CS342, Assignment 2 - Report

Harmeet Singh - 1404037, BSc MathStat

Abstract:

In this report I aim to analyse my developed construction of models capable of moderately classifying images of fish to their corresponding label. I will begin by performing some preliminary analysis by going through the training images. Next I will move onto implementations of three feature engineering techniques, namely Image Thresholding, Histogram of Ordinary Gradients and Data Augmentation. Following on, I will then explore the performance of a basic MLP without any feature engineering, tuning it with cross validation and subsequently move onto the MLP with the feature engineering techniques that we discussed implemented on top. Having constructed these models, I will end with implementing a convolutional neural network framework in order to find an optimal log_loss score required for the competition.

Data Exploration and Feature Engineering

Task 1:

Initially, I began by exploring individual images in each of the 8 folders within the training set. Nearly all of the images are roughly 1280 x 720 in resolution. This is quite a large resolution for us to work with when training our MLPs and CNNs, therefore, I immediately resized the images to a small and computationally fast resolution of 32 x 32.

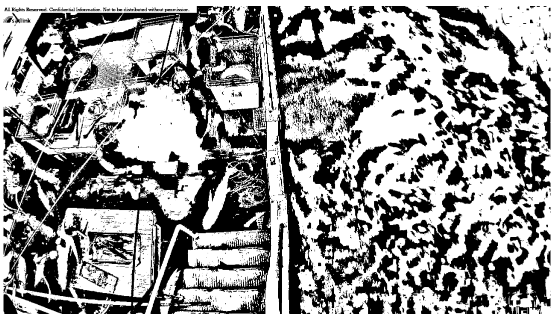
Although I knew this project was going to be difficult, I did not know how much until I observed the various differences from image to image. These included, different fisherman, different boats, quantity of fish per image etc. One more important discovery was that each folder contained some images that were taken in low light (most likely during deep night), and therefore had high a intensity of green due to the night vision mode of the camera. Since for the entirety of the project we are working with arrays of RGB values, it is good to use these to compare the intensity in pixel values of similar images. For example, let us consider two images of ALB (Albacore Tuna) fish from the training set, both of which are taken in low light.

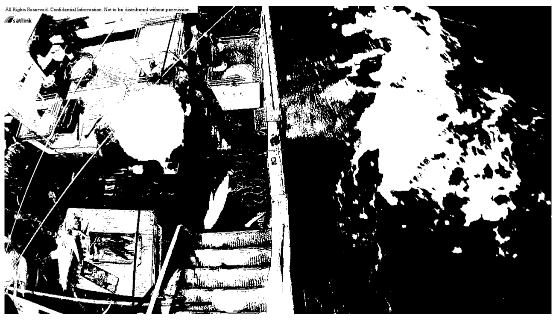
By observing (a) in Figure 1 of the Appendix, we can see that most pixels are are of low intensity, and their corresponding RGB values are also very low which due to the very low light in the image. This is in contrast to (c) which is shows that there exist many bright pixels in the image, even though it was taken in night vision. This makes sense since a good night vision image is visually clear and have a strong shade of green. This is supported by (d) which shows there exist more green pixels on high intensity (i.e. more than 120) than both red and blue. By these findings we see that images like img_00019 may not provide sufficient detail for our models to pick up information. Therefore, we shall try to tackle this problem with a feature engineering approach called image threshold shown on the next page.

Task 2:

Image thresholding:

Image threshold is a technique used to transform a grey scale image into a binary image, i.e. more simply each pixel is replaced by a black pixel if it is less than a fixed intensity and similarly replaced with a white pixel if greater than a fixed intensity. We will be implementing this technique in the hope to tackle the low light issue discussed in the previous page. We begin by transforming the image into a grey scale via the function *cv2.cvtColor* with the image and *'cv2.COLOR_BGR2GRAY'* as parameters. Next we simply achieve our threshold image via *cv2.adaptiveThreshold* which checks the size of pixel values and turns them black or white as we just discussed. The function also contains a Adaptive Method, which allows us to specify how our thresholding value is calculated. After some experimenting, I have determined that *cv2.ADAPTIVE_THRESH_GAUSSIAN_C* and *cv2.THRESH_OTSU* perform well in distinguishing the shape of a fish from the surrounding environment, where the former determines threshold values which are the weighted sum of the neighbourhood values with Gaussian weights.

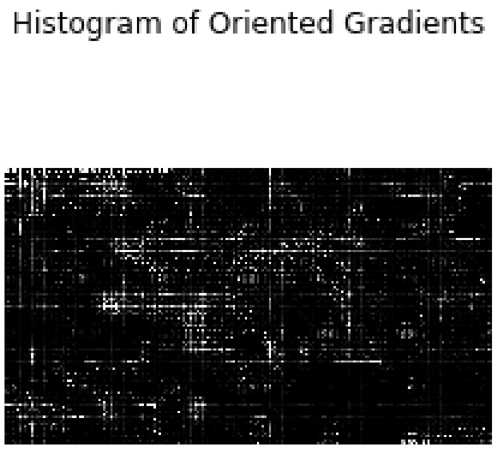

(a) A Produced using Gaussian adaptive threshold


(b) A Produced using Otsu's binarization

As you can see, both adaptive thresholds do well in outlining the shape of the fish. However, (b) has far less groups of small white pixel areas, evident by observing the sea water (shown in the right side of (a)). As such, I have chosen to implement Otsu's binarization for the MLP in order to prevent the model from learning irrelevant areas of the image.

Histogram of Gradients:

This is another useful and well known feature engineering technique aimed at object detection. We begin by performing an optional global image normalisation, which is a kind of pre-processing serving to deduct any impact of illumination effects. Next, we calculate the vertical and horizontal gradients of the image in the kernels (-1,0,1) and $(-1, 0, 1)^T$. This again reduces illumination effects allows by allowing detection of silhouette and texture information. Continuing from these steps, we move further into computing gradient histograms, normalising across blocks and then finally achieving a flattened feature vector which we use for window classification. Below is an example of converting a input image into a Histogram of Oriented gradients.



From the HoG above, you can slightly make out a similar shape to that of the Bigeye Tuna on the left, nonetheless, for this example the HoG has not performed well in detecting the fish. But this does not mean it can't! We will have to wait and test it out with the MLP.

Data Augmentation:

This is a much more simpler feature engineering approach than the previous two. Over here we are concerned with doing more with what we've got. It is true that with more data, comes more information. Since we have 1719 images of ALB fish in our training set and only 67 images of LAG, our trained model would be more likely to give a accurate prediction of ALB fish rather than LAG. But what can we do if we only have this little data? We can make more ourselves! We will apply numerous random transformations to our currently existing images in LAG, so that we can create further images of the same fish and more importantly, don't produce the same picture twice.

```

def gen_imgs(img, amount):
  i = 0
  for batch in datagen.flow(img, batch_size=1, save_to_dir = 'augmented', save_prefix = '
    BET',save_format = 'jpeg'):
    i += 1
  if i > amount:
    break

def data_aug(fish):
  for k in range(0,10):
    img = img_read(img_get(fish)[k])
    img = img.reshape((1,) + img.shape)
    gen_imgs(img, 50)
```

Listing 1: Generating 500 augmented images of a particular class of fish.

Above is a part of the implementation of data augmentation that I have constructed in my script. The *gen_imgs* function loops through *datagen.flow* which was initialised prior to this (see script) and outputs an augmented version of itself into a specified folder, in this case 'augmented'. The *data_aug* function reads the first 10 images in a specified folder for a class of fish and calls the *gen_imgs* function to output a total of 500 augmented images.

Artificial Neural Networks and Deep Learning

Task 3:

To begin this task, I started with the default MLPClassifier, i.e. just *.MLPClassifier()* which had 1 hidden layer with 100 neurons. This of course gave a very poor score of 8.04727. I then extended the number of hidden layers to 4, with 10 neurons each. This improved the log_loss greatly and down to 1.90521. This is because the MLP was able to disect information efficiently through its multiple hidden layers. Furthermore, I kept on improving my model by tuning the parameters and testing them out with cross validation. The following are some of the results I achieved:

```
mlp1 = MLPClassifier(hidden_layer_sizes=(10,10,10,10),activation='logistic',solver='adam',
    momentum = 0.5)

#cVal1 = 2.2002026525318135

mlp2 = MLPClassifier(hidden_layer_sizes=(10,10,10,10),activation='logistic',solver='sgd',
    momentum = 0.

#cVal2 = 2.0922429092974477

mlp3 = MLPClassifier(hidden_layer_sizes=(10,10,10,10),activation='logistic',solver='sgd',
    momentum = 0.5)

#cVal3 = 1.9838959032447363

mlp4 = MLPClassifier(hidden_layer_sizes=(10,10,10),activation='logistic',solver='sgd',
    momentum = 0.5)

#cVal4 = 2.0007166038443303

mlp5 = MLPClassifier(hidden_layer_sizes=(10,10,10,10),activation='tanh',solver='sgd',
    momentum = 0.5)

#cVal5 = 2.1740325078100544
```

Listing 2: cross validation results for vairous MLPs

From above, mlp3 provided the lowest log_loss value after cross validation, and on kaggle it provides a score of 1.65160. As such, I have continued to use this model for the next task.

Task 4:

In this section, we implement the model from Task 3 but with the proposed feature engineering methods discussed in Task 2. Beginning with image threshold, my MLP model with the Gaussian adaptive threshold achieved a kaggle score of 1.65429, whereas the MLP model with the Otsu binarization achieved a score of 1.65294. Although the difference is extremely small, the Otsu binarization model still outperformed the model with the Gaussian adaptive threshold. This supports my assertion in Task 2, since the image for the Gaussian contained many small patches of white and black pixels in contrast to the Otsu which was in general more consistent in local environments. This allowed the Otsu model to better gather information from important objects instead of picking up a lot of different information randomly from various parts of the image as in the Gaussian.

Another observation is that the scores for both models were not improvements to the score of the MLP without any feature engineering. This is most likely because we used resized 32 x 32 images for the the Gaussian adaptive thresholds and Otsu binarization. The images that I showed for these two thresholds in Task 2 were made via the original images of resolution roughly 1280 x 720. This means that the model was not able to identify the features optimally and so only performed as well as it did for the 32 x 32 images without image threshold. The reason I have not made any improvements to the model by increasing resolution is because it in turn increases the computational time by a huge amount. In this case, it is not feasible to use such a model and one should move onto something else.

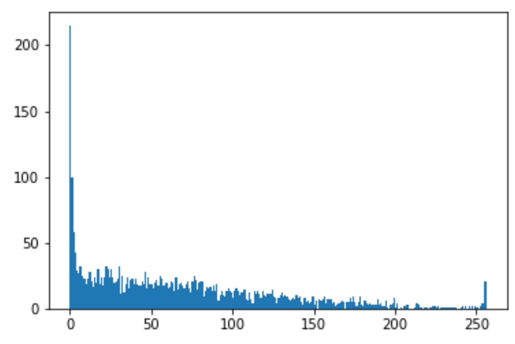
Task 5:

I began the CNN model using a basic CNN architecture from the 'basic script' specified in Lab 6. This involved three hidden layers with 8 neurons each, a *relu* activation function, *softmax* output and the 'adam' optimizer. This gave me a log loss score of 16.02551! Horrified by this result, I began experimenting with this architecture and tried to learn more about the general CNN framework. By this I came across special layers such as MaxPooling, which attempts to reduce the dimensionality of each feature map whilst also trying to retain the most important information. This in turn reduces the complexity of our framework and it becomes more like something the Occam's razor would prefer!

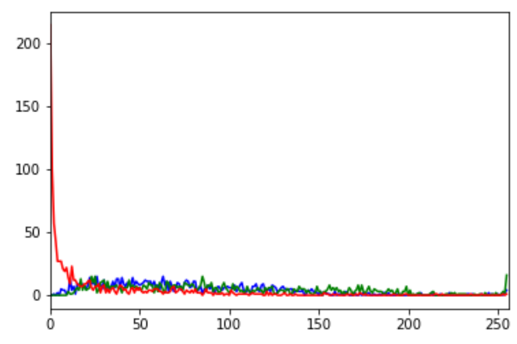
Furthermore, I also discovered the Dropout core layer. To implement this, we flatten the feature map into 1D (i.e. the map is reduced from the RGB state to just 1D) using the Dense layer. Then we initialise a fully connected layer with this Dense layer and apple a *relu* activation. This core layer is very useful since it tries to prevent overfitting in our model.

By using these new findings and altering parameter values in for example the no. of hidden layers or the momentum of the sgd function, I achieved my best score of 1.49077 .

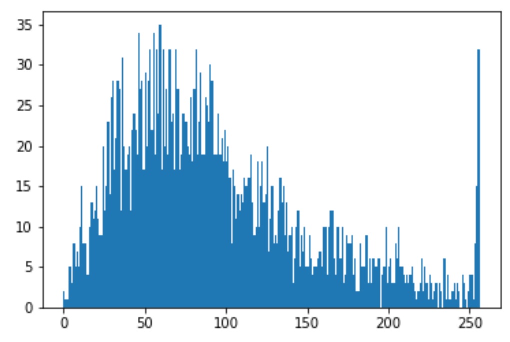
Appendix



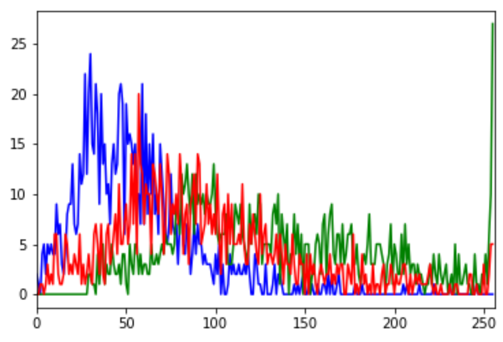
(a) Histogram for img_00019, bins=256



(b) BGR plot for img_00019



(c) Histogram for img_00097, bins=256



(d) BGR plot for img_00097

Figure 2: