Final Project

Classification of Big Cats from Google Image Dataset

DATA 603 Platforms for Big Data Processing

University of Maryland – Baltimore County

Prof. James Klucar

Team 5

Kumra, Shouray (skumra1)

Mironov, Sanaa (has1)

Purushothaman, Govind Ram (gx54176)

Thakkar, Mahima Ashokbhai (db59735)

Introduction

A semester-long project on big data processing doing a simple classification model. The binary classification will be on predicting an image to be a big cat or not. The list of big cats within the Google Image Dataset includes Jaguar, Lynx, Tigers, Lions, Leopards, and Cheetahs.

Dataset

The Google Open Images V5 features segmentation masks for 2.8 million object instances in 350 categories. Unlike bounding-boxes, which only identify regions in which an object is located, segmentation masks mark the outline of objects, characterizing their spatial extent to a much higher level of detail. These masks cover a broader range of object categories and a more significant total number of instances than any previous dataset.

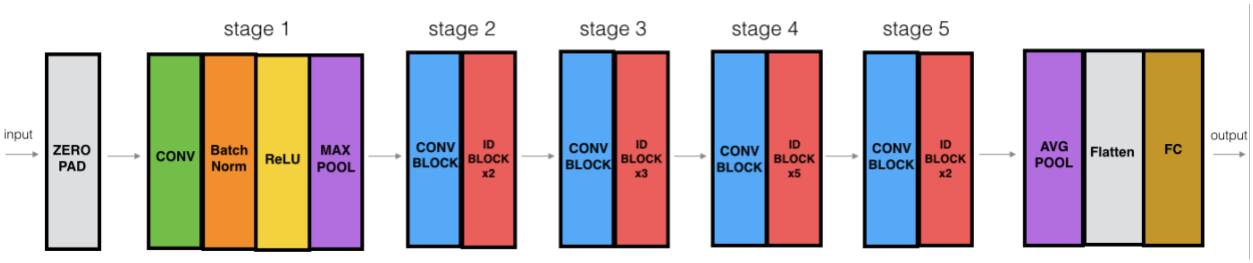
The segmentation masks on the training set (2.68M) have been produced by an interactive segmentation process, where professional human annotators iteratively correct the output of a segmentation neural network. This is considered efficient than manual drawing alone, while at the same time delivering accurate masks. In addition to the masks, 6.4M new human-verified image-level labels are present, reaching a total of 36.5M over nearly 20,000 categories.

Why ResNet50?

ResNet50 is a 50 layer deep, pre-trained Deep learning model for image classification of the Convolutional Neural Network, mostly used to analyze visual imagery. A pre-trained model is a more practical approach than collecting loads of data and training the model ourselves. It is trained on a million images of 1000 categories from the ImageNet database. The model has over 23 million trainable parameters, which indicates a deep architecture that makes it better for image recognition and classification. Also, ResNet50 has excellent generalization performance with fewer error rates than other pre-trained models like AlexNet, GoogleNet, or VGG19. The network has an image input size of 224-by-224.

Further, in a deep convolutional neural network like VGG16, several layers are stacked and trained to the task, and the network learns several low/mid/high-level features at the end of its layers. Whereas in residual learning, instead of trying to learn some features, we try to learn

some residual by subtracting the feature learned from the input of the layer. This is done by directly connecting the input of the nth layer to some (n+x) the layer. Training these kinds of networks is more accessible than training simple deep convolutional neural networks, and the problem of degrading accuracy is also resolved.



MOBILENET

MobileNet is a separable convolution (conv) module which is composed of depth wise and point wise conv. Depth wise conv is performed independently for every input channel of the image, this helps in reducing the computational cost by omitting conversation in channel.

Mobile net independently performs conv. MobileNet is efficiently used to predict the category we want if available in ImageNet.

Advantages of using MobileNet:

- MobileNets is easy to apply as it is light weight.
- Uses depth wise separable conv, which is a streamlined architecture and human understandable.
- Reduces size of the network
- Reduces number of parameters, resulting in availability of more space.
- Small Convolutional Neural Network, therefore, it does not forget the weights at large scale and gives out reliable predictions.
- Fast in performance as compared to few other pre-trained models.

Applications:

- Object detection
- Fine grain classification Face recognition

Disadvantages:

- There is a slight reduction in the accuracy when compared with other pre-trained models. In our project it was less accurate for on class but performed decently with other classes.

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Part 1: Preprocessing the data

Part will walk you through the data processing part of the project. Below will be detailed steps of the process we took to achieve our outcome.

To find labels within the Google open image dataset were done using google OID. In order to find a class of images you wanted to extract you search the OID labels by examining the OID label hierarchy to find labels. Keras Models use ResNet image labels. ResNet has 1000 labels. It's a good idea to see the different format between the two. ResNet labels are lower case, but Google OID are not.
To start the preprocessing of the files, a Spark configuration must be made, and Spark must be launched to work on UMBC cluster ecosystem.

Reading in labels through Spark syntax, you read in the CSV file 'class_description_boxable.csv'. To read in the file with a specific format you want, a schema must be created. We created a simple schema with a struct type LabelName and LabelText. The CSV file imported all labels within the Google Image Dataset, however, we are only interested in Big Cats. To remove any unwanted labels we call a filter function on the dataframe and set LabelText to Tiger, Lion, Cheetah, Leopard, Jaguar (Animal), and Lynx. Good practice to check if new dataframe removed all unwanted label and decreased the overall dataframe size.

Next, we download Image ID files into a another dataframe, each image id file will be in a separate dataframe, resulting in 3 dataframe. Again, a schema was defined for how we wanted to few and read in the Image ID data. For example, Confidence should be a double, not a string, this can be defined within our schema. Once the files has been read in, these files will be join together to create one large dataframe called image labels. The image label dataframe will hold all of the image labels form our original data set. The size of the dataframe is over 9 million images. This happened because of the label hierarchy structure, we have multiple labels for a single image. A big cat can be labeled as a Mammal and a Tiger or whatever specified name it has. To combat this problem, we group the image label by 'Source' and then use distanced 'ImageID' to filter out multiple labels from the same source.

With dataframe so large, we filter image labels through the means of using one dataframe to filter another one. This is done by using a smaller data frame to join up with a large dataframe. Since we filtered the dataframe image labels easier, we can join image labels with image ids. This will remove anything from image ids that is not in image labels. To make this dataframe even more

robust we add in filter of confidence greater than 99%. This will filter out the large dataframe, image id, with the interested labels and labels that have a high confidence interval.

To make sure that the dataframe image id has distinct we run another filter on it with confidence greater than 99% on the ImageID label. This resulted in a smaller dataframe that went from a size of over 9 million to a little over 3500 labels.

Parquet File Read

We create a new dataframe that reads in image parquet files from the UMBC cluster. The file we are reading in is images_coalesced.parquet which did another processing for us.
The Google Open Image dataset were too large to load into the HDFS ecosystem so the file were first written to an AVRO file which then were written to a parquet file. The parquet files broken down data to make it easier to read in and be able to mange the large dataset.
When reading in the image parquet file, we extract only the columns we are interested in like ImageID, Subset the image is from, and the raw data of the image itself. And format this column to match our other dataframe we have built so far.
Repeating the process of removing unwanted field by joining a smaller dataframe, image ids, with our current large dataframe, image parquet.

Bounding Box

At this point the data within our dataframes are not human-readable language and we do not have the full picture. We can combat this by reading in the bounding boxes of the whole dataset. We join three dataframe into one bounding box dataframe. Again, this will have all the bounding boxes of all the images within the original dataset. When reading in the three CSV files, a schema was not created so the data read in is not the format we want. This will be corrected later on.
Repeating the process of removing unwanted field by joining a smaller dataframe, image ids, with our current large dataframe, bounding box. This join removes any image id that is not within the dataframe image id. The bounding box dataframe now has only the images we are interested in.
Still we are unable to read the data from the bounding box. To fix this problem, we join the bounding box dataframe with the label dataframe. This is joining the label code to the label name to make it human-readable dataframe.

Extract image chips:

All the CSV files have been read and formatted to our desire and turned into a dataframe. Now we want to be able to extract an image chip, not just an image itself. The image chip will have information such as the label name, id, bounding box size, confidence interval within each image. To expand our bounding box dataframe to have all the information it needs, we join the images parquet dataframe. This will add the raw ImageID of an image from the parquet data to the bounding box dataframe. To be able to extract each image chip, we define a UDF to extract the portion of the image defined within the bounding box.

UDF extract image chip uses the PIL library to read in one image at a time. When reading in the image it also extracts the image size. Then it starts converting the bounding box x and y directions to float number. It calculates the size of the picture based on the bounding boxes. The bounding box coordinates are given as fractions of the number of pixels in the image. It takes the image and crops it to the size of the bounding box it just calculated. The image is saved to a byte-buffer by getting the raw bytes of the image before returning it and repeating the same process on the next image.

Parquet File Write

Now that all the data from the original Google Open Image dataset has been extracted and clean up to our desired format. We can write the dataframe to a parquet file. This is done by defining another UDF that will write each image chip file to the HDFS and evaluate it on the dataframe. The UDF will write the image chips from every spark node in parallel across the cluster. This also avoids pulling every image chip back to the notebook later when needed.

Outside of the UDF function you write the file by using the dataframe you want to write.praquet file to the path. Warning: this might cause you to run out of memory on your cluster. Make sure to increase capacity and if need be you can use write.mode(‘overwrite’).praquet to write over whatever was not saved.

Loading 2 Pre-built Model Code Breakdown:

In a separate notebook run a Keras model in parallel with Spark on all the image chips that were extracted. Since this is a new notebook, Spark must be configured and launched with the model file uploaded.

The process is the same for both models when it comes to reading the data.
Start out by reading in the data. This is the data from the parquet file that was created in the pre-processing section. Once the data has been read in and saved to a model, verify that the amount of data that is within the parquet file matches the parquet file made in the preprocess section.

MobileNet Model
Run Keras on image chip data that was created in the preprocess section. Similar to the UDF extract image chip, we evaluate image chip instead by running it through MobileNet prediction model.

The UDF function: import all of the libraries the Keras MobileNet production model would need. Then it takes in one image chip file which is being sent from the dataframe that read in parquet file. It first loads the MobileNet model data and weights. Then it loads in each image chip with the target size for the MobileNet model to read. Once the image has been loaded, it will then prepare the image by turning it into an array, reshaping it and Keras preprocess input function. This is all done so that MobileNet model can read each file that is being evaluated. Once the images can be read by the model then we predict what that image is and attach a confidence score to it. This information is stored in to a Python dictionary; the first index is a string (name) and the second index is a floating number of the confidence score. The UDF will return a MapType which again has the prediction name and the confidence score.

Once the evaluation is complete, we can extract the predication label and confidence score by creating another UDF to wrap the Keras evaluation that is defined as a MapType. MapType is similar to a Python dictionary, but Spark sticks to a strong data type, this allows for setting hard datatype.
Predication:

Finally, viewing the results of the prediction by grouping by the predicted label and sorting them by the count.

This will output Image label and the confidence score of the image type. MobileNet prediction are not as accurate as ResNet prediction would have been because of the model were built. Though we have high confidence score with the MobileNet prediction, another model would have been better.

ResNet Model

We were unsuccessful in compiling the model on our dataset. However, reading in the data and running it thought an image chip evaluate is the same. Instead of running the images on MobileNet prediction model the images would have been compiled with ResNet50 predication model.

Challenges and Errors faced:

The main issue we faced at the start of the project was a memory issue. Whenever we wrote to the parquet file, it would exceed the physical memory of 20GB. This was later fixed by increasing capacity to 50GB

```
Lost task 1278.1 in stage 6.0 (TID 3311, worker1.hdp-internal, executor 20): ExecutorLostFailure (executor 20 exited caused by one of the runni
ng tasks) Reason: Container killed by YARN for exceeding memory limits.  35.2 GB of 32 GB physical memory used. Consider boosting spark.yarn.ex
ecutor.memoryOverhead or disabling yarn.nodemanager.vmem-check-enabled because of YARN-4714.
```

the second major issue we faced is running the ResNet Model on our dataset. We continued to have issues with the file format that was being read in by the UDF. Because of this we were not able to run any prediction that would have been better than the MobileNet prediction on our dataset.

Reference

https://towardsdatascience.com/the-w3h-of-alexnet-vggnet-resnet-and-inception-7baaaecccc96

https://medium.com/@nina95dan/simple-image-classification-with-resnet-50-334366e7311a

All notebooks were adopted from Professor James Klucar