训练 YOLOv3 :基于自定义对象检测的深度学习

YOLOv3 是机器视觉领域最流行的实时对象检测技术。

在前面的文章中，我们分享了 how to use YOLOv3 in an OpenCV application. 非常受欢迎，很多读者询问如何使用新对象（如自定义数据）训练 YOLOv3.

本教程中，实现 YOLOv3 的 1-class 对象检测的例子。本教程为初学者编写。继续以节日为素材，我们实现雪人检测。

本文中，将分享训练过程、辅助脚本，以及在一些公开的图像和视频上的执行效果。可用同样的方式训练多对象检测器.

为了简化教程，请先下载代码。

DOWNLOAD CODE

1. Dataset

深度学习的第一个任务，就是准备数据集。我们使用谷歌 OpenImagesV4 数据集中的雪人图像，可在线直接下载。这个数据集很大，包含 600 种对象。数据集中同时包括了对象标注边界框。数据集 500GB 以上，我们只需要下载“snowman”对象部分。

As with any deep learning task, the first most important task is to prepare the dataset. We will use the snowman images from Google’s OpenImagesV4 dataset, publicly available online. It is a very big dataset with around 600 different classes of object. The dataset also contains the bounding box annotations for these objects. As a whole, the dataset is more than 500GB, but we will download the images with ‘Snowman’ objects only.Copyright Notice

我没有这些图像的版权，因此我们将重点阐述使用这些图像做训练的代码，而不是这些图像本身。OpenImages 中每个图像都有 url 地址和许可信息。使用这些数据（学术、非商业、商业）需要自付法律风险。

We do not own the copyright to these images, and therefore we are following the standard practice of sharing source to the images and not the image files themselves. OpenImages has the originalURL and license information for each image. Any use of this data (academic, non-commercial or commercial) is at your own legal risk.

1.1 下载数据[约 1hour]

首先安装 awscli （用于下载 snowman 图像）

```
sudo pip3 install awscli
```

接着获取相关 openImages 文件。 class-descriptions-boxable.csv 和 train-annotations-bbox.csv(1.11GB) ，用来定位包含我们感兴趣的对象的文件。

```
wget https://storage.googleapis.com/openimages/2018_04/class-descriptions-boxable.csv
```

```
wget https://storage.googleapis.com/openimages/2018_04/train/train-annotations-bbox.csv
```

接着将上面下载的.csv 文件拷贝到与下载脚本代码同目录中，使用如下脚本下载数据：

Next, move the above .csv files to the same folder as the downloaded code and then use the following script to download the data

```
python3 getDataFromOpenImages_snowman.py
```

图像下载到了 JPEGImages 目录，相应的标签数据写入 labels 目录。共下载 539 张图片，770 个雪人对象。下载时间约 1 小时（看网速）。JPEGImages 和 labels 共 136MB。

注意：Win10 下上面的 python 脚本执行时报错，主要时从 csv 中解析 snowman 图像的边界框信息时的错误。我们可用 excel 打开第二个 csv 文件，手动筛选出目标内容，拷贝到新的 csv 文件中，直接按读取第一个 csv 的方式读取 snowman 的边界框数据。

The images get downloaded into the **JPEGImages** folder and the corresponding label files are written into the **labels** folder. The download will get 770 snowman instances on 539 images. The download can take around an hour which can vary depending on internet speed. Both the JPEGImages and labels together should be less than 136 MB.

对于多对象检测器，需要对每个类准备更多的图像，也可以下载 test-annotations-bbox.csv 和 validation-annotations-bbox.csv，修改 python 脚本的 runMode 后重新运行获取更多类别的图像。本例仅针对雪人，770 个实例足够了。

For multiclass object detectors, where you will need more samples for each class, you might want to get the test-annotations-bbox.csv and validation-annotations-bbox.csv files too and then modify *runMode* in the python script and rerun it to get more images for each class. But in our current snowman case, 770 instances are sufficient.

1.3 训练集-测试集分割

机器学习过程首先要将数据随机的分为两部分。Any machine learning training procedure involves first splitting the data **randomly** into two sets.

1. **训练集 Training set** ：用于训练模型的数据。根据数据的数量，可随机选择 70%--90%用于训练。This is the part of the data on which we train the model. Depending on the amount of data you have, you can randomly select between 70% to 90% of the data for training.
2. **测试集 Test set** ：用于测试模型的数据。通常选 10%—30%的数据。不能有图像同时包含在训练集和测试集中。This is the part of the data on which we test our model. Typically, this is 10-30% of the data. No image should be part of the both the training and the test set.

将 JPEGImages 目录中的图像拆分为训练集和测试集。运行 splitTrainAndTest.py 脚本，传递 JPEGImages 目录的完整路径作为参数。

We split the images inside the JPEGImages folder into the train and test sets. You can do it using the **splitTrainAndTest.py** scripts as follows, passing on the full path of the JPEGImages folder as an argument.

```
python3 splitTrainAndTest.py /full/path/to/snowman/JPEGImages/
```

上面的脚本把数据拆分成 90%训练集和 10%的测试集，并生成 snowma_train.txt 和 snowman_test.txt 文件。

要成为机器视觉、机器学习和 AI 的专家，请查看我们的资源：

The above script splits the data into a train (90%) and a test set (10%) and generates two files **snowman_train.txt** and **snowman_test.txt**

Become an expert in **Computer Vision**, **Machine Learning**, and **AI**! Check out our courses

AI COURSES BY OPENCV.ORG

2. Darknet

本教程使用 Joseph Redmon 使用 C 写的深度学习框架 Darknet。

In this tutorial, we use Darknet by Joseph Redmon. It is a deep learning framework written in C.

2.1 下载并编译 Darknet

首先下载源码到本地并编译.

```
cd ~
git clone https://github.com/pjreddie/darknet
cd darknet
```

make

2.2 微调代码开启定期存储模型功能 Modify code to save model files regularly

在本地编译代码前，修改一下源代码，定期存储中间过程 weight。在 examples/detector.c 文件的 135 行 After we make sure the original repo compiles in your system, let’s make some minor modifications in order to store the intermediate weights. In the file examples/detector.c, change line#135 from

```
if(i%10000==0 || (i < 1000 && i%100 == 0)){
```

改为

```
if(i%1000==0 || (i < 2000 && i%200 == 0)){
```

原来的代码在前 1000 次迭代中，每迭代 100 次存储一次中间 weight，后面每迭代 10000 次存储一次中间 weight。本例中由于只是训练单类检测，我们希望训练尽快的收敛。为了能更好的监控中间过程，在前 2000 次迭代中，每迭代 200 次存储一次中间 weight，以后则每 1000 次迭代存储一次中间 weight。修改代码后，重新编译代码，运行 make 命令。The original repo saves the network weights after every 100 iterations till the first 1000 and then saves only after every 10000 iterations. In our case, since we are training with only a single class, we expect our training to converge much faster. So in order to monitor the progress closely, we save after every 200 iterations till we reach 2000 and then we save after every 1000 iterations. After the above changes are made, **recompile** darknet using the **make** command again.

我在 *NVIDIA GeForce GTX 1080 GPU* 上运行测试程序。下面看一下更多必要的依赖。We ran the experiments using an *NVIDIA GeForce GTX 1080 GPU*. Let’s now get into some more details required to run the training successfully.

在 windows 下，需要安装 cygwin 进行快平台编译，过程到不复杂。但建议下载 alexeyab 的 darknet 实现，下载 opencv3.4 后即可使用 vs2017 直接编译，对 win10 更友好，而且对算法进行了增强，在训练过程中提供了可视化的 loss 监控界面。

使用 Alexeyab 版本的 darknet，训练过程请参考：https://blog.csdn.net/haithink/article/details/88843297

3. 数据标注

在 labels 目录中已经生成了标注文件。标注文件中的每一行都对应了图像中的一个边界框，格式如下：We have shared the label files with annotations in the **labels** folder. Each row entry in a label file represents a single bounding box in the image and contains the following information about the box:

```
<object-class-id> <center-x> <center-y> <width> <height>
```

第一项 object-class-id 是一个整数，表示对象的类别，范围是 0 至类别数-1。本例是单类识别，因此一直是 0。The first field **object-class-id** is an integer representing the class of the object. It ranges from 0 to (number of classes – 1). In our current case, since we have only one class of snowman, it is always set to 0.

第二项和第三项，center-x 和 center-y 分别是边界框的 x 和 y 的中心点坐标，已经进行了归一化（分别除以图像的宽和高）。The second and third entry, **center-x** and **center-y** are respectively the x and y coordinates of the center of the bounding box, normalized (divided) by the image width and height respectively.

第四和第五项，分别是边界框的宽和高，已经进行了归一化（分别除以图像的宽和高）。The fourth and fifth entry, **width** and **height** are respectively the width and height of the bounding box, again normalized (divided) by the image width and height respectively.

使用如下符号来举例：Let’s consider an example with the following notations:

x – 边界框中心的 x 坐标（像素）x-coordinate(in pixels) of the center of the bounding box
y – 边界框中心的 y 坐标（像素）y-coordinate(in pixels) of the center of the bounding box
w – 边界框宽（像素）width(in pixels) of the bounding box

h - 边界框高(像素)height(in pixels) of the bounding box
W - 整个图像宽度（像素）width(in pixels) of the whole image
H - 整个图像高度（像素）height(in pixels) of the whole image

标注文件中的变量值关系如下：Then we compute the annotation values in the label files as follows:

center-x = x/W

center-y = y/H

width = w/W

height = h/H

上面的四个浮点变量值在 0—1 之间。The above four entries are all floating point values between 0 to 1.

训练的时候将 labels 目录中的标注文件拷贝到 JPEGImages 中。图像文件（jpg）和标记文件（txt）同名，后缀不同。

4. 下载预训练模型 Download Pre-trained model

要训练自己的对象检测器，从使用大量数据集训练出来的模型作为起点，是一个很好的思路，即使其中不包含我们要检测的对象。此过程叫做迁移学习。When you train your own object detector, it is a good idea to leverage existing models trained on very large datasets even though the large dataset may not contain the object you are trying to detect. This process is called **transfer learning**.

有别于从 0 开始学习，我们使用包含卷积权重训练的 ImageNet 作为预训练模型。使用这些权重作为我们的起始权重，可以加快学习。现在将其下载到本地 darknet 目录。Instead of learning from scratch, we use a pre-trained model which contains convolutional weights trained on ImageNet. Using these weights as our starting weights, our network can learn faster. Let' s download it now to our darknet folder.

```
cd ~/darknet
wget https://pjreddie.com/media/files/darknet53.conv.74 -O ~/darknet/darknet53.conv.74
```

5. 数据文件

在 darknet.data 文件（在发布代码中找），需要指定我们的对象检测器和相关路径信息。In the file **darknet.data**(included in our code distribution), we need to provide information about the specifications for our object detector and some relevant paths.

```
classes = 1
train  = /path/to/snowman/snowman_train.txt
valid  = /path/to/snowman/snowman_test.txt
names = /path/to/snowman/classes.names
backup = /path/to/snowman/weights/
```

Classes 参数指定类别数量，本例是 1. The *classes* parameter needs the number of classes. In our case, it is 1.

指定早期生成的 snowman_train.txt 和 snowman_test.txt 文件的绝对路径，分别包含用于训练和测试的文件列表。You need to provide the absolute paths of the files **snowman_train.txt** and **snowman_test.txt** generated earlier, which have the list of files to be used for training(*train* parameter) and validation(*valid* parameter) respectively.

Names 字段指定包含所有类别名称的文件。我们创建了一个 classes.names 的文件，内部包含 snowman 类名。也需要配置为绝对路径。The *names* field represents the path of a file which contains the names of all the classes. We have included the **classes.names** file which contains the class name 'snowman'. You will need to provide its absolute path in your machine here.

最后，backup 参数指向已存的目录，用于存放训练过程的中间权重文件。Lastly, for the *backup* parameter, we need to give the path to an existing directory where we can store the intermediate weights files as the training progresses.

6. YOLOv3 配置参数 configuration parameters

有了 darknet.data 和 classes.names 文件，YOLOv3 还需要 darknet-yolov3.cfg 配置文件。也包含在我们的代码库中了。这个文件是基于范例配置文件 yolov3-voc.cfg 的（来自 darknet 代码），用于在 VOC 数据集上进行训练。所有重要的训练参数都存储在此配置文件中。我们看一下内部参数的意义和内容。Along with the darknet.data and classes.names files, YOLOv3 also needs a configuration file **darknet-yolov3.cfg**. It is also included in our code base. It is based on the demo configuration file, yolov3-voc.cfg (comes with darknet code), which was used to train on the VOC dataset. All the important training parameters are stored in this configuration file. Let us understand what they mean and what values to set them to.

6.1 YOLOv3 中的 Batch 超参数 Batch hyper-parameter in YOLOv3

下面详细介绍 batch 和 subdivision 参数。Let’s learn more about batch and subdivision parameter.

```
[net]
# Testing
# batch=1
# subdivisions=1
# Training
batch=64
subdivisions=16
```

Batch 参数指定训练过程中的批大小（**每批次训练使用的图片数量**）。The **batch** parameter indicates the **batch size** used during training.

我们的训练集包含一百多张图片，但训练几百万张图象也是正常的。训练过程就是基于在训练数据集上的误差来迭代更新神经网络权重。Our training set contains a few hundred images, but it is not uncommon to train on million of images. The training process involves iteratively updating the weights of the neural network based on how many mistakes it is making on the training dataset.

单次训练使用训练集中的全部图像来更新权重是不现实的。因此，单次迭代会使用这些图像的一个小子集，这个子集的大小就叫批尺寸（batch size）。It is impractical (and unnecessary) to use all images in the training set at once to update the weights. So, a small subset of images is used in one iteration, and this subset is called the **batch size**.

当 batch size 设置为 64，表示单次迭代使用 64 张图像更新神经网络参数。When the batch size is set to 64, it means 64 images are used in one iteration to update the parameters of the neural network.

6.2 YOLOv3 的 Subdivisions 配置参数 Subdivisions configuration parameter in YOLOv3

即使单批次使用 64 张图片来训练神经网络，但 GPU 可能也没有足够的内存处理这 64 张图像。幸运的是，Darknet 提供了一个 subdivisions 参数，可以指定 GPU 上单次处理图像的数量（再细分）。DarknetEven though you may want to use a batch size of 64 for training your neural network, you may not have a GPU with enough memory to use a batch size of 64. Fortunately, Darknet allows you to specify a variable called **subdivisions** that lets you process a fraction of the batch size at one time on your GPU.

如果看到 Out of memory 错误，可以设置 subdivisions=1，已 2 的倍数逐步增加 subdivisions 参数（如 2、4、8、16）直到训练过程成功。**GPU 单次提取 batch/subdivisions 个图像进行处理**，但完整的批或迭代还是以 64 张图片为单位。You can start the training with subdivisions=1, and if you get an *Out of memory error*, increase the subdivisions parameter by multiples of 2(e.g. 2, 4, 8, 16) till the training proceeds successfully. The GPU will process $batch/subdivision$ number of images at any time, but the full batch or iteration would be complete only after all the 64 (as set above) images are processed.

在测试过程中，batch 和 subdivisions 都设置为 1.During testing, both batch and subdivision are set to 1.

6.3 Width, Height, Channels

这些参数指定输入图像的尺寸和通道数。These configuration parameters specify the input image size and the number of channels.

```
width=416
height=416
channels=3
```

训练前首先将输入图像 resize 为 width x height 大小。这里我们使用默认的 416x416，改为 608x608 可能会获得更好的效果，但花费时间更长。Channels=3 表示要处理的输入图像是 3 通道的 RGB 格式。The input training images are first resized to **width** x **height** before training. Here we use the default values of 416×416. The results might improve if we increase it to 608×608, but it would take longer to train too. **channels**=3 indicates that we would be processing 3-channel RGB input images.

6.4 动能和衰减 Momentum and Decay

配置文件中包括几个控制权重更新的参数。The configuration file contains a few parameters that control how the weight is updated.

```
momentum=0.9
decay=0.0005
```

上面提到神经网络权重更新是基于小批次图像的，而不是全部数据集。因此权重更新的波动很大，需要使用动能（momentum）参数来抑制迭代中权重较大的变更。In the previous section, we mentioned how the weights of a neural network are updated based on a small batch of images and not the entire dataset. Because of this reason, the weight updates fluctuate quite a bit. That is why a parameter **momentum** is used to penalize large weight changes between iterations.

典型的神经网络都有百万个权重数据，因此很容易在训练数据上过拟合。过拟合即模型在训练数据上表现良好，但在测试数据集上很差。就像神经网络已经把训练集上的所有图像的答案都记下来了，但并没有学到潜在的概念。缓解这个问题的一个思路是抑制权重中较大的值。decay 参数就是控制这个惩罚项的。通常默认值就可以，如果发生过拟合需要调整这个参数。A typical neural network has millions of weights and therefore they can easily overfit any training data. Overfitting simply means it will do very well on training data and poorly on test data. It is almost like the neural network has memorized the answer to all images in the training set, but really not learned the underlying concept. One of the ways to mitigate this problem is to penalize large value for weights. The parameter **decay** controls this penalty term. The default value works just fine, but you may want to tweak this if you notice overfitting.

6.5 Learning Rate, Steps, Scales, Burn In (warm-up)

```
learning_rate=0.001
policy=steps
steps=3800
scales=.1
burn_in=400
```

学习率参数控制当前批次数据应该采用的学习积极程度。通常介于 0.01 到 0.0001 之间。The parameter **learning rate** controls how aggressively we should learn based on the current batch of data. Typically this is a number between 0.01 and 0.0001.

在训练刚开始时，我们从零信息开始，所以学习率需要很高。但当神经网络看到很多数据后，权重变化需要变得保守些。换句话说，学习率应该随着时间推移在变小。在配置文件中，制定了控制学习率变化的策略是 steps。上例中，学习率时 0.001，在 3800 次迭代中是不变的，然后乘以 scales 获得一个新的学习率。也可以指定多个 steps 和 scales。At the beginning of the training process, we are starting with zero information and so the learning rate needs to be high. But as the neural network sees a lot of data, the weights need to change less aggressively. In other words, the learning rate needs to be decreased over time. In the configuration file, this decrease in learning rate is accomplished by first specifying that our learning rate decreasing **policy** is **steps**. In the above example, the learning rate will start from 0.001 and remain constant for 3800 iterations, and then it will multiply by **scales** to get the new learning rate. We could have also specified multiple steps and scales.

从上面的描述中，我们知道训练开始时学习率较大，而后逐渐变小。虽然这种说法在很大程度上是正确的，但经验表明，如果我们在开始的短时间内使用较小的学习率，会提高整体的训练速度。这是由 burn_in 参数控制的。有时将 burn-in 阶段也叫

warm-up（预热）阶段 In the previous paragraph, we mentioned that the learning rate needs to be high in the beginning and low later on. While that statement is largely true, it has been empirically found that the training speed tends to increase if we have a lower learning rate for a short period of time at the very beginning. This is controlled by the **burn_in** parameter. Sometimes this burn-in period is also called **warm-up** period.

6.6 数据增强 Data augmentation

我们知道收集数据很费时间。本例中，我们首先需要收集 1000 张图片，然后要手动标注边界框。需要一个 5 人团队花 1 天的时间来完成。We know data collection takes a long time. For this blog post, we first had to collect 1000 images, and then manually create bounding boxes around each of them. It took a team of 5 data collectors 1 day to complete the process.

我们要通过变换来最大化使用数据。这个过程就是图像增强。例如，将雪人图像旋转 5 度，还是一个雪人图像。配置文件中的 Angle 参数控制图像进行随机的旋转± angle。We want to make maximum use of this data by cooking up new data. This process is called data augmentation. For example, an image of the snowman rotated by 5 degrees is still an image of a snowman. The **angle** parameter in the configuration file allows you to randomly rotate the given image by ± angle.

同样，如果变换整张图像颜色的 saturation(饱和度)、exposure(曝光)、hue(色调)，图像也还是雪人图像。Similarly, if we transform the colors of the entire picture using **saturation**, **exposure**, and **hue**, it is still a picture of the snowman.

```
angle=0
saturation = 1.5
exposure = 1.5
hue=.1
```

我们使用默认值进行训练。We used the default values for training.

6.7 迭代次数 Number of iterations

最后指定训练过程的迭代次数。Finally, we need to specify how many iterations should the training process be run for.

```
max_batches=5200
```

对多分类检测器，max_batches 参数设置会更大，需要运行更多次数（如 yolov3-voc.cfg）。在 n 个类别的对象检测器中，可以设置运行 2000*n 批次的训练。本例中，只有 1 个类别，max_batches 设置为 5200 应该是合理的。For multi-class object detectors, the max_batches number is higher, i.e. we need to run for more number of batches(e.g. in yolov3-voc.cfg). For an n-classes object detector, it is advisable to run the training for at least 2000*n batches. In our case with only 1 class, 5200 seemed like a safe number for max_batches.

7. 训练 YOLOv3

现在我们知道了训练必要的概念，就开始训练过程。在 darknet 目录，运行如下命令：Now that we know what all different components are needed for training, let's start the training process. Go to the darknet directory and start it using the command as following:

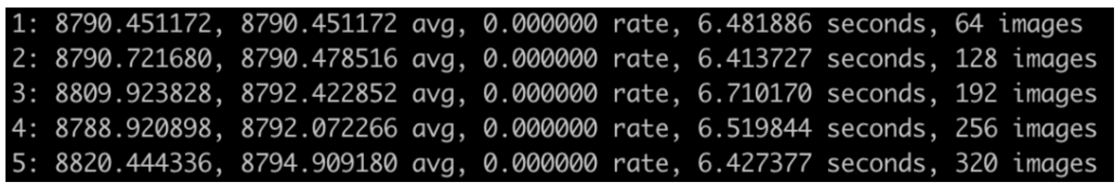
```
cd ~/darknet
./darknet detector train /path/to/snowman/darknet.data /path/to/snowman/darknet-
yolov3.cfg ./darknet53.conv.74 > /path/to/snowman/train.log
```

确保将 darknet.data 和 darknet-yolov3.cfg 放置在正确的目录。将训练日志保存到数据集目录的 train.log 文件中，便于在训练过程中跟踪 loss。Make sure you give the correct paths to darknet.data and darknet-yolov3.cfg files in your system. Let's also save the training log to a file called train.log in your dataset directory so that we can progress the loss as the training goes on.

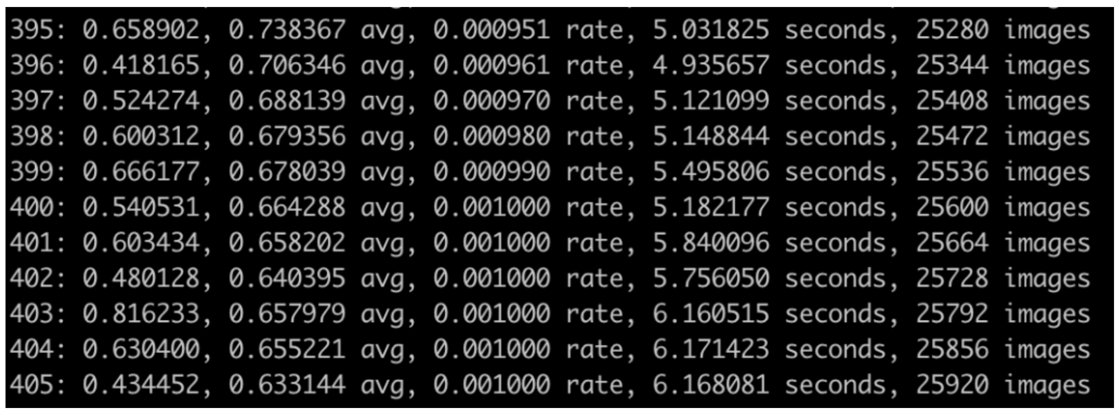
在训练过程中跟踪 loss 值可使用 grep 命令（win 中是 FINDSTR）搜索 train.log 文件。A useful way to monitor the loss while training is using the grep command on the train.log file

```
grep  "avg"  /path/to/snowman/train.log
```

其中显示批次数、当前批次的 loss、平均 loss 值、当前学习率、批次所用时间、累计使用图像数。下图中可见每批次使用图像数增加 64.这是因为我们设置的 batch 大小为 64. It shows the batch number, loss in the current batch, average loss till the current batch, current learning rate, time taken for the batch and images used till current batch. As you can see below the number of images used till each batch increases by an increment of 64. That is because we set the batch size to 64.



```
1: 8790.451172, 8790.451172 avg, 0.000000 rate, 6.481886 seconds, 64 images
2: 8790.721680, 8790.478516 avg, 0.000000 rate, 6.413727 seconds, 128 images
3: 8809.923828, 8792.422852 avg, 0.000000 rate, 6.710170 seconds, 192 images
4: 8788.920898, 8792.072266 avg, 0.000000 rate, 6.519844 seconds, 256 images
5: 8820.444336, 8794.909180 avg, 0.000000 rate, 6.427377 seconds, 320 images
```



```
395: 0.658902, 0.738367 avg, 0.000951 rate, 5.031825 seconds, 25280 images
396: 0.418165, 0.706346 avg, 0.000961 rate, 4.935657 seconds, 25344 images
397: 0.524274, 0.688139 avg, 0.000970 rate, 5.121099 seconds, 25408 images
398: 0.600312, 0.679356 avg, 0.000980 rate, 5.148844 seconds, 25472 images
399: 0.666177, 0.678039 avg, 0.000990 rate, 5.495806 seconds, 25536 images
400: 0.540531, 0.664288 avg, 0.001000 rate, 5.182177 seconds, 25600 images
401: 0.603434, 0.658202 avg, 0.001000 rate, 5.840096 seconds, 25664 images
402: 0.480128, 0.640395 avg, 0.001000 rate, 5.756050 seconds, 25728 images
403: 0.816233, 0.657979 avg, 0.001000 rate, 6.160515 seconds, 25792 images
404: 0.630400, 0.655221 avg, 0.001000 rate, 6.171423 seconds, 25856 images
405: 0.434452, 0.633144 avg, 0.001000 rate, 6.168081 seconds, 25920 images
```

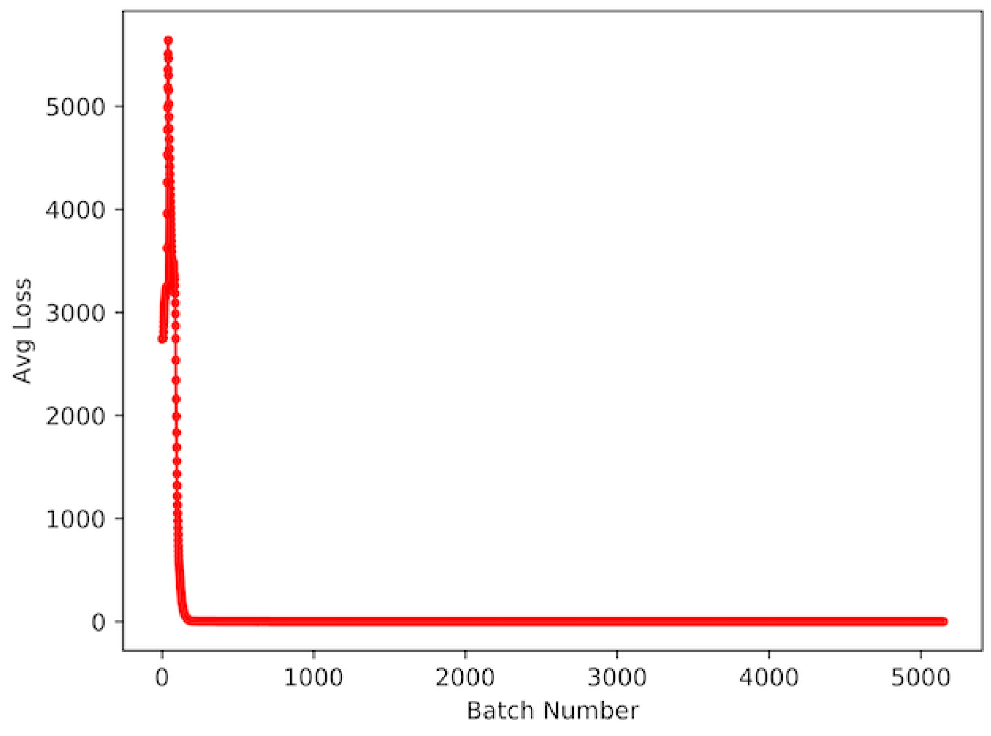
而学习率在前 400 批次从 0 逐步增大到 0.001，而后一直到第 3800 次迭代都保持不变，最后逐步变为 0.0001. As we can see the learning rate increases gradually from 0 to 0.001 by the 400th batch. It would stay there till the 3800th batch when it would again change to 0.0001.

使用 AlexeyAB 版本的 darknet，训练过程中提供了查看 loss 的可视化界面。

7.1 何时停止训练 When do we stop the training?

随着训练的推进，log 文件包含每批次的 loss 值。当 loss 低于某个阈值时应该停止训练。下面绘制的是雪人检测器训练的 loss-batch number 的点图。可使用如下脚本生成：As the training goes on, the log file contains the loss in each batch. One could argue to stop training after the loss has reached below some threshold. Below is the loss plotted against the batch number for our snowman detector. We generate the plot using the following script:

```
python3 plotTrainLoss.py /full/path/to/train.log
```



但是最终测试需要使用学习到的权重获取 mAP。Darknet 初始代码不含计算 mAP 功能。我们提供了在 darknet 代码中直接计算 mAP 的功能，便于跟踪精度，在保存权重文件时同时存储 mAP。将在后续文章给出。**同时也可查看一下 AlexeyAB 的 darknet 分支中计算 mAP 的实现。** But the actual test should be seeing the mAP using the learned weights. The original darknet code does not have a code to compute mAP. We are working on providing code to compute mAP directly in the darknet code so that you can monitor the precision and recall along with mAP when the weights files are saved. It would come up as a follow-up post. In the meanwhile, you might want to check out AlexAB 's fork of darknet for computing mAP.

在雪人检测器中，仅在配置文件中使用 5200 迭代。因此可一直运行到结束。最后训练的权重文件 **darknet-yolov3_final.weights** 的 mAP（mean Average Precision）为 70.37%。点击下载 here. For the snowman detector, we have only 5200 iterations in the configuration file. So you might just let it run till the end. Our final trained weights file, **darknet-yolov3_final.weights** got a mean Average Precision(mAP) of **70.37%**. You can download it here.

8. 测试模型 Testing the model

除了 loss 和 mAP，我们还需要在新的数据上测试权重文件，查看结果确保我们对准确率满意。在早期文章中，我们讲了如何使用 OpenCV 测试 YOLOv3 模型。已经有测试雪人检测器的代码了。只需在 **object_detection_yolo.py** 代码中调整 *modelConfiguration* 和 *modelWeights* 文件路径，在图像或视频上测试雪人检测模型。如：Along with the loss and mAP, we should always test our weights file on new data and see the results visually to make sure we are happy with the results. In an earlier post, we described how to test the YOLOv3 model using OpenCV. We have included the code for testing your snowman detector. You will need to give the correct path to the *modelConfiguration* and *modelWeights* files in **object_detection_yolo.py** and test with an image or video for snowman detection, e.g.

```
python3 object_detection_yolo.py --image=snowmanImage.jpg
```

订阅&下载代码 Subscribe & Download Code

如果你喜欢本文并要下载代码（C++和 Python），以及本文中的范例图像，请订阅（subscribe）我们的 newsletter 。同时可获得免费的 Computer Vision Resource。在我们的 newsletter 中共享 OpenCV 教程和范例（C++和 Python），机器视觉和机器学习算法和新闻。If you liked this article and would like to download code (C++ and Python) and example images used in this post, please subscribe to our newsletter. You will also receive a free Computer Vision Resource Guide. In our newsletter, we share OpenCV tutorials and examples written in C++/Python, and Computer Vision and Machine Learning algorithms and news.