

A Developer Diary

{about:"code learn and share"}

[Home](#)[Data Science](#)[Java](#)[JavaScript](#)[jBPM](#)[Tools](#)[Tips](#)[About](#)

September 7, 2019 By [Abhisek Jana](#) — [Leave a Comment \(Edit\)](#)

Imagenet PreProcessing using TFRecord and Tensorflow 2.0 Data API

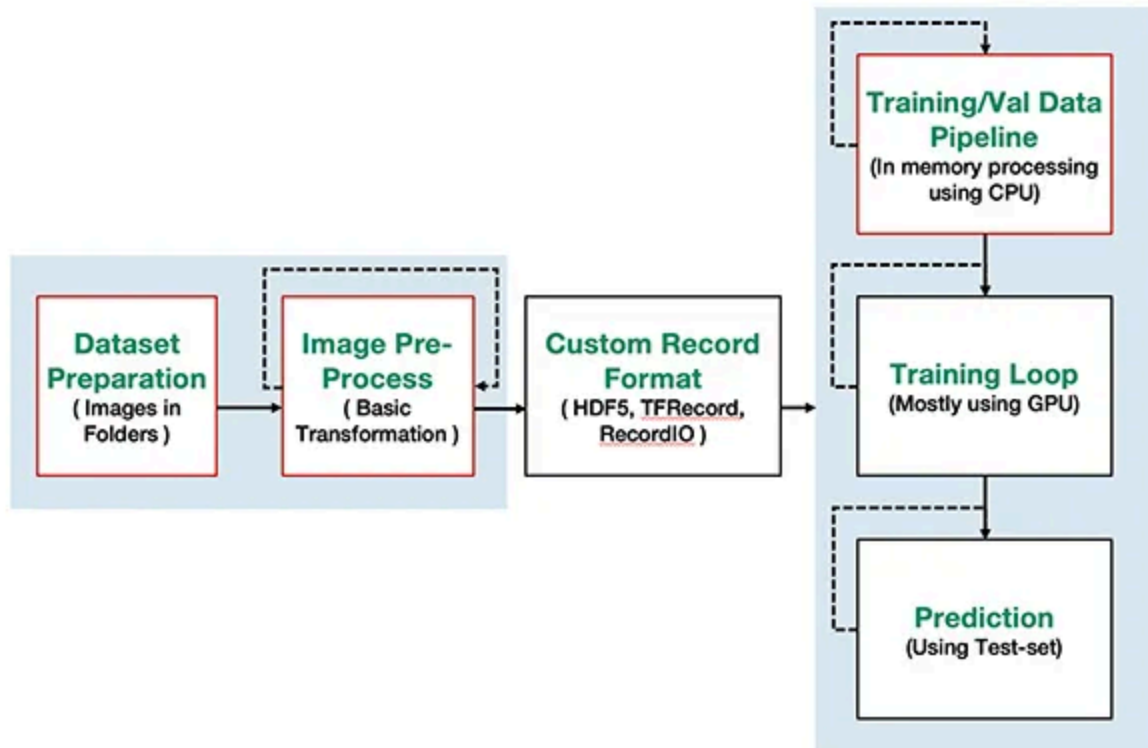


Image PreProcessing is the first step of any Computer Vision application. Although beginners tends to neglect this step, since most of the time while learning, we take a small dataset which has only couple of thousand data to fit in memory. However in real life that's not the case and learning to have an efficient pipeline for Image PreProcessing can be really helpful when working on a tight deadline. In this Imagenet PreProcessing using TFRecord and Tensorflow 2.0 we will learn not only about how to effectively use TFRecord and new TensorFlow 2.0 Data API features, we will also learn how to use available computational resources fully.

Scope:

Let's assume that we want to replicate the **AlexNet** using 2015 Imagenet data. Now, Imagenet is around **166GB**, hence its probably not a good idea to plan to store the entire dataset in computer memory, hence we must look out for building an efficient pipeline.

Now, look at the process diagram of typical Convolutional Neural Network application. This is at a very high-level and focuses on the data preparation part and not the real-time active/online learning and prediction.



In the above diagram the data-preparation steps are highlighted in red. In this tutorial we will mainly focus on the Image Pre-Processing step.

In case you want to understand how to prepare the Imagenet data please refer the following tutorial to know more on the Data Preparation step.



How to prepare Imagenet dataset for Image Classification

Imagenet is one of the most widely used large scale dataset for benchmarking Image Classification algorithms. In case you are starting with Deep Learning and want to test your model against the imagine dataset or just trying out to implement existing publications, you can download the dataset from the imagine website. The downloaded dataset is ... [Continue reading](#)

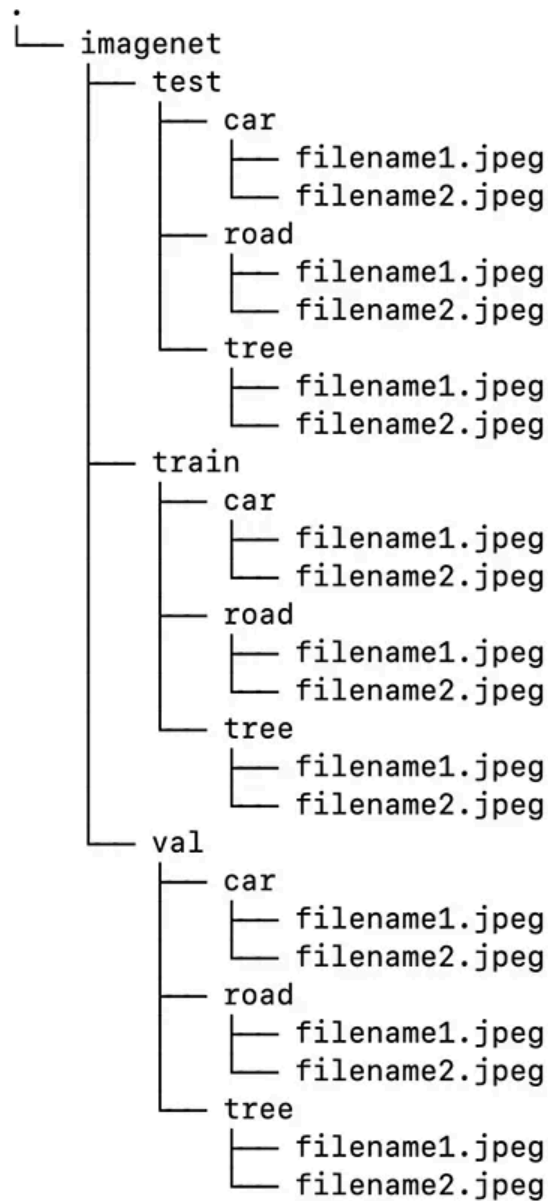


A Developer Diary



Assumptions:

- The images (train/val/test) are organized within their respective class label. In the Dataset Preparation step we will organize them accordingly. Here is a sample structure of how it should look like.



- There is a JSON file with the class label as number. Here is an example of that. This also should have been generated from the Dataset Preparation step. (Later during Training we will be using sparse categorical cross entropy loss function rather than categorical cross entropy)

```

{
  "Persian_cat": 0,
  "barracouta": 1,
  "pug": 2,
  "whiskey_jug": 3,
  "pot": 4,
  "cassette": 5,

```

```
"solar_dish": 6,  
"tub": 7,  
"gorilla": 8,  
"microphone": 9,  
"cabbage_butterfly": 10,  
....  
....  
}
```

- We can also directly have one hot encoding in this file rather than using number encoding. Here just for simplicity I am using number encoding for the classes.

```
{  
    "car": [1,0,0],  
    "road": [0,1,0],  
    "tree": [0,0,1]  
}
```

Objectives:

It's important to define what we want to do in the beginning.

Functional Objectives:

As per the AlexNet paper, we will perform following operations:

1. Mean RGB Calculation
2. Image Preprocessing:
 - Image Resize
 - Create TFRecord and store them in filesystem

Technical Objectives:

We will also try to use as much as computation power we may have in the system we are using, which should also lead to faster processing times. As you see the first picture, this step could be executed more than once, hence having a faster pipeline will help us in long run.

- Use more than one CPU core
- Reduce Processing Timeframe
- The size of the TFRecord files should be same as the original data size.
 - In case you have enough storage you can ignore this. More on this in later section.

Implementations:

1. Mean RGB Calculation:

In the Mean RGB Calculation we will calculate the mean values of R,G & B channels across all the images. You can always use a subsample dataset derived from the main dataset and use that for this task, which theoretically should give you same result as long as the sample dataset is a good representation of the original dataset.

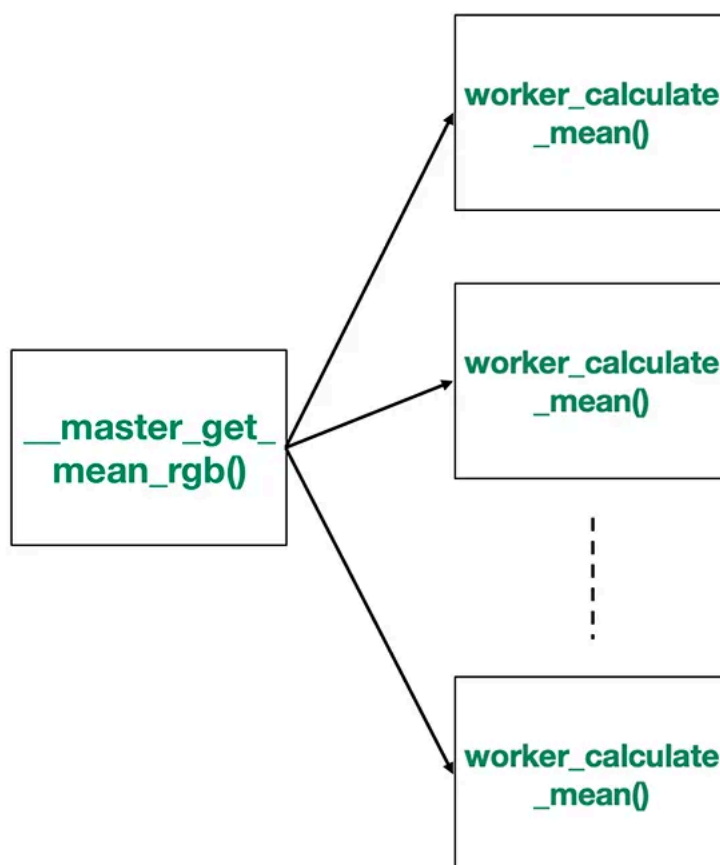
There are two ways we can implement this:

- Use the original images and calculate the mean rgb.
 - This is helpful when the rgb mean normalization is part of the pre-processing. So that when we create the TFRecord, you can use this.
 - This will be bit complex to implement since we need to perform the thread management by ourself.
- Use the generated TFRecord files after pre-processing
 - This is helpful when you want to perform the rgb mean normalization during your training pipeline.
 - This will also be more accurate since the mean of rgb will be calculated on the resized, cropped images.

- Easy to implement since Tensorflow data API will perform most of the work.

Implementing using python multi-threading:

The `__master_get_mean_rgb()` function creates many child threads of `__worker_calculate_mean()` function in order to calculate the mean of a subset of images and return them to the `get_mean_rgb()` function, which will then save the final values in JSON format to the disk.



`__worker_calculate_mean()`:

This is a very straight forward function. We will be using progressbar package to create a nice processing bar in the console. The function's argument is a list of image paths and an identifier as `core_num` (just to show in the progress bar, not required for actual functionality).

We first create 3 empty lists named R,G and B, then loop through the image paths and read them using opencv. Next, we are calling the `cv2.mean()` function to get the mean of each layer. We will take only first 3 values, since the 4th one is for the alpha channel which we don't need.

Then append the values to the R, G, B lists. Remember opencv uses B,G,R format and not R,G,B. Once the loop is completed, we calculate the mean of the batch of images and return that to the calling function.

```
def __worker_calculate_mean(files, core_num):
    (R, G, B) = ([], [], [])

    widgets = [
        'Calculating Mean - [' + str(core_num) + ']',
        progressbar.Bar('#', '[' + ']' ),
        ' [' + progressbar.Percentage(), ']' ,
        '[' + progressbar.Counter(format='%(value)02d/%'
(max_value)d'), ']'

    ]

    bar = progressbar.ProgressBar(maxval=len(files),
widgets=widgets)
    bar.start()

    for i, file in enumerate(files):
        image = cv2.imread(file)
        (b, g, r) = cv2.mean(image)[:3]
        R.append(r)
        G.append(g)
        B.append(b)
        bar.update(i + 1)
```

```
bar.finish()  
return np.mean(R), np.mean(G), np.mean(B)
```

__master_get_mean_rgb():

We will be using multiprocessing package to create and manage threads. First we get the core count of the processor by calling

`multiprocessing.cpu_count()` and then split the list of images in batches based on the core count.

So if we have 1M images and 10 cores in CPU, we will have 100K images per CPU core to process.

Use the Pool class to create a pool of threads, then invoke `starmap()` function. The first argument of this function is the `__worker_calculate_mean()` method and the 2nd argument is the parameters need to be passed.

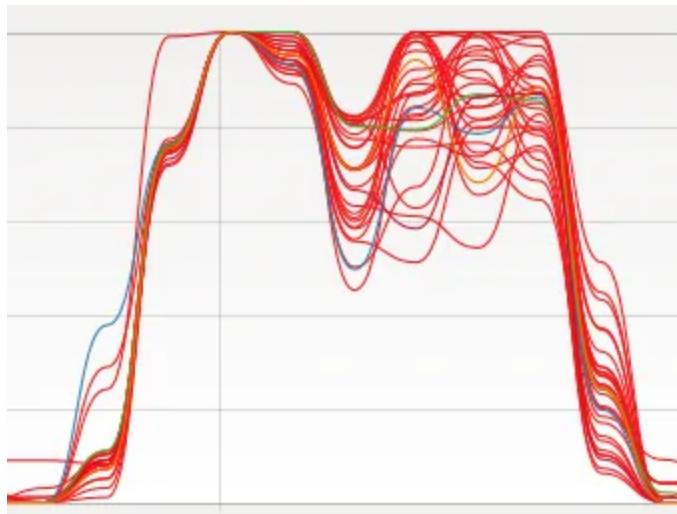
After all threads have completed, loop through the results and append the batch means to the local R,G and B list. Later return the mean values.

```
def __master_get_mean_rgb(files):  
    (R, G, B) = ([], [], [])  
  
    cpu_core = multiprocessing.cpu_count()  
    item_per_thread = int(len(files) / cpu_core) + 1  
  
    split_file_list = [files[x:x + item_per_thread] for x  
in range(0, len(files), item_per_thread)]  
    p = multiprocessing.Pool(cpu_core)  
    results = p.starmap(__worker_calculate_mean,  
zip(split_file_list, list(range(len(split_file_list)))))  
    p.close()  
    p.join()
```

```
for val in results:  
    R.append(val[0])  
    G.append(val[1])  
    B.append(val[2])  
  
return np.mean(R), np.mean(G), np.mean(B)
```

Using this approach, we should be able to speedup the computations. I have Threadripper 1950x with 32 Threads which significantly improved the performance by completing in just 5mins for using all 1.2M images. I was reading data from WD Black 4TB HDD, if you use nvme drive or regular SSD the performance will be even better.

As you see in the below picture, all the cpu cores in my machine were utilized.



These RGB values need to be stored in a JSON file in the disk for later use.

```
def get_mean_rgb(image_dir, output_file):  
    files = glob.glob(image_dir)  
    R, G, B = __master_get_mean_rgb(files)  
  
    with open(output_file, "w+") as f:  
        f.write(json.dumps({"R": R, "G": G, "B": B}))
```

You can execute the script using following command:

```
python mean_rgb_calc.py -i  
"/media/4TB/datasets/ILSVRC2015/ILSVRC2015/Data/CLS-  
LOC/test/**/*.*.JPEG" -o "imagenet_test_mean_rgb.json"
```

You can see the progress in the console:

```
Calculating Mean - [5]  
[#####]  
[100%] [1313/1313]  
Calculating Mean - [13] [#####] [100%]  
[1313/1313]  
Calculating Mean - [28] [#####] [100%]  
[1313/1313]  
Calculating Mean - [15] [#####] [100%]  
[1313/1313]  
Calculating Mean - [23] [#####] [100%]  
[1313/1313]  
Calculating Mean - [0] [#####] [100%]  
[1313/1313]  
Calculating Mean - [21] [#####] [100%]  
[1313/1313]  
Calculating Mean - [19] [#####] [100%]  
[1313/1313]  
Calculating Mean - [26] [#####] [100%]  
[1313/1313]  
Calculating Mean - [1] [#####] [100%]  
[1313/1313]  
Calculating Mean - [8] [#####] [100%]  
[1313/1313]  
Calculating Mean - [7] [#####] [100%]  
[1313/1313]
```

Here is the output JSON file:

```
{  
    "R": 122.58534800031481,  
    "G": 116.7101693473191,  
    "B": 104.37388196859331  
}
```

Implementing using TFRecord:

If you have completed Step 2 (image PreProcessing) and saved the data using TFRecord then those files can be used for RGB Mean calculation as well.

This code will be simple since TensorFlow's Data API will take care of creating multiple threads for efficiency.

I strongly encourage to skip this section and come back after reading through the TFRecord creation in case you are new to TFRecord.

__master_get_mean_rgb_from_tfrecord():

First lets read the tfrecords files using tensorflow's Data API, then call the `parse_image()` function to parse each TFRecord to image tensor and label. We will set `repeat` to `1` so that every record should be accessed only once. You can set any batch size, I am setting it as `1024`.

In order to make sure everything happens using parallel calls, set `tf.data.experimental.AUTOTUNE` to the `num_parallel_calls` in the `map` function and `buffer_size` in the `prefetch` function. Based on the available hardware tensorflow will automatically set the number of parallel threads.

Next, loop through the dataset by calling `take()` function. Set `count` to `-1` in order to retrieve all the data in loop.

Inside the for loop:

- Change the data type to `float64` using `tf.cast()` function.
- Calculate the mean using `tf.reduce_mean()`.
 - Set the `axis` to `(0,1,2)` since the shape of the `image` will be `(1024, 256, 256, 3)` and we want to calculate the mean for each channel. The final output will be a vector with 3 values `[R,G,B]`
- Set the mean for each batch in the `rgb_mean_arr` array.

Finally, convert the python list to jumpy array and call `np.mean` with `axis` set to `0` to calculate the final rgb mean.

```
def __master_get_mean_rgb_from_tfrecord(files):
    def parse_image(record):
        features = {
            'label': tf.io.FixedLenFeature([], tf.int64),
            'image_raw': tf.io.FixedLenFeature([],
tf.string)
        }
        parsed_record =
tf.io.parse_single_example(record, features)
        image =
tf.io.decode_jpeg(parsed_record['image_raw'], channels=3)
        label = tf.cast(parsed_record['label'], tf.int32)
        return image, label

    record_files = tf.data.Dataset.list_files(files)

    dataset =
tf.data.TFRecordDataset(filenamees=record_files,
compression_type="GZIP")

    dataset = dataset.map(parse_image,
```

```

num_parallel_calls=tf.data.experimental.AUTOTUNE) \
    .repeat(1) \
    .batch(1024) \

.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)

rgb_mean_arr = []

for i, (image, label) in
enumerate(dataset.take(count=-1)):
    rgb_mean_arr.append(tf.reduce_mean(tf.cast(image,
tf.float64), axis=(0, 1, 2)))

return np.mean(np.array(rgb_mean_arr), axis=0)

```

The `get_mean_rgb()` function now needs to be updated to accommodate both the options.

```

def get_mean_rgb(image_dir, output_file,
useTFRecord=False):
    files = glob.glob(image_dir)

    if useTFRecord:
        rgb_mean =
__master_get_mean_rgb_from_tfrecord(files)
        R, G, B = rgb_mean[0], rgb_mean[1], rgb_mean[2]
    else:
        R, G, B = __master_get_mean_rgb(files)

    with open(output_file, "w+") as f:
        f.write(json.dumps({"R": R, "G": G, "B": B}))

```

Let's create the main function.

```
if __name__ == '__main__':  
    ap = argparse.ArgumentParser()  
    ap.add_argument("-i", "--image_dir", required=True,  
help="path to the input image dir. e.g.  
/home/Dataset/train/**/*.jpg", )  
    ap.add_argument("-o", "--output_file_name",  
required=True, help="path to the json output")  
    ap.add_argument("-tf", "--use_tfrecord",  
required=False, type=bool, default=False, help="path to  
the json output")  
    args = vars(ap.parse_args())  
  
    get_mean_rgb(args["image_dir"],  
args["output_file_name"], args["use_tfrecord"])
```

You can execute the script using following command:

```
python mean_rgb_calc.py -i  
"/media/4TB/datasets/ILSVRC2015/ILSVRC2015/tf_records/test/*.tfrecord"  
-o "imagenet_test_mean_rgb_tf.json" -tf True
```

The output JSON will be:

```
{  
    "R": 122.10927936917298,  
    "G": 116.5416959998387,  
    "B": 102.61744377213829  
}
```

Notice, there is a difference between both the approaches. The 2nd one is more appropriate though.

Please find the full code in github.

2. Image Preprocessing:

In this Image Preprocessing section, we will first resize the images and crop them as per AlexNet paper and then will store them in TFRecord format for faster processing at training time.

Image Resize:

We will be using opencv for the image processing tasks.

`scale_image()`:

The `scale_image()` method will take raw image as vector input `[h,w,3]` and will upscale/downscale the shortest side to 256 pixel. This is a very straight forward code.

```
def scale_image(image, size):  
    image_height, image_width = image.shape[:2]  
  
    if image_height <= image_width:  
        ratio = image_width / image_height  
        h = size  
        w = int(ratio * 256)  
  
        image = cv2.resize(image, (w, h))  
  
    else:  
        ratio = image_height / image_width  
        w = size  
        h = int(ratio * 256)
```

```
image = cv2.resize(image, (w, h))
```

```
return image
```

center_crop():

This function will crop the center part of the resized image. We initially need to make sure the longer side is not 257pixel since that will make incorrect sizes.

In case the width & height are not equal to 256px we will just resize them to 256px X 256px at the end. This will happen when the longer side has 257px.

```
def center_crop(image, size):
    image_height, image_width = image.shape[:2]

    if image_height <= image_width and abs(image_width -
size) > 1:

        dx = int((image_width - size) / 2)
        image = image[:, dx:-dx, :]
    elif abs(image_height - size) > 1:
        dy = int((image_height - size) / 2)
        image = image[dy:-dy, :, :]

    image_height, image_width = image.shape[:2]
    if image_height is not size or image_width is not
size:
        image = cv2.resize(image, (size, size))

    return image
```

process_image():

The process image just invokes the above functions and returns the values.

```
def process_image(image, size):  
    image = scale_image(image, size)  
    image = center_crop(image, size)  
    return image
```

TFRecord Creation:

Once the image pre-processing has been completed, we can now store them in TFRecord format.

Note – We have not done the RGB Mean normalization here, we will perform that during training.(read more on that later)

worker_tf_write():

This function will take a list of image paths and store all them in **TFRecord** format. TFRecord supports **GZIP** as compression format so that we get around 4% storage benefit, which can be passed as an argument.

We will loop through each image files and call **process_image()** function by passing the image vector. This function will return the resized image.

We can store the image vector directly in TFRecord however the size will be 5-7 times more. Hence we will compress the image vector to **jpg** file and store that as **base64** (Binary data in String format).

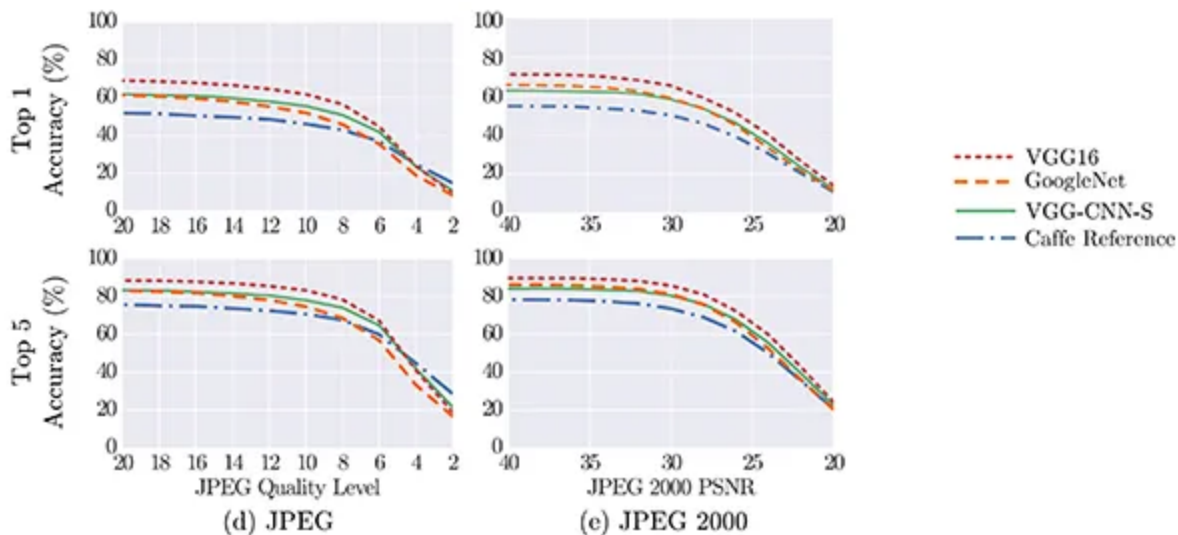
The **cv2.imencode()** function will encode the image vector to jpg file. We can define the quality of the jpg compression using the **encode_param**. There will be small decode compute cost need to be paid during training, however that can be alleviated by processing in CPU instead of the GPU (The next tutorial will have more details on this).

Also, we are not normalizing the images using Mean RGB since when we convert it to compressed JPG, we will loose all that information, so we will perform the mean RGB normalization during training. However if you are saving

the images as vector and not compressed jpg encoded image, you can actually perform both **RGB Mean normalization** and **float32** conversion before storing the image vector in TFRecord.

With **70%** JPG Compression Quality our training output size was **~13GB** and with **90%** JPG Compression Quality it was **~25GB**. We are seeing a significant reduction in size as we are resizing the images to 256×256 from the original **~138GB** training data.

In case you are interested to find out how the model performance degrade over higher JPG Compression Quality, here is the reference chart from a recent study. Compression quality of 70%-80% wont effect accuracy of the model much. Again, not all the imagenet images are having 100 quality. I found many images with 70%-80% quality.



If you want to know more on this, please refer this paper:

<https://arxiv.org/abs/1604.04004>

If the encoding is successful, then convert it to bytes using **tobytes()** function and store it.

The label name will be captured by parsing the image path and retrieve the label number using the **label_map**. I will be storing the labels as

`NumberEncoding`, however you can convert them to `OneHotEncoding` and store them.

Using the `tf.train.Example()` convert the label and raw image into `TFRecord` data.

Then using `tf_writer.write()` function, write the record to the file.

```
def _int64_feature(value):
    return
    tf.train.Feature(int64_list=tf.train.Int64List(value=
[value]))

def _bytes_feature(value):
    return
    tf.train.Feature(bytes_list=tf.train.BytesList(value=
[value]))

def worker_tf_write(files, tf_record_path, label_map,
size, image_quality, tf_record_options, number):
    encode_param = [int(cv2.IMWRITE_JPEG_QUALITY),
image_quality]
    tf_record_options =
tf.io.TFRecordOptions(compression_type=tf_record_options)

    with tf.io.TFRecordWriter(tf_record_path,
tf_record_options) as tf_writer:
        for i, file in enumerate(files):
            image = process_image(cv2.imread(file), size)
            is_success, im_buf_arr = cv2.imencode(".jpg",
image, encode_param)
```

```

if is_success:
    label_str = file.split("/")[-2]

    label_number = label_map[label_str]

    image_raw = im_buf_arr.tobytes()
    row =
tf.train.Example(features=tf.train.Features(feature={
    'label':
_int64_feature(label_number),
    'image_raw':
_bytes_feature(image_raw)
}))

    tf_writer.write(row.SerializeToString())
else:
    print("Error processing " + file)

```

Note: In the GitHub code you can see reference of progress bar, however for simplicity purpose I have removed it from above code sample.

master_tf_write():

We will invoke the `worker_tf_write()` using multiple threads from `master_tf_write()` function. The code is simple to read through.

```

def master_tf_write(split_file_list, tf_record_paths,
size, image_quality, label_map, tf_record_options):
    cpu_core = multiprocessing.cpu_count()

    p = multiprocessing.Pool(cpu_core)
    results = p.starmap(worker_tf_write,
                        zip(split_file_list,
tf_record_paths, repeat(label_map), repeat(size),

```

```
repeat(image_quality), repeat(tf_record_options),

list(range(len(tf_record_paths))))))
    p.close()
    p.join()
```

create_tf_record():

The `create_tf_record()` function has 8 different arguments. Get the list of files first using the `glob.glob()` function, then shuffle the list. Afterwards, find how many tfrecord files need to be created based on the `split_number`. In a loop create the list of tfrecord files and then invoke `master_tf_write()` function by passing all the required parameters.

```
def create_tf_record(image_folder, record_path,
                    identifier, label_map, size=256, split_number=1000,
                    image_quality=90, tf_record_options=None):
    print("creating " + identifier + " records")

    files = glob.glob(image_folder)

    random.shuffle(files)

    split_file_list = [files[x:x + split_number] for x in
                      range(0, len(files), split_number)]

    tf_record_paths = []

    for i in range(len(split_file_list)):
        tf_record_paths.append(record_path + identifier +
                              "-" + str(i) + ".tfrecord")

    master_tf_write(split_file_list, tf_record_paths,
                    size, image_quality, label_map, tf_record_options)
```

JSON Config:

In case we need to create the tfrecords for train, test and validation images at once, we can define a JSON Config file to have all the necessary configurations.

```
{
  "label_map": "label_map_100.json",
  "split_number": 6000,
  "image_folder":
"/media/4TB/datasets/ILSVRC2015/ILSVRC2015/Data/CLS-
LOC_100/",
  "record_path":
"/media/4TB/datasets/ILSVRC2015/ILSVRC2015/tf_records_159/",
  "image_type": "JPEG",
  "crop_size": 256,
  "image_quality": 90,
  "tf_record_compression": "GZIP",
  "batch": [
    "val",
    "test",
    "train"
  ]
}
```

__name__:

In the `main` method we can first parse the json and call `create_tf_record()` inside the for loop for each split type.

```
if __name__ == '__main__':
    ap = argparse.ArgumentParser()
    ap.add_argument("-c", "--config", required=True,
help="path to the config JSON.", )
    args = vars(ap.parse_args())
```



```

with open(args["config"], "rb") as file:
    config = json.loads(file.read())

with open(config["label_map"], "rb") as file:
    label_map = json.loads(file.read())

for indentifier in config["batch"]:
    image_folder = config["image_folder"] +
indentifier + "/*/*." + config["image_type"]
    record_path = config["record_path"] + indentifier
+ "/"

    if not os.path.isdir(record_path):
        os.makedirs(record_path)

        create_tf_record(image_folder, record_path,
indentifier, label_map, config["crop_size"],
config["split_number"], config["image_quality"],
                        config["tf_record_compression"])

```

You can execute the script using following command:

```
python imagenet_preprocessing.py -c pre_process_config.json
```

The output will look like following:

```

creating val records
Processing Images - [1] [#####]
[100%] [1686/1686]
Processing Images - [0] [#####]
[100%] [6000/6000]
creating test records
Processing Images - [1] [#####]

```

```
[100%] [2004/2004]
Processing Images - [0] [#####]
[100%] [6000/6000]
```

You can find the source code in github.

[GitHub Code](#)

Conclusion:

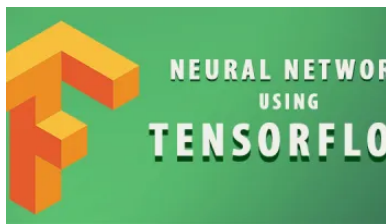
This preprocessing steps can be extended for any data sets. Next we will learn how to use the tfrecord to fetch the data at training time.

Related



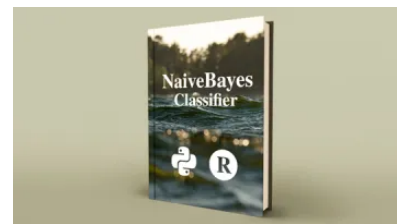
How to prepare Imagenet dataset for Image Classification

In "Computer Vision"



Implement Neural Network using TensorFlow

In "Data Science"



Introduction to Naive Bayes Classifier using R and Python

In "Data Science"

Filed Under: [Computer Vision](#), [Deep Learning](#), [PreProcessing](#), [Python](#), [TFRecord](#)

Tagged With: [AlexNet](#), [Computer Vision](#), [imagenet](#),

Subscribe to stay in loop


* indicates required

Email Address *

Leave a Reply

Logged in as Abhisek Jana. [Edit your profile](#). [Log out?](#) Required fields are marked *

Comment *



This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

Copyright © 2024 A Developer Diary

