Object Localization and Object Detection

Topics Covered

1. Introduction to SSD
2. Working of SSD
3. Implementation of SSD
4. Tensorflow 2.0 Object Detection API
5. Object Detection using TF 2.0 OD API
6. Summary

Introduction to SSD

The paper about SSD: Single Shot MultiBox Detector (by C. Szegedy et al.) was released at the end of November 2016 and reached new records in terms of performance and precision for object detection tasks, scoring over 74% mAP (*mean Average Precision*) at 59 frames per second on standard datasets such as PascalVOC and COCO. To better understand SSD, let's start by explaining where the name of this architecture comes from:

- **Single Shot:** this means that the tasks of object localization and classification are done in *a* single forward pass of the network
- **MultiBox:** this is the name of a technique for bounding box regression developed by Szegedy et al.
- **Detector:** The network is an object detector that also classifies those detected objects

SSD has two components: a **backbone** model and **SSD head**.

Backbone model usually is a pre-trained image classification network as a feature extractor. This is typically a network like ResNet trained on ImageNet from which the final fully connected classification layer has been removed.

We are thus left with a deep neural network that is able to extract semantic meaning from the input image while preserving the spatial structure of the image albeit at a lower resolution. For ResNet34, the backbone results in a 256 7x7 feature maps for an input image.

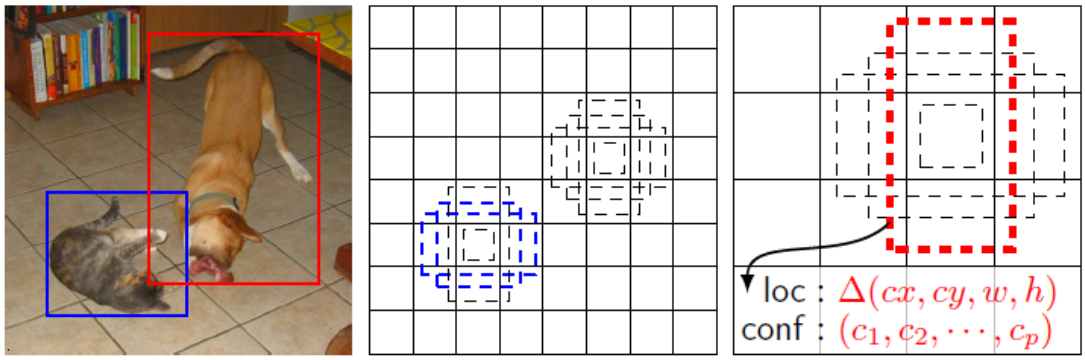
The SSD head is just one or more convolutional layers added to this backbone and the outputs are interpreted as the bounding boxes and classes of objects in the spatial location of the final layers activations.

Working of SSD

The SSD is based on the use of convolutional networks that produce multiple bounding boxes of various fixed sizes and scores the presence of the object class instance in those boxes, followed by a non-maximum suppression step to produce the final detections the SSD can also be implemented using various other backbone models.

The SSD model works as follows, each input image is divided into grids of various sizes and at each grid, the detection is performed for different classes and different aspect ratios. And a score is assigned to each of these grids that says how well an object matches in that particular grid. And non-maximum suppression is applied to get the final detection from the set of overlapping detections. This is the basic idea behind the SSD model.

Here we use different grid sizes to detect objects of different sizes, for example, look at the image given below when we want to detect the cat smaller grids are used but when we want to detect a dog the grid size is increased which makes the SSD more efficient.



(a) Image with GT boxes (b) 8×8 feature map (c) 4×4 feature map

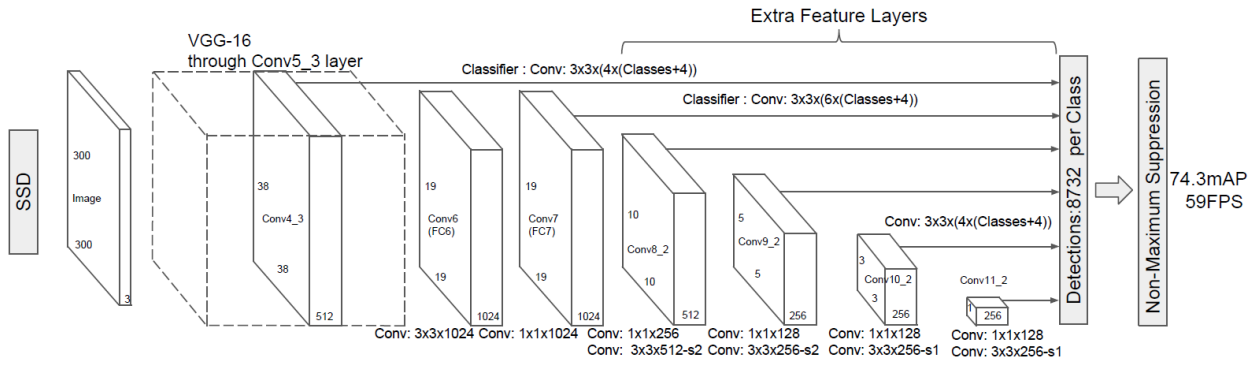
Let's dive into more details on how the objects are detected.

Detections using multi-scale feature maps

The multi-scale feature maps are added to the end of the truncated backbone model. These multi-scale feature maps reduce in size progressively, which allows the detections at various scales of the image. The convolutional layers used here vary for each feature layer.

Detection using the convolutional predictors

The addition of each extra layer produces a fixed number of predictions using the convolutional filters in them. These additional layers are shown at the top of the model in the given diagram below. For example, a feature layer of size m x n with p channels, the minimal prediction parameter that gives a decent detection is a 3 x 3 x p small kernel. Such kernel gives us the score for a category or a shape offset relative to the default box coordinates.



Usage of Default Boxes and aspect ratios

The default bounding boxes are associated with every feature map cell at the top of the network. These default boxes tile the feature map in a convolutional manner such that the position of each box relative to its corresponding cell is fixed.

For each box out of k at a given location, c class scores are computed and 4 offset relatives to the original default box shape. This results in the total of $(c+4)k$ filters that are applied around each location in the feature map, yielding $(c+4)kmn$ outputs for a feature map of size $m \times n$.
This allows the usage of different default box shapes in several feature maps and makes the model efficiently discretize the space of possible output box shapes.

The table given below gives the details about all the operations in the SSD head

TYPE / NAME	GRID SIZE	KERNEL SIZE
Conv 6	19×19	3x3x1024
Conv 7	19×19	1x1x1024
Conv 8_2	10×10	1x1x256 3x3x512-s2
Conv 9_2	5×5	1x1x128 3x3x256-s2
Conv 10_2	3×3	1x1x128 3x3x256-s1
Conv 11_2	1×1	1x1x128 3x3x256-s1

Additional Notes On SSD

The SSD paper makes the following additional observations:

- More default boxes result in more accurate detection, although there is an impact on speed
- having MultiBox on multiple layers results in better detection as well, due to the detector running on features at multiple resolutions.
- 80% of the time is spent on the base VGG-16 network: this means that with a faster and equally accurate network SSD's performance could be even better.
- SSD confuses objects with similar categories (e.g. animals). This is probably because locations are shared for multiple classes.
- SSD-500 (the highest resolution variant using 512x512 input images) achieves best mAP on Pascal VOC2007 at 76.8%, but at the expense of speed, where its frame rate drops to 22 fps. SSD-300 is thus a much better trade-off with 74.3 mAP at 59 fps.
- SSD produces worse performance on smaller objects, as they may not appear across all feature maps. Increasing the input image resolution alleviates this problem but does not completely address it.

Implementation of SSD

Input and Output: The input of SSD is an image of fixed size, for example, 512x512 for SSD512. The fixed size constraint is mainly for efficient training with batched data. Being fully convolutional, the network can run inference on images of different sizes.

The output of SSD is a prediction map. Each location in this map stores classes confidence and bounding box information as if there is indeed an object of interests at every location. Obviously, there will be a lot of false alarms, so a further process is used to select a list of most likely prediction based on simple heuristics.

To train the network, one needs to compare the ground truth (a list of objects) against the prediction map. This is achieved with the help of *priorbox*, which we will cover in details later.

Multi-scale Detection

The resolution of the detection equals the size of its prediction map. Multi-scale detection is achieved by generating prediction maps of different resolutions. For example, SSD512 outputs seven prediction maps of resolutions 64x64, 32x32, 16x16, 8x8, 4x4, 2x2, and 1x1 respectively. You can think there are 5461 "local prediction" behind the scene. The input of each prediction is effectively the receptive field of the output feature.

Priorbox

We know the ground truth for object detection comes in as a list of objects, whereas the output of SSD is a prediction map. We also know in order to compute a training loss, this ground truth list needs to be compared against the predictions. The question is, how?

- There can be multiple objects in the image. In this case which one or ones should be picked as the ground truth for each prediction?
- There can be locations in the image that contains no objects. How to set the ground truth at these locations?

Intuitively, object detection is a local task: what is in the top left corner of an image is usually unrelated to predict an object in the bottom right corner of the image. So one needs to measure how relevance each ground truth is to each prediction, probably based on some distance based metric.

This is where priorbox comes into play. You can think it as the *expected bounding box prediction* – the average shape of objects at a certain scale. We can use priorbox to select the ground truth for each prediction. This is how:

- We put one priorbox at each location in the prediction map.
- We compute the intersect over union (IoU) between the priorbox and the ground truth.
- The ground truth object that has the highest IoU is used as the target for each prediction, given its IoU is higher than a threshold.
- For predictions who have no valid match, the target class is set to the *background* class and they will not be used for calculating the localization loss.

Basically, if there is significant overlapping between a priorbox and a ground truth object, then the ground truth can be used at that location. The class of the ground truth is directly used to compute the classification loss; whereas the offset between the ground truth bounding box and the priorbox is used to compute the location loss.

More on Priorbox

The size of the priorbox decides how "local" the detector is. Smaller priorbox makes the detector behave more locally, because it makes distanced ground truth objects irrelevant. It is good practice to use different sizes for predictions at different scales. For example, SSD512 uses 20.48, 51.2, 133.12, 215.04, 296.96, 378.88 and 460.8 as the sizes of the priorbox at its seven different prediction layers.

In practice, SSD uses a few different types of priorbox, each with a different scale or aspect ratio, in a single layer. Doing so creates different "experts" for detecting objects of different shapes. For example, SSD512 use 4, 6, 6, 6, 6, 4, 4 types of different priorboxes for its seven prediction layers, whereas the aspect ratio of these priorboxes can be chosen from 1:3, 1:2, 1:1, 2:1 or 3:1. Notice, experts in the same layer take the same underlying input (the same receptive field). They behave differently because they use different parameters (convolutional filters) and use different ground truth fetch by different priorboxes.

Hard negative mining

Priorbox uses a simple distance-based heuristic to create ground truth predictions, including backgrounds where no matched object can be found. However, there can be an imbalance between foreground samples and background samples, as background samples are considerably easy to obtain. In consequence, the detector may produce many false negatives due to the lack of a training signal of foreground objects.

To address this problem, SSD uses hard negative mining: all background samples are sorted by their predicted background scores in the ascending order. Only the top K samples are kept for proceeding to the computation of the loss. K is computed on the fly for each batch to keep a 1:3 ratio between foreground samples and background samples.

Data augmentation

SSD use a number of augmentation strategies. A "zoom in" strategy is used to improve the performance on detecting large objects: a random sub-region is selected from the image and scaled to the standard size (for example, 512x512 for SSD512) before being fed to the network for training. This creates extra examples of large objects. Likewise, a "zoom out" strategy is used to improve the performance on detecting small objects: an empty canvas (up to 4 times the size of the original image) is created. The original image is then randomly pasted onto the canvas. After which the canvas is scaled to the standard size before being fed to the network for training. This creates extras examples of small objects and is crucial to SSD's performance on MSCOCO.

Pre-trained Feature Extractor and L2 normalization

Although it is possible to use other pre-trained feature extractors, the original SSD paper reported their results with VGG_16. There is, however, a few modifications on the VGG_16: parameters are subsampled from fc6 and fc7, dilation of 6 is applied on fc6 for a larger receptive field. It is also important to add apply a per-channel L2 normalization to the output of the conv4_3 layer, where the normalization variables are also trainable.

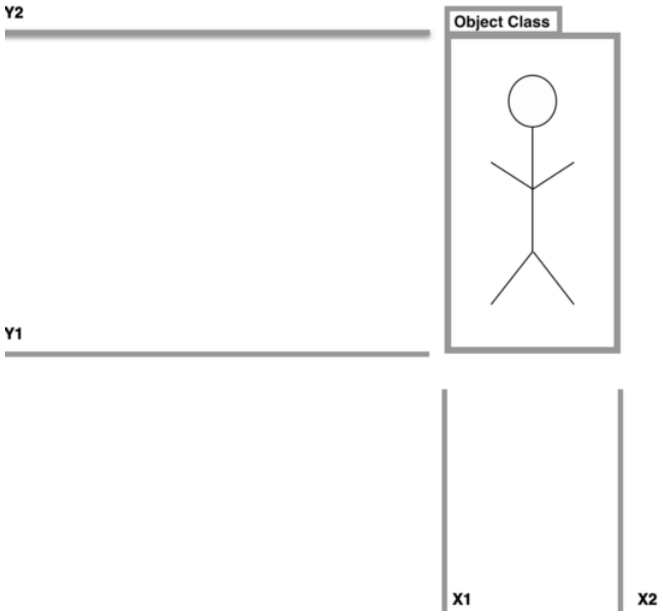
Post-processing

Last but not least, the prediction map cannot be directly used as detection results. For SSD512, there are in fact 64x64x4 + 32x32x6 + 16x16x6 + 8x8x6 + 4x4x6 + 2x2x4 + 1x1x4 = 24564 predictions in a single input image. SSD uses some simple heuristics to filter out most of the predictions: It first discards weak detection with a threshold on confidence score, then performs a per-class non-maximum suppression, and curates results from all classes before selecting the top 200 detections as the final output. To compute mAP, one may use a low threshold on confidence score (like 0.01) to obtain high recall. For a real-world application, one might use a higher threshold (like 0.5) to only retain the very confident detection.

Tensorflow 2.0 Object Detection API

The TensorFlow2 Object Detection API is an extension of the TensorFlow Object Detection API. The TensorFlow2 Object Detection API allows you to train a collection state of the art object detection models under a unified framework, including Google Brain's state of the art model EfficientDet.

More generally, object detection models allow you to train your computer to identify objects in a scene with bounding boxes and class labels. There are many ways you can use deep learning techniques to model this problem and the TensorFlow2 Object Detection API allows you deploy a wide variety of different models and strategies to achieve this goal.



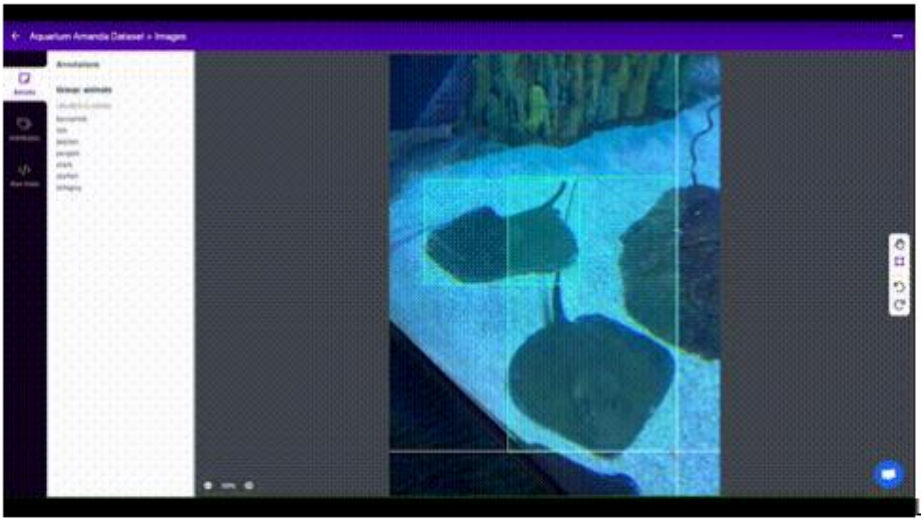
Object Detection using TF 2.0 OD API

we will train the smallest EfficientDet model (EfficientDet-D0) for detecting our custom objects on GPU resources provided by Google Colab. That said, the TensorFlow 2 Object Detection library has many models available in their model zoo.

to get a feel for the new deep learning technologies available in the TensorFlow Object Detection API, you might consider utilizing a public object detection dataset

Open Source Labeling Solutions

If you have unlabeled images, and want to train a detector to detect your custom objects, we recommend trying your hand at free, **open source labeling solutions**.



Labeling objects

you may consider solutions to label your own object detection dataset, such as **LabelImg**, **VoTT**, **SuperAnnotate**, or **LabelMe**.

Regardless of which tool you use, we recommend exporting your annotations at **VOC XML**, which you can later **convert to any format you need**. We find VOC XML is less error prone.

Install TensorFlow 2 Object Detection Dependencies

Once you have a **labeled dataset**, you are ready to dive into the training procedures. Google Colab also provides free GPU resources for training, so make sure that is switched on by selecting Runtime --> Change Runtime Type --> GPU.

Then, we install `tensorflow_gpu=="2.2.0"` as a **backbone** for our training job.

After that we install the object detection library as a python package.

Next up, we run the TF2 model builder tests to make sure our environment is up and running. If successful, you should see the following outputs at the end of the cell execution printouts.

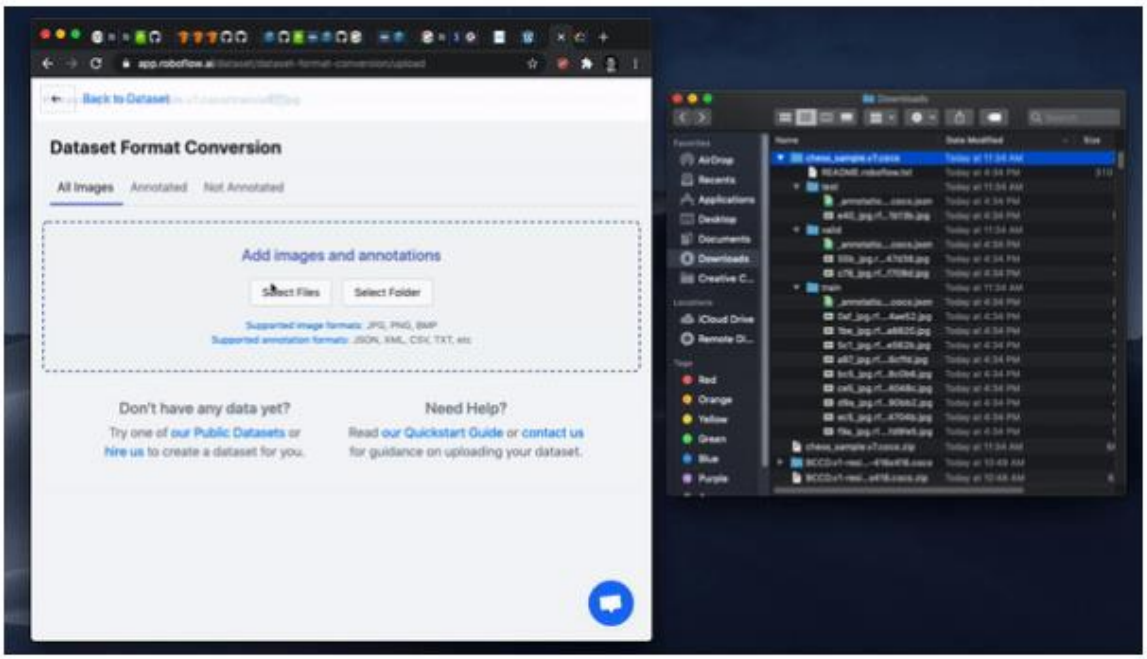
```
[       OK ] ModelBuilderTF2Test.test_create_ssd_models_from_config
[ RUN      ] ModelBuilderTF2Test.test_invalid_faster_rcnn_batchnorm_update
[       OK ] ModelBuilderTF2Test.test_invalid_faster_rcnn_batchnorm_update
[ RUN      ] ModelBuilderTF2Test.test_invalid_first_stage_nms_iou_threshold
[       OK ] ModelBuilderTF2Test.test_invalid_first_stage_nms_iou_threshold
[ RUN      ] ModelBuilderTF2Test.test_invalid_model_config_proto
[       OK ] ModelBuilderTF2Test.test_invalid_model_config_proto
[ RUN      ] ModelBuilderTF2Test.test_invalid_second_stage_batch_size
[       OK ] ModelBuilderTF2Test.test_invalid_second_stage_batch_size
[ RUN      ] ModelBuilderTF2Test.test_session
[  SKIPPED ] ModelBuilderTF2Test.test_session
[ RUN      ] ModelBuilderTF2Test.test_unknown_faster_rcnn_feature_extractor
[       OK ] ModelBuilderTF2Test.test_unknown_faster_rcnn_feature_extractor
[ RUN      ] ModelBuilderTF2Test.test_unknown_meta_architecture
[       OK ] ModelBuilderTF2Test.test_unknown_meta_architecture
[ RUN      ] ModelBuilderTF2Test.test_unknown_ssd_feature_extractor
[       OK ] ModelBuilderTF2Test.test_unknown_ssd_feature_extractor
----------------------------------------------------------------------
Ran 20 tests in 52.705s

OK (skipped=1)
12345678910111213141516171819202l
```

Model tests confirming our TF2 environment setup has been successful

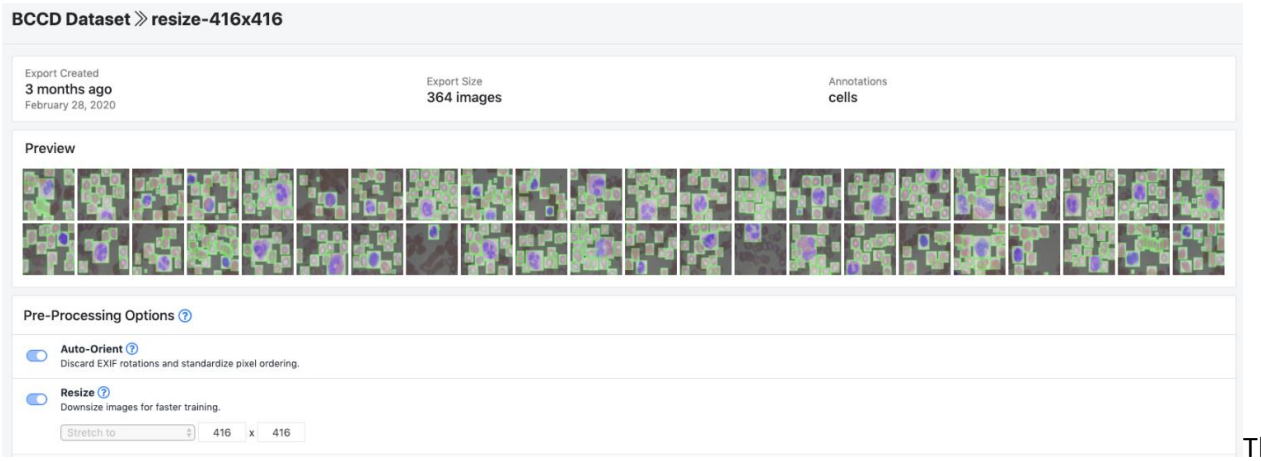
Prepare TensorFlow 2 Object Detection Training Data

Once our programming environment has been properly installed, we need to acquire a version of our dataset in **TFRecord format**.



Drag and drop to upload data to Roboflow in any format

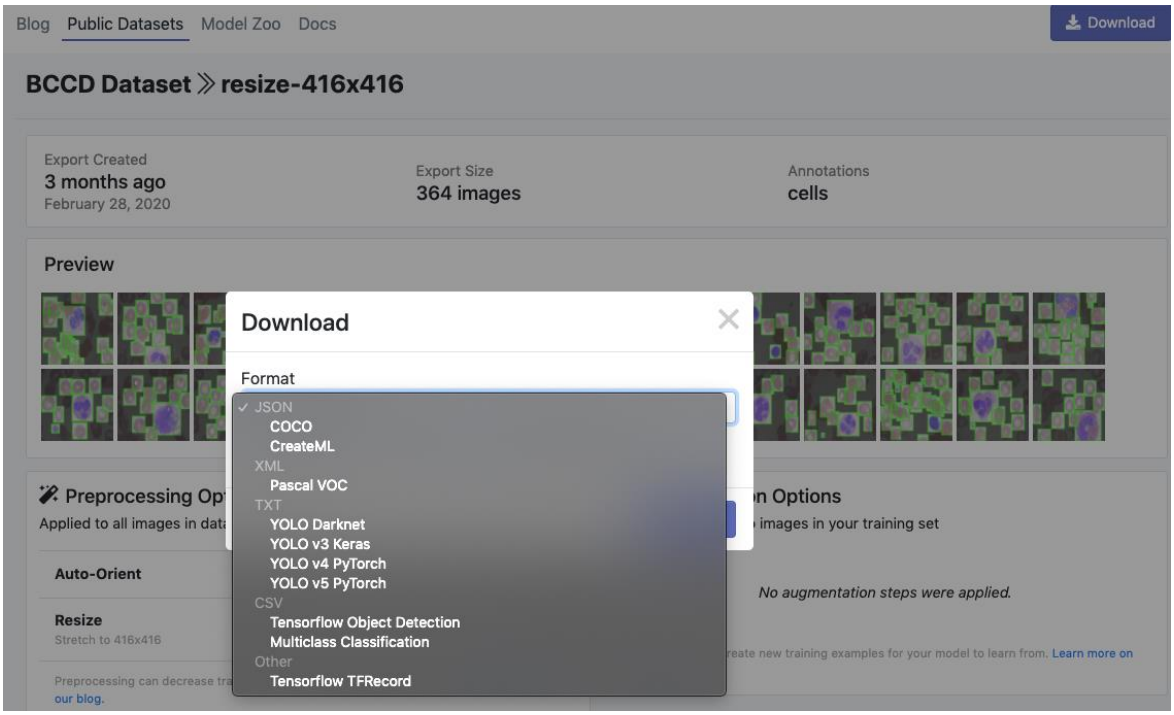
After upload you will be prompted to choose options to version your dataset including **preprocessing** and **augmentations**.



The augmentation and preprocessing selections for our **BCCD dataset**

After selecting these options, click Generate and then Download. You will be prompted to choose a data format for your export. Choose Tensorflow TFRecord format.

Note: in addition to creating TFRecords in Roboflow, you can easily inspect the health of your dataset and its annotations as well as preprocess and augment your data to improve model performance.



Selecting **TensorFlow TFRecord** export format from Roboflow

After export, you will receive a curl link to download your data into our training notebook.

```
#Downloading data from Roboflow
#UPDATE THIS LINK - get our data from Roboflow
%cd /content
!curl -L "[YOUR LINK HERE]" > roboflow.zip; unzip roboflow.zip; rm roboflow.zip
```

```
/content
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   891  100   891    0     0   1242      0 --:--:-- --:--:-- --:--:--  1240
100 4873k  100 4873k    0     0  5087k      0 --:--:-- --:--:-- --:--:-- 5087k
Archive:  roboflow.zip
 extracting: train/cells.tfrecord
 extracting: train/cells_label_map.pbtxt
 extracting: valid/cells.tfrecord
 extracting: valid/cells_label_map.pbtxt
 extracting: test/cells.tfrecord
 extracting: test/cells_label_map.pbtxt
 extracting: README.roboflow.txt
 extracting: README.dataset.txt
```

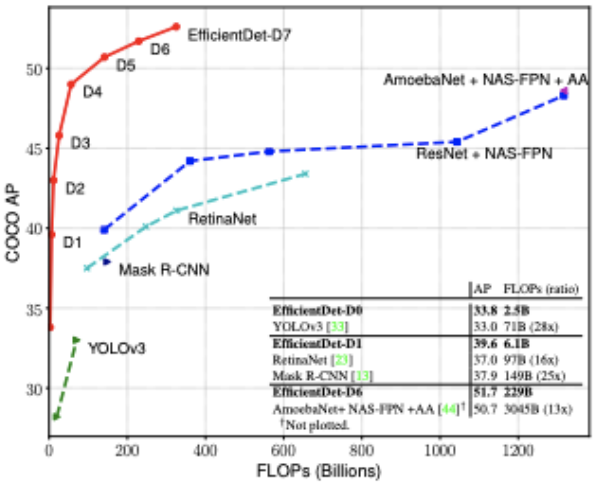
Downloading our training **TFRecords** in the **TF2 Object Detection Colab Notebook**

Lastly, we map our training data files to variables for use in our **computer vision training pipeline configuration**.

Write Custom TensorFlow 2 Object Detection Training Configuration

Next, we write a specialized training configuration file based on our choice of an **object detection model** to instruct the training procedure we plan to run later in the notebook.

By changing the chosen_model variable, you can select among available models for training. We have coded in the first few models in the EfficientDet model series for your exploration. If you want to use even bigger EfficientDet models, you will likely need to increase your compute resources beyond Google Colab!



EfficientDet model family is among state of the art for object detection

You may also consider adding any model you desire in the **TensorFlow 2 Object Detection model zoo**.

Each model has a model_name, a base_pipeline_file, a pretrained_checkpoint, and a batch_size. The base_pipeline_file is a shell of a training configuration specific to each model type, provided by the authors of the TF2 OD repository. The pretrained_checkpoint is the location of a pretrained weights file saved from when the object detection model was pretrained on the COCO dataset. We will start from these weights, and then fine tune into our particular custom dataset task. By using pretraining, our model does not need to start from square one in identifying what features might be useful for object detection.

With all of those inputs defined, we edit the base_pipeline_file to point to our custom data, the pretrained_checkpoint, and we also specify some training parameters. To train for longer, increase the num_steps and to train faster, try increasing the batch_size to a level that your GPU can handle. Remember to decrease the number of steps with the same factor that you increase batch size to keep the training length the same.

Train Custom TensorFlow 2 Object Detector

Now we are ready to train!
We kick of training with the following command:

```
!python /content/models/research/object_detection/model_main_tf2.py \
  --pipeline_config_path={pipeline_file} \
  --model_dir={model_dir} \
  --alsologtostderr \
  --num_train_steps={num_steps} \
  --sample_1_of_n_eval_examples=1 \
  --num_eval_steps={num_eval_steps}
1234567
```

Our training command references our pipeline_file and the model_dir where we would like the model to be saved during training.

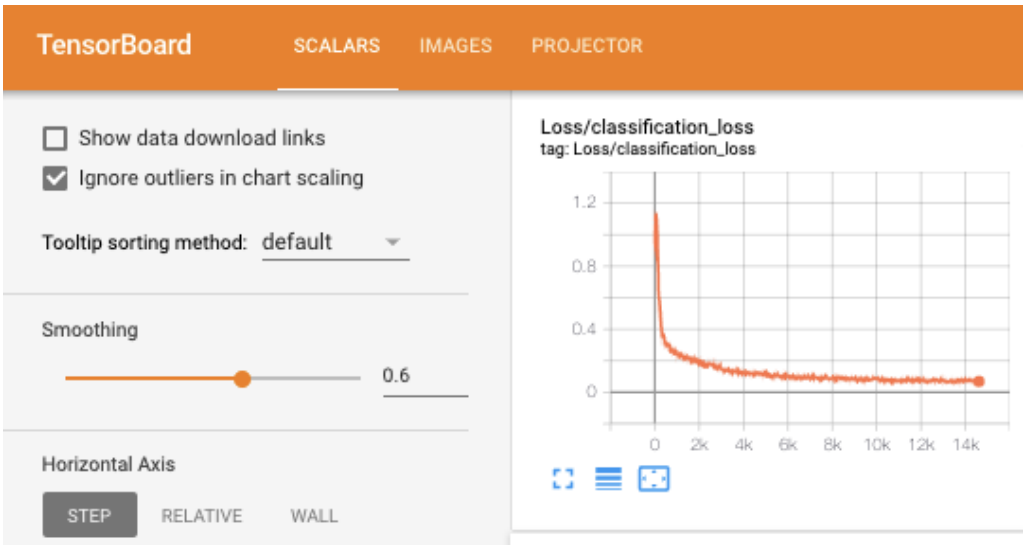
In Colab, even with free GPU, you should expect training to be a lengthy multi-hour process. Note that Colab will stop your kernel's session after a period of inactivity (around 45 minutes), so you may need to keep interacting in your browser tab.

Training Evaluation

At this time of writing, training time evaluation metrics are still under construction for EfficientDet. The code is there so you can try it! If you find out how to execute this part of the notebook This does not affect other architectures, and TensorBoard eval is still valid.

TensorBoard Output

To examine our training success, we output the TensorBoard showing how our models loss function has decreased over the training procedure. The lower the loss, the better!



TensorBoard output to visualize our training procedure

Export Trained TensorFlow 2 Object Detector Weights

Next, we use the exporter_main_v2.py to export our model from TF2 OD library checkpoints to a .pb frozen graph file. The .pb file will be much easier for us to deploy to applications and move to new settings. Stay tuned for more to come on this front.

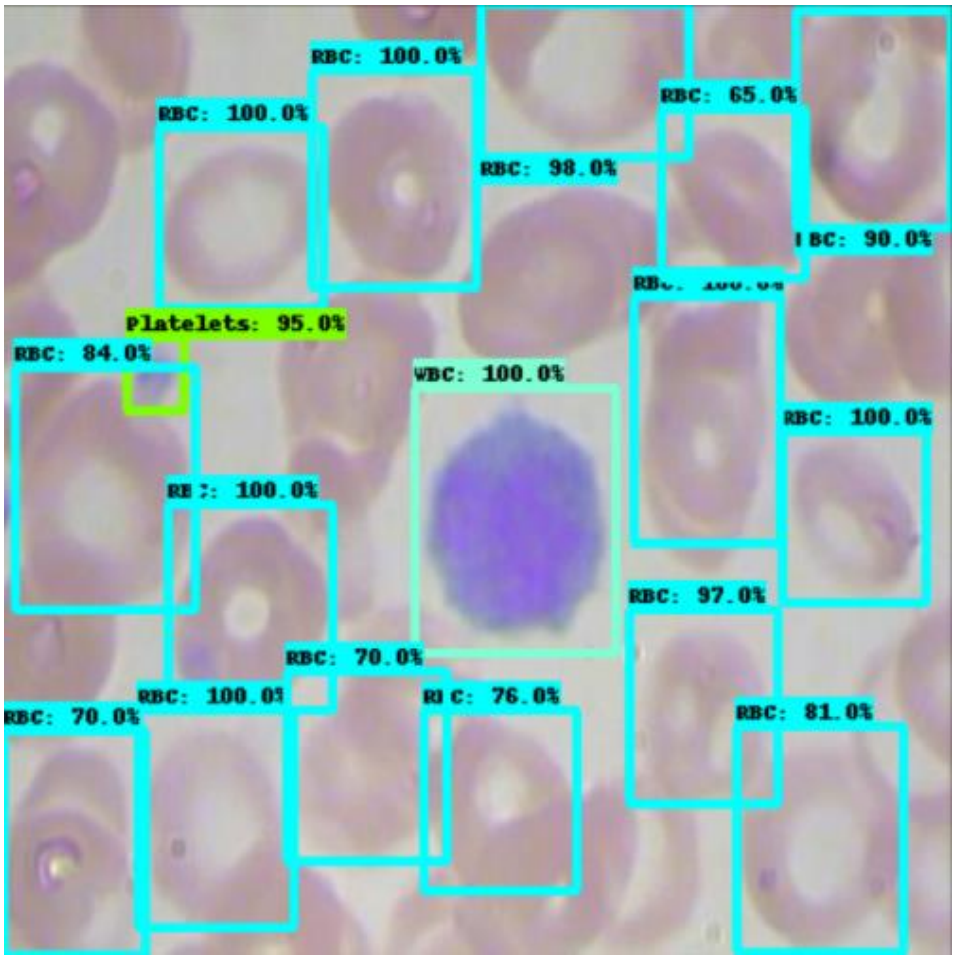
Use Trained TensorFlow 2 Object Detection For Inference on Test Images

Now that we have a trained TensorFlow2 object detector, we can use our model's saved weights to make **test inference on images that the model has never seen**.

TFRecord does not allow you to access the underlying images, so we recommend making another export of your dataset in COCO JSON format to access the underlying test images.

We rebuild our custom object detector from the saved checkpoints. Generally, you want to choose the last checkpoint but you may also want to choose the checkpoint where your object detector did the best on your validation set.

We draw from our test images randomly and send the image through the network for prediction.



Test inference on an image that our model has never seen

Our model does a pretty good job of detecting different types of cells in the blood stream! With the right dataset, you can apply this technology to teach the model to recognize any object in the world.

Summary

- Backbone model in SSD usually is a pre-trained image classification network as a feature extractor.
- The SSD head is just one or more convolutional layers added to this backbone and the outputs are interpreted as the bounding boxes and classes of objects in the spatial location of the final layers activations.
- More default boxes result in more accurate detection, although there is an impact on speed
- having MultiBox on multiple layers results in better detection as well, due to the detector running on features at multiple resolutions.
- The TensorFlow2 Object Detection API allows you to train a collection state of the art object detection models under a unified framework, including Google Brain's state of the art model EfficientDet.