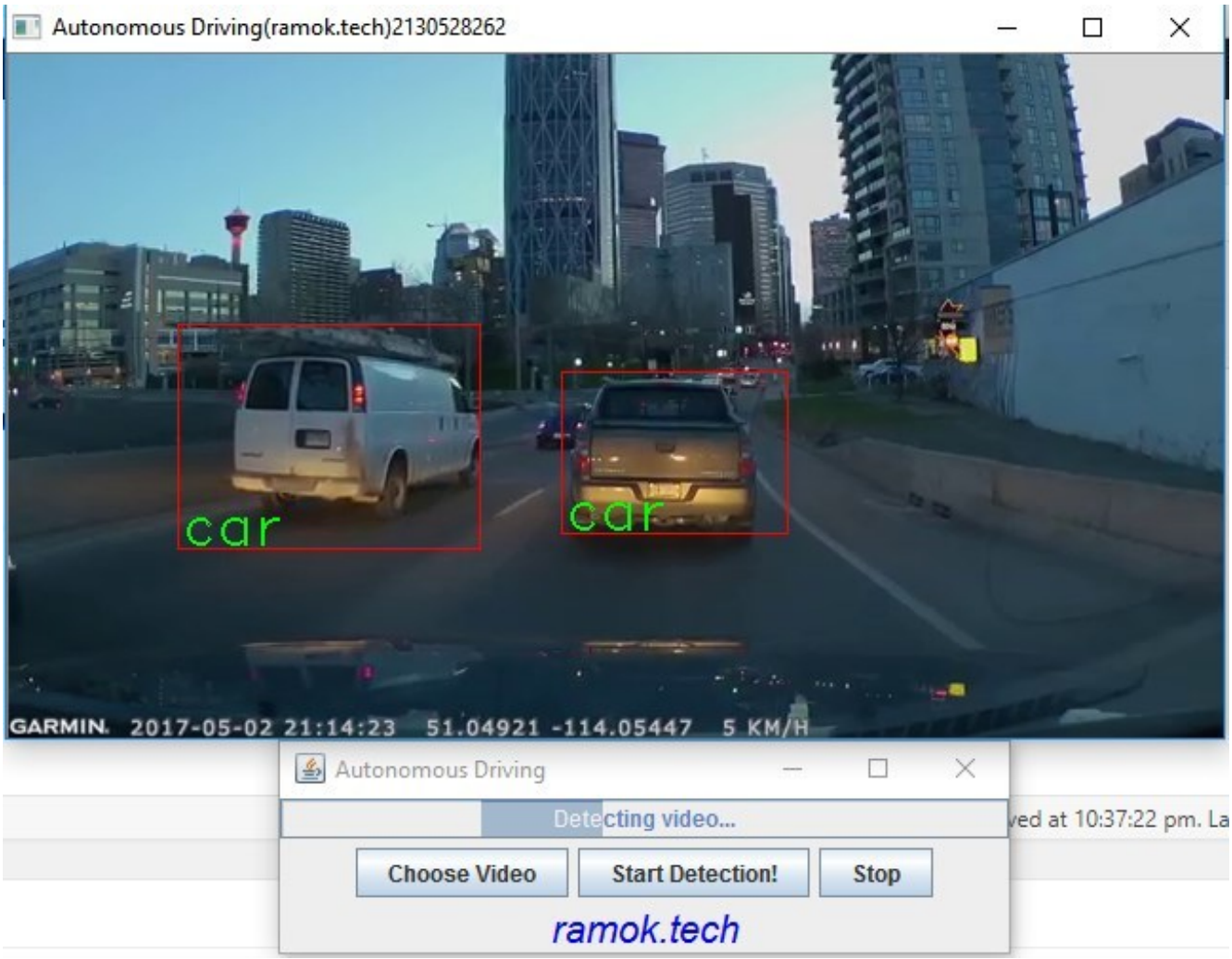
Java构建汽车无人驾驶：汽车目标检测

Java Autonomous Driving: Car Detection

原文地址： https://dzone.com/articles/java-autonomous-driving-car-detection-1

在这篇文章中，我们将用Java构建一个实时视频对象检测应用程序，用于汽车检测，这是自动驾驶系统的一个关键组件。 在之前的文章中，我们能够构建一个图像分类器（猫与狗）；现在，现在我们要检测物体（即汽车，行人），并用边框（矩形）标记它们。 随意下载代码或使用自己的视频运行应用程序（简短视频示例）。



目标检测本质

Object Classification（物体分类）

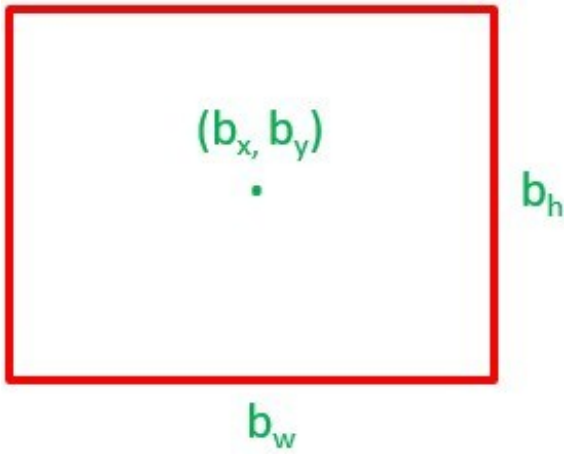
首先，我们遇到了对象分类的问题，比如我们想知道在一个图片中是否包含一个特定的物体，在下图中即图像中是否包含汽车。

我们在前一篇文章中提到了使用现有的架构VGG-16和迁移学习去构建图像分类器。

Object Localization（物体定位）

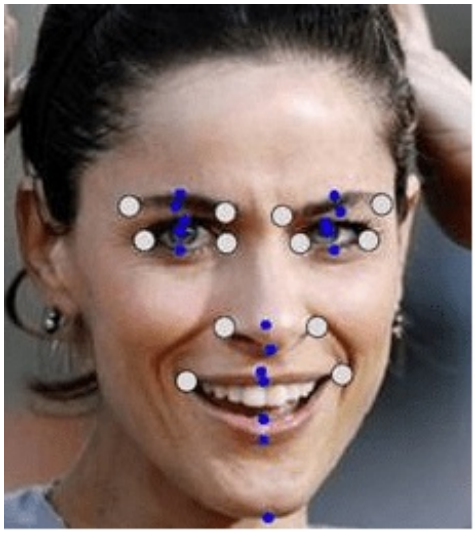
现在我们可以说，我们拥有高置信度去判断图片中是否含有某一特定物体，我们就面临着图像中物体定位的挑战。通常，这是通过用矩形或边界框标记对象来完成的。

除了图片分类，我们需要额外去识别物体在图像中的位置。这通过定义边界框来完成。边界框通常表现为物体的中心$center(b^x, b^y)$，矩形的高$height(b^h)$，矩形的宽$width(b^w)$。



现在，我们就需要为我们的训练集数据中为图像中的每一个物体都定义这四个变量。此外，网络不仅会输出图像类别编号（即20%cat [1]，70%dog [2]，10%tiger [3]）的概率，还会输出上面定义边界框的四个变量值。

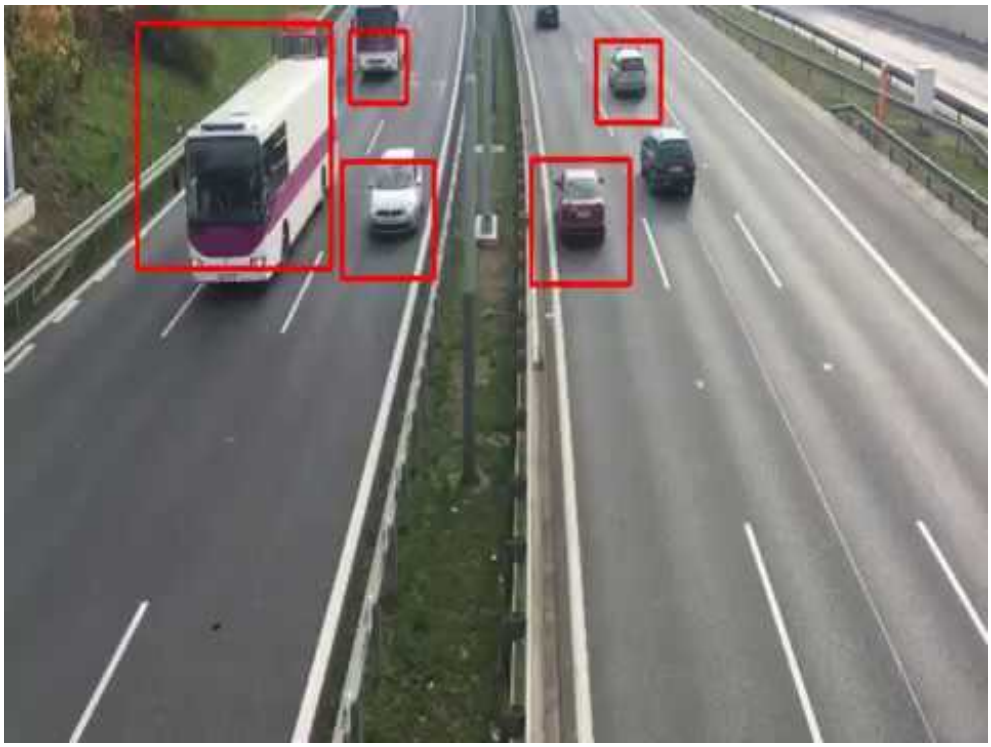
提供边界框点（中心，宽度，高度），我们的模型通过给我们更详细的图像内容视图输出/预测更多的信息。



不难想象，在图像训练数据中加入更多点可以让我们更深入地了解图像。 例如，在人脸上（即嘴巴，眼睛）的点可以告诉我们这个人在微笑，哭泣等等。

Object Detection（目标检测）

我们可以更进一步的定位图像中的多个物体。

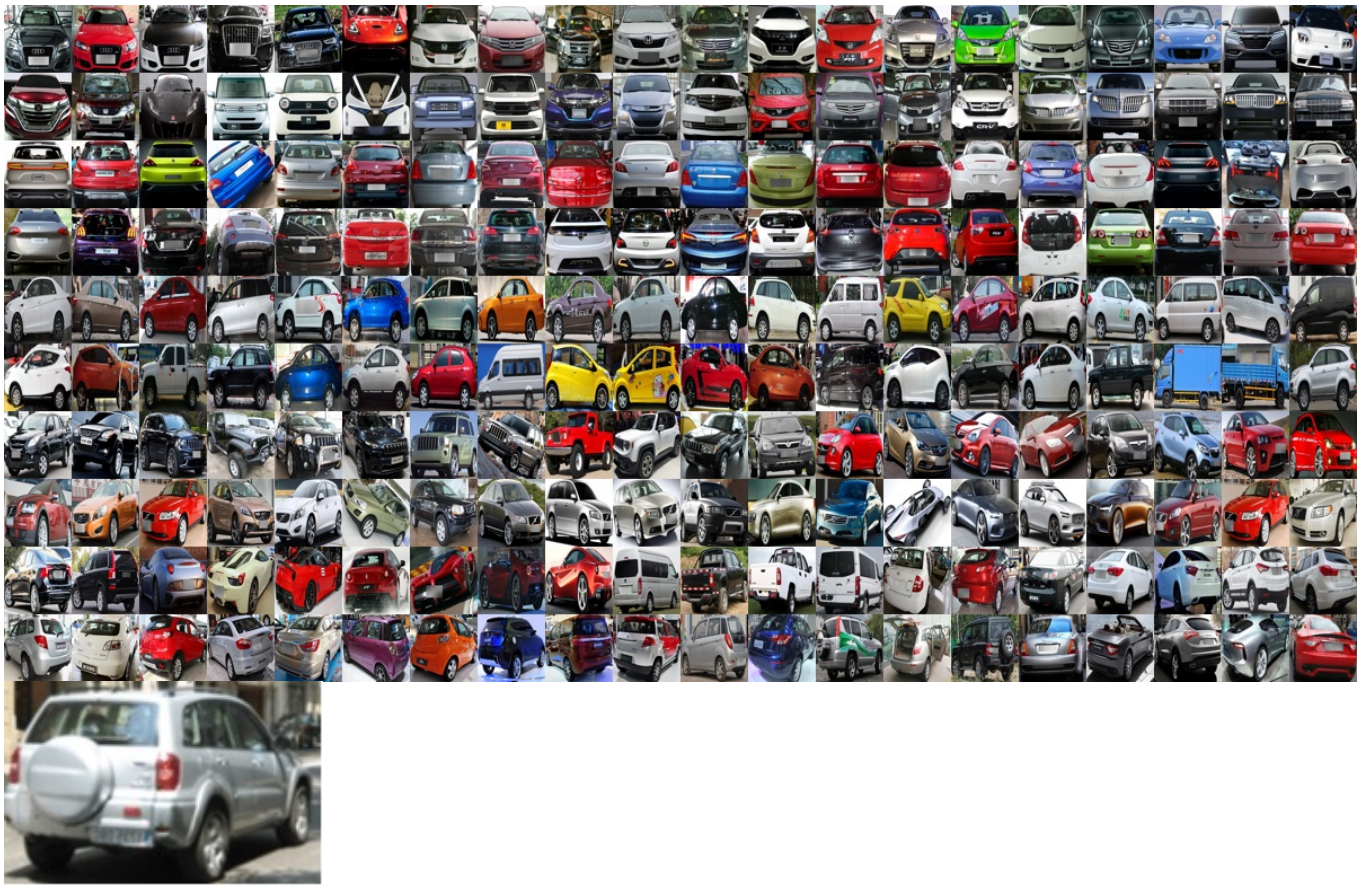


虽然结构没有太大变化，但这里的问题变得更加困难，因为我们需要准备更多的数据（多边界框）。 原则上，我们只是将图像分成较小的矩形，对于每个矩形，我们都有相同的额外的五个变量 - $P^c, (b^x, b^y), b^h, b^w$ - 和正常的预测概率（20%cat [1]，70%dog[2]）。 稍后我们会看到更多细节，但现在让我们假设结构与物体定位相同，不同之处在于我们有多个结构。

Sliding Window Solution（滑动窗口解决方案）

这是一个非常直观的解决方案。这个想法是不使用一般的汽车图像，而是尽可能的裁剪图像使得图像中只含有汽车。

我们使用裁剪之后的图像，去训练一个网络结构类似于VGG-16或者其他深度网络的卷积神经网络。



这种方法的结果很好，但是这种模型只能用于检测图像中是否含有汽车，所以在检测真实场景的图像中会有问题，因为它里面包含有其他的物体（比如说树、人、交通标志等）。
除此之外，现实图像一般也比训练数据的尺寸要大。

为了克服这些问题，我们可以只分析图像中的一部分并且分析这一部分是否含有汽车的一部

分。更精确的说法是，我们使用滑动的矩形框扫描整个图片，在每一次扫描的时候使用我们的模型来判断这一部分是否含有汽车。让我们看一个示例：

总之，我们使用正常的卷积神经网络（VGG-16）来训练带有不同大小裁剪图像的模型，然后使用矩形扫描物体（汽车）的图像。 我们可以看到，通过使用不同大小的矩形，我们可以计算出不同的车辆形状和位置。

这个算法并不复杂，并且奏效。但是这个算法拥有两个缺点。

1. 第一个问题是性能问题。我们必须调用模型预测结果很多次。每一次矩形框移动，我们都需要调用模型用于获取预测结果 - 并且我们需要使用不同的矩形框大小来重复做这一件事情。其中一个解决性能问题的方法是增加矩形框的步长（使用大步长），但是这样我们可能会检测不出来某些物体。在过去，模型主要是线性的，并且具有手动设计的特征，因此预测并不昂贵。 所以这个算法用来做得很好。 目前，网络规模（VGG-16的参数为1.38亿），这种算法速度很慢，对于像自主驾驶这样的实时视频对象检测几乎没有用处。

另外一个导致算法性能不佳的原因是：当我们移动矩形框时（向右和向下移动），许多共用的像素点并没有重用而是重复计算了。在下一部分，我们将使用最新的卷积来克服这个问题。

2. 即使使用不同大小的矩形框来扫描图像，我们依旧可能无法检测出图像中包含的物体。模型可能无法输出准确的边界框信息；例如：这个矩形中只包含物体的一部分。在下一个部分中我们将探讨 YOLO (you only look once) 算法，这个将会为我们解决这个问题。

Convolutional Sliding Window Solution（卷积滑动窗口解决方案）

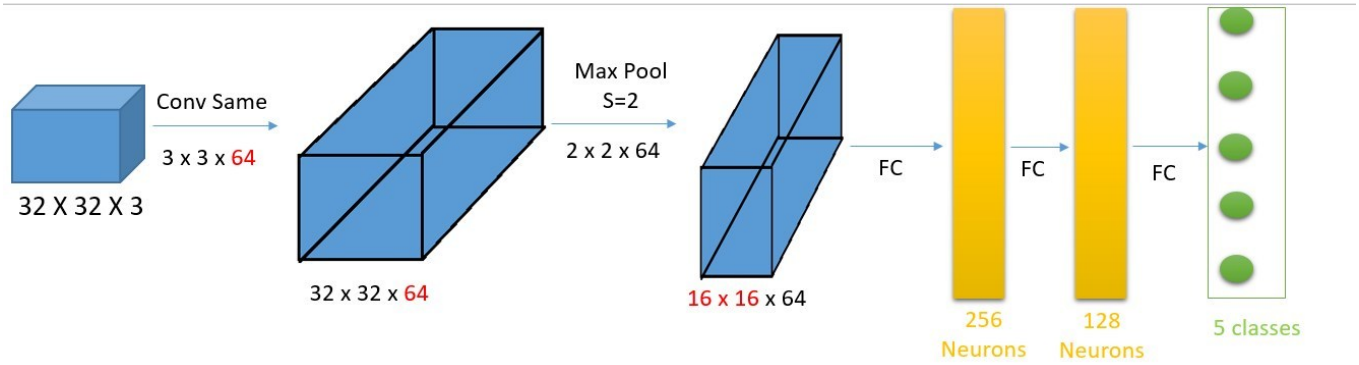
我们看到滑动窗口因为无法重用大量已经计算过后的数值有性能问题。每一次滑动窗口移动之

后，我们都需要计算模型中大量的参数（可能上百万的参数），为了获取一个预测值。在现实中，我们可以引入卷积结构来重用这些计算结果。

Turn Fully Connected Layers Into Convolution（从全连接网络转向卷积神经网络）

我们在前一篇文章的最后看到，图像分类结构 - 无论其大小和配置如何 - 训练不同层数的全连接层网络，输出的数量取决于分类物体的种类。

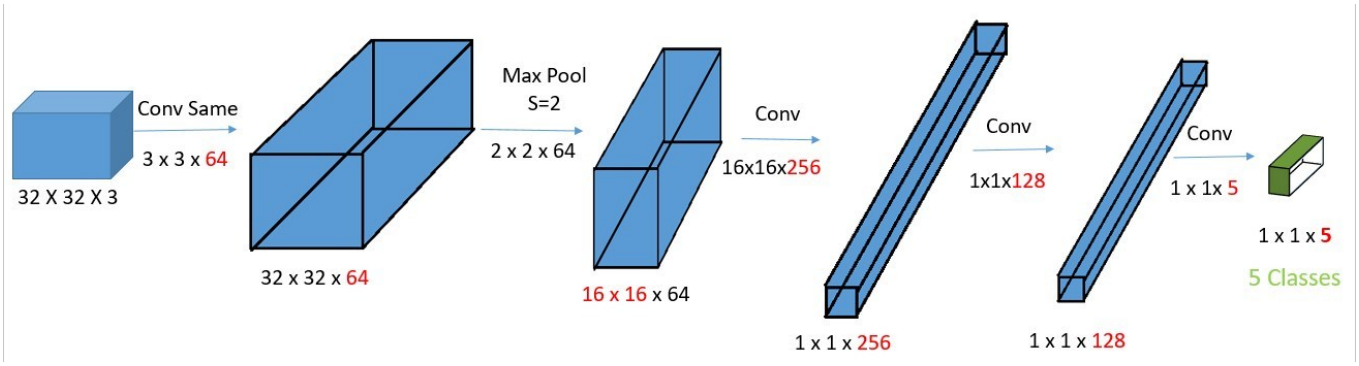
为了简单起见，我们将以较小的网络模型为例，但对于任何卷积网络来说，相同的逻辑都是有效的。（VGG-16,AlexNet）



有关卷积更加直观详细的解释请查看之前一篇文章。

这个简单的网络使用大小为`32 x 32 x 3`的彩色图片作为网络的输入，使用SAME`3x3x64`的卷积（这种卷积方式不会改变图像的宽和高）获取了一个大小为`32 x 32 x 64`的输出（提示，输出数据的第三个维度和卷积和的维度64大小相等，这通常用于对输入数据的增益）。接下来使用最大池化削减宽度和高度，并且保持第三个维度的数据不发生变化（16 x 16 x 64）。在此之后，我们使用神经元个数为256和128的两层全连接网络。在最后我们输出五个种类的概率（通常使用soft-max）。

让我们看看如何用卷积层替换全连接层，同时保持数学效果相同（输入16 X 16 X 64的线性函数）。

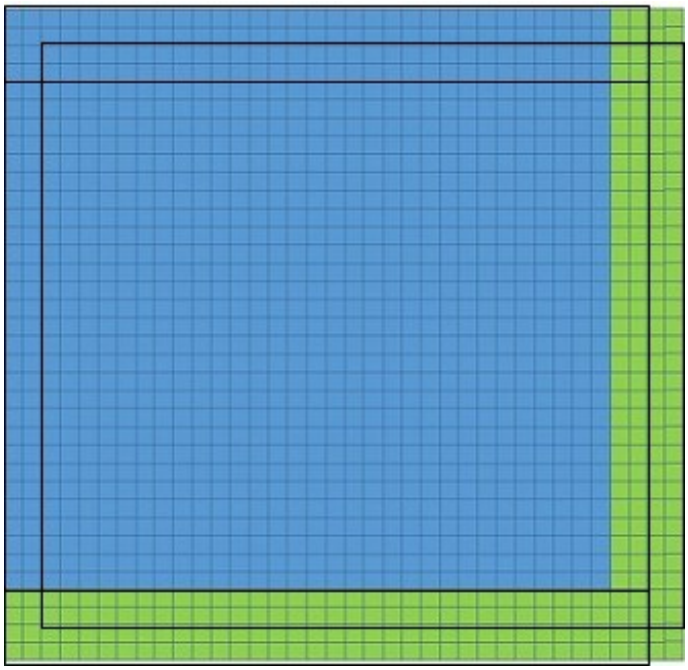


我们只是使用卷积核来代替全连接层。在现实中，`16x16x256`的卷积核实际上是一个`16x16x64x256`的矩阵（多个卷积核），因为卷积核的第三个维度和输入的第三个维度总是一样的。为了简便，我们将其认为是`16 x 16 x 256`。这意味着，这相当于一个全连接层，因为输出1 X 1 X 256的每个元素都是输入16 X 16 X 64的每个元素的线性函数。

我们为什么要将全连接（FC）层转换为卷积层的原因是因为这会给我们在选择输出方式时带来更大的灵活性。 借助FC，您将始终具有相同的输出大小，即类数。

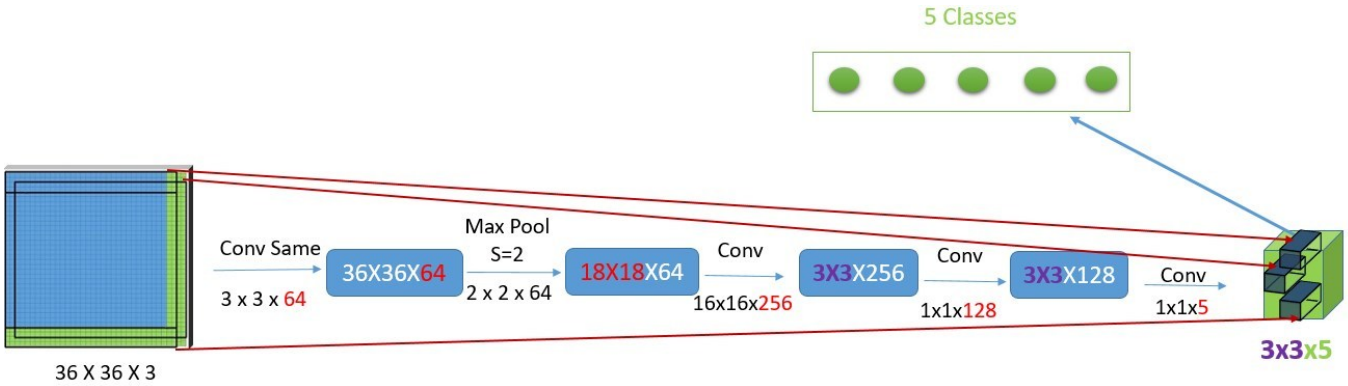
Convolution Sliding Window（卷积滑动窗口）

想要看到使用卷积网络替代全连接层背后的思想，我们需要输入图片的大小大于原来的图片大小-`32x32x3`。让我们使用`36x36x3`的图片作为输入。



这个图像 (with green 36 X 36)比原有的图像(blue 32 X 32)大四行四列。如果我们使用步长为2的滑动窗口和全连接层网络，我们需要移动原有图像大小9次才可以覆盖整个图像（图中的黑色矩形示范了其中的三种移动结果）。与此同时，调用模型9次。

接下来让我们尝试将更大图像应用在我们只使用卷积层的新模型上。



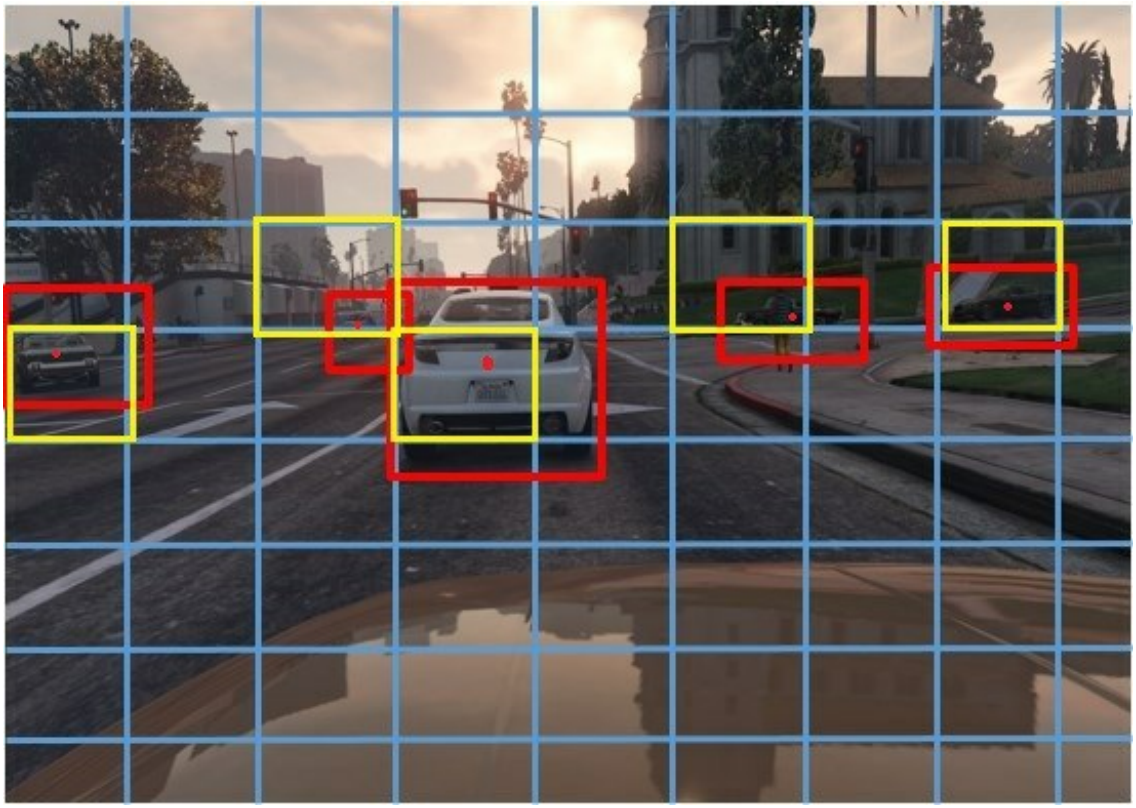
正如同我们所看到的，相比于全连接层的输出只能为`1x1x5`，输出从`1x1x5`变成了`3x3x5`。回想到我们必须移动9次才能覆盖整个图片 - 等一下，这是不是正好等于`3x3`？是的，这些3x3大小的单元表示了每一个滑动窗口的`1x1x5`的分类概率输出结果。

这是一种非常先进的技术，因为我们只需一次就可以立即获得全部九个结果，而无需多次执行带有数百万参数的模型。

YOLO (You Only Look Once)

虽然我们通过引入卷积滑动窗口的方式解决了性能问题。但是我们的模型依旧不能输出非常准确的边界框，即便使用大量不同大小的边界框。让我们看YOLO是如何解决这个问题。

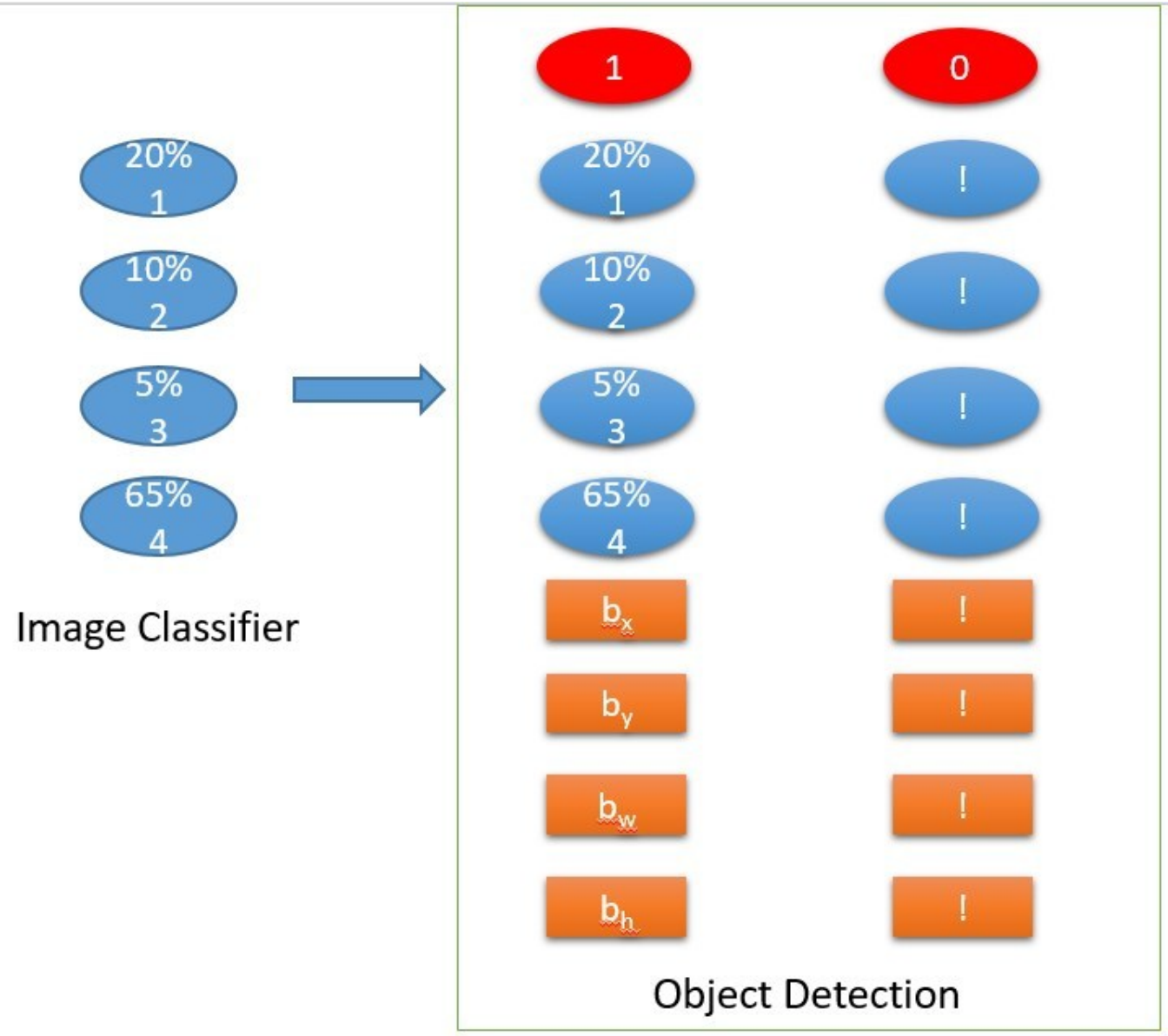
首先，我们通常在每一个图片中标注我们想要检测的物体。每一个物体都会使用四个变量，通过边界框的方式进行标注 - 记住这四个变量是物体的中心$center(b^x, b^y)$，矩形的高$height(b^h)$，矩形的宽$width(b^w)$。每个图像都被会分割成更小的矩形 - 通常分为`19x19`个矩形，为了简单起见，这里分割为`8x9`。



红色的边界框和物体都是由很多蓝色边界框的一部分组成的，所以我恩枝江对象和边界框分配给拥有对象中心的黄色框。我们使用额外的四个变量（除了告知物体为汽车，还额外提供物体的中心，宽度和高度）训练我们的模型，并且将这些变量分配给拥有中心点的边界框。在神经网络使用这种标注好的数据集训练完成以后，我们将用该模型用于预测这四个变量的值（除了识别物体的类别），值或者边界框。

我们不是使用预定义的边界框大小进行扫描，而是试图拟合对象，我们让模型学习如何用边界框标记对象; 因此，边界框现在是灵活的（被学习）。 这样，边界框的精度就高得多并且灵活多了。

让我们来看看现在我们如何表示输出，除了像1-车，2-行人这样的类之外，我们还有额外的四个变量（b^x，b^y，b^h，b^w）。 实际上，还增加了另一个变量P^c，它会告诉我们图像是否有任何我们想要检测的对象。



1. $P^c = 1$(红色标注) 意味着这里至少有一个我们想要被检测出来的对象，所以就值得去查看被识别物体的分类概率和边界框的值。
2. $P^c = 0$（红色标注）意味着里面检测出任何我们想要检测出来的对象，我们不需要关心模型预测出来的分类概率和边界框的值。

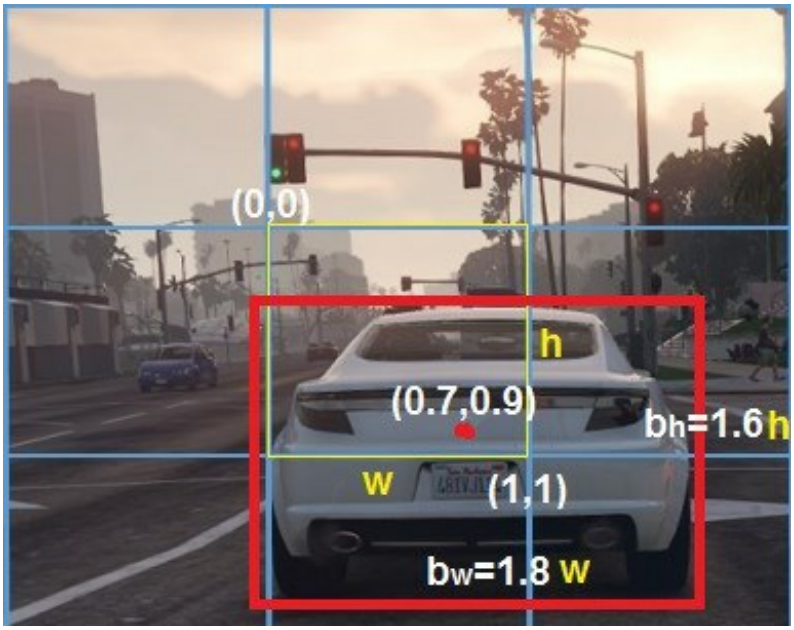
Bounding Box Specification（边界框识别）

我们需要使用特殊的方法来标记我们的数据以便于YOLO算法能正确的运转。`YOLO V2`格式需要边界框的维度值（b^x，b^y，b^h，b^w）为原始图片宽高的相对值。假设我们拥有一个300x400大小的图片，并且边界框的维度为 $B^{width} = 30$，$B^{height} = 15, B^X = 150, B^Y = 80$。这些值需要转换为

$B^{width} = 30/300$，$B^{height} = 15/400, B^X = 150/300, B^Y = 80/400$。

这篇文章将会讲述如何借助`BBox Label Tool`工具花费很小的代价来标注我们的数据。这个工具标记的边界框（给我们的是左上的点和右下的点）与`YOLO V2`格式有些许的不一样。但是将其转化成为我们想要的数据是非常简单的事情。

除了YOLO需要训练数据的标签，在内部实现中，预测方式也有所不同。



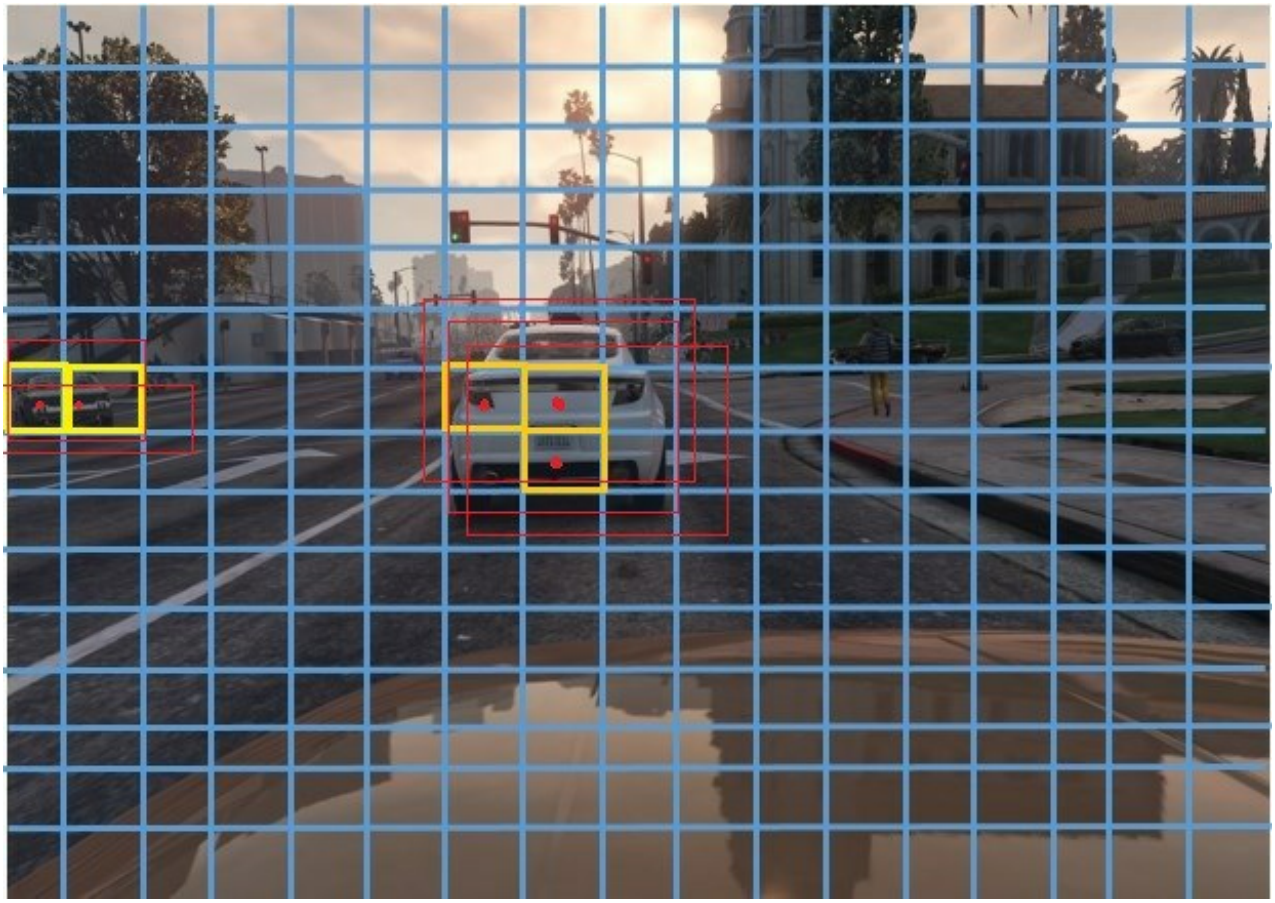
YOLO预测的边界框是相对于拥有对象中心的框（黄色）进行定义。黄色框左上角的点定义为(0,0)，并且将右下角的点定义为（1,1）。所以中心店（b^x，b^y）的值范围一定在$0-1$之间（sigmoid激活函数会确保这个结果），因为中心点一定在这个黄色的边界框之中。b^h和b^w是相对于黄色框的宽和高计算出来的相对值，所以其值可以大于1。在这个图片中我们可以看到b^w是黄色边框宽度的1.8倍，高度是黄色边框高度的1.6倍。

预测之后，我们可以看到预测框与开始标记的实际边界框相交多少。基本上，我们试图最大化它们之间的交集，所以理想情况下，预测的边界框与标记的边界框完全相交。

在原则上就是这样！你提供更加专业的使用边界框（b^x，b^y，b^h，b^w）标注的训练数据，分割图像，并分配给包含中心的框（负责检测对象），训练卷积滑动窗口神经网络，并且预测出对象和位置。

Two More Problems（还有两个问题）

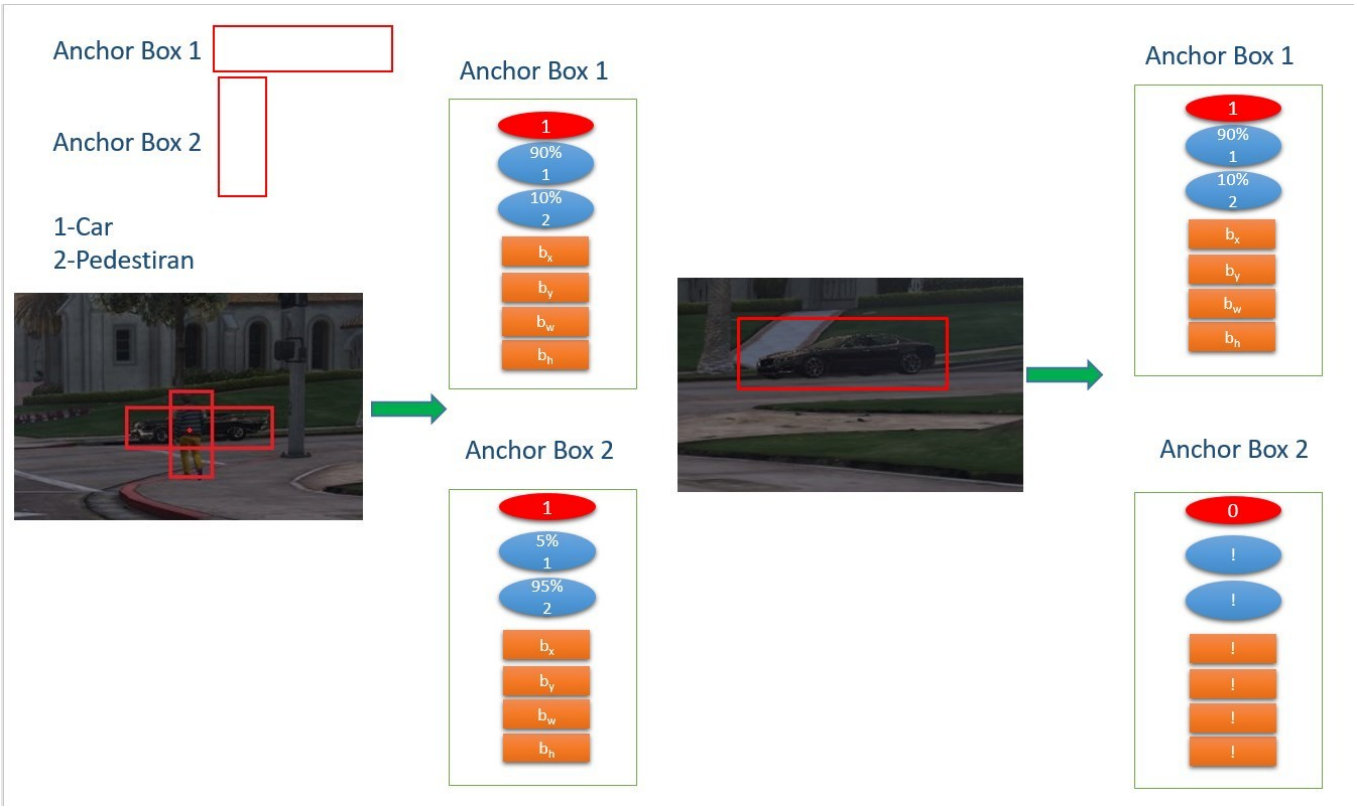
即使我们试图在本篇文章提供更加详细的解释，但是在事实中，我们还有两个小问题需要去解决。



首先，即使在训练时间内，对象被分配到一个盒子（包含对象中心的盒子），在测试时间（预测时），几个盒子（黄色）可能认为它们具有对象的中心（带有红色 ），因此为同一个对象定义自己的边界框。 这是用非最大抑制算法解决的。 目前，Deeplearning4j没有提供实现，所以请在GitHub上找到一个简单的实现（removeObjectsIntersectingWithMax）。 它的作用是首先选择最大Pc概率的盒子作为预测（它不仅具有1或0值，而是可以在0-1范围内）。 然后，删除每个与该框相交超过特定阈值的框。 它再次启动相同的逻辑，直到没有剩下的边界框。

其次，由于我们预测了多个物体（汽车，行人，交通信号灯等），因此可能发生两个或更多物体的中心是一个箱体。 这些情况通过引入锚箱来解决。 使用锚箱，我们选择几个边界框的形

状，我们发现更多用于我们想要检测的对象。 YOLO V2文件通过k-means算法实现，但也可以手动完成。 之后，我们修改输出以包含我们之前看到的相同结构（Pc，bx，by，bh，bw，C1，C2 ...），但对于每个选定的锚箱形状。 所以，我们现在可能有这样的事情：



Application（应用）

训练深度网络需要付出很多努力，并需要有意义的数据和处理能力。 正如我们在之前的文章中所做的那样，我们将使用转移学习。 这一次，我们不打算修改架构并用不同的数据训练，而是直接使用网络。

我们打算使用Tiny YOLO; 以下问题引用于网上：

Tiny YOLO基于Darknet参考网络，速度比普通的YOLO模型快得多，但是相比普通的模型准确率有所降低。 要使用VOC训练的版本：

```
wget https://pjreddie.com/media/files/tiny-yolo-voc.weights
./darknet detector test cfg/voc.data cfg/tiny-yolo-voc.cfg tiny-yolo-voc.we
ights data/dog.jpg
```

好吧，这不是完美的，但兄弟，它确实很快。 在GPU上，它运行速度大于200 FPS。

Deeplearning4j 0.9.1的当前发行版本不提供TinyYOLO，但0.9.2-SNAPSHOT提供。 所以首先，我们需要告诉Maven在哪里加载SNAPSHOT版本：

```
<repositories>
    <repository>
        <id>a</id>
        <url>http://repo1.maven.org/maven2/</url>
    </repository>
    <repository>
        <id>snapshots-repo</id>
        <url>https://oss.sonatype.org/content/repositories/snapshots</url>
        <releases>
            <enabled>false</enabled>
        </releases>
        <snapshots>
            <enabled>true</enabled>
            <updatePolicy>daily</updatePolicy>
        </snapshots>
    </repository>
</repositories>
<dependencies>
    <dependency>
        <groupId>org.deeplearning4j</groupId>
        <artifactId>deeplearning4j-core</artifactId>
        <version>${deeplearning4j}</version>
    </dependency>
    <dependency>
        <groupId>org.nd4j</groupId>
        <artifactId>nd4j-native-platform</artifactId>
        <version>${deeplearning4j}</version>
    </dependency>
```

接下来我们要用非常简短的代码来加载模型：

```
private TinyYoloPrediction() {
    try {
        preTrained = (ComputationGraph) new TinyYOLO().initPretrained();
```

```
            prepareLabels();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
```

`prepareLabels()` 只是使用PASCAL VOC数据集中用于训练模型的标签. 运行 `preTrained.summary()` 去查看模型架构的时候不要感到有任何压力。

视频的每一帧是`CarVideoDetection`使用`JavaCV`进行捕获的：

```
FFmpegFrameGrabber grabber;
grabber = new FFmpegFrameGrabber(f);
grabber.start();
while (!stop) {
    videoFrame[0] = grabber.grab();
    if (videoFrame[0] == null) {
        stop();
        break;
    }
    v[0] = new OpenCVFrameConverter.ToMat().convert(videoFrame[0]);
    if (v[0] == null) {
        continue;
    }
    if (winname == null) {
        winname = AUTONOMOUS_DRIVING_RAMOK_TECH + ThreadLocalRandom.cur
rent().nextInt();
    }
    if (thread == null) {
        thread = new Thread(() -> {
            while (videoFrame[0] != null && !stop) {
                try {

TinyYoloPrediction.getINSTANCE().markWithBoundingBox(v[0], videoFrame[0
].imageWidth, videoFrame[0].imageHeight, true, winname);
                } catch (java.lang.Exception e) {
                    throw new RuntimeException(e);
                }
            }
        });
        thread.start();
    }
    TinyYoloPrediction.getINSTANCE().markWithBoundingBox(v[0], videoFra
```

```
me[0].imageWidth, videoFrame[0].imageHeight, false, winname);
30.     imshow(winname, v[0]);
```

因此，代码正在从视频获取帧并传递到TinyYOLO预先训练好的模型。 从那里，图像帧首先被缩放到416 X 416 X 3（RGB），然后传给TinyYOLO用于预测和标记边界框：

```
1.  public void markWithBoundingBox(Mat file, int imageWidth, int
    imageHeight, boolean newBoundingBOx,String winName) throws Exception {
2.      int width = 416;
3.      int height = 416;
4.      int gridWidth = 13;
5.      int gridHeight = 13;
6.      double detectionThreshold = 0.5;
7.      Yolo2OutputLayer outputLayer = (Yolo2OutputLayer) preTrained.getOut
    putLayer(0);
8.
9.          INDArray indArray = prepareImage(file, width, height);
10.         INDArray results = preTrained.outputSingle(indArray);
11.         predictedObjects = outputLayer.getPredictedObjects(results, det
    ectionThreshold);
12.         System.out.println("results = " + predictedObjects);
13.         markWithBoundingBox(file, gridWidth, gridHeight, imageWidth, im
    ageHeight);
14.     imshow(winName, file);
15. }
```

预测之后，我们应该已经拥有了边界框尺寸的预测值。我们已经实现了非最大抑制算法（removeObjectsIntersectingWithMax）因为，正如我们所提到的，YOLO算法在测试时，在预测每一个对象的时候回拥有不只一个边界框。相比（b^x，b^y，b^h，b^w），我们将会使用`topLeft`和`bottomRight`左上点和右下点。`gridWidth`和`gridHeight`使我们打算将图片分割成为多少个更小的边界框，在我们的案例中被分割为`13x13`。w和h为原始图像的尺寸。

在此之后，除了播放视频的线程之外我们将会启用另外一个线程，我们更新视频以获取检测到的对象的矩形和标签。

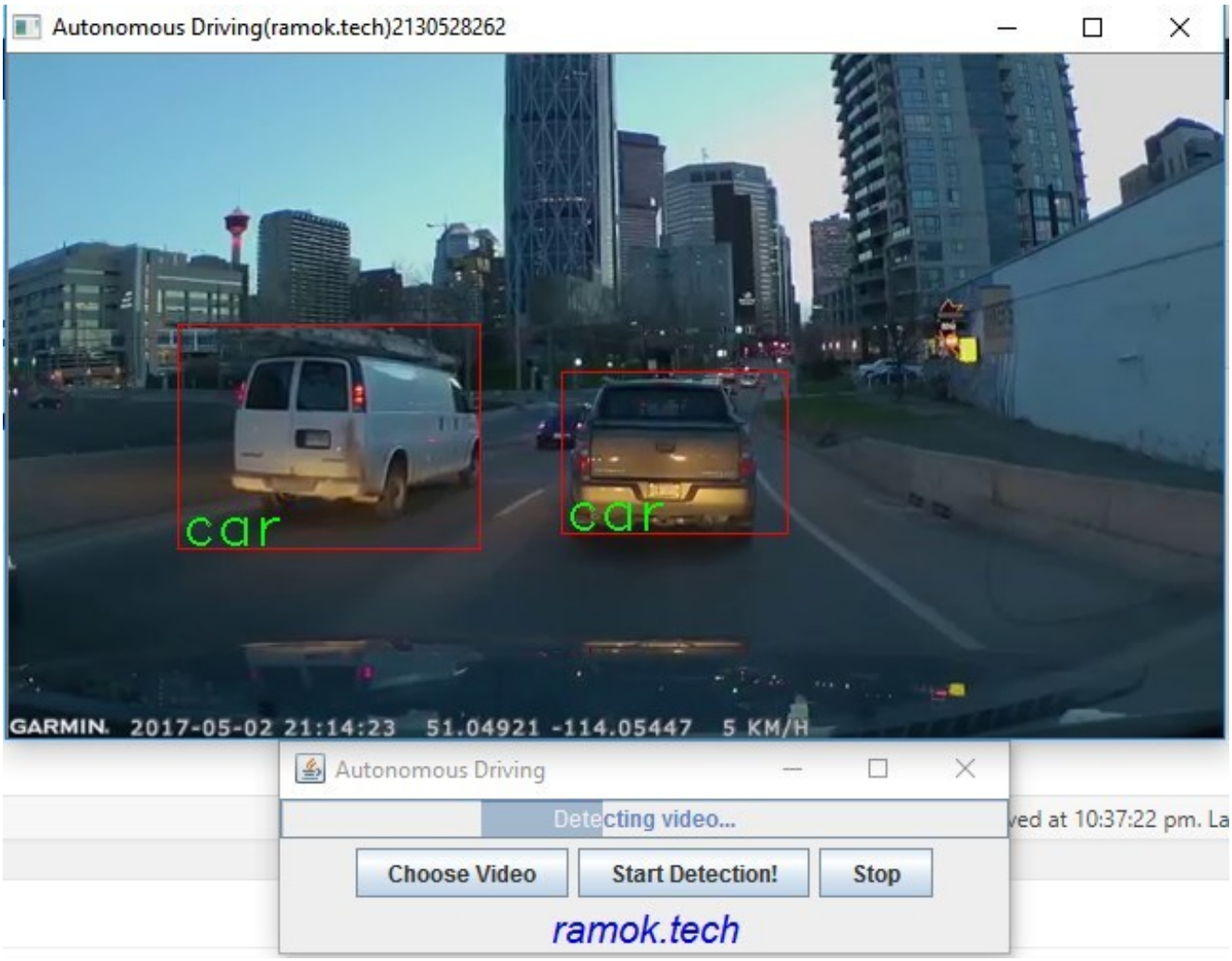
考虑我们运行在CPU上时，（实时）预测的速度非常的快；如果运行在GPU上，我们会获得更好的实时检测效果。

Running Application（运行应用）

即使你的电脑上没有安装Java，这个应用可以在没有任何Java背景知识的条件下下载和运行。你可以尝试使用自己的视频。

在源代码中执行`RUN`类即可运行。如果你不想使用IDE来打开这个工程，你可以运行`mvn clean install exec:java`命令。

在运行这个应用之后，你将会看到如下图所示的结果：



Enjoy!

---

github代码地址： https://github.com/klevis/AutonomousDriving
gitee开源中国地址： https://gitee.com/sjsdfg/AutonomousDriving

更多文档可以查看 https://github.com/sjsdfg/deeplearning4j-issues。
你的star是我持续分享的动力