Motif & Textbox Recognition on IVC artifacts

Qi Mo **Michael Tishman**

Department of Computer Science
Florida Institute of Technology
Melbourne, FL 32901
qmo2015@my.fit.edu
mtishman2013@my.fit.edu

Abstract

Motif and Textbox recognition on IVC artifacts: Bounding box generation by deep learning using Python, Tensorflow and Keras. In this project, we transformed Images of Indus Valley Civilization(IVC) artifacts into different variants by 2D/3D rotations and rescaling. Textboxes and motifs on IVC artifacts are located and labeled. Keras-retinanet was used to train an object detector to generate bounding boxes of textboxes and motifs.

1 Introduction

IVC artifacts are fairly uniform bricks discovered from Indus Valley Civilization(IVC). The IVC artifacts include the forms of animals called motifs and a short text describing the animals. The objective of this project is to train a Neural Network that recognizes the text boxes and motifs on the artifacts and generates their bounding boxes. Training a Neural Network such as an object detector usually requires a large set of images. However, the only source we can obtain images of IVC artifacts from is the Internet, and we only managed to obtain 26 unique images from it. Therefore, an image transformation technique was adopted to form a larger dataset. We created a Python script that transforms the 26 unique images into different variants. The Python script also kept track of and saved the coordinates of the bounding boxes. Once we had our dataset, we used Tensorflow, Keras and Retinanet to create and train an object detector. Dataset was split into training dataset and testing dataset, and uploaded to Google Colab. A Google Colab ipynb notebook was created to read the dataset, train the object detector, test the object detector and evaluate the results and performance.

2 Data Generation

In this section we discuss the need to perform data generation for this project. The data that needs to be generated for this project is our inputs: the images of the Indus River Valley Civilization artifacts. The amount of available images for these artifacts would not be enough to create and train a meaningful neural network alone. Because of this, producing additional dataset images would be required.

2.1 Data Augmentation

One of the best ways to grow your dataset is to augment the small dataset you are currently using. Image augmentation is the process of manipulating an image by performing various transformations (resize, offset, rotate) and creating different variations of the same image.

2.2 Rotation Calculations

The augmentations were done with a projection matrix, which is a matrix that represents three dimensional space in a two dimensional matrix. A projection matrix, also called a camera matrix, can be represented with the equation $x = PX$, where x is a 2D point matrix, P is the camera matrix, and X is the 3D point matrix. This can be visualized with a generic camera matrix shown in Figure 1.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} p_1 & p_2 & p_3 & p_4 \\ p_5 & p_6 & p_7 & p_8 \\ p_9 & p_{10} & p_{11} & p_{12} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Figure 1. Generic Camera Matrix

Our representation of this equation can be seen in Figure 2.

```
# Final transformation matrix
return np.dot(A2, np.dot(T, np.dot(R, A1)))
```

Figure 2. Final Transformation Matrix in Code

```
# Projection 3D -> 2D matrix
A2 = np.array([[f, 0, w / 2, 0],
               [0, f, h / 2, 0],
               [0, 0, 1, 0]])
```

Figure 3. Camera Matrix

“A2” in our equation represents the camera matrix where “f” is the focal point, or the change in the zoom distance, “w” is the width and “h” is the height of the image. The width and height are halved in order to keep the image within the image frame.

```
# Translation matrix
T = np.array([[1, 0, 0, dx],
              [0, 1, 0, dy],
              [0, 0, 1, dz],
              [0, 0, 0, 1]])
```

Figure 4. Translation Matrix

The 3D point matrix in our project is the dot product of a transformation matrix (T) and rotation matrix (which is a dot product of the original image and a rotation matrix (R). The translation matrix performs the positional offsets for each of the axis: dx, dy, and dz.

```
# Rotation matrices around the X, Y, and Z axis
RX = np.array([[1, 0, 0, 0],
               [0, np.cos(theta), -np.sin(theta), 0],
               [0, np.sin(theta), np.cos(theta), 0],
               [0, 0, 0, 1]])

RY = np.array([[np.cos(phi), 0, -np.sin(phi), 0],
               [0, 1, 0, 0],
               [np.sin(phi), 0, np.cos(phi), 0],
               [0, 0, 0, 1]])

RZ = np.array([[np.cos(gamma), -np.sin(gamma), 0, 0],
               [np.sin(gamma), np.cos(gamma), 0, 0],
               [0, 0, 1, 0],
               [0, 0, 0, 1]])

# Composed rotation matrix with (RX, RY, RZ)
R = np.dot(np.dot(RX, RY), RZ)
```

Figure 5. Rotation Matrix

The rotational matrix (R) is responsible for the rotational augmentations done in the respective axis RX, RY and RZ. Theta, Phi, and Gamma represent the angle of rotation

The final matrix result contains the distances for each point in the image from the original image. To find where a point was moved to, we used the following algorithm shown in Figure 6 to follow the four corners of our augmented bounding box.

```
tlx = self.top_left_corner[0]
tly = self.top_left_corner[1]
ntlx = ((mat[0, 0]*tlx) + (mat[0, 1]*tly) + (mat[0, 2])) / ((mat[2, 0]*tlx) + (mat[2, 1]*tly) + (mat[2, 2]))
ntly = ((mat[1, 0]*tlx) + (mat[1, 1]*tly) + (mat[1, 2])) / ((mat[2, 0]*tlx) + (mat[2, 1]*tly) + (mat[2, 2]))

brx = self.bottom_right_corner[0]
bry = self.bottom_right_corner[1]
nbrx = ((mat[0, 0]*brx) + (mat[0, 1]*bry) + (mat[0, 2])) / ((mat[2, 0]*brx) + (mat[2, 1]*bry) + (mat[2, 2]))
nbry = ((mat[1, 0]*brx) + (mat[1, 1]*bry) + (mat[1, 2])) / ((mat[2, 0]*brx) + (mat[2, 1]*bry) + (mat[2, 2]))

trx = self.top_right_corner[0]
tryy = self.top_right_corner[1]
ntrx = ((mat[0, 0]*trx) + (mat[0, 1]*tryy) + (mat[0, 2])) / ((mat[2, 0]*trx) + (mat[2, 1]*tryy) + (mat[2, 2]))
ntryy = ((mat[1, 0]*trx) + (mat[1, 1]*tryy) + (mat[1, 2])) / ((mat[2, 0]*trx) + (mat[2, 1]*tryy) + (mat[2, 2]))

blx = self.bottom_left_corner[0]
bly = self.bottom_left_corner[1]
nblx = ((mat[0, 0]*blx) + (mat[0, 1]*bly) + (mat[0, 2])) / ((mat[2, 0]*blx) + (mat[2, 1]*bly) + (mat[2, 2]))
nbly = ((mat[1, 0]*blx) + (mat[1, 1]*bly) + (mat[1, 2])) / ((mat[2, 0]*blx) + (mat[2, 1]*bly) + (mat[2, 2]))

tl = int(ntlx), int(ntly)
br = int(nbrx), int(nbry)

tr = int(ntrx), int(ntryy)
bl = int(nblx), int(nbly)
```

Figure 6. Algorithm to follow new augmented corners

By utilizing this method, we are able to take a single image and produce thousands of augmented images to be used as our dataset. Our starting data set was approximately 26 images, and after our data augmentation we ended up with over 5,000 images per unique image. For the purposes of this project, we cut this dataset down to increase processing speeds since we did not have access to a high power CPU.

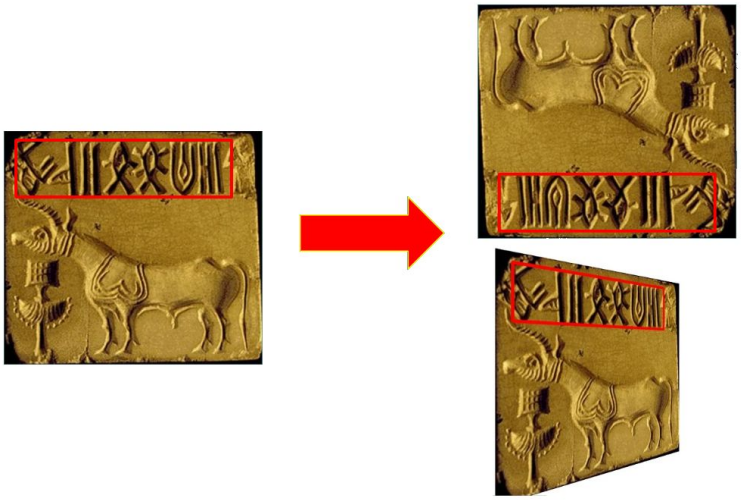

Figure 7. Augmentation example

In addition to bulking up our dataset, data augmentation helps fight overfitting in our model. Overfitting is where the model stops improving because it starts to memorize specific patterns and fails to continue learning. One of the best ways to prevent overfitting is to add more training data, and when it's not possible to do so, (such as our case), the next best solution is to augment the data to increase the quantity of information given [1].

2.3 Data Annotations

In order to use our images for a neural network, we need to have it in a format that a neural network model would understand. The main focus point of our dataset images are the manual bounding boxes that were printed on top of the image. These bounding boxes are recorded in a comma separated value sheet (CSV) along with the file name and a label. Regardless of the type of augmentation performed on the dataset images, the recorded points will match up the corners of the bounding boxes. To verify that our annotation points are accurate after all the augmentations, we created a test that uses the points from the annotations file to draw points on the corners of the bounding boxes found within the image. After hundreds of manual tests, we were able to verify that our annotations file has the correct information in it for each of the respective images in our dataset.

3 Motif & textbox Recognition

In this section we discuss the model, tools and platform used for this project, the training procedure, and the performance and results of training and testing.

3.1 RetinaNet and gitHub/fizyr/keras-retinanet

3.1.1 RetinaNet

The model selected for this project is RetinaNet, whose original paper was published by Facebook AI Research (FAIR) on Aug 7th, 2017. The original paper can be found at https://arxiv.org/abs/1708.02002.

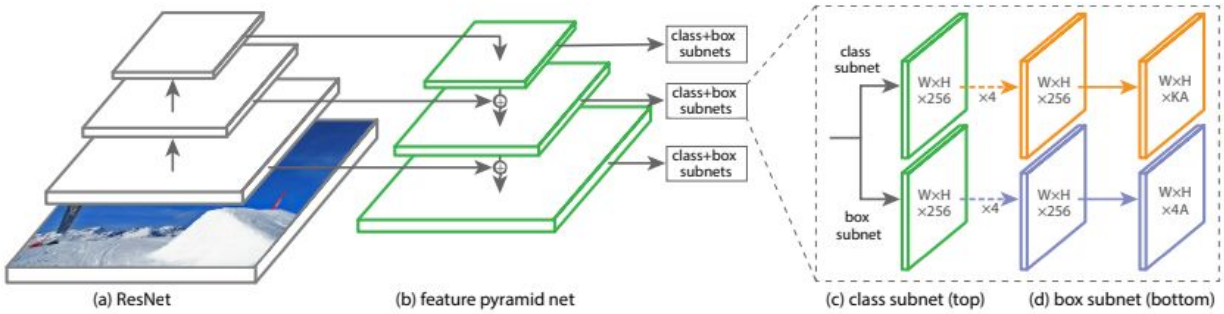

Figure 8. RetinaNet Network Architecture, taken from the original paper

RetinaNet is a one-stage object detection model that is applied over a regular, dense sampling of object locations, scales, and aspect ratios.
The structure of the model, as shown in figure 1, shows that the model utilizes a Feature Pyramid Network (FPN) [4] backbone on top of a feedforward ResNet architecture [5], which means it rescales the input image into feature maps with different resolutions and attaches subnetworks to each feature map. A feature map is a rescaled and filtered variant of the original image where each spatial location(pixel) represents a portion of the original image. The subnets attached to the backbone are a class subnet, for classifying anchor boxes, and a box subnet, for regressing from anchor boxes to ground-truth object boxes.
An example of how the model works is: suppose there is a dataset of images of fruits, and each image contains an apple and an orange. The model rescales and converts each image to multiple feature maps. For each feature map, the class subnet checks each region generated by the box subnet and classifies the region. After training, the model can predict a list of boxes on a new image that might contain an apple or an orange, then give each box a score, which is the probability that the box contains either an apple or an orange. The model can then classify each box, decide the specific class (apple or orange) the box contains, and update the score accordingly.

The output layers of the class subnets and box subnets contain W*H*KA nodes and W*H*4*A nodes, respectively, where W and H are the weight and height of the feature map, K is the number of classes in the dataset, and A is the number of anchor boxes, which are introduced by the Region Proposal method in Faster R-CNN [3]. For instance, if one of the feature maps is 800x600, there is only one class, and the number of anchors is 9(by default), then the number of output nodes will be 800 x 600 x 1 x 9. For the box net, it is similar to the class net, only it produces 4 outputs per spatial location to anchor. The 4 outputs predict the offsets between the anchor and the bounding box. Therefore, the number of output nodes for a 800x600 feature map is 800 x 600 x 4 x 9. *Note: W, H, K, A are not part of output data, they simply denote how many output notes there are for each feature map.*

The RetinaNet paper also introduces a novel loss called *Focal Loss* [2] that adds a factor $(1 - pt)^{\gamma}$ to the standard cross entropy criterion. Focal loss is calculated by $FL(pt) = -\alpha t(1 - pt)^{\gamma} \log(pt)$, details can be found on the original paper.

3.1.2 gitHub/fizyr/keras-retinanet

https://github.com/fizyr/keras-retinanet is an implementation of RetinaNet object detection using Tensorflow and Keras. The repository is organized by https://github.com/hgaiser. The training and inferencing for our project are done using this repository.

Keras-retinanet is used in various projects in the tech industry, for instance, Microsoft Research for Horovod on Azure, Improving Apple Detection and Counting Using RetinaNet, and Improving RetinaNet for CT Lesion Detection with Dense Masks from Weak RECIST Labels.

The required labelling format for the training is as follow:

```
path/to/image.jpg,x_min,y_min,x_max,y_max,class_name
```

A full example:

```
dataset/img_001.jpg,837,346,981,456,motif
dataset/img_001.jpg,215,312,279,391,text_box
dataset/img_002.jpg,22,5,89,84,motif
dataset/img_002.jpg,82,85,288,108,text_box
```

The training script in keras-retinanet has a feature to allow random transformation such as 2D rotation and zooming in/out. Because of that, only the 3D rotation part of our data generation work was utilized in the training.

3.2 Environment

3.2.1 Hardware

CPU: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, 2808 Mhz, 4 Core(s), 8 Logical Processor(s)

Memory: 12 GB

GPU: Nvidia GTX 1050 Ti

3.2.2 Platform

The training is done on **Google Colab**. Because it is easy to share and use, and it has a feature to enable GPU as accelerator, which makes utilizing GPU easier.

3.3 Training

Complete training and testing process is as follow:

1. Open a new notebook on Google Colab
2. Upload folders dataset and test_dataset which contain training dataset and testing dataset.
3. Upload folders annotations.csv and test_annotations.csv which contain training annotations and testing annotations.
4. Clone gitHub repository Keras-retinanet
5. Install required packages
6. Read both annotations files into Pandas dataframe
7. Save all class names
8. Start the training by running train.py
9. Load the saved checkpoint
10. Test the trained model by having it inferencing every image in the test dataset
11. Record the results and performance

The version of Tensorflow has to be 2.2.0rc3, which is not a stable release. The latest stable release of tensorflow has a conflict with Keras that prevents tensorboard from working.

Python package Pandas is used to read csv files into Dataframes. Data in Dataframes is easy to access. The data is saved into w new annotations files with headers removed.

Script keras_retinanet/bin/train.py is the training script. The usage of it is

Train.py --freeze-backbone --random-transform --weights {model} --batch-size batch_size --steps number_of_steps --epochs number_of_epochs --tensorboard-dir tensorboard_output_dir
Where argument freeze-backbone means the backbone layer of the remains unchanged random-transform means allowing random transformation for training
weights is the argument to load pretrained weights and bias Tensorboard-dir enables the Tensorboard callback which allows usage of tensorboard.

After each epoch, the model will automatically save a checkpoint to ./snapshots folder.

The training model is converted to an inferencing model using methods models.load_model(model_path) and models.convert_model(loaded_model) where model_path is the latest saved checkpoint and loaded_model is the loaded model.

Testing the model is done by model.predict_on_batch(image), where image is an image row from a pandas dataframe. When given an image row from a dataframe, the model will produce a list of coordinates of boxes with scores and class names. The scores are possibilities of the class name for each box.
Thres_score is the minimum score. We set it to 0.6, meaning that only boxes with scores higher than 0.6 will be printed out. Matplotlib and openCV are used to draw the resulting images.

3.5 Training Performance

The following table and graphs show that the runtime is roughly the same for each epoch with the exception of the first epoch which takes about 13 seconds more than others. The total loss, classification loss and the box regression loss are all in a downward trend, meaning, within 10 epochs, the model becomes more accurate the more epochs there are.

10 Epochs; 500 steps per epoch

Epochs	time(in seconds)	loss	regression_loss	regression_loss
1	375	1.9284	1.3682	0.5602
2	362	1.1172	0.8994	0.2177
3	361	0.8529	0.6943	0.1586
4	362	0.6942	0.5729	0.1212
5	366	0.6029	0.5047	0.0982
6	366	0.5292	0.4493	0.0799
7	369	0.4851	0.4174	0.0677
8	369	0.4428	0.3843	0.0585
9	371	0.4156	0.3645	0.0523
10	371	0.3894	0.3444	0.0453

Table 1. Performance per Epoch

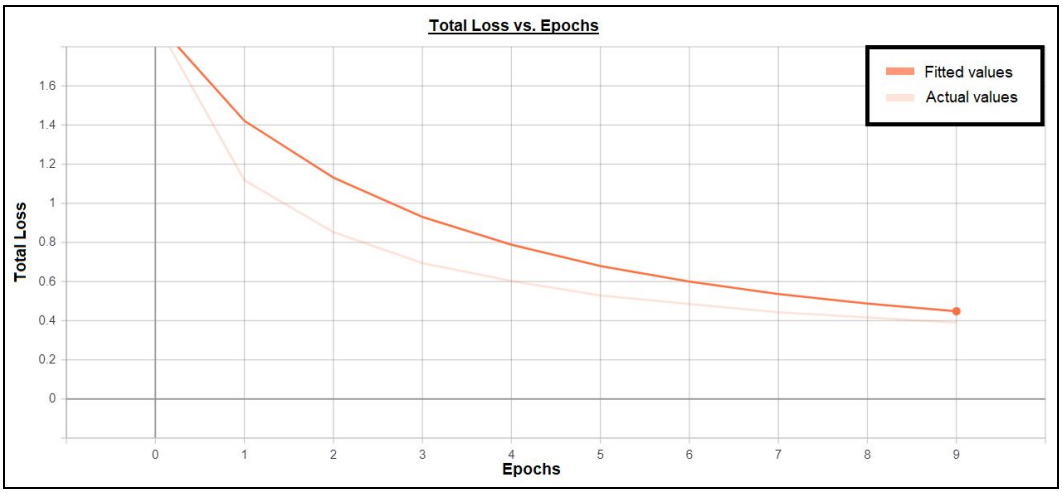

Figure 9. Total loss vs epochs

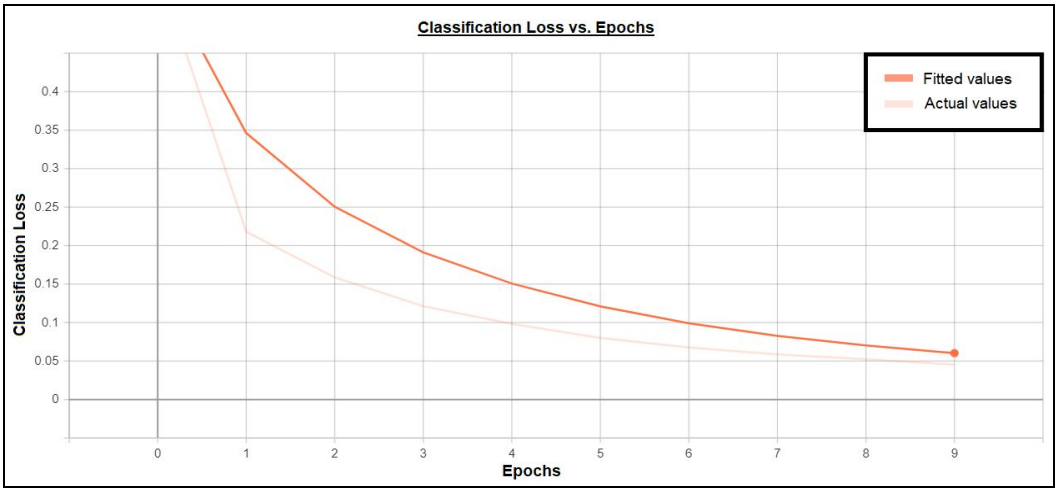

Figure 10. Classification loss vs epochs

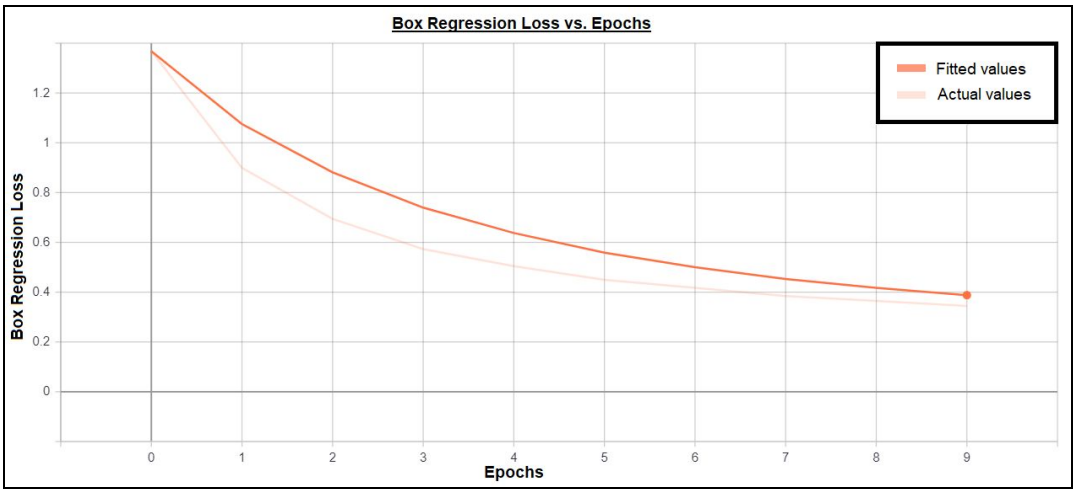

Figure 11. Box regression loss vs epochs

3.6 Results

The testing dataset contains 160 images. There are variants of 4 distinct images. We had the trained model predict the results of all 160 images and recorded the results. We manually check the accuracy of the results, because it is hard to factor in the margin of error for the predictions, since we labeled the boxes manually, some coordinates might not be precise. The results show that although all bounding boxes contain what they are supposed to contain, some bounding boxes are much larger, containing more than just their intended targets. We decided that bounding boxes that are much larger than motifs or textboxes are incorrect bounding boxes.

total	**160**
Correct textboxes (not too big)	**120 (all wrong textboxes are from variants of the same image)**
Correct motifs (not too big)	**113**
Correct both	**73**
Wrong both	**0**

Table 4. Results

Some examples of the printed images and bounding boxes:

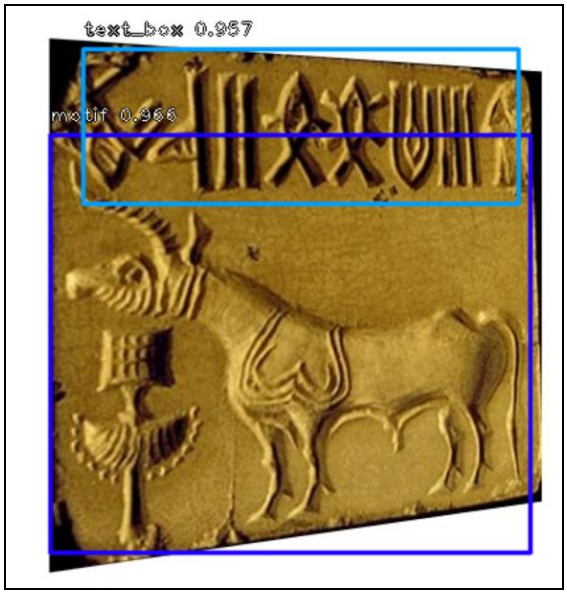

Figure 12. IRV artifact result example

Correct predictions for both textbox and motif. Some minor errors are ignored such as the small gap between the tail and the right side of the box, because our manual labeling is not precise and the image quality is not high.

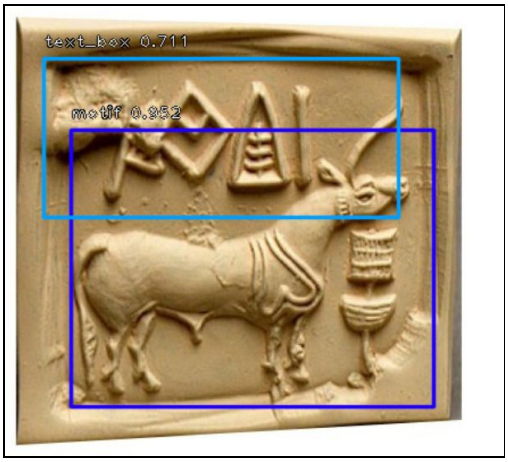

Figure 13. Too much text detected

Correct prediction for motif but wrong prediction for textbox. The horn is not part of the textbox. All the incorrect results for textbox come from the variants of this image.

Figure 14. Too much motif detected

Correct prediction for textbox but wrong prediction for motif. Bounding box for the motif is way bigger than it should be.

None of the results have wrong predictions for both motif and textbox

5 Conclusion

The image transformation approach to make up a large dataset of images with only a few images, while working to an extent, it's not an ideal approach for training neural networks(at least not ideal for object detection models that only take two corners into consideration such as RetinaNet). The accuracy of bounding box generation by the trained model based on 160 predictions is not great, with 25% of textbox bounding boxes and 29% of motif bounding boxes being larger than intended. The low accuracy may be due to the fact that the training dataset does not have enough distinct images. Although our training dataset only includes 880 images, because the model itself allows random transformation during training, the number of actual training instances is a few times more than 880 by this project's standard(a transformed image counts as a new image). However, the 880 images are transformed from 22 distinct images. 26 distinct IVC artifacts images were what we could find on the Internet, which 4 of them are transformed into 160 images to serve as testing data. The training dataset needs more distinct images. While we do not know how well the model will perform when trained with more distinct images and how many distinct images are needed, an online source suggests that training with 160 distinct images can achieve 90% accuracy. In this article, https://www.curiousily.com/posts/object-detection-on-custom-dataset-with-tensorflow-2-and-keras-using-python/, the author trains the RetinaNet model with 180 high-quality images of car plates and gets good results.

Another reason that causes the low accuracy is the nature of IVC artifact images. Text boxes and motifs' bounding boxes sometimes overlap each other. For example,

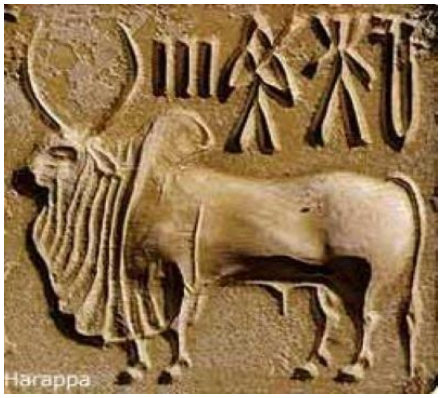

Figure 15. Original IRV artifact

Figure 16. Incorrect bounding box

Figure 8 is one of the training images. In this image, the bounding box for the goat is almost the entire image.

The text box is completely included by the motif bounding box. This may not be an issue when training with only one class, but when training with 2 classes, the model might get "confused", thinking the goat and the text are combined into one class name "motif".

There are a few images in the training dataset like this one and because of the way our dataset is constructed, the bounding boxes generated are sometimes larger, including unintended regions. Also, since we labelled the pre-transformed images manually, the bounding boxes we drew may not be perfectly precise, which hurts the accuracy of predictions.

Future Improvement

A future improvement that can be done to this project will be training with more distinct images. Like stated above, image transformation only works when there are enough distinct images before the transformation. A new way to fetch a dataset of distinct IVC artifact images must be found, since there is a lack of them on the internet.

If more distinct IVC artifacts are not to be found and the transformation technique is to be kept, a new model of neural network must be adopted, preferably a neural network that takes 8 corners into account. Object detection models might not be ideal since most major ones only deal with 2 corners. Regular CNNs with Tensorflow and Keras predicting 8 numbers could potentially have better results.

References

[1] Chollet François. “4.4 Overfitting and Underfitting.” *Deep Learning with Python*, by Chollet François, Manning Publications Co., 2018, pp. 104–105.

[2] Lin, Tsung-Yi, et al. "Focal loss for dense object detection." *Proceedings of the IEEE international conference on computer vision*. 2017.

[3] Ren, Shaoqing, et al. "Faster r-cnn: Towards real-time object detection with region proposal networks." *Advances in neural information processing systems*. 2015.

[4] T.-Y. Lin, P. Dollar, R. Girshick, K. He, B. Hariharan, and ´ S. Belongie. Feature pyramid networks for object detection. In CVPR, 2017. 1, 2, 4, 5, 6, 8

[5] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In CVPR, 2016. 2, 4, 5, 6, 8