Using Synthetic Images as Training Data for Object Orientation Prediction

Markus Fjellheim
M.Fjellheim@stud.uis.no
University of Stavanger,Norway

Renny Octavia Tan
ro.tan@stud.uis.no
University of Stavanger,Norway

Nils Magne Fossaaen
nm.fossaen@stud.uis.no
University of Stavanger,Norway

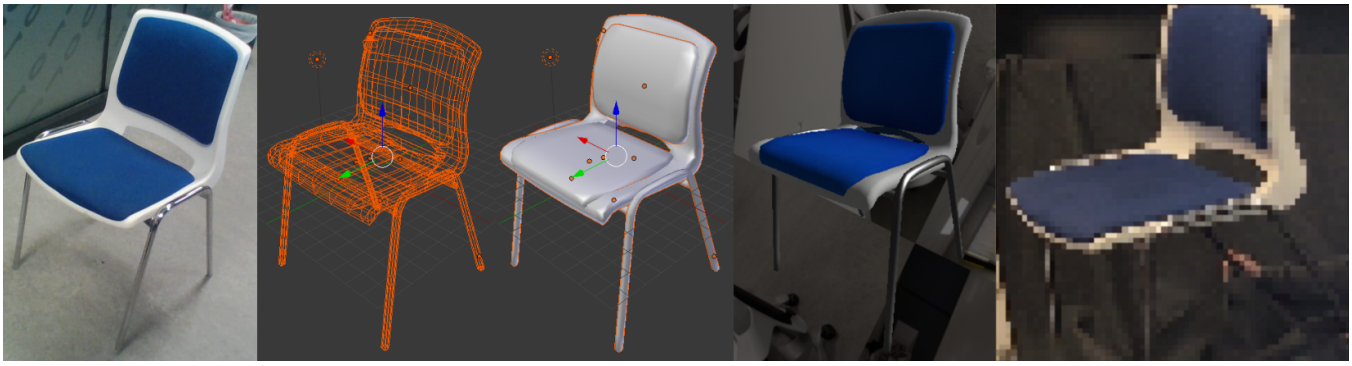


Figure 1: The pipeline. Left to right: real reference image, two synthetic images, synthetic image with random background, input image to be predicted

ABSTRACT

In this paper, we are documenting our project related to detecting objects in 2D images as well as determining its 3D orientations relative to the camera. We use Yolo for the object detection task and Convolutional neural network (CNN) for the regression task in estimating the object orientation. We generate our own synthetic images as training data using computer rendering. Compared to the work described in [6], we are trying to solve the orientation prediction as a regression problem rather than as a classification problem. We test our model on both synthetic test data, and on images of real objects.

KEYWORDS

datasets, neural networks, rendering, object detection, orientation prediction, convolutional neural network

ACM Reference Format:
Markus Fjellheim, Renny Octavia Tan, and Nils Magne Fossaaen. 2019. Using Synthetic Images as Training Data for Object Orientation Prediction. In *Proceedings of Octiba Nima (Dat550).* Uis, Stavanger, Rogaland, Norway , 9 pages. https://github.com/uis-dat550-spring19/Octiba-Nima/

Dat550, April 2019, Stavanger, Norway

https://github.com/uis-dat550-spring19/Octiba-Nima/

1 INTRODUCTION

Our goal is to detect the position and orientation of objects in pictures. Use cases for such an application is to give spatial awareness to automated systems as self driving cars, automatic vacuums, drones for local spatial navigation, or surveillance systems. Our strategy is to first generate multiple synthetic images of objects with known spatial information. We use these images to train a machine learning algorithm to predict the spatial data based on the synthetic images. We can now predict the position and orientation of objects in real life images. The flow of data can be seen in figure 2. First the images generated at the image generator is used to train an orientation prediction model (step 1 and 2). Then real world images are cropped in the object detection module (step 3 and 4). Then the model is applied to the real world cropped images to predict the objects orientation. The object detection module also gives information of where in the picture the object was detected.

2 OBTAINING THE TRAINING DATA

We choose to generate the training data our self rather than downloading training data from online sources or taking pictures of objects in real life. The Advantage to generate our own data is that we can adjust the data generated to our own liking, and if we later change our minds, the parameter deciding how the training data is generated can quickly be changed and re-generated. Taking pictures of real life objects would take a long time and it would take just as long if we wanted to for example change the background, as we would have to take all the pictures again. Images downloaded online might not suit the input images we would use our model for, and might be missing spatial orientation labels. Another advantage to generating our own data is we can generate as much data as we want, only having to balance overfitting against processing power required to train our model.

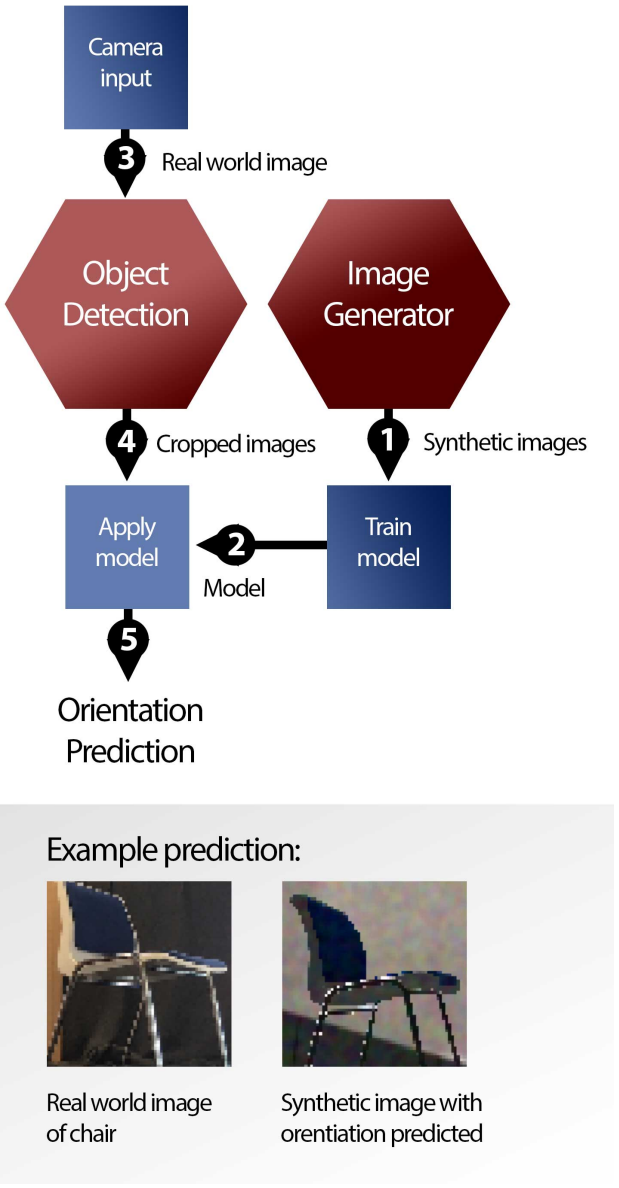


Figure 2: A flow chart showing the flow of data

3 CHOOSING THE TECHNOLOGY

To generate images efficiently, we wanted a rendering library that was both high performing, and one that gave us enough control over the rendering process to create training data that matches our needs.

3.1 WebGL

WebGL is a web API based on OpenGL used on the web. This was the first rendering API considered as we already knew how to use it. Even though WebGL allow for both fast and customizable rendering, its interface with the other modules (object detection and orientation prediction) would be more complicated. We decided to instead look for a python based OpenGL interface instead, as that would be less complicated.

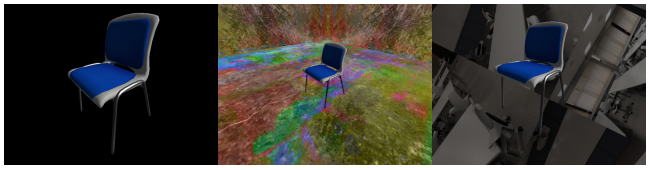


Figure 3: Left to right: An image of a synthetic chair with a black background, synthetic chair with colorful background, synthetic chair with random background images.

3.2 OpenGL and ModernGL

We first looked at PyOpenGL for python. This library is built on OpenGL, which is both simple and give us a good control over the rendering process. The problem is the package is a bit outdated, often depending on python 2. Also the OpenGL functionality taken use of was outdated, somewhat limiting customization of the rendering. Performance would also be worse. We looked into moderngl instead. Moderngl is another python library that builds on OpenGL, has good performance, and gives a high control over the rendering. It also is a slightly simpler API than pyOpenGL. Now as we have the tools to render our objects, we just need a way to model and load them.

3.3 3d modeling software

Blender is a free open source 3d modeling program. We used this to create various models of chairs to be visualized using the rendering module. We took pictures of a real chair, and used the reference images to model the chair in Blender. We took pictures of the surfaces of the chair in a diffuse white light condition and used those images as textures. In our experiments we only use chairs, but other objects can also be used

4 RENDERING

The rendering pipeline consists of taking images of real chairs, these images are modeled in blender and saved as object files. The model data is sent to buffers (vertex and index buffers) in the graphics card together with a shader program. The shader program tells the graphics card how to interpet the vertex data in the vertex buffers. We implemented the pong lighting model as it is simple and gives good enough results for our purpose. The texture gives the color of the surface. The color and angle of the light coming from above decides the color and brightness of the light bouncing off the surface. For the real and synthetic chairs (see figure 1), rendering the legs were challenging as the matrial is reflective. The view and light reflection angle off the legs made the chairs sometimes more easy to spot, and sometimes more hard. Initially we made the legs grey, but this made the legs consistently easy to spot. To make the training data more similar to the real chair, specular mapping was introduced. This made the legs much more reflective.

4.1 Data structure

The generated images are 3 dimensional numpy ndarrays. The first dimension is the image height, the second, is the image width, the third one is the three color channels (red, green, and blue). Each color value is an uint8 value between 0 and 255. In our experiments, we generate images of 64 by 64.

4.2 Challenges

One of the biggest challenges we faced, was that the training data had to be very similar to the real world images. The models usually had great success at fitting the training data, and the loss was even good on the generated test data, but not the real images. Even a small difference between the real and generated images, could throw the predictions way off. To combat this, we focused on making the generated data as similar to the training data as possible. To get the model more robust to differences in lighting, like white balance, we experimented with varying lighting conditions. Random noise was added for general robustness. We also randomized the light direction. But the change that proved to give the best improvement was making the proportion of the object's size to the image size match the training and real test data more precisely.

4.3 Describing the orientation

There would not be much use of training data, if it was not labeled with the orientations of the rendered objects. For the orientations to be useful for the orientation prediction module, the ways of describing orientation must follow some criteria. One criteria is that the description of orientation must not be redundant, that is, there must not exist two descriptions describing the same orientation. If this were the case, the orientation prediction module, might predict the correct orientation, but since the target description is different, the prediction is assumed to be wrong. This excludes using Euler angles, as it is redundant. Another problem with Euler angles is gimbal lock, which brings us to the next criteria. Two orientations that are almost the same, must give almost the same values describing the orientation. This must be the case for the gradient descent to work while fitting the model. In this paper [6], they use classes to describe the different orientations. They use spherical coordinates to describe the angle, only allowing the azimuth angle to change. Since they split the azimuth angle into 16 classes (22.5 degrees angles per class), they don't have a problem with the first criteria as the angle jump from 0 degrees to 360 degrees. Since we use regression to predict angles, we would have a problem with predicting angles close to 0 or 360 degrees. We might have a prediction of 359 degrees, while the true angle is 1 degree and get an error of 358 degrees while the error should be 2 degrees. Our first approach was using quaternions. Using qaternions is straight forward as they are used internally to describe orientation while rendering the objects. A quaternion consists of 4 real numbers multiplied with some imaginary components. We used these 4 numbers as the orientation. Unfortunately, we found that quaternions had a similar problem as the azimuth situations, as two almost identical quaternion orientations could have very different values. Our next approach was to describe orientation using two vectors of 3 dimensions. This gives 6 values. We let the first vector be the forwards direction, the second vector be the upwards direction.

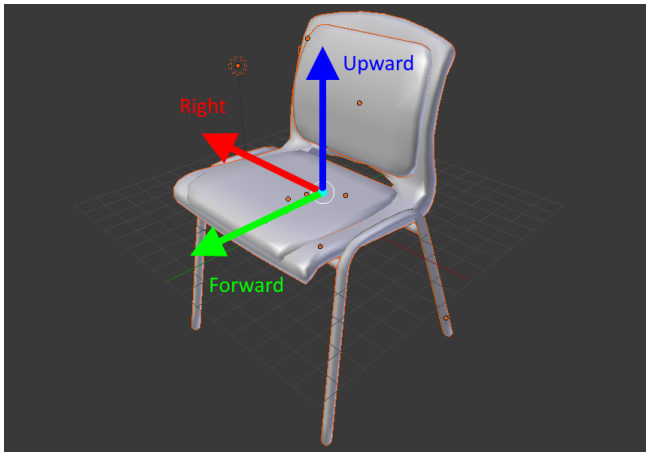


Figure 4: A chair showing the forwards, upwards and rightwards directions

If the lengths of each of the vectors are 1, we satisfy the first criteria. Each orientation can only be described one way. The second criteria is also fulfilled, as slightly small differences in orientation will give a small difference in the vectors positions and their coordinate values. Since the size of the vectors do not matter, we need to re-size the predictions to unit vectors. From experimentation, the prediction gives very large vectors, so a potential problem of small vectors radically changing direction due to small displacements, will not occur. An advantage to this approach compared to [6] is that we can describe any orientation, even upside down. The downside, is we need 6 output values, which also have to be interpreted as the predicted forwards vector might not be orthogonal to the upwards vector.

4.4 Background

Initially we set the background of the generated images to be black. Since this might make the model unfit to deal with the noise that comes with a varying background, we rendered a box around the model.

The background has different colors and shapes. The orientation of the chair relative to the background is also randomized for each training image. In the process of making the training data even more similar to the real images, we also added a background made out of images taken of the surroundings of where the real chair would be located. The background images are translated and rotated randomly for each training sample to maximize the use of the images.

5 OPTIMIZATION

To speed up the training process, the rendering module also support limiting of the angles of the images. It can be set to only show the models from above, or models that are never tilted more than 90 degrees. This limits what the orientation prediction model can recognize, but also makes it better at the more limited problem for it to solve.

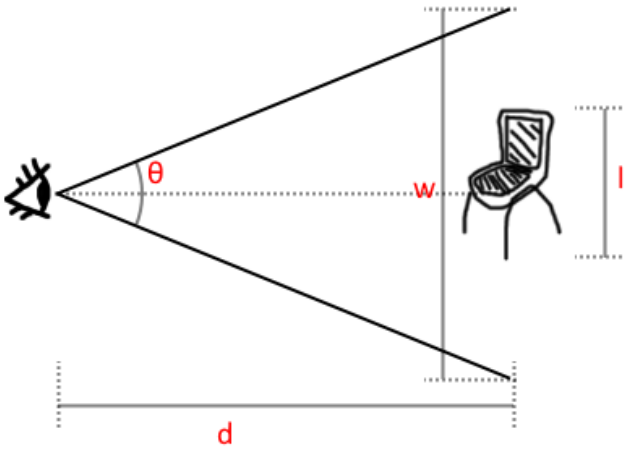


Figure 5: Variable definition for the distance calculation

6 VISUALIZATION

After the model is trained and we want to test our model, we do so by capturing live images of a real chair, predicts its position and orientation, and render a 3d model of the chair at that predicted position. The input data to the visualization is the bounding box of the chair in the image from the object detection module and the predicted orientation from the orientation prediction module. Some metadata about the real chair and the capturing of the chair is also used, for example the real size of the chair and the field of view of the web camera used for the capturing.

To calculate the position of the chair, the bounding box from the object detection module is used. The first calculation is the distance from the camera. In figure 5 we see the variables used in this calculation. θ is the horizontal field of view. We imagine the chair being distance d from the camera (the point of view of the rendering) (the eye in figure 5). w is the width of the visible plane at d distance from the camera. l is the visible with of the chair. We want to calculate a new distance D that is so that the chair will fill the same fraction of the field of view as the bounding box width from the object detection module. Currently we only know the variable θ and the bounding box width. We start by calculating w.

$$\frac{w}{2 \cdot d} = \tan\frac{\theta}{2}$$
$$w = 2 \cdot d \cdot \tan\frac{\theta}{2}$$

We now have an expression for w. We don't have a value for d, but that will not be a problem as it will be cancelled in later calculations. Let's introduce a new varialbe r. r is the width of the bounding box relative to the width of the image. The visible size (width or height) of an object inside the field of view of the camera is invertedly proportional to the objects distance. We can see this from the cosine rule.

$$c^2 = a^2 + b^2 - 2ac * \cos C$$

Where a, b, and c are sides in a triangle and C is the angle opposite of a. In figure 5, if we let a and b be the sides adjacent to the camera, c will be the visible plane at distance d from the camera. If we increase the distance d by a factor, a and b must be increased by the same factor, assuming θ (or C from the cosine rule) is not changed. This will lead to c (or w from figure 5) being increased by the same factor. Let's call this factor f.

$$(af)^2 + (bf)^2 - 2(af)(cf) * \cos C =$$
$$= f^2a^2 + f^2b^2 - f^2 2ac * \cos C =$$
$$= f^2(a^2 + b^2 - 2ac * \cos C) = f^2c^2 = (fc)^2$$

We want to adjust the distance to the chair in figure 5 so that the size of the chair is r size relative to the width of the visible plane at distance d from the camera. In other words, we want to make $\frac{l}{w'} = r$. w' being the new w as d is adjusted to be D.

$$r = \frac{l}{w'} = \frac{l}{2D\tan\frac{\theta}{2}}$$

$$D = \frac{l}{2r\tan\frac{\theta}{2}}$$

If we let the camera position be in the origin, forward being in the positive y direction, right being in the positive x direction and up being in the positive z direction (right handed coordinate system), D will be the chairs y coordinate (y_{chair}).

$$D = y_{chair}$$

To calculate the x and z coordinate, we use the tangent rule.

$$\tan\frac{\theta}{2} = \frac{x_{chair}}{2 \cdot y_{chair} \cdot (y_{boundingBox} - 0.5)}$$

$$x_{chair} = (h_{boundingBox} - 0.5) \cdot 2 \cdot y_{chair} \cdot \tan\frac{\theta}{2}$$

$h_{boundingBox}$ is the horizontal coordinate of the bounding box from the real image of the chair. It is given in relative coordinates, 0 being all to the left of the image, and 1 being all to the right. To calculate the z_{chair} coordinate, we do a similar calculation as when calculating the x_{chair} coordinate, except we also have to take the aspect ration into account ($\frac{\text{image width}}{\text{image height}}$).

$$z_{chair} = \frac{(v_{boundingBox} - 0.5) \cdot 2 \cdot y_{chair} \cdot \tan\frac{\theta}{2}}{aspect}$$

To calculate the orientation, we set the orientation of the model to the predicted orientation. If the models position is offset from the forward direction of the camera, (x_{chair} or z_{chair} is not zero), we rotate the chair to be oriented towards the camera so the correct side of the model is still being rendered.

7 OBJECT ORIENTATION PREDICTION

We formulate this module as a regression problem, where we focus on predicting the forwards and upwards vectors which we use to define the orientation in section 4.3.

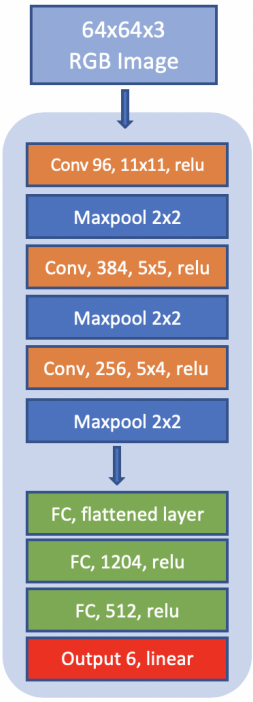


Figure 6: Simple CNN architecture

7.1 Architecture

We choose Convolutional Neural Network (CNN) for this task, as CNN is generally an effective machine learning algorithm when working with images. There are three different architectures that we tried.

We initially experimented with building our own simple layers of CNN (let's call this simple CNN) as Fig. 6. This architecture has three convolutional layers and two hidden layers in the fully connected part. As this task is not for object detection, we expected that with this simple architecture, we will be able to extract high level information from the edges and shapes which can be sufficient for predicting the spatial features.

The second architecture we use is the VGG-16[8]. We realized from the first experiment that, to tune and design a CNN architecture from scratch will take a lot of time, and also to build a robust model we need to train on huge amount of training images. Therefore, we try to utilize an architecture that have been widely used. VGG-16 also has the option to use pre-trained weights from ImageNet.

We use more deeper network for the third architecture which is Residual Neural Network, ResNet-50[3]. Residual Neural Network has become popular, which introduce the concepts of residual block and identity shortcut that can handle the vanishing gradient problem in deep network. There is also an option to use pre-trained weights from ImageNet for this architecture.

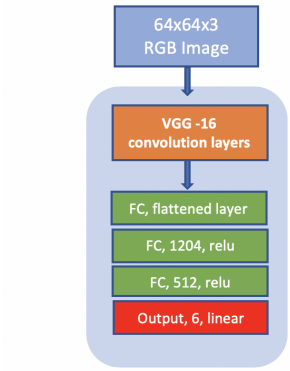


Figure 7: Modified VGG-16 architecture. For the original architecture, please refer to paper [8]

7.2 Modifications and Implementation Options of the VGG-16 architecture

Before we can use the VGG-16 architecture for our task, we have to make some adjustments. VGG-16 was designed for image size 224x224x3 as input, while having 7x7x512 as output from its last convolutional base. Meanwhile, we have image size 64x64x3 as input, and by using the same architecture, will resulted in output of 2x2x512 from the last convolutional base. Therefore, the input to the fully connected layer (top layer) at the end will not be the same. Due to this reason, we can only use the convolutional base from original the design of VGG-16 and we build our own fully Connected layers on top of it.

Both of VGG-16 architecture and the pre-trained weights are trained for classification problem, therefore we must train the fully connected layers for doing regression. We adjusted the number of outputs in the output layer of the fully connected part to 6 and use linear activation function instead of softmax (due to regression problem). The modified VGG-16 architecture that we use for our VGG-16 best model is as seen in figure 7

Options that we have for the implementation is as follow [5]:

- Option 1 : Not using pre-trained weights, it means that we have to train the model from scratch through all of the layers with initial random weights.
- Option 2 : Use pre-trained weights, open all layers for training and train the model from scratch throughout all of the layers.
- Option 3 : Use pre-trained weights, partially freeze the convolutional base from training, it means that the weights in the freezed layers will not be updated, and then train only the unfreezed layers and the fully connected layers.
- Option 4 : Use pre-trained weights, freeze convolutional base (all of the layers) and only train on the Fully connected layers.

The benefit of using pre-trained weights is that we can make use of the learning done on much larger and varied training sets with longer running time. In this case, the images from ImageNet. This is usually useful when we have small training set and limited computational power.

7.3 Modifications and Implementation Strategy of the ResNet-50 architecture

The modifications to the architecture is done to the top layer. While the top layer of VGG-16 consist of 3 fully connected layers including the output layer, ResNet-50 only has 1 average pooling layer and 1 output layer in the top layer. The modification that we did is only to change the output layer to 6 outputs and with linear activation function due to regression.

We also have the option whether to use pre-trained weights or not. If using pre-trained weights, we can choose to freeze all layers and train only the top layers, or freeze partially and trained the unfreezed layers and top layers, or open all layers for training. The original architecture of ResNet-50 can be seen in [3].

7.4 Customized Loss Function

We use customized loss function in our implemetations which is a modification of a standard mean squared error function. The loss function is customized to make it more suitable with our target properties which is vector orientations. As explained in earlier chapter, our target for prediction is an orientation vector which consists of two 3-dimensional vectors (forwards and upwards) as illustrated in figure 4. In the output layer, the vectors are represented by 6 numbers of output, where the first three outputs represent the forwards vector and the last three outputs represent the upwards vector.

For orientation of the object, the most important feature from the vector is the direction. It does not really matter of how long the vectors are, but as long as the directions are correct then it is sufficient. Our training data have vector of magnitude 1, while the predictions (output from the linear activation function) might have various magnitudes, and in fact have large numbers. By using standard Mean Squared Error, we will punish the predictions which have different magnitudes than the training data, even though the directions are correct. Therefore, to avoid this, in our customized loss function, we scaled the magnitudes of prediction vectors to magnitude of 1 before calculating the mean squared error. Formula can be seen below.

$p_0, p_1, ..., p_5$ is the prediction from the model.

$$\vec{p_1} = [p_0, p_1, p_2]$$
$$\vec{p_2} = [p_3, p_4, p_5]$$
$$\hat{\vec{p_1}} = \frac{\vec{p_1}}{|\vec{p_1}|}$$
$$\hat{\vec{p_2}} = \frac{\vec{p_2}}{|\vec{p_2}|}$$

The arrows mean vectors. The hats mean unit vectors. The bars mean length of vector. $t_0, t_1, ..., t_5$ is the true orientation.

$$\hat{\vec{t_1}} = [t_0, t_1, t_2]$$
$$\hat{\vec{t_2}} = [t_3, t_4, t_5]$$

t_0, t_1, t_2 and t_3, t_4, t_5 are already such that $[t_0, t_1, t_2]$ and $[t_3, t_4, t_5]$ will be unit vectors. The loss function is calculated like this:

$$loss = (\hat{\vec{t_1}} - \hat{\vec{p_1}})^2 + (\hat{\vec{t_2}} - \hat{\vec{p_2}})^2$$

$...^2$ means dot product with self.

The Comparisons of using default mean squared error and customized loss function can be seen in figure 8. By using the customized loss function, we see that the model can learn better and faster. The other thing we try is to change the activation from linear to tanh and use standard mean squared error as the loss function. Since tanh will return values between -1 to 1, we thought this can handle our issue of having very large output, however, since the input to the activation function is large numbers (positive and negative), we end up with output of only 1 or -1 in the beginning, it will take time for the model to adjust.

Just to explain a bit about the training loss and validation loss curves in the picture 8. In most of the case, training loss should be better than the validation loss, but in the picture it is the other way around. We found an explanation from from Keras [2] that, dropout mechanism is turned off at the validation. Training loss is calculated from the average of losses over each batch of the training data, while the validation loss is computed using the model at the end of the epoch (last batch). In general, the loss on early batches is higher than at the end of the epoch, so naturally, validation loss will be lower.

We realize that the vectors resulted from the predictions might not always be orthogonal to each other. To render the prediction, these vectors must be orthogonal. To overcome this issue we treat the forwards vector as the base of the orientation, and the upwards vector prediction as the indicator of on which plane those two vectors are located, and we then adjusted the upward vector 90 degrees from the forwards vector in that plane to get the orthogonal vectors.

7.5 Angle discrepancy calculation

To measure the performance of our model we use the below formula which will return the angle discrepancy. The angle discrepancy is calculated like this:

$$\hat{\vec{p_3}} = \hat{\vec{p_1}} \times \hat{\vec{p_2}}$$
$$\hat{\vec{t_3}} = \hat{\vec{t_1}} \times \hat{\vec{t_2}}$$

$\times$ means cross product.

$$\sin\theta = \frac{|\hat{\vec{p_1}} \times \hat{\vec{t_1}} + \hat{\vec{p_2}} \times \hat{\vec{t_2}} + \hat{\vec{p_3}} \times \hat{\vec{t_3}}|}{2}$$

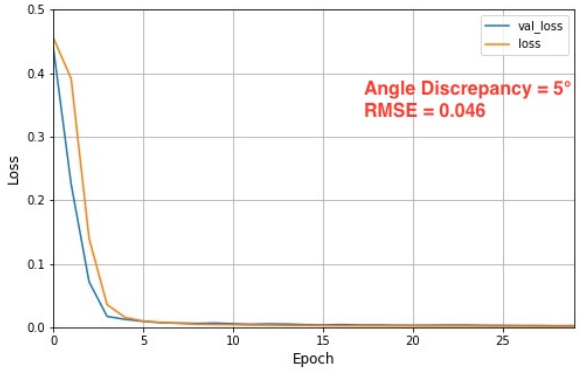
$$\theta = \arcsin \frac{|\hat{\vec{p_1}} \times \hat{\vec{t_1}} + \hat{\vec{p_2}} \times \hat{\vec{t_2}} + \hat{\vec{p_3}} \times \hat{\vec{t_3}}|}{2}$$

This is true as long as θ is less than $90°$. θ is the angle discrepancy.

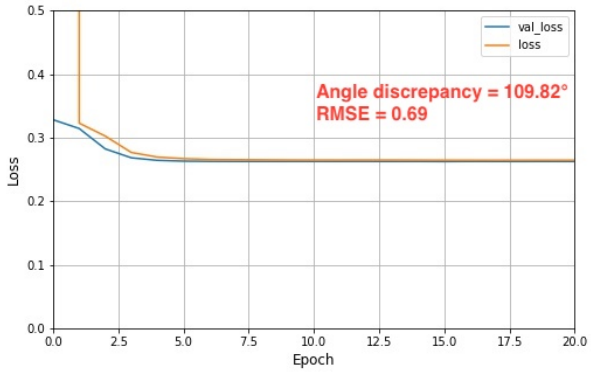
8 EXPERIMENTS

8.1 Dataset

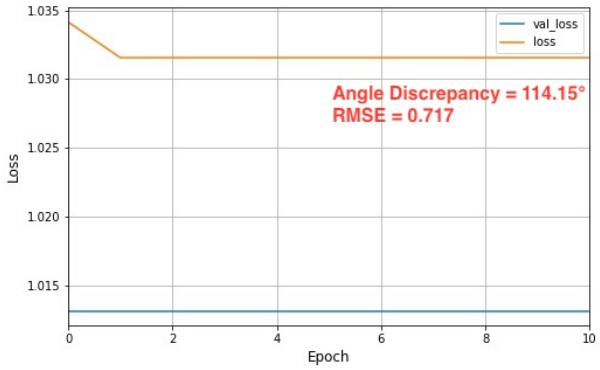
We use 64 x 64 x 3 (RGB) synthetic images in the training and test data. Considering our resource limitation, the number of dataset that we use is 10000 rendered images, which we split between training data and validation data in proportion of 90/10. To reduce complexity, we use only one type of chair as our training and test data. We have two types of test data. The first test data is the synthetic images which are generated with label of vector orientations (from now on will be called as synthetic test data), this is used for evaluating the model which will enable us to calculate the accuracy.



(a) Output Activation: linear, Loss Function: custom loss function



(b) Output Activation: linear, Loss Function: mean squared error



(c) Output Activation: Tanh, Loss Function: mean squared error

Figure 8: Comparison of learning history for different combination of activation function in output layer, and the loss function. Base model : VGG16

The other test data is the cropped real world images (let's call this real test data), this real test data does not have label or orientations.

8.2 Parameters Tuning and Training

The tuning in this project lays heavily in synchronizing between tuning of the parameters of the model and updating the training data to be as similar as possible to the real world images. Our goal is to get a model which not only have good accuracy when we test it with synthetic test data, but the model has to be able to transfer the learning from the training data to predict the orientation of objects in real test data.

This phase is the most challenging and time consuming. Many fixes, transformations, additions to the training images i.e. lighting, color, background, texture, noise, were implemented. Every time there is a change, we have to re-train the model with the latest type of training images, re-analyse and repeat the process if we think there is something that we can improve from the training data.

For the model parameters tuning, in general it is initialized with several random parameter searches for each architecture with small data size, and train on more data for the best parameters resulting from the random search. We also do further tuning, such as adding/reducing nodes, adding/reducing layers, change the number of batches, and try several optimizers. After we get the best models, we make comparisons and tested them with the real test data.

8.3 Results

The comparison of the model performance can be seen in table 1 for the options in VGG-16, and 2 for comparison of the best models of different architectures. We show comparisons of the Root Mean Squared Error (RMSE), Angle Discrepancy (formula can be seen in 7.5), and runtime per epoch during model training. The RMSE is calculated after scaling the predicted vectors to the unit length.

Table 1 shows the comparisons of all VGG-16 implementation options as mentioned in section 7.2. These models are implemented in the same setup as figure 7. We can see that the use of pre-trained weights has good impact on the accuracy, which means that the transfer of learning is useful for our case. The more freezed layers, the more efficient the time used for training. Option 3, which combine the use pre-trained weights and partially freezed layers returns the best result (accuracy and efficiency).

For ResNet-50 models, we dealt with issues of overfitting. We have played around with several parameters i.e. tried to freeze the layers from top to bottom with increasing number at a time, change the optimizer, freeze the batch normalization layers. The biggest improvement that we can see is when we freeze the batch normalization layers and also use a small learning rate (SGD with learning rate 0.0001). There are many discussions in Internet forums regarding the issue of overfitting of implementation of ResNet-50 in Keras, and one of the issue mentioned is related to batch normalization. Due to limited time, we will need to do more research for further work. In this experiment, we will just document the best result that we can get at this point.

Please note that these angle discrepancies and RMSE can only be calculated when the models are used to predict on the synthetic test data, because then we can compare the target values. When

Table 1: Comparison of VGG-16 models with different implementation options. Please see the explanation about each option in section 7.2

No.	VGG-16 models	RMSE	Angle Discrepancy	Runtime/epoch (during training)	no.epoch
1.	VGG-16 - Option 1	0.65	97.82°	510 s	8
2.	VGG-16 - Option 2	0.10	10.13°	502 s	25
3.	VGG-16 - Option 3	0.047	5°	191 s	30
4.	VGG-16 - Option 4	0.169	19.07°	156 s	25

Table 2: Comparison of best models

No.	Models	RMSE	Angle Discrepancy	Runtime/epoch (during training)	no.epoch
1.	VGG-16 - Option 3	0.047	5°	191 s	30
2.	Simple CNN	0.055	4.22°	460 s	34
3.	Resnet50	0.47	68.78°	60 s	30

Figure 9: Prediction results to the real test data from the best models

doing prediction to the real test data, where we don't have information about the orientation, we can only see the result by visual observation. Naturally, it will perform less accurate when we apply the model to the real test data compare to the synthetic one, as their properties are more close to the training data.

Now we pick two models (no.1 and no.2) from table 2 which have lowest angle discrepancy and run prediction on the real test data. Please see side-by-side comparison for these two models in figure 9. By visual evaluation, we see that model no.1 (trained using VGG-16 with pre-trained weight) is predicting the orientation of real test data much better than model no. 2 (trained using simple CNN), even though model 2 has lower angle discrepancy when predicting synthetic test data. We believe that the use of pre-trained weights plays an important part which gives a lot more information or learning to the model 1, which cannot be acquired by model 2 which only can learn from the train data. The network depth might also cause model 1 to be able to extract more features.

9 OBJECT DETECTION

We separate the action of detecting the chair from the prediction of orientation. Therefore, we would have two models, one model for detecting objects and another module for predicting orientation.

9.1 Research

Initially we considered writing a simple model for the object detection. Slicing up an image in many subsets and train a neural network to recognize a chair in any of the subsets. If a chair was detected with a certain confidence we send it further for orientation analysis. This method though easy to implement would be very expensive to compute and would also raise problems with detecting the same chair several times.After some reading we decided to go for the YOLO algorithm for object detection.

9.2 YOLO

You Only Look Once, is an CNN algorithm developed by [7]. Look into their papers for exact details on how this object detection algorithm works. The main difference from other algorithms is that it considers the object detection problem a regression problem and predicts the bounding boxes directly whereas other algorithms have separate models for classification, bounding box placement and bounding box size. This makes other models like Faster-RCNN much slower and harder to optimize as each part of the problem needs to be trained separately. YOLO also performs better at context problems like mistaking a small patch of background for an object. Other models might loose context by only looking at parts of the image at a time. A tradeoff with the YOLO algorithm is that it is not as accurate as the best algorithms on very small objects. Since this is not a problem in our use case the speed it delivers and accuracy at normal sized objects is acceptable.

9.3 OpenCV

We also need to be able to read images from webcam or other sources and be able to perform easy tasks as resizing and cropping. A good library for anything computer vision related is OpenCV [1], an open source software library for computer vision and machine learning. OpenCV also supports Darknet, the neural net module on which YOLO is based and we are therefore able to load YOLO directly in OpenCV using the DNN module, provided we give it the appropriate configuration file, pre-trained weights and a file containing labels for all the classes the model is trained to classify. OpenCV handles image files as Numpy nd.arrays which is also convienient as this is how our other modules handles images. One important "quirk" with OpenCV is that it arranges its color layers in the BGR order as opposed to RGB!

9.4 Our implementation

We have decided to divide our project in three distinct modules. Each module with their own scope and encapsulation. For the object detection module we need two public methods, one for using a webcamera and another for a single image. We use callback functions as our way to create an interface between modules. The object detection module is divided in three parts; getting the YOLO model, applying the model to an image and returning the cropped classified images through one of the public interfaces. We use a YOLO v3 configuration pre-trained on the COCO dataset [4].

10 CONCLUSION

Based on our experiment, to predict orientation using regression is possible, as long as the cost function is such that it fulfills the criterias described in section 4.3. The orientation prediction models proved extremely sensitive to small differences between the synthetic images and the real test data if the synthetic training data has too low variation. To make the predictions of the real test data, we either have to make the training data extremely similar to the test data, or make the train data more general with more variations. What approach is better depend on the application. The use of pre-trained weights is useful in our case to compensate for the lack of robustness of our model.

10.1 Future Work

We only worked on one object (a chair). To make the model more general one could train the orientation model on different types of chairs, alternatively using multiple models for the orientation of different models, and a classifier to distribute the input images to the correct orientation prediction model.

11 AUTHOR AND CONTRIBUTIONS

We divide this project into three modules, image rendering and visualization, object detection module, and orientation prediction module. We believe that this is a good way split our work, and each of the author have contributed equally. Markus Fjellheim is responsible for rendering and visualization, Renny Octavia Tan is responsible for the object orientation prediction, and Nils Magne Fossaaen is responsible for the object detection. Even though we have responsibility for different modules, we also had input across our main responsibilities.

12 THE GITHUB REPOSITORY

https://github.com/uis-dat550-spring19/Octiba-Nima/

REFERENCES

[1] G. Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* (2000).
[2] François Chollet et al. 2015. Keras. https://keras.io/getting-started/faq/.
[3] Shaoqing Ren Kaiming He, Xiangyu Zhang and Jian Sun. 2015. *Deep Residual Learning for Image Recognition.* Technical Report. Microsoft Research.
[4] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. 2014. Microsoft COCO: Common Objects in Context. In *ECCV*.
[5] Pedro Marcelino. 2018. Transfer learning from pre-trained models. Retrieved April 24, 2019 from https://towardsdatascience.com/transfer-learning-from-pre-trained-models-f2393f124751
[6] Ruizhongtai (Charles) Qi. 2015. *Learning 3D Object Orientations From Synthetic Images.* Technical Report. Stanford University, 450 Serra Mall, Stanford, CA 94305, USA.
[7] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: An Incremental Improvement. *arXiv* (2018).
[8] Karen Simonyan and Andrew Zisserman. 2015. *VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION.* Technical Report. Visual Geometry Group, Department of Engineering Science, University of Oxford, Wellington Square, Oxford, OX1 2JD, United Kingdom.