

CS5339 Project Report

A0163158A: Mannu Malhotra

A0065571A: Patrick Cho Chung Ting

The Problem

For this project, we used the Kaggle dataset from The Nature Conservancy Fisheries Monitoring. They have installed multiple cameras in fishing boats to capture the activities. However, labelling these images manually is too labor intensive. Hence, there is a need to develop an image classification algorithm.

Training Data:

The training data given consists of 3777 images. Each image is taken from a camera attached on a fishing boat and can be of different dimensions. Moreover, the images are taken both in the day and at night. The figure below shows some sample images from the dataset.



Figure 1: Sample Images from the dataset

The objective is to classify each image into one of eight categories (ALB, BET, DOL, LAG, NoF, OTHER, SHARK, YFT). In particular, NoF represents “No Fish” in the scenario where the image contains no fish and OTHER represents the existence of fish in the image which is not a species of one of the other 6 categories.

Test Data and Evaluation:

The test data consists of 13153 images. For each image, the hypothesis needs to output a set of probabilities for each of the 8 categories. The loss function is calculated using the log loss function.

Initial Explorations with dataset:

When we first looked into the dataset, we realized a couple of challenges. Namely,

1. The item to classify can exist in a very small part of the image. Unlike other image classification datasets in which the item to be classified is usually considered the subject of the image, in this case, the item to be classified may take up only a small portion of the image.
2. Night vs Day photos can differ by a lot
3. Different cameras tend to produce very different photos. For example, the image on the far right in figure 1 is taken by a camera which always produces photos with a green tint.

Method 1: Support Vector Machines

For our first model, we wanted to try out something simple to implement a model and so we went ahead with out-of-the-box SVMs which have been used frequently for image classifications.

Resizing of Images:

Before inputting the images into the SVM, we have to resize the images for two reasons. Firstly, the original image sizes differ while SVMs require inputs to be of same dimensions. Secondly, given that we only have 3777 training examples, having a feature size of roughly $500*500*3=750,000$ might be too large. Although SVMs tend to do well even when the number of features exceeds the number of training examples due to the regularization term, C , we believed that further adding an Radial Basis Function (RBF) kernel would further increase the complexity of the model and result in overfitting. Hence, we decided to decrease the number of features to roughly the number of training examples. Concretely, we resized the images to $32*32*3=3072$ features.

Grid Search for tuning of parameters:

For the SVM, we used grid search to tune the parameters of C and Gamma and used a simple linear function together with the RBF kernel. Tuning of the regularization parameter, C , as well as kernel parameter gamma is important to prevent overfitting. Concretely, we select parameters such that the cross validation error is minimized in order to reduce overfitting.

Probability in SVM Models:

One problem that lies in the SVM Models is that it does not naturally come with the notion of probabilities. SVMs try to create a classification boundary with a large margin and do not assign confidence levels (or probability of prediction being correct). Although Scikit-learn's implementation of SVMs contains the function `predict_proba`, this is done as a separate step after the SVM is trained through a process known as Platt Scaling. In short, Platt Scaling adds a layer of logistic regression after the margin has been decided upon by the SVM such that points further away from the margin are given a higher probability. While this approach does endow SVMs with the notion of probabilities, optimizing the hinge loss function in an SVM may not in turn optimize the log loss function which we are being evaluated upon. This is one disadvantage of using SVMs when log loss is the evaluation metric.

Results:

The SVM using a RBF kernel with $C=100$ and $\text{gamma}=0.01$ gave a public log loss of 1.25219 and a private log loss of 2.31062¹.

Method 2: Convolutional Neural Networks

After getting a high log loss using the SVM model, we used a CNN model instead for prediction.

¹ The public dataset uses 8% of the test data while the private dataset uses 92% of the test data. It seems that the public dataset is more similar to the training dataset as compared to the private dataset. Hence, while the leader on the public leaderboard had a log loss of approximately 0.3, the leader on the private leaderboard could only achieve a log loss of approximately 0.8.

The configurations of the CNN network can be referred to in Appendix A.

Different input image sizes:

In our CNN we tried resizing input images (test and training) to the following resolutions:

- 1) 32*32
- 2) 64*64
- 3) 128*128

Following are our observation on different size of input image:

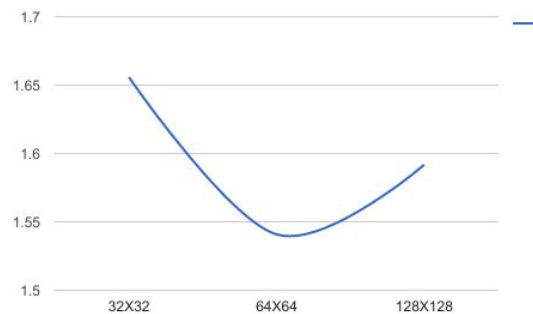


Figure 2: Plot showing relation between input image resolution to CNN model and log loss

For 32*32 image size our log loss was quite high, which is quite obvious for such low resolution image. It would be quite difficult for model to extract feature from such small size.

Log loss went down as expected when we doubled the image resolution from 32*32 to 64*64.

However, interestingly, the log loss went up when we chose 128*128 resolution for our input images.

One possible reason is that our network only uses 2 max pooling layers. A smaller image would be able to combine local features of bigger patches of the original image as compared to larger images. For instance, if the original image were 512*512, reducing its size to 128*128 and having 2 max pool layers would result in a final image size of 32*32 before the fully connected layers. This allows only patches of 16*16 in the original image to have interactions through the convolutions. Compare this with a resizing to 64*64 which would allow patches of 32*32 in the original image to have interactions through the convolutional layers. This larger area might be required to combine the different low-level features of a fish.

After determining the ideal input image size we tried tuning the CNN for the following parameters:

1. Number of hidden layers
2. Number of hidden units

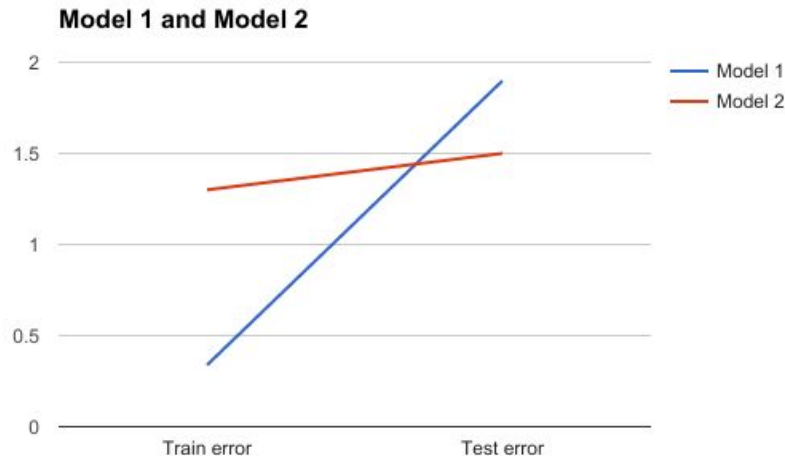
Following were the experimental values we tried:

1. Number of units in hidden layers:

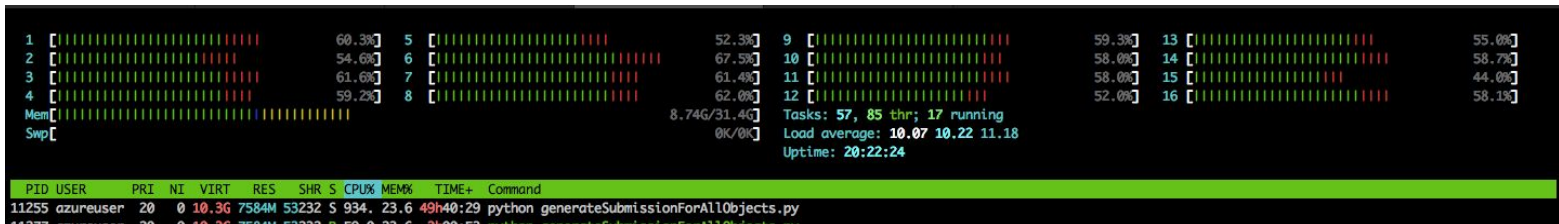
Model 1: 40X3X3 for 1st convolutional layers and 80X3X3 for 2nd convolution layer

Mode 2: 4X3X3 for 1st convolutional layers and 8X3X3 for 2nd convolution layer

As shown in diagram model 1 having huge number of layers over fitted a lot



Another challenge we faced during training model 1 was huge CPU time. With so many units and with image size 64*64 model was taking huge amount of time, Hence spawned a 16 cores virtual machine for running the model and it took around 50 CPU hours to run the model.



Clipping of results:

Clipping helped to decrease our log loss by .01 factor. When calculating log loss, a high penalty is given when either a prediction of very high probability turned out not to be the correct class or a prediction of very low probability turned out to be the correct class. Clipping helps to avoid such hard punishments by setting a maximum and minimum certainty.

Maximum and minimum certainty we took: 0.9 and 0.1/7 (0.0142857143)

Method 3: Object Proposal + Convolutional Neural Networks

In our final approach, We hypothesized that the small subject relative to image size makes image classification on an entire image difficult. We reasoned that if we could crop smaller parts of the images that could potentially be fish and use this cropped image for classification instead, the image classifier would do much better.

Object Proposal:

The first step in this approach is to come up with an algorithm for determining bounding boxes within an image that could potentially be fish. One way to do this would be to use an object proposal algorithm. This algorithm would generate a large number of bounding boxes that are potentially objects. A classifier would then be able to classify these potential objects into the actual object classes.

After doing some research on object proposal algorithms, we found out that Selective Search and EdgeBox work pretty well and decided to go with EdgeBox because of its speed in generating object proposals. EdgeBox works by first running an edge detection algorithm on an image. The main hypothesis of EdgeBox is that “the number of contours wholly enclosed by a bounding box is indicative of the likelihood of the box containing an object”. Concretely, edges tend to represent object boundaries. Therefore, if the contours of edges all lie within a box, then it is likely that the box contains an object. Of course, penalties also need to be given to bigger boxes since we would not want a box such as the entire image as an object proposal. Using this simple hypothesis, a score can be given to each bounding box such that any edge that passes the boundary of the box is penalized and the size of the box is also penalized. The highest scoring 1000 boxes are then used as object proposals.

After generating these object proposals, we compared to ground truth labels contributed by members of the Kaggle community. Concretely, we calculated the Intersection Over Union (IOU) of the generated object proposals with the ground truth bounding boxes and determined a box to contain that particular type of fish if the IOU exceeded a threshold of 0.5. Figure 3 below shows the object proposals on the left and the boxes with IOU exceeding 0.5 on the right.

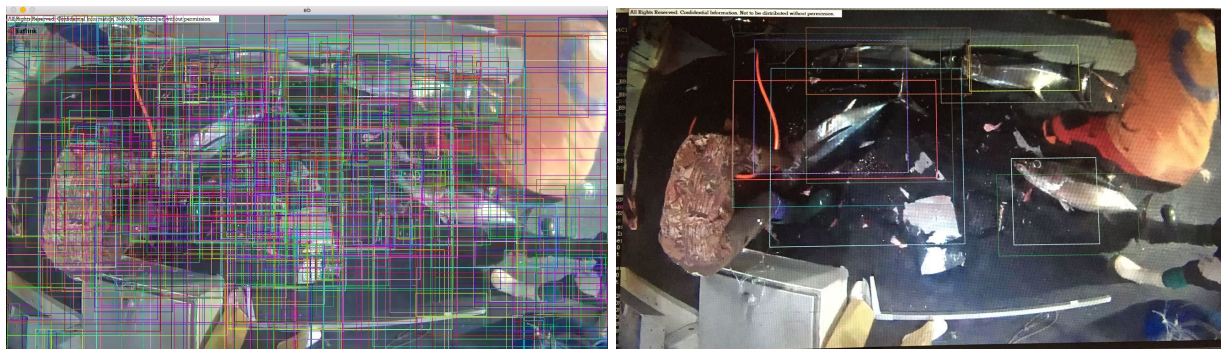


Figure 3: Example of result of Object Proposal Algorithm

Training Error:

After garnering object proposals, the next step would be to train a classifier to classify these object proposals into the correct fish category. We used the same CNN that we built in Method 2 for this purpose. To test our hypothesis that the CNN would do better on images in which the fish are the main subject of the image (e.g. take up more space in the image), we compared the training log loss using 3 different training sets. The first training set consists of the original images and got a training log loss of approximately 1.35.

Inspiration from R-CNN:

This approach of generating object proposals prior to classifying is very similar to the R-CNN algorithm proposed by Ross Girshick et al. in their paper titled “Rich feature hierarchies for accurate object detection and semantic segmentation”. In R-CNN, an object detection algorithm is built by first running an object proposal algorithm before running a CNN to classify each box into different object classes. A catch-all category of nothing is added and most boxes fall into this category. They also combine this CNN classification with a bounding box regressor to garner more accurate bounding boxes for object detection. This approach was the motivation for our 3rd method.

Combining Object Proposal with CNN:

The main difference between R-CNN and our algorithm lies in the objective. While R-CNN is an object detection algorithm, our ultimate objective in this problem is to do object classification. Hence, we needed a method to combine these 1000 object classifications into one single object classification for the entire image. The approach of simply averaging the scores of these 1000 objects would not be accurate since most objects would simply be in the category of “NoF”. Hence, we went with the approach of discarding all bounding boxes in which the highest probability class is “NoF”. If there were no bounding boxes left, we would assign a probability of 0.05 to each class of the other classes and a probability of 0.86 to NoF . If not, we would simply average the values of all the bounding boxes. Such an approach is similar to the idea of each box (that is not NoF) getting a vote on what to classify the original image as. For instance, if 3 boxes think it should be ALB while 1 box thinks it should be BET, ALB would get a higher vote percentage and hence ALB will be the more likely category. However, if the 1 box thinking it should be BET is extremely confident, it is still possible that BET is the category with the highest probability.

Testing Time and Possible Improvements:

One problem with this approach is the required time to generate a submission. Given that the test set has 13153 images and we need to run the CNN 1000 times per image, this equates to running the CNN 13153000 times. This makes prediction at test time extremely slow. In fact, we could not generate a submission on Kaggle prior to the end of the competition. We will attempt to generate a submission soon and update once we get the results of this method. One possible way to improve test time performance is to run the convolution layers on the entire image first. Since many of these bounding boxes overlap, the convolutions done on these bounding boxes can be reused. Finally, we need to resize these bounding boxes to fit to the Fully Connected layers which have a constant number of input dimensions. This can be done via some form of max pooling and is, in fact, the main thesis behind Fast-RCNN which is the improved version of R-CNN. However, we did not have the time to implement this approach.

Conclusion:

The final results of our Kaggle Performance can be found in Appendix B. The source code used for all of our experiments in this project can be found in Appendix C.

Appendix A:

CNN Configuration:

1. ZeroPadding2D((1, 1), input_shape=(3, 64, 64), dim_ordering='th')

To pad the input image with 1 (for both height and width). Doing this ensures that the output of the convolution is still 3*64*64.

2. Convolution2D(4, 3, 3, activation='relu', dim_ordering='th')

2D convolution layer, to create a convolution kernel that is convoluted to the layer input to produce a tensor of outputs.

3. ZeroPadding2D((1, 1), dim_ordering='th')
4. Convolution2D(4, 3, 3, activation='relu', dim_ordering='th')
5. MaxPooling2D(pool_size=(2, 2), strides=(2, 2), dim_ordering='th')
6. ZeroPadding2D((1, 1), dim_ordering='th')
7. Convolution2D(8, 3, 3, activation='relu', dim_ordering='th')
8. ZeroPadding2D((1, 1), dim_ordering='th')
9. Convolution2D(8, 3, 3, activation='relu', dim_ordering='th')
10. MaxPooling2D(pool_size=(2, 2), strides=(2, 2), dim_ordering='th')
11. Flatten()

After passing the input from various convolution and pooling layer we have to flatten the input to be able to prepare the input data for dense layers.

12. Dense(32, activation='relu')

To apply a fully connected layer of input units = 64.

13. Dropout(0.5)

14. Dense(32, activation='relu')

15. Dropout(0.5)

To prevent from overfitting we used a dropout layer with 1/2 rate.

16. Dense(8, activation='softmax')

Kept final layer as softmax with eight labels.

Note: number of input layers (16) and input units (32 fully connected) used was arbitrarily number, we did tuning later

Appendix B:









Final Results:

On kaggle leaderboard, we were able to finish on 136th position out of more than 2000 teams.

← → ↻ Secure https://www.kaggle.com/c/the-nature-conservancy-fisheries-monitoring/leaderboard ☆

OverviewDataKernelsDiscussionLeaderboardMoreMy SubmissionsSubmit Predictions

⚠ This competition has completed. This leaderboard reflects preliminary final standings. The result will become final after the competition organizers verify the results.

134	▼ 60	CV626		1.84761	1	6d
135	▲ 149	ahb1		1.84948	4	2d
136	▲ 180	mannuM		1.85130	1	7d
137	▲ 30	ZepplinRules		1.85527	2	2d
138	▲ 18	PradeepG		1.85534	8	1d
139	▼ 97	swordFish-[Anjith Happy Anirud]		1.85695	5	7d
140	▲ 17	skydream		1.85871	1	7d
141	▲ 139	DimaSakovich		1.86206	5	3d

Shown in the image is our position on kaggle leaderboard

with a log loss of 1.8513.

Appendix C:

Source Code:

Code for SVM can be found at:

<https://github.com/patrickcho168/cs5339/blob/master/svm.py>

Code for CNN can be found at:

https://github.com/patrickcho168/cs5339/blob/master/original_model.py

Code for EdgeBox can be found at:

<https://github.com/patrickcho168/cs5339/tree/master/edgebox>

Code for Combining Object Proposal and CNN for Image Classification can be found at:

<https://github.com/patrickcho168/cs5339/blob/master/generateSubmissionForAllObjects.py>

Code snippet for Clipper can be found at:

<https://github.com/patrickcho168/cs5339/blob/master/clipper.py>

References:

R. B. Girshick et al., "Rich feature hierarchies for accurate object detection and semantic segmentation," CoRR, vol.1311.2524. 2013.

C. L. Zitnick and P. Dollar, "Edge Boxes: Locating Object Proposals From Edges," EECV. Sep, 2014.

R. Girshick, "Fast R-CNN," ICCV. 2015.

(2016, Nov 15)

<https://www.kaggle.com/anokas/the-nature-conservancy-fisheries-monitoring/finding-boatids/comments>

(2016, Nov 29)

<https://www.kaggle.com/c/the-nature-conservancy-fisheries-monitoring/discussion/25902>

(2016, Nov 14)

<https://www.kaggle.com/zfturbo/the-nature-conservancy-fisheries-monitoring/fishy-keras-lb-1-25267/code>