Week 4

本文档基于上海交通大学 交龙战队2022年视觉培训博客编写。

传送门：https://sjtu-robomaster-team.github.io/vision-learning-1/

概念：

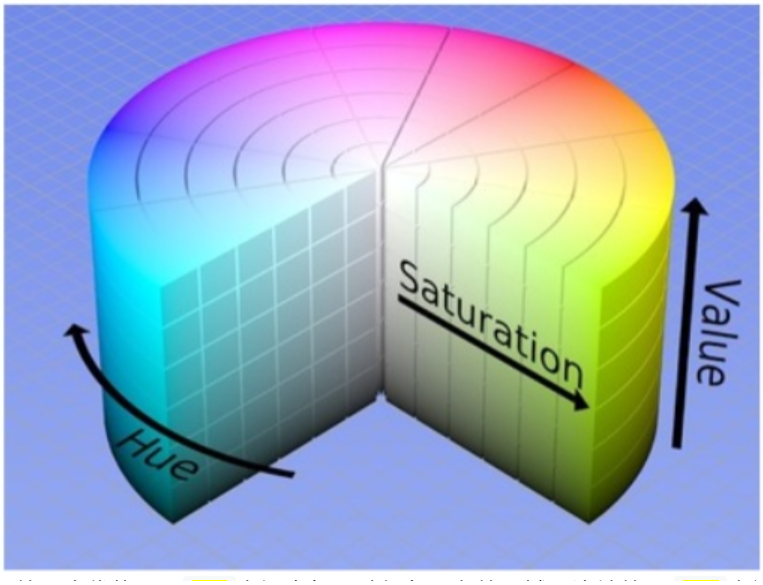
目标检测 - 传统视觉：

图像的构成与三原色

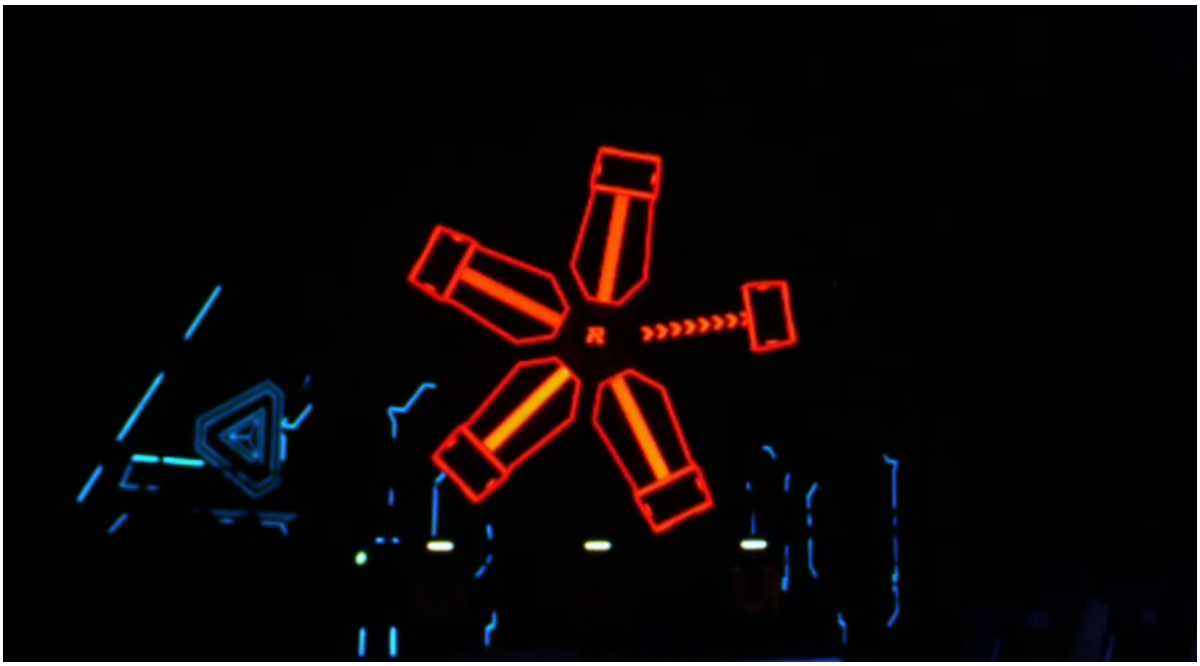
图像的组成与颜色空间：RGB与HSV

`RGB`空间通过图像三原色通道值

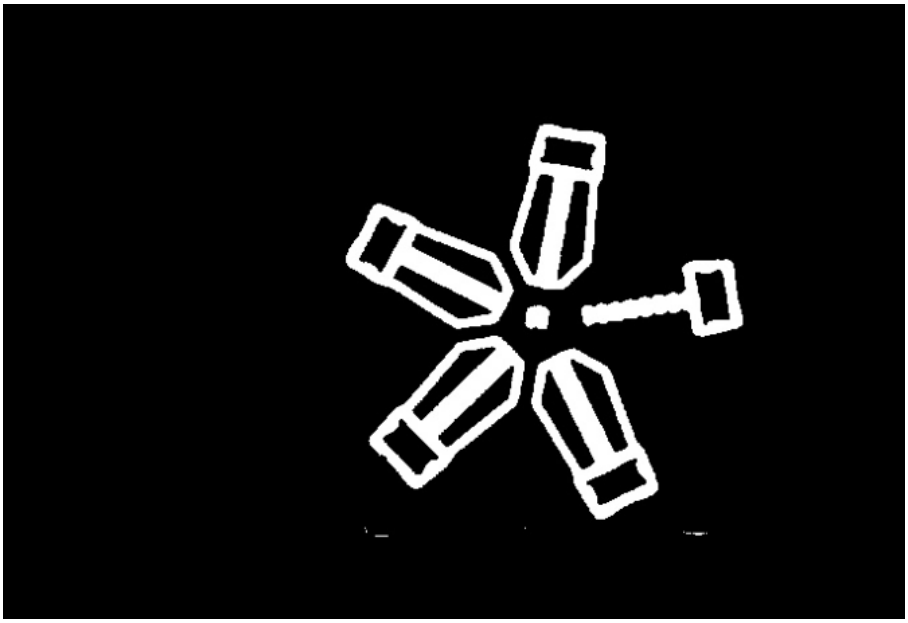
`HSV`空间通过色相(H)，饱和度(S)，和亮度(V)表示空间内的每一个颜色。下图为`HSV`空间的示意图：



基于轮廓的传统视觉的一般流程



1. 二值化



设定阈值法

通过设定一个阈值以实现二值化是最简单的二值化方法。 对于一张图片，他的每一个像素都处在区间$[0, 255]$之中。我们只需要设定一个中间值$threshold$，将所有灰度大于$threshold$的像素点转换为白色，所有灰度小于$threshold$的像素点转换为黑色。这样一个最简单的二值化就完成了。

在`OpenCV`中提供了函数

`double cv::threshold(InputArray src, OutputArray dst, double thresh, double maxval, int type)`

实现图像的二值化。 下面讲解这个函数中的参数作用：

```
src:    输入
dst:    输出
thres:  设定的二值化阈值
maxval:使用 THRESH_BINARY 或 THRESH_BINARY_INV 进行二值化时使用的最大值
type:   二值化算法类型
```

二值化算法主要分为以下几种：

```
THRESH_BINARY:      将小于 thres 的值变为 0 ，大于 thres 的值变为 255
THRESH_BINARY_INV:  将小于 thres 的值变为 255, 大于 thres 的值变为 0
THRESH_TRUNC:       将大于 thres 的值截取为 thres, 小于 thres 的值不变
THRESH_TOZERO:      将小于 thres 的值变为 0 , 大于 thres 的值不变
THRESH_TOZERO_INV:  将大于 thres 的值变为 0 , 小于 thres 的值不变
```

通道相减法

对于赛场上的装甲板与能量机关，另一种常见的方法是红蓝通道相减。C++中的代码实现如下：

```
int main(int argc, char ** argv)
{
    cv::Mat src = cv::imread("/home/nvidia/Downloads/energy.jpg");
    assert(src.channels() == 3);       // 检测是否为三通道彩色图片
```

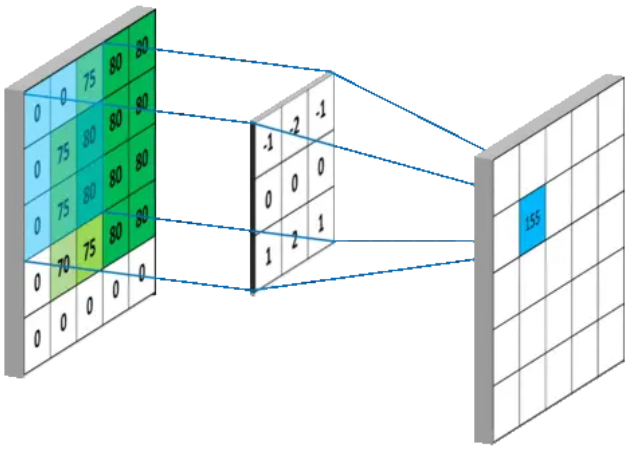
```
    cv::Mat channels[3];
    cv::split(src, channels);          // 三通道分离

    cv::Mat red_sub_blue = channels[2] - channels[0];     // 红蓝通道相减

    ......

    return 0;
}
```

值得注意的是，红蓝通道相减通常会出现二值图的中空现象。例如：

因此，我们上赛季的代码中使用**设定阈值**的方法进行二值化。

2. 滤波（模糊）

领域运算



均值滤波，中值滤波，高斯滤波

0	0	75	80	80
0	75	80	80	80
0	75	80	80	80
0	70	75	80	80
0	0	0	0	0

*

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

=

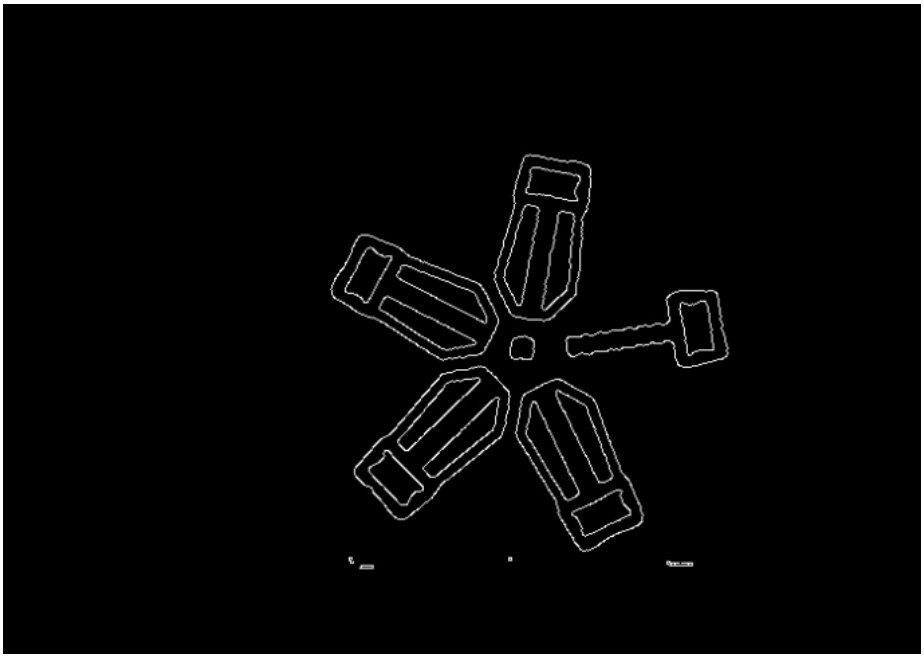
	43			

形态学运算

- 腐蚀：取邻域的最小值
- 膨胀：取邻域的最大值
- 开运算：先腐蚀再膨胀
- 闭运算：先膨胀再腐蚀

轮廓提取

我们使用`opencv`中的`findContours()`函数进行轮廓提取。进行轮廓提取后的大致效果如下图



`findContours()`函数的参数如下

```
void cv::findContours(InputArray image, OutputArrayOfArrays contours, OutputArray
hierarchy, int mode, int method, Point offset = Point())
```

`image`为需要进行轮廓提取的图像，`contours`为提取到的轮廓序列，`hierarchy`中记录了轮廓间的拓扑结构，`mode`指示提取出的轮廓的储存方法，`method`指示使用的轮廓提取方法。

其中，`mode`分为一下几类：

```
RETR_EXTERNAL:      只列举外轮廓
RETR_LIST:          用列表的方式列举所有轮廓
RETR_TREE:          用树状的结构表示所有的轮廓，在这种模式下会在hierachy中记录轮廓的父子关系
```

需要注意的是，`hierachy`的记录格式如下：**对于每一个轮廓，hierarchy都包含4个整型数据，分别表示：后一个轮廓的序号、前一个轮廓的序号、子轮廓的序号、父轮廓的序号。**

`method`分为以下几类：

```
CHAIN_APPROX_NONE:      绝对的记录轮廓上的所有点
CHAIN_APPROX_SIMPLE:    记录轮廓在上下左右四个方向上的末端点（轮廓中的关键节点）
```

`findContours()` 的返回值为一个类型为 `std::vector<std::vector<cv::Point>>` 的对象。可以理解为每个轮廓是一个由 `cv::Point` 构成的 `vector`，而轮廓是这些 `vector` 的集合。

轮廓筛选

轮廓筛选最基本的思想就是用轮廓自身的几何性质以及轮廓间的几何关系，实现对目标轮廓的约束，排除不感兴趣的轮廓。

在这一步骤中，我们有几个常用的函数：

- `double cv::contourArea(InputArray contour, bool oriented = false)`
  - 这个函数可以用来求出一个轮廓的大小
  - 第一个参数为输入的轮廓
  - 另外，若第二个参数为 `true`，则函数会返回一个带有符号的浮点数，符号基于轮廓的方向
  - 如果第二个参数为 `false`，则函数会返回轮廓面积的绝对值。
- `double cv::arcLength(InputArray curve, bool closed)`
  - 这个函数可以用来求出一个轮廓的周长
  - 第一个参数为输入的轮廓
  - 第二个参数为轮廓是否是封闭的
  - 返回轮廓的周长
- `Rect cv::boundingRect(InputArray array)`
  - 这个函数输入一个轮廓，返回最小的包含轮廓的正向外接矩形（不带有旋转）
- `RotatedRect cv::minAreaRect(InputArray points)`
  - 这个函数输入一个轮廓，返回轮廓的最小外接矩形（带有旋转）
- `void cv::convexHull(InputArray points, OutputArray hull, bool clockwise=false, bool returnPoints=true)`
  - 此函数被用来求解轮廓的凸包
  - 第一个参数为输入的轮廓，第二个参数为输出的凸包
  - 第三个参数如果为 `true`，则返回顺时针的轮廓，如果为 `false`，则返回逆时针。
  - 第四个参数如果为 `true`，则用点表示凸包，如果为 `false`，则用点的索引表示凸包，在 `hull` 的类型为 `vector` 的情况下，第四个参数失效，依靠 `vector` 的类型决定。

列举几个常用轮廓筛选的手段：

- 面积/周长大小约束

面积/周长大小约束是最简单的约束之一，即通过轮廓所包含区域的大小或是轮廓的周长大小筛选指定的轮廓。 这种方法虽然简单粗暴，但对于一些环境干扰小的简单环境往往能够取得相当不错的效果。

- 轮廓凹凸性约束

这种方法能通过轮廓的凹凸性对凹轮廓或凸轮廓进行有针对性的筛选。 一般来说可以通过将**轮廓的凸包**与**轮廓本身**进行比较来实现。

- 与矩形相似性约束

在轮廓筛选时常常会需要筛选一些较规则的形状，如矩形轮廓等。 在这种情况下，一般来说我们可以通过将轮廓的**最小外接矩形**与**轮廓本身**进行比较来实现筛选。

- 拓扑关系约束

例如必须是外轮廓啊，必须有父轮廓啊，这里就不细说了。

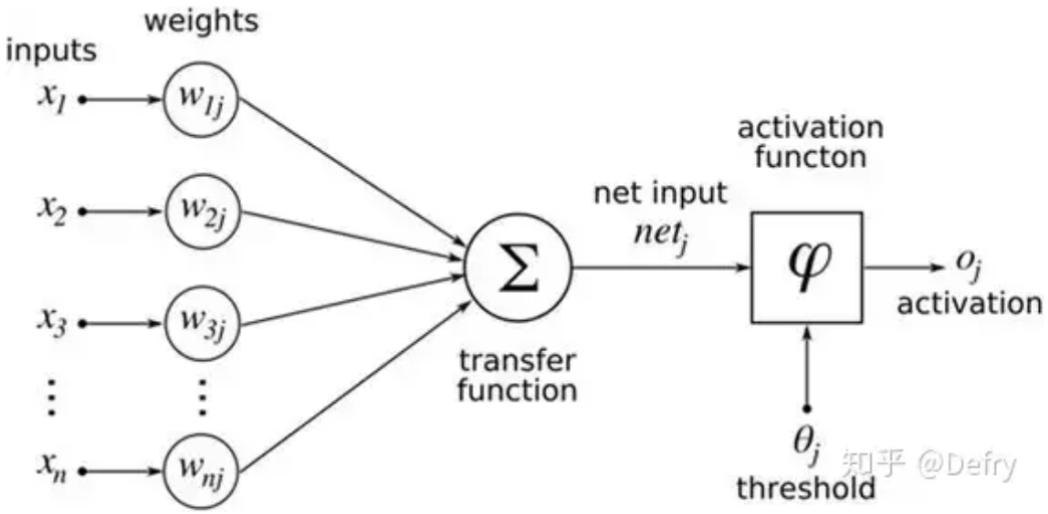
目标检测：深度学习

基础知识：多层感知器（MLP，人工神经网络）

神经元

在讨论多层感知器前，我们先聊聊构成多层感知器的基本单位：神经元。**多层感知器**（英语：Multilayer Perceptron，缩写：**MLP**）是一种前向结构的人工神经网络，映射一

神经做的事情很简单。假设我们有一个向量$[a,b,c]$，神经元将这个向量乘上另一个由反向传播 确定的**权重向量**$[w1,w2,w3,w4]$得到 $output = w_1a + w_2b + w_3c + w_4$ ，再将$output$输出给下一个神经元。示意图如下：



对于任意一个线性函数，我们都可以通过对神经元的排列组合来计算/模拟。

然而，由于神经元之间的计算均为线性，即便有再多的神经元也无法计算一个最简单的非线性函数（如`f(x) = x if x > 0 else 0`）。为了引入非线性因素，我们在每个神经元计算完成后，使用**激活函数（activation function）**对输出再次进行处理。

常用的激活函数有以下三种：

- ReLU：

$$f(x) = max(0, x)$$

- Sigmoid:

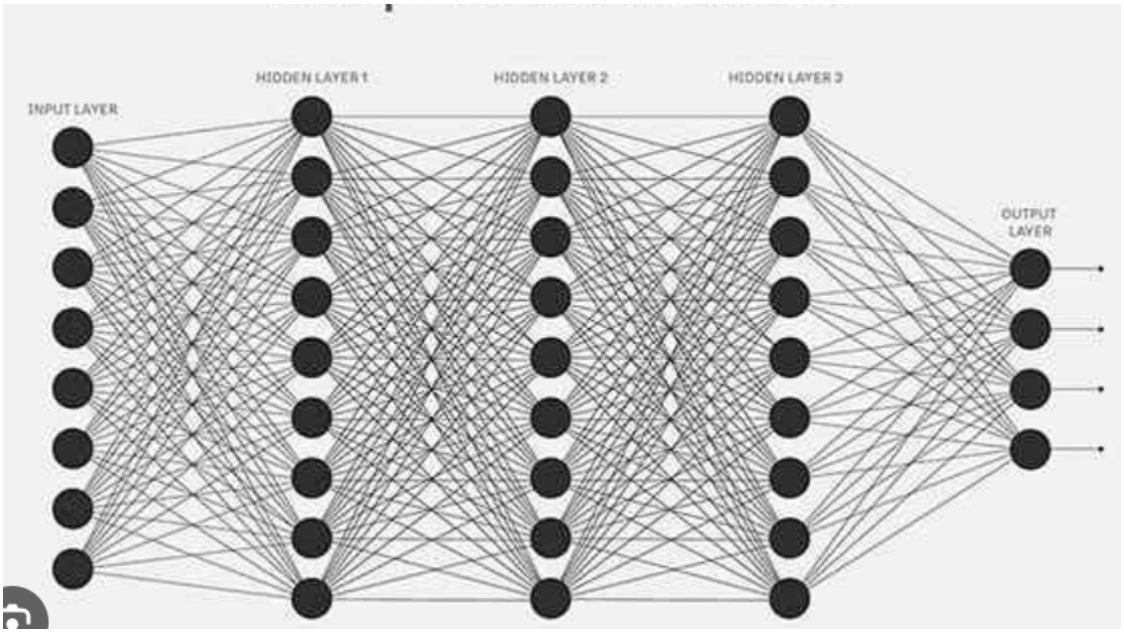
$$f(x) = \frac{1}{1 + e^{-x}}$$

- Tanh:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

人工神经网络

当许多神经元组合在一起时，就形成了神经网络（MLP）。



以上都没看懂？没关系，你可以只将多层感知器理解为一个万能的**函数拟合器**。通过标签，损失函数与反向传播，两层mlp理论上可以拟合**世界上任何一个函数**！

YOLO（You Only Look Once）

前置知识：特征提取

由于mlp中全连接层恐怖的计算量，我们不可能将一张几百万像素图像直接放进mlp中进行运算。为了降低运算量，我们一般会使用一些特征提取的算法对图像进行处理、降维，来降低输入mlp中的像素的数量。YOLO使用卷积神经网络（CNN）进行特征提取，感兴趣的同学可以课后自行了解。

简介

YOLO是一种单阶段、快速的深度学习目标检测模型。在原作者的论文中，它大致分为以下步骤

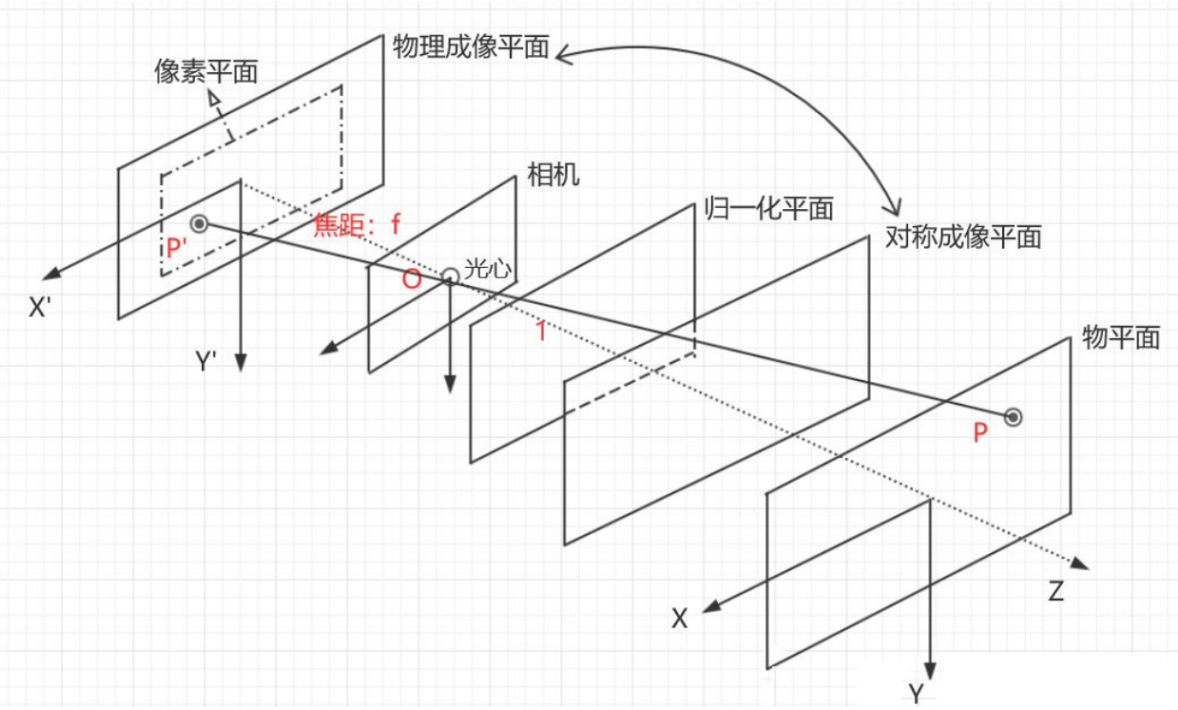
1. 使用CNN对图像进行处理，将原图像转化为7 * 7 * 1024的特征图
2. 将此特征图放入mlp，运算得到7 * 7 * 30的结果向量。
3. 每个30维的向量包含2组物体的信息，包括中心点位置，宽高，类别概率。

因此，你可以理解为：YOLO将图像特征提取后，用一个1024 - 30的mlp拟合了 特征向量-物体信息 的 函数。

注：实际上，YOLO将7 * 7 * 1024的特征图拉开为了一个50176维的向量，放入两层感知器后得到一个1470维的结果向量，再拆成了7 * 7 * 30的矩阵。

目标定位：相机成像原理与pnp算法

相机成像原理



(图中的坐标系仅供参考，并不是相机坐标系)

工业相机基础知识

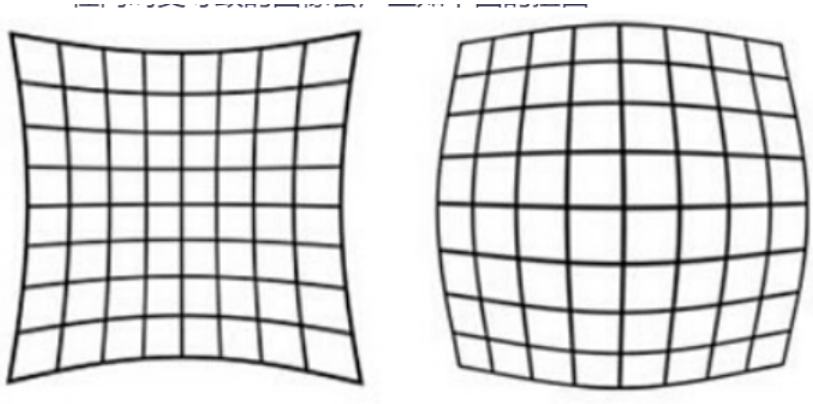
1. 光心：凸透镜的中心
2. 焦距：**成像平面**到**光心**的距离
3. 光圈：控制光线进入的通路的大小
4. 曝光时间：快门速度，控制每一张图片进光的时间
5. 增益：模拟信号的放大系数。增益越大，图象越亮，但噪声也会越大。
6. 内参与外参
    - 外参：相机在世界坐标系下的位置
    - 内参：相机的焦距、畸变参数等

坐标系简介

- 图像坐标系（又叫像素坐标系）
- 相机坐标系
- 陀螺仪坐标系（有时我们会叫成机器人坐标系）
- 世界坐标系

相机标定

由于工业相机使用凸透镜成像，图像会不可避免地出现畸变现象。例如：



为了解决这个问题，我们采用**相机标定**的方法确定相机的**内参**，

标定方法

一般来说，我们使用标定板辅助标定。

opencv提供了相机标定的api

```
double cv::calibrateCamera(InputArrayOfArrays objectPoints,
                    InputArrayOfArrays imagePoints,
                    Size imageSize,
                    InputOutputArray cameraMatrix,
                    InputOutputArray distCoeffs,
                    OutputArrayOfArrays rvecs,
                    OutputArrayOfArrays tvecs,
                    int flags = 0,
                    TermCriteria criteria =
TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, DBL_EPSILON) )
```

- objectPoints：棋盘格上的角点对应的世界坐标系中的位置
- imagePoints：棋盘格上找到的角点
- imageSize：图片的大小
- cameraMatrix：输出的相机内参矩阵
- distCoeffs：输出的相机畸变矩阵
- rvecs：相机坐标系与世界坐标系的旋转向量
- tvecs：相机坐标系与世界坐标系的平移向量

pnp解算

透视变换

参考"相机成像"部分的示意图。相机成像的过程就是典型的透视变换。

PNP测距

- 原理

  相关教程 论文
- opencv实现：

```
bool cv::solvePnP(  InputArray objectPoints,
                    InputArray imagePoints,
                    InputArray cameraMatrix,
                    InputArray distCoeffs,
                    OutputArray rvec,
                    OutputArray tvec,
                    bool useExtrinsicGeuss = false,
                    int flages = cv::SOLVEPNP_ITERATIVE,
)
```

其中

- objectPoints为世界坐标系中的点
- imagePoints为像素坐标系中的点
- cameraMatrix为相机内参
- disCoeffs为相机畸变矩阵
- rvec为求出来的旋转向量
- tvec为求出来的平移向量

在我们的代码中，“世界坐标系”被定义为目标装甲板的坐标系。因此，rvec与tvec实际上是相机相对与目标装甲板的变换向量。

在相机相对与陀螺仪坐标系的位姿已知的情况，我们可以解出目标装甲板相对陀螺仪坐标系的位姿。

目标跟踪：卡尔曼滤波

卡尔曼滤波

在比赛中的自瞄系统中，如果想要准确预测地方车辆的运动轨迹，就必须要精确地求出对方车辆的速度。而卡尔曼滤波可以高效、较准确，实时地根据一系列带有时间戳的坐标点推算出敌方车辆的速度。

为了方便研究，我们可以先把这个二维空间内的速度求解问题转换到一维。 假设我们有一系列处在函数$f(x) = kx + b$上的一系列离散点，我们可以通过哪些方法通过这些离散点推算出$f(x)$的斜率k。

对于这一问题，最简单的方法是通过差分的方式求解速度。即通过公式：$v = \frac{x2 - x1}{t2 - t1}$这是最朴素的思路，但是问题在于：

1. 受噪声干扰大
2. 求出的速度与实际速度有一定延迟（由于求解的是平均速度，因此这一方法求出的速度事实上是时刻$\frac{t2 + t1}{2}$的瞬时速度）
3. 输出的速度不连续，在不断跳跃。

因此，我们需要更优秀的算法求出运动物体的速度，他要有下面的特点：

1、实时计算速度

2、输出结果的波动较小

3、抗噪声能力强

直观地理解卡尔曼滤波

下面举一个常用的卡尔曼滤波的例子： 假设有一辆自动驾驶的小车想要通过一个隧道，他想要知道自己每一时刻在隧道中的位置，但是隧道中没有了GPS，因此你没有可以直接获得位置信息的手段。这种情况下，应该如何得到小车在隧道中的问题。

在试图解决问题之前，先看看我们有哪些可以用来计算小车位置的数据： 1、轮胎编码器 2、惯性测量单元 3、供给小车前进的燃料稳定（可以理想为小车在做匀速直线运动）

把这些信息进行分类，我们大致有两种信息：

1. 测量量
    - 编码器
    - 惯性测量单元
2. 预测量
    - 匀速直线模型

这里引入一点简单的数学：

1. 通过**预测量**，我们可以得到如下的方程 $x_2 = x_1 + v_1\Delta t \ v_2 = v_1$
2. 通过**观测量**，我们可以得到如下的方程 $x_2 = x_1 + \Delta x \ v_2 = v_1 + a\Delta t$
   $x_2 = v_1\Delta t + \frac{1}{2}at^2$

其中x_1, v_1为上一时刻小车的位置和速度，x_2, v_2为这一时刻小车的位置和速度。 Δt为前后两时刻之间的时间差，Δx为编码器得到的汽车位移，a为惯性测量单元得到的小车加速度。

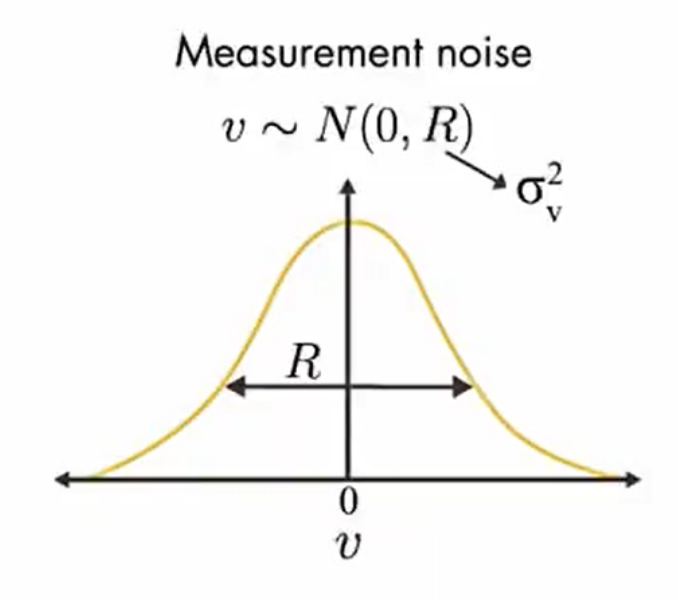
通过这些变量，公式，以及得到的传感器数据，观察上面的5个算式，我们可以用3种不同的算法得到小车的位置x，2种不同的方法得到小车的速度。

我们的目标是求出小车的位置，因此我们来看一下位置的三种求解方法： 第一个**预测的方法**利用牛顿运动定律约束前后之前位置的关系 后两个**观测的方法**利用传感器的数据间接推算小车位置 显然，这三个计算方法都不可避免的**存在误差。**

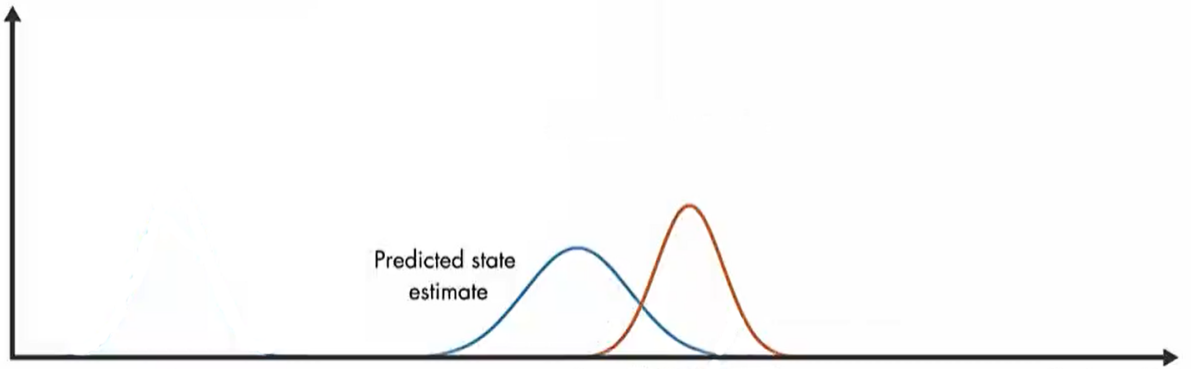
在我校的实验课程中，你应该了解过我们在测量和计算时会受到两种误差的干扰：

- **随机误差** 由于在测定过程中一系列有关因素微小的随机波动而形成的具有相互抵偿性的误差。 一般来说呈现正态分布
- **系统误差** 一种非随机性误差。 如违反随机原则的偏向性误差，在抽样中由登记记录造成的误差等。他会使得数据产生像一个方向的偏移

这里我们不妨假设我们的计算模型绝对正确，对数据的使用过程中也没有出现原理上的问题，也就是说**不存在系统误差。** 那么再考虑上述的三个计算结果，由于每个结果都存在**随机误差**，因此他们的实际分布应该是一个概率上的**正态分布。**

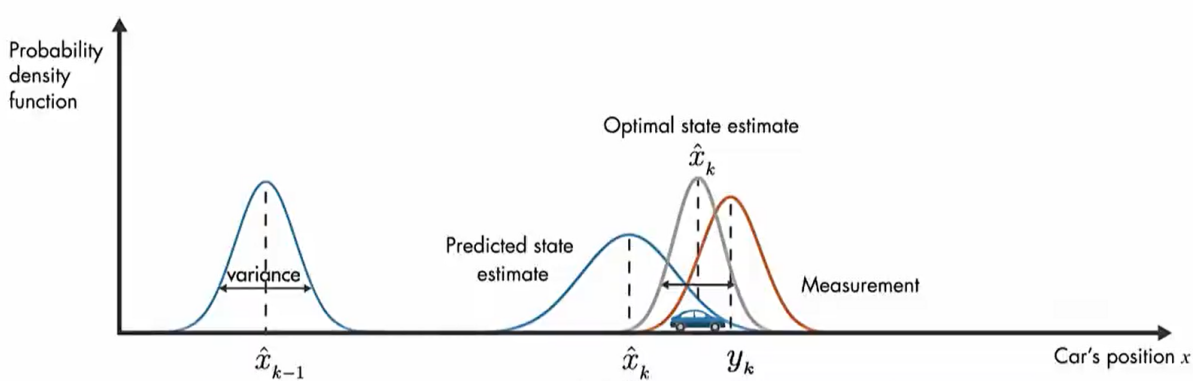


如果呈现在坐标轴上，他们应该大致分布成这样：



可以看到，这会是多个独立的正态分布曲线，每个曲线代表一种方式计算出的小车的位置的分布区间。

而卡尔曼滤波所做的，就是推算出一个合适的每个计算结果的权重，并以此预测物体的未来状态



如图，他通过预测量和观测量一同推算出小车最可能出现在的最终位置。

简单来讲，卡尔曼滤波器就是根据上一时刻的状态，预测当前时刻的状态，将预测的状态与当前时刻的测量值进行加权，加权后的结果才认为是当前的实际状态，而不是仅仅听信当前的测量值。

更详细的教程可以参考https://zhuanlan.zhihu.com/p/45238681

Git

关于Git的基础知识，参考菜鸟教程

环境配置

- Ubuntu20.04系统安装
- apt包管理器

- vcpkg包管理器
- cmake
- conda虚拟环境
- openvino&相机驱动