

Data Retrieval

The URL links of the subject images are stored in a structured table, from the "open_images" dataset from BigQuery, whose link is -> <https://cloud.google.com/bigquery/public-data/openimages>

The following SQL query was done on the BigQuery platform of Google Cloud, to retrieve images of the desired labels, for both train and test.

```
SELECT thumbnail_300k_url, img.subset, d.label_display_name
FROM `bigquery-public-data.open_images.images` AS img
INNER JOIN `bigquery-public-data.open_images.labels` AS lbl ON img.image_id =
lbl.image_id
INNER JOIN `bigquery-public-data.open_images.dict` AS d ON d.label_name = lbl.label_name
WHERE lbl.source = 'human'
AND d.label_display_name IN ('Lemon','Baseball bat','Sea lion','Dolphin','Banana')
AND img.subset IN ('train','test');
```

This is the BigQuery link of the same:

<https://bigquery.cloud.google.com/savedquery/493893702004:67048603b3d14a5f9eedfb2364964430>

From the query, created a csv file which stored the image URLs, the subset they belonged to (train, test) and the labels.

Data Exploration:

- Performed some preliminary descriptive statistics for the data, which included the total count, column names, unique values, etc.
- There were 140 URLs which were blank, so removed those rows. After removing the NA values, there were a total of ___ images, and they were divided in the ratio of 85% train and 15% test images.
- Also another important step was to check the distribution of the 5 labels/classes, whether there was any imbalance. All the 5 labels were equally distributed, with roughly around 935 images on an average for each label.
- Another important observation was that there were 92 URLs which were repeated, and no URL was repeated more than twice, and surprisingly for each of the same images, the labels were assigned differently.

Reading Data and Pre-processing:

From the URLs, stored the train and test images and resized them to 160 X 160 pixels greyscale. Normalized the images and converted the labels to one-hot encoded vector forms.

CNN models used:

1. Building the CNN architecture from scratch:

The initial architecture of the CNN was designed with 3 convolutional layers, each followed by a maxpool layer, and two dense fully connected layers, and a softmax layer for the 5 classes:

conv - pool - conv - pool - conv - pool - flatten - fc1 - fc2 - dropout - softmax

This was not running on my laptop, hence reduced this to 2 convolutional layers, each followed by a maxpool layer and just one dense fully connected layer:

conv – pool – conv – pool – flatten – fc1 – dropout – softmax

Tried with 3 different optimizer functions (Gradient Descent, Adam, Momentum), however none were converging, and as a result at all times the system crashed after running the CNN, without giving any output.

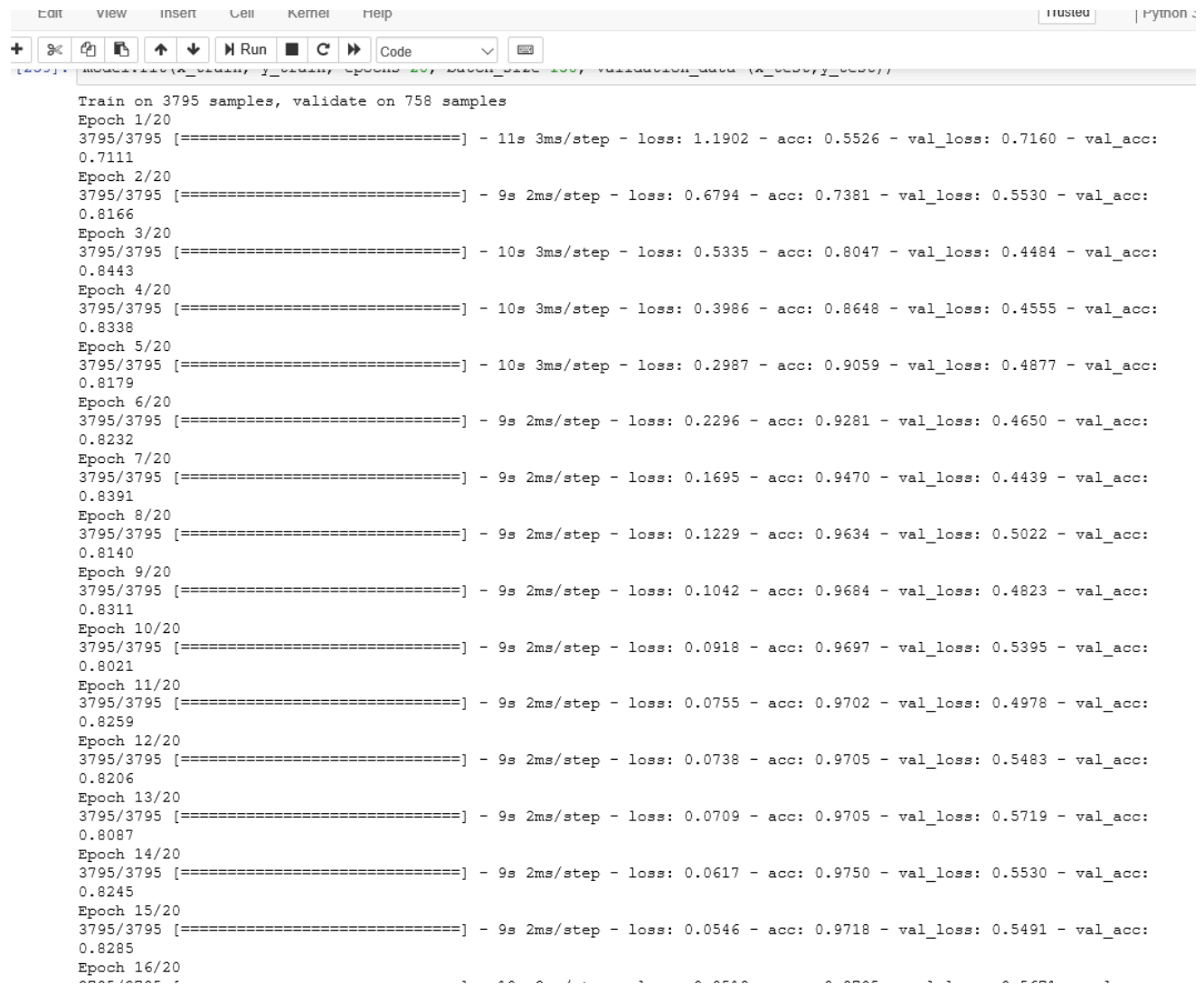
Steps taken to ensure CNN model runs:

- Used greyscale images instead of color channels, reduced the color dimension to 1, still didn't work.
- Reduced the image dimensions from the initially decided 250 X 250 to 160 X 160
- Dropped one conv, maxpool and a fully connected layer as well.
- Started with Adam optimizer, which took long time, so switched to simple gradient descent optimizer, which also took time.
- Reduced the batch size from 250 to 128 to 64.

2. Pre-trained VGG16 model using Keras:

Used the VGG16 pre-trained model. Basically acts as a feature extraction by passing the images through the network, which has all the weights parameters tuned on the ImageNet dataset. Made use of the entire model weights, added 2 dense layers and a output softmax layer.

The output was achieved as follows:



```
Train on 3795 samples, validate on 758 samples
Epoch 1/20
3795/3795 [=====] - 11s 3ms/step - loss: 1.1902 - acc: 0.5526 - val_loss: 0.7160 - val_acc: 0.7111
Epoch 2/20
3795/3795 [=====] - 9s 2ms/step - loss: 0.6794 - acc: 0.7381 - val_loss: 0.5530 - val_acc: 0.8166
Epoch 3/20
3795/3795 [=====] - 10s 3ms/step - loss: 0.5335 - acc: 0.8047 - val_loss: 0.4484 - val_acc: 0.8443
Epoch 4/20
3795/3795 [=====] - 10s 3ms/step - loss: 0.3986 - acc: 0.8648 - val_loss: 0.4555 - val_acc: 0.8338
Epoch 5/20
3795/3795 [=====] - 10s 3ms/step - loss: 0.2987 - acc: 0.9059 - val_loss: 0.4877 - val_acc: 0.8179
Epoch 6/20
3795/3795 [=====] - 9s 2ms/step - loss: 0.2296 - acc: 0.9281 - val_loss: 0.4650 - val_acc: 0.8232
Epoch 7/20
3795/3795 [=====] - 9s 2ms/step - loss: 0.1695 - acc: 0.9470 - val_loss: 0.4439 - val_acc: 0.8391
Epoch 8/20
3795/3795 [=====] - 9s 2ms/step - loss: 0.1229 - acc: 0.9634 - val_loss: 0.5022 - val_acc: 0.8140
Epoch 9/20
3795/3795 [=====] - 9s 2ms/step - loss: 0.1042 - acc: 0.9684 - val_loss: 0.4823 - val_acc: 0.8311
Epoch 10/20
3795/3795 [=====] - 9s 2ms/step - loss: 0.0918 - acc: 0.9697 - val_loss: 0.5395 - val_acc: 0.8021
Epoch 11/20
3795/3795 [=====] - 9s 2ms/step - loss: 0.0755 - acc: 0.9702 - val_loss: 0.4978 - val_acc: 0.8259
Epoch 12/20
3795/3795 [=====] - 9s 2ms/step - loss: 0.0738 - acc: 0.9705 - val_loss: 0.5483 - val_acc: 0.8206
Epoch 13/20
3795/3795 [=====] - 9s 2ms/step - loss: 0.0709 - acc: 0.9705 - val_loss: 0.5719 - val_acc: 0.8087
Epoch 14/20
3795/3795 [=====] - 9s 2ms/step - loss: 0.0617 - acc: 0.9750 - val_loss: 0.5530 - val_acc: 0.8245
Epoch 15/20
3795/3795 [=====] - 9s 2ms/step - loss: 0.0546 - acc: 0.9718 - val_loss: 0.5491 - val_acc: 0.8285
Epoch 16/20
3795/3795 [=====] - 9s 2ms/step - loss: 0.0550 - acc: 0.9725 - val_loss: 0.5675 - val_acc: 0.8255
```

```
3795/3795 [=====] - 10s 3ms/step - loss: 0.2987 - acc: 0.9059 - val_loss: 0.4877 - val_acc: 0.8179
Epoch 6/20
3795/3795 [=====] - 9s 2ms/step - loss: 0.2296 - acc: 0.9281 - val_loss: 0.4650 - val_acc: 0.8232
Epoch 7/20
3795/3795 [=====] - 9s 2ms/step - loss: 0.1695 - acc: 0.9470 - val_loss: 0.4439 - val_acc: 0.8391
Epoch 8/20
3795/3795 [=====] - 9s 2ms/step - loss: 0.1229 - acc: 0.9634 - val_loss: 0.5022 - val_acc: 0.8140
Epoch 9/20
3795/3795 [=====] - 9s 2ms/step - loss: 0.1042 - acc: 0.9684 - val_loss: 0.4823 - val_acc: 0.8311
Epoch 10/20
3795/3795 [=====] - 9s 2ms/step - loss: 0.0918 - acc: 0.9697 - val_loss: 0.5395 - val_acc: 0.8021
Epoch 11/20
3795/3795 [=====] - 9s 2ms/step - loss: 0.0755 - acc: 0.9702 - val_loss: 0.4978 - val_acc: 0.8259
Epoch 12/20
3795/3795 [=====] - 9s 2ms/step - loss: 0.0738 - acc: 0.9705 - val_loss: 0.5483 - val_acc: 0.8206
Epoch 13/20
3795/3795 [=====] - 9s 2ms/step - loss: 0.0709 - acc: 0.9705 - val_loss: 0.5719 - val_acc: 0.8087
Epoch 14/20
3795/3795 [=====] - 9s 2ms/step - loss: 0.0617 - acc: 0.9750 - val_loss: 0.5530 - val_acc: 0.8245
Epoch 15/20
3795/3795 [=====] - 9s 2ms/step - loss: 0.0546 - acc: 0.9718 - val_loss: 0.5491 - val_acc: 0.8285
Epoch 16/20
3795/3795 [=====] - 10s 3ms/step - loss: 0.0519 - acc: 0.9705 - val_loss: 0.5671 - val_acc: 0.8206
Epoch 17/20
3795/3795 [=====] - 10s 3ms/step - loss: 0.0459 - acc: 0.9731 - val_loss: 0.6886 - val_acc: 0.7757
Epoch 18/20
3795/3795 [=====] - 10s 3ms/step - loss: 0.0527 - acc: 0.9739 - val_loss: 0.5897 - val_acc: 0.8285
Epoch 19/20
3795/3795 [=====] - 9s 2ms/step - loss: 0.0498 - acc: 0.9713 - val_loss: 0.5819 - val_acc: 0.8338
Epoch 20/20
3795/3795 [=====] - 10s 3ms/step - loss: 0.0434 - acc: 0.9758 - val_loss: 0.5987 - val_acc: 0.8311
Out[259]: <keras.callbacks.History at 0x23684923550>
```

With a batch size of 150, and at the end of 20 epoch, the model achieved 97.58% train accuracy and 81% test accuracy. It was able to reach such an accuracy because of the fact that all of its layers weights were used, which were trained on 1.2 Mn images.

3. Pre-trained VGG16, with modifying the CNN layers:

This is also using a pre-trained VGG16 model, but the main difference here is that instead of just using the weights, we can also alter the architecture of the model, and build our own layers over the low level layers. The idea here is that a few low level layers will be freezed/fixed, such that the parameters won't get updated, but the rest network is subject to being trained on the image dataset that we provide, and also has the flexibility to modify and change the architecture as well.

I kept and freezed the layers only till block2_pool. Then on that, I built a conv + max pool layer, followed by a dense layer, dropout and finally the output softmax layer.

The following is the VGG16 architecture:

Layer (type)	Output Shape	Param #
input_11 (InputLayer)	(None, 160, 160, 3)	0
block1_conv1 (Conv2D)	(None, 160, 160, 64)	1792
block1_conv2 (Conv2D)	(None, 160, 160, 64)	36928
block1_pool (MaxPooling2D)	(None, 80, 80, 64)	0
block2_conv1 (Conv2D)	(None, 80, 80, 128)	73856
block2_conv2 (Conv2D)	(None, 80, 80, 128)	147584
block2_pool (MaxPooling2D)	(None, 40, 40, 128)	0
block3_conv1 (Conv2D)	(None, 40, 40, 256)	295168
block3_conv2 (Conv2D)	(None, 40, 40, 256)	590080
block3_conv3 (Conv2D)	(None, 40, 40, 256)	590080
block3_pool (MaxPooling2D)	(None, 20, 20, 256)	0
block4_conv1 (Conv2D)	(None, 20, 20, 512)	1180160
block4_conv2 (Conv2D)	(None, 20, 20, 512)	2359808
block4_conv3 (Conv2D)	(None, 20, 20, 512)	2359808
block4_pool (MaxPooling2D)	(None, 10, 10, 512)	0
block5_conv1 (Conv2D)	(None, 10, 10, 512)	2359808
block5_conv2 (Conv2D)	(None, 10, 10, 512)	2359808
block5_conv3 (Conv2D)	(None, 10, 10, 512)	2359808
block5_pool (MaxPooling2D)	(None, 5, 5, 512)	0
Total params: 14,714,688		
Trainable params: 14,714,688		
Non-trainable params: 0		

From the output below, it is evident that the algorithm has reached a minima, such that there are no further updates in the weights in the consecutive epochs.

```

Train on 3795 samples, validate on 758 samples
Epoch 1/20
3795/3795 [=====] - 1004s 265ms/step - loss: 12.8308 - acc: 0.2040 - val_loss: 13.2475 - val_
_acc: 0.1781
Epoch 2/20
3795/3795 [=====] - 979s 258ms/step - loss: 12.8308 - acc: 0.2040 - val_loss: 13.2475 - val_
acc: 0.1781
Epoch 3/20
3795/3795 [=====] - 975s 257ms/step - loss: 12.6319 - acc: 0.2161 - val_loss: 14.7572 - val_
acc: 0.0844
Epoch 4/20
3795/3795 [=====] - 980s 258ms/step - loss: 12.3338 - acc: 0.2348 - val_loss: 14.7572 - val_
acc: 0.0844
Epoch 5/20
3795/3795 [=====] - 1027s 271ms/step - loss: 12.3338 - acc: 0.2348 - val_loss: 14.7572 - val_
_acc: 0.0844
Epoch 6/20
3795/3795 [=====] - 1010s 266ms/step - loss: 12.3338 - acc: 0.2348 - val_loss: 14.7572 - val_
_acc: 0.0844
Epoch 7/20
3795/3795 [=====] - 1020s 269ms/step - loss: 12.3338 - acc: 0.2348 - val_loss: 14.7572 - val_
_acc: 0.0844
Epoch 8/20
3795/3795 [=====] - 1097s 289ms/step - loss: 12.3338 - acc: 0.2348 - val_loss: 14.7572 - val_
_acc: 0.0844
Epoch 9/20
3795/3795 [=====] - 1144s 301ms/step - loss: 12.3338 - acc: 0.2348 - val_loss: 14.7572 - val_
_acc: 0.0844
Epoch 10/20

```

Loss/Evaluation:

As has been confirmed earlier from the descriptive statistics, about the distribution of the classes/labels, all the labels have almost equal distribution in the instances in the train and test data, and there is no unbalanced class/ class skewness problem. Hence, accuracy can be a choice for the evaluation metric. For other scenarios though, we can go for precision and recall (this needs to be done for each class/label though). Log-loss/categorical cross-entropy is the ideal loss metric of choice when the probabilities for each class is being predicted. Log-loss basically penalizes more those instances whose probabilities are very far away from the ground truth. In simple terms, that means, if a particular label is present (ground truth is 1), log-loss penalizes more if the probability value for that label is say 0.3, rather than if it's probability value would have been, say 0.7.
