The Variable Bandwidth Mean Shift and Data-Driven Scale Selection,” in *Computer Vision, 2001. ICCV 2001,* Vancouver, BC, Canada, Canada, 2001.

[50] D. Comaniciu, V. Ramesh and P. Meer, “Real-Time Tracking of Non-Rigid Ob jects using Mean Shift,” *IEEE Conf. Comp. Vis. Patt. Recogn.,* vol. 2, pp. 142-149, 2000.

[51] M. Ester, H.-P. Kriegel, J. Sander and X. Xu, “A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise,” *AAAI,* pp. 226-231, 1996.

[52] WikiPedia, “DBSCAN,” Wikipedia, 9 12 2017. [Online]. Available: https://en.wikipedia.org/wiki/DBSCAN#cite_note-minpts-5. [Accessed 28 12 2017].

[53] A. Quadros, J. Underwood and B. Douillard, “An occlusion-aware feature for range images,” in *ICRA,* Saint Paul, MN, USA, May 14-18 2012.

[54] P. K. N. Silberman, D. Hoiem and R. Fergus, “Indoor segmentation and support inference from rgbd images,” in *ECCV,* 2012.

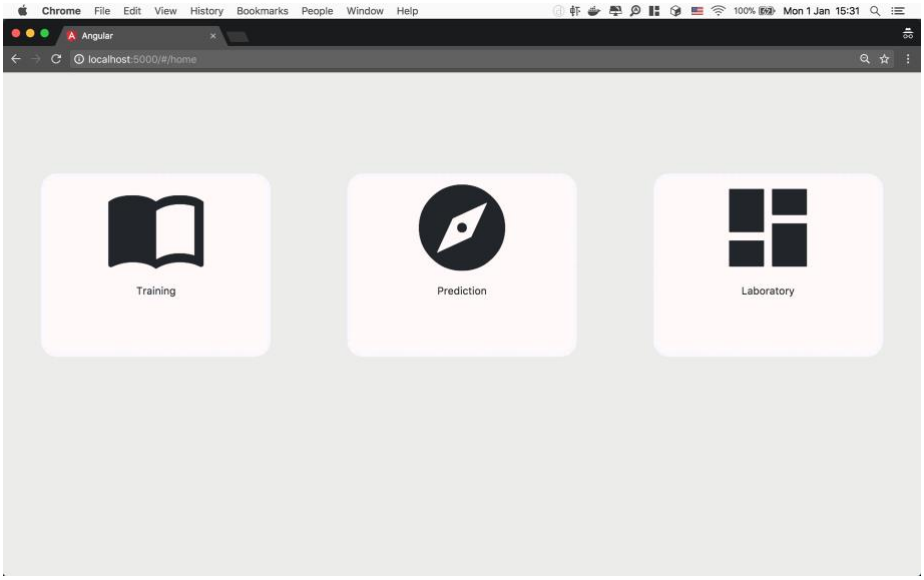
Appendix

I Web Application

This is the product for the company to train, predict and trial on 3D CNN. The purpose of it is to simplify the process, to hide the technique details and concepts from users.

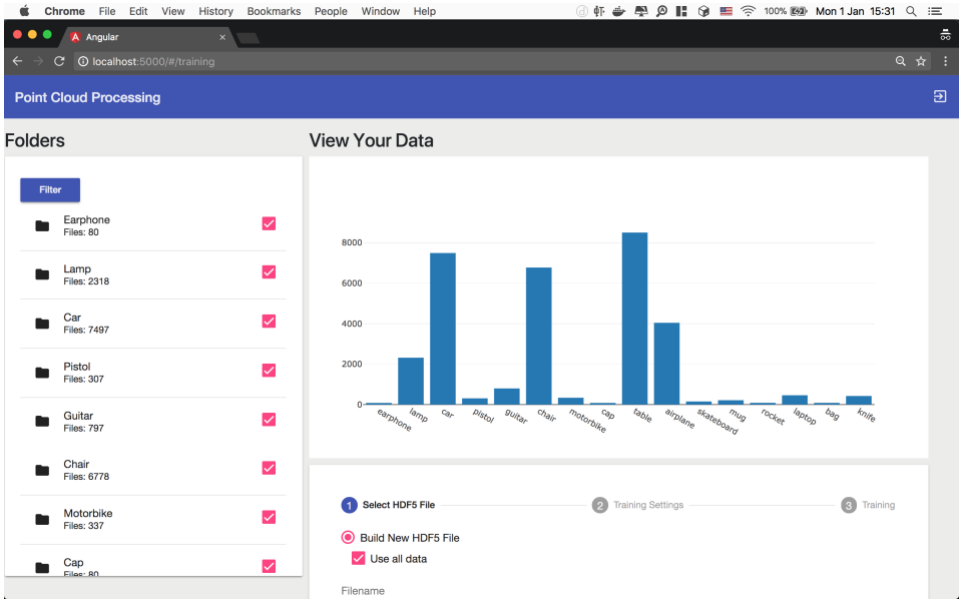
I.i Home Page.

This application contains three main sections, training, predicting and laboratory for visualizing the process in layers.



I.ii Training Page.

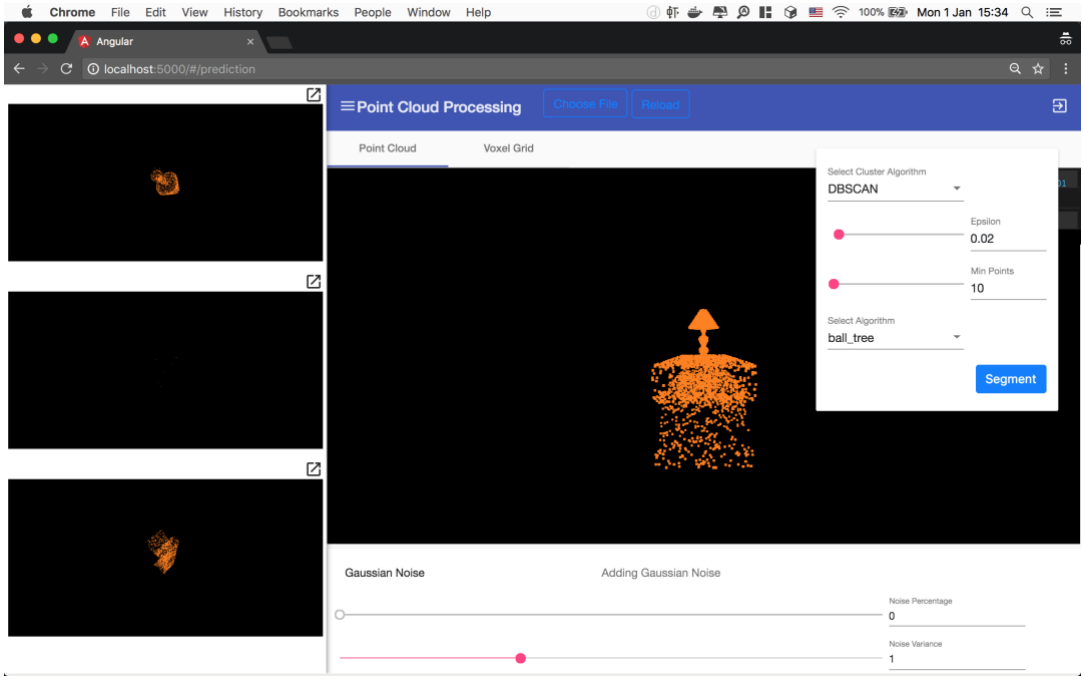
In this page, users can pre-process their data, adjust hyperparameters and start training.



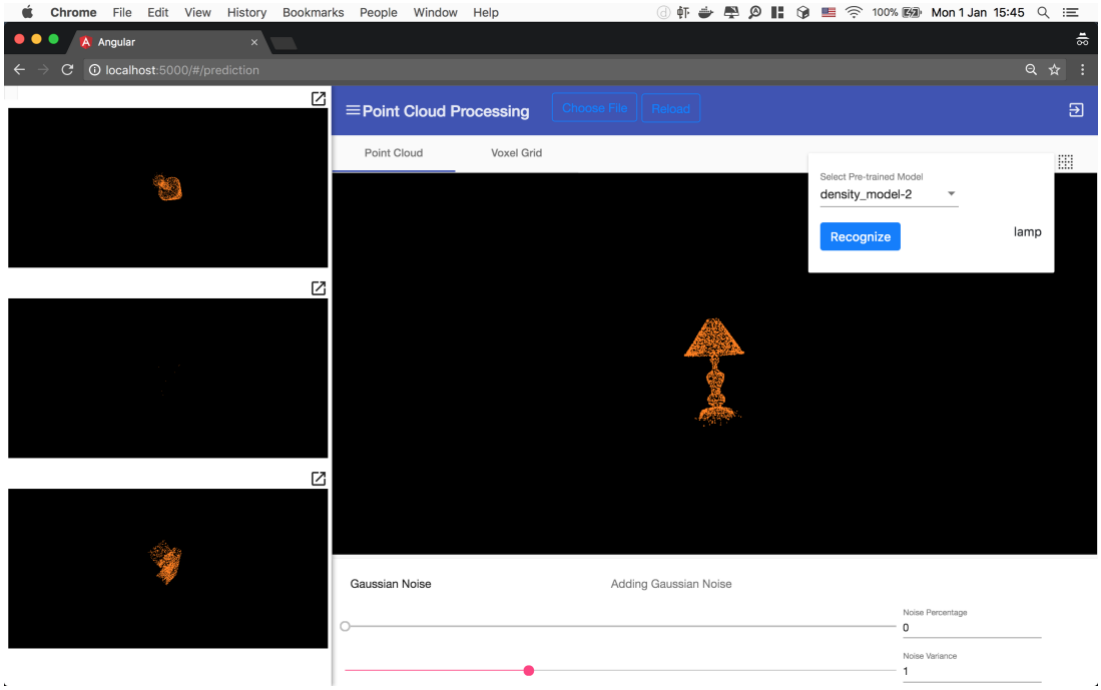
I.iii Prediction Page.

In this page, users can perform clustering and prediction.

a. Clustering.

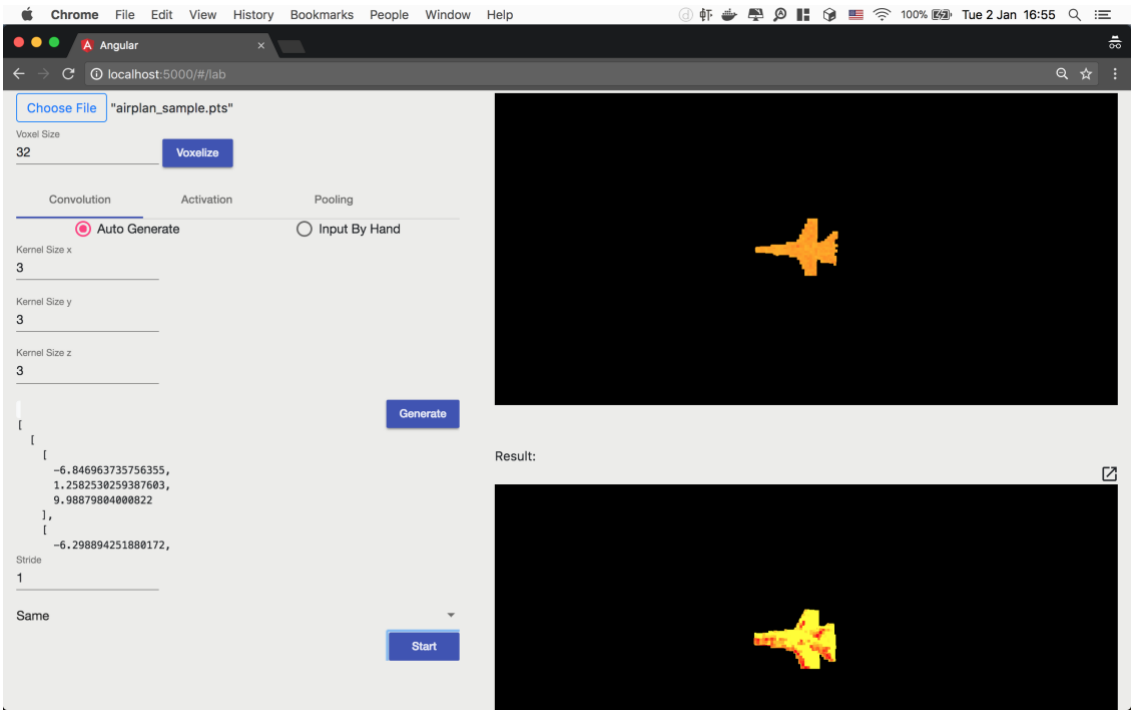


b. Prediction.



I.iv Laboratory Page.

In this page, users can visualize the convolution, activation and max pooling operations.



II Data Processing Implementation.

II.i Data loader.

```
def find_data(data_path, max_file_num=None, folder_filter=(None, None)):
    """
    Find file in each folder according to the 'data_path'.
    Giving the max number of files via `max_file_num`, it will read first `max_file_num` in each
folder. Read all if there is no enough file inside.
    `folder_filter` is a tuple like (100, 2000) which indicates the number of files will between
100 and 2000.
    """
    min_th, max_th = folder_filter
    if max_file_num is not None and folder_filter is not None:
        if min_th is not None:
            assert(max_file_num > min_th), "`max_file_num` should be greater than `" + min_th + "`
in " + folder_filter

    data = []
    label = []
    for entry in os.scandir(data_path):
        if entry.is_dir():
            target_dir_path = os.path.join(data_path, entry.name, 'points')
            path, dirs, files = os.walk(target_dir_path).__next__()
            file_count = len(files)
            if (min_th is None or file_count >= min_th) and (max_th is None or file_count <=
max_th):
                count = 0
                for pts_data in os.scandir(target_dir_path):
                    if (max_file_num is None) or (count < max_file_num):
                        data.append(os.path.join(data_path, entry.name, 'points', pts_data.name))
                        label.append(entry.name)
                        count += 1
                    else:
                        break
    return data, label
```

II.ii Normalization & Density Converter.

```
def normalize_point_cloud(pts):
    # find minimum value along each axis
    minimum_val = [pts[0][0], pts[0][1], pts[0][2]]
    # find the smallest
    for pair in pts:
        if pair[0] < minimum_val[0]:
            minimum_val[0] = pair[0]
        if pair[1] < minimum_val[1]:
            minimum_val[1] = pair[1]
        if pair[2] < minimum_val[2]:
            minimum_val[2] = pair[2]
    # move it to first quadrant
    rectified_pts = np.empty(pts.shape)
    for index, pair in enumerate(pts):
        point = np.zeros(3)
        point[0] = pair[0] - minimum_val[0]
        point[1] = pair[1] - minimum_val[1]
        point[2] = pair[2] - minimum_val[2]
        rectified_pts[index] = point
    # biggest value of all axes
    maximum_val = pts[0][0]
    for pair in rectified_pts:
        for val in pair:
            if val > maximum_val:
                maximum_val = val
    # normalize all the axises to (0,1)
    normalized_pts = rectified_pts/maximum_val
    return normalized_pts
```

```
def density_grid(normalized_pts, dim=(32,32,32)):
    x_grid_length = 1/dim[0]
    y_grid_length = 1/dim[1]
    z_grid_length = 1/dim[2]
    output = np.zeros((dim[0],dim[1],dim[2]))
    epsilon = 0.000000000001
    max_volume_size = 0
    for pair in normalized_pts:
        x_loc = int(pair[0]/(x_grid_length + epsilon))
        y_loc = int(pair[1]/(y_grid_length + epsilon))
        z_loc = int(pair[2]/(z_grid_length + epsilon))
        output[x_loc, y_loc, z_loc] += 1
        if output[x_loc, y_loc, z_loc] > max_volume_size:
            max_volume_size = output[x_loc, y_loc, z_loc]
    output = output/max_volume_size
    return output
```

II.iii Data Reshape

```
def data_reshape(data, dim=[32,32,32]):
    """
    Reshape to add one colour channel and voxelization
    """
    x_reshaped = []
    for i in range(len(data)):
        if i % 20 == 0:
            print("Process: ", str('{0:.2f}'.format(i/len(data))) + "%", end='\r')
        point_cloud = PyntCloud.from_file(data[i], sep=" ", header=0, names=["x","y","z"])
        vox = density_converter(point_cloud.xyz, dim=dim)
        vox_chan = np.array(vox).reshape(vox.shape + (1,))
        x_reshaped.append(vox_chan)
    print("Process: ", " 100% ")
    return x_reshape
```

II.iv Data Shuffling.

```
def data_swap(data, index_a, index_b):
    temp = data[index_a]
    data[index_a] = data[index_b]
    data[index_b] = temp
    return data

def data_random_position(data):
    import random
    return random.randint(0,len(data)-1)

def data_shuffling(data, label):
    for i in range(len(data)):
        target = data_random_position(data)
        data = data_swap(data, i, target)
        label = data_swap(label, i, target)
    return data, label
```

II.v Data Encapsulation.

```
def write_big_files_to_h5(data_path, folder_filter=(None, None), dim=[32,32,32]):

    data, label = find_data(data_path, folder_filter=folder_filter)

    _shuffled_data_raw, _shuffled_label_raw = data_shuffling(data, label)

    _shuffled_data = data_reshape(_shuffled_data_raw)
    _shuffled_label, _label_ref = data_onehot_encode(_shuffled_label_raw)

    # Create hdf5
    hdf5_path = os.path.join(data_path, 'small_shuffled_sets.h5')
    hdf5_file = h5py.File(hdf5_path, mode='w')

    hdf5_file.create_dataset("voxels", data=_shuffled_data[:])
    hdf5_file.create_dataset("labels", data=_shuffled_label[:])
    hdf5_file.create_dataset('label_ref', data=np.array(_label_ref).astype('|S9')) # ASCII
```

III CNN Tensorflow Implementation.

III.i CNN Architecture.

```
def cnn_model(x_train_data, label_size, keep_rate=0.7, training=True, seed=None):

    if seed is not None:
        tf.set_random_seed(seed)

    with tf.name_scope("layer_a"):
        # conv => 32*32*32
        conv1 = tf.layers.conv3d(inputs=x_train_data, filters=16,
kernel_size=[3,3,3], padding='same', activation=tf.nn.relu, name="conv1",
reuse=tf.AUTO_REUSE)
        # conv => 32*32*32
        conv2 = tf.layers.conv3d(inputs=conv1, filters=32, kernel_size=[3,3,3],
padding='same', activation=tf.nn.relu, name="conv2", reuse=tf.AUTO_REUSE)
        # pool => 16*16*16
        pool3 = tf.layers.max_pooling3d(inputs=conv2, pool_size=[2, 2, 2],
strides=2, name="pool3")

    with tf.name_scope("layer_b"):
        # conv => 16*16*16
        conv4 = tf.layers.conv3d(inputs=pool3, filters=64, kernel_size=[3,3,3],
padding='same', activation=tf.nn.relu, name="conv4", reuse=tf.AUTO_REUSE)
        # conv => 16*16*16
        conv5 = tf.layers.conv3d(inputs=conv4, filters=128, kernel_size=[3,3,3],
padding='same', activation=tf.nn.relu, name="conv5", reuse=tf.AUTO_REUSE)
        # pool => 8*8*8
        pool6 = tf.layers.max_pooling3d(inputs=conv5, pool_size=[2, 2, 2],
strides=2, name="pool6")

    with tf.name_scope("layer_c"):
        # conv => 8*8*8
        conv7 = tf.layers.conv3d(inputs=pool6, filters=256, kernel_size=[3,3,3],
padding='same', activation=tf.nn.relu, name="conv7", reuse=tf.AUTO_REUSE)
        # conv => 8*8*8
        conv8 = tf.layers.conv3d(inputs=conv7, filters=512, kernel_size=[3,3,3],
padding='same', activation=tf.nn.relu, name="conv8", reuse=tf.AUTO_REUSE)
        # pool => 4*4*4
        pool9 = tf.layers.max_pooling3d(inputs=conv8, pool_size=[2, 2, 2],
strides=2, name="pool9")

    with tf.name_scope("batch_norm"):
        cnn3d_bn = tf.layers.batch_normalization(inputs=pool9, training=training,
name="bn", reuse=tf.AUTO_REUSE)

    with tf.name_scope("fully_con"):
        flattening = tf.reshape(cnn3d_bn, [-1, 4*4*4*512])
        dense = tf.layers.dense(inputs=flattening, units=1024,
activation=tf.nn.relu, name="full_con", reuse=tf.AUTO_REUSE)
        # (1-keep_rate) is the probability that the node will be kept
        dropout = tf.layers.dropout(inputs=dense, rate=keep_rate,
training=training, name="dropout")

    with tf.name_scope("y_conv"):
        y_conv = tf.layers.dense(inputs=dropout, units=label_size, name="y_pred",
reuse=tf.AUTO_REUSE)

    return y_conv
```

III.ii CNN Input and Measurement Initializer.

```
def set_placeholders(x_shape, y_shape):

    with tf.name_scope('inputs'):
        x_input = tf.placeholder(tf.float32, shape=x_shape, name="x_input")
        y_input = tf.placeholder(tf.float32, shape=y_shape, name="y_input")

    return {'x_input': x_input, 'y_input': y_input}



def get_measurement(placeholders, seed=None, keep_rate=0.5, learning_rate=0.01,
training=True, device='/cpu:0'):

    x_input = placeholders['x_input']
    y_input = placeholders['y_input']

    device_name = device

    with tf.device(device_name):

        prediction = cnn_model(x_input, y_input.shape[1].value, keep_rate,
training=training, seed=seed)
        tf.add_to_collection("logits", prediction)

        with tf.name_scope("cross_entropy"):
            cost =
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=prediction,
labels=y_input), name="cross_entropy")

        with tf.name_scope("training"):
            optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)
            tf.add_to_collection("optimizer", optimizer)

        correct = tf.equal(tf.argmax(prediction, 1), tf.argmax(y_input, 1))
        accuracy = tf.reduce_mean(tf.cast(correct, 'float'), name="acc")

    return {'prediction': prediction, 'cost': cost, 'optimizer': optimizer,
'accuracy': accuracy}
```

III.iiiCNN Runner.

```
def run_nn(data, label, placeholders, measurments, session, summary_op=None, batch_size=32,
epoch_step=0, training=True, device='/cpu:0'):

    x_input = placeholders['x_input']
    y_input = placeholders['y_input']

    prediction = measurments['prediction']
    cost = measurments['cost']
    optimizer = measurments['optimizer']
    accuracy = measurments['accuracy']

    iterations = int(len(data)/batch_size) + 1

    summary_step = 8
    summary_counter = int(epoch_step*(len(data)/summary_step))

    acc = 0
    with session.as_default():
        # mini batch
        for itr in range(iterations):
            mini_batch_x = data[itr*batch_size: (itr+1)*batch_size]
            mini_batch_y = label[itr*batch_size: (itr+1)*batch_size]

            if training and summary_op is None:
                _optimizer, _acc, _cost = session.run([optimizer, accuracy, cost], feed_dict={x_input:
mini_batch_x, y_input: mini_batch_y})
                print('\tLost for', itr + 1, "/", iterations, _cost, end='\r')
                acc += _acc

            elif training and summary_op is not None:
                _optimizer, _acc, _cost, _summary = session.run([optimizer, accuracy, cost,
summary_op['summary']], feed_dict={x_input: mini_batch_x, y_input: mini_batch_y})
                print('\tLost for', itr + 1, "/", iterations, _cost, end='\r')
                summary_op['train_writer'].add_summary(_summary, summary_counter)
                summary_op['train_writer'].flush()
                acc += _acc

            elif not training and summary_op is not None:
                _acc, _summary = session.run([accuracy, summary_op['summary']], feed_dict={x_input:
mini_batch_x, y_input: mini_batch_y})
                print('\tAccuacy for', itr + 1, "/", iterations, _acc, end='\r')
                acc += _acc
                summary_op['test_writer'].add_summary(_summary, summary_counter)
                summary_op['test_writer'].flush()

            else:
                _acc = session.run(accuracy, feed_dict={x_input: mini_batch_x, y_input: mini_batch_y})
                print('\tAccuacy for', itr + 1, "/", iterations, _acc, end='\r')
                acc += _acc

            summary_counter += int(batch_size/summary_step)

        print("\n")

        if not training:
            print("\tTesting Accuracy:", acc/iterations)

        return acc/iterations
```

III.iv Transfer Learning Architecture.

```
def new_model(n_classes, stop_layer):
    with tf.name_scope('inputs'):
        x_input = tf.placeholder(tf.float32, shape=[None, 32, 32, 32, 1], name="x_input")
        y_input = tf.placeholder(tf.float32, shape=[None, n_classes], name="y_input")
    prediction = cnn3d_model(x_input, n_classes, seed=1234, stop_layer=stop_layer, keep_rate=0.5)
    tf.add_to_collection("logits", prediction)
    with tf.name_scope("cross_entropy"):
        cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=prediction,
labels=y_input), name="cross_entropy")
    with tf.name_scope("training"):
        optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)
        tf.add_to_collection("optimizer", optimizer)
    correct = tf.equal(tf.argmax(prediction, 1), tf.argmax(y_input, 1))
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32), name="acc")
    return {'prediction': prediction, 'cost': cost, 'optimizer': optimizer, 'correct': correct,
'accuracy': accuracy, 'x_input': x_input, 'y_input': y_input}
```

III.v Running Transfer Learning Model

```
import os
model_path = os.path.join(os.getcwd(), 'trained_model', 'model-2')
layers = ['conv1', 'conv2', 'pool3', 'conv4', 'conv5', 'pool6', 'conv7', 'conv8', 'pool9', 'bn',
'full_con', 'dropout', 'y_pred']
learning_rate = 0.0005
keep_rate = 0.5
batch_size = 16
x_ = data[:]
y_ = label[:]

split_point = int(0.7*len(x_))
x_train = x_[:split_point]
y_train = y_[:split_point]
x_test = x_[split_point:]
y_test = y_[split_point:]

n_classes = len(y_train[0])
device_name = '/gpu:1'
stop_layer = 3

with tf.Session(graph=tf.Graph(), config=tf.ConfigProto(allow_soft_placement=True)) as sess:
    with tf.device(device_name):
        model = new_model(n_classes, stop_layer)

    valid_vars = layers[:stop_layer]
    restore_vars = remove_layers_from_trained_model(valid_vars, model_path)
    restore_map = {variable.op.name : variable for variable in tf.global_variables() if
variable.op.name in restore_vars}
    tf.contrib.framework.init_from_checkpoint(os.path.join(os.getcwd(), 'trained_model', 'model-
2'), restore_map)
    sess.run(tf.global_variables_initializer())

    graph = tf.get_default_graph()

    import math
    iterations_all = math.ceil(len(x_)/batch_size)
    iterations_train = math.ceil(len(x_train)/batch_size)
    iterations_test= math.ceil(len(x_test)/batch_size)

    for epoch in range(8):
        for itr in range(iterations_train):
            mini_batch_x = x_train[itr*batch_size: (itr+1)*batch_size]
            mini_batch_y = y_train[itr*batch_size: (itr+1)*batch_size]
            _optimizer, _cost, _acc = sess.run([model['optimizer'], model['cost'],
model['accuracy']], feed_dict={model['x_input']: mini_batch_x, model['y_input']: mini_batch_y})
            print('\tLost for', itr+1, "/", iterations_train, _cost, end='\r')

        print('\n')

        acc = 0
        for itr in range(iterations_test):
            mini_batch_x = x_test[itr*batch_size: (itr+1)*batch_size]
            mini_batch_y = y_test[itr*batch_size: (itr+1)*batch_size]
            _acc = sess.run(model['accuracy'], feed_dict={model['x_input']: mini_batch_x,
model['y_input']: mini_batch_y})
            acc += _acc
            print('\tLost for', itr+1, "/", iterations_test, end='\r')

        print('\n')
        print('Test Acc:', acc/iterations_test)
```