实验六：回归应用实例——目标跟踪

本次实验旨在让同学们理解并掌握回归在目标跟踪中的应用，学习计算机视觉常用库OpenCV的基本使用。

1. 目标跟踪

目标跟踪是计算机视觉领域中的一个重要任务，它旨在在视频序列中准确的跟踪特定的目标。

1.1 目标跟踪的方法

目标跟踪的方法按照模式划分可以分为2类：

- **生成式模型**： 生成式模型的核心思想是通过建立目标模型或者提取目标特征尝试捕捉目标的外观模型，并在后续帧中搜索与该模型特征最为匹配的区域，逐步迭代实现目标定位。这一过程可以视为模板匹配。
- **判别式模型**： 判别式模型的策略是将目标模型和背景信息同时考虑在内, 通过区分目标模型和背景信息, 选择置信度最高的候选样本作为预测结果，将目标提取出来, 从而得到当前帧中的目标位置。 这类方法一般是在追踪过程中训练一个目标检测器，使用目标检测器去检测下一帧预测位置是否是目标，然后再使用新检测结果去更新训练集进而更新目标检测器。 判别式方法已经成为目标跟踪中的主流方法，因为有大量的机器学习方法可以利用。

1.2 目标跟踪的基本流程

- **目标初始化**

  在视频序列的起始帧中识别和标定目标，根据应用需求和目标特性选择合适的跟踪算法。

- **特征提取**

  从目标区域提取适合的特征（如颜色、纹理、形状等），用于后续的跟踪过程。鉴别性的特征表示是目标跟踪的关键之一。

- **预测与跟踪**

  在每个后续帧中确定目标的位置。并根据目标的外观变化，适时更新调整预测模型，防止跟踪过程发生漂移。

- **跟踪维护**

  处理目标跟踪失效，检测并处理跟踪过程中可能发生的目标被遮挡的情况。

2. 回归方法实现简单的目标跟踪

2.1 OpenCV库简介

OpenCV是一个基于BSD许可（开源）发行的跨平台计算机视觉库，有很强大的图片、视频处理功能，可实现图像处理和计算机视觉方面的很多通用算法。官方文档

pip进行安装：

In [ ]:
```
pip install opencv-python
```

安装后，在Python代码中通过导入cv2模块来使用OpenCV的功能：

In [ ]:
```
import cv2
```

2.2 算法描述

可以把跟踪问题抽象为一个线性回归模型的求解。设代表目标图像的输入为z，权重w，输出为$f(z) = w^T z$，目标是通过学习权重向量 w，能够最小化样本x_i经模型输出$f(x_i)$和期望值y_i的最小均方差的解:

$$\min_w \sum_i (f(x_i) - y_i)^2 + \lambda \|w\|^2$$

其中：

- $f(x_i) = w^T x_i$ 是模型对于输入特征 x_i 的预测值。
- y_i 是样本的期望输出（通常是高斯函数加权，中心在目标位置，即越靠近目标的区域为正样本的可能性越大）。
- $\lambda \|w\|^2$ 是正则化项，用于惩罚过大的权重值，以提高模型的泛化能力。

这里使用的是**岭回归**，因为它可以防止过拟合，提升模型的泛化能力（不仅训练误差小，测试误差也小）。

于是，权重向量 w 的最优解为：

$$w = (X^T X + \lambda I)^{-1} X^T y$$

其中：

- X 每一行包含了一个训练样本的特征。
- I 是单位矩阵，与 X 的列数相同。

2.3 目标初始化

在初始帧进行目标选取，确定目标区域的位置以及大小（这里通过手动选择标注）：

In [ ]:
```python
# 手动选择目标区域
roi = cv2.selectROI("tracking", frame, False, False) #目标区域的矩形框
cx, cy, w, h = roi #cx为矩形框中最小的x值,cy为矩形框中最小的y值,w为这个矩形框的宽,h为这个矩形框的高
# 计算矩形框的左上角坐标
x = int(cx - w // 2)
y = int(cy - h // 2) #这里的y是坐标值，不是上述公式中目标函数y的值
#提取目标区域图像
sub_image = frame[y:y+h, x:x+w, :]
```

注意，**手动选取区域的位置很重要**！要把目标完整选取在框内，同时框的边界与目标之间留有一点点距离，便于提取目标周围的背景的信息！

2.4 特征提取

提取目标区域的特征。这里使用方向梯度直方图（Histogram of Oriented Gradient）提取目标的纹理特征。HOG特征的核心思想是利用图像局部区域像素点的梯度方向信息来描述这些区域的外观和形状，将一张图像转化为特征向量。HOG特征原理

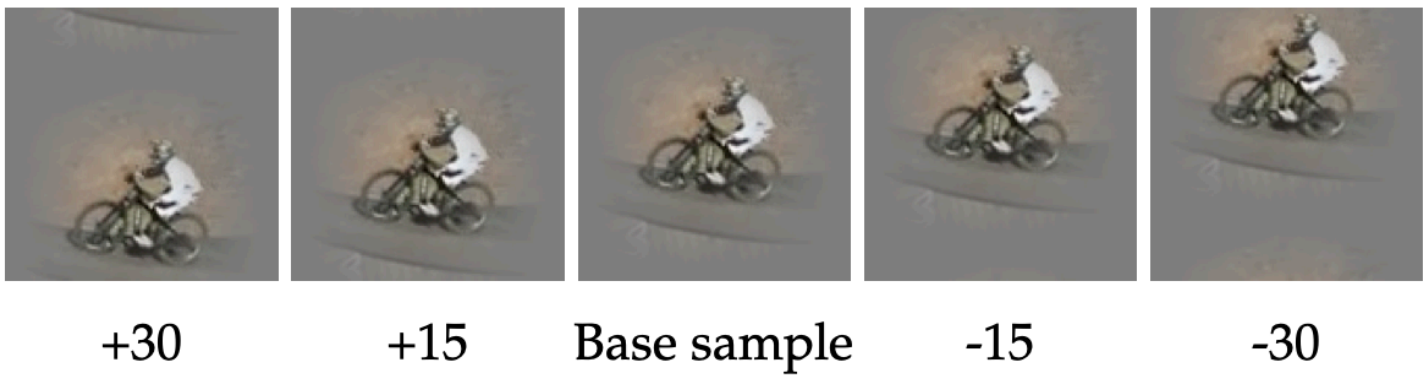
OpenCV中封装了提取HOG特征的类`HOGDescriptor`，实现过程中只需构造一个hog对象来实现：

In [ ]:
```python
import cv2
class HOG():
    def __init__(self, winSize):
        self.winSize = winSize #窗口大小
        self.blockSize = (8, 8) #块大小
        self.blockStride = (4, 4) #块在窗口中滑动的步长
        self.cellSize = (4, 4) #块中的单元（cell）大小
        self.nbins = 9 #方向梯度直方图的柱数
        self.hog = cv2.HOGDescriptor(winSize, self.blockSize, self.blockStride,
                                     self.cellSize, self.nbins)

    def get_feature(self, image):
        winStride = self.winSize
        hist = self.hog.compute(image, winStride, padding = (0, 0))
        w, h = self.winSize
        sw, sh = self.blockStride
        w = w // sw - 1
        h = h // sh - 1
        return hist.reshape(w, h, 36).transpose(2, 1, 0)

hog = HOG(winSize) #winSize为目标区域的大小(pw, ph)
resized_image = cv2.resize(sub_image, (pw, ph)) #将目标区域图像缩放到winSize大小
feature = hog.get_feature(resized_image) #提取目标区域图像的HOG特征
```

2.5 正负样本生成

在训练目标检测器时一般选取目标区域为正样本，目标的周围区域为负样本。为了解决检测器训练时缺少样本的问题，在初始帧图像中通过循环移位采集大量样本，即把目标向上、向下分别移动不同的像素得到新的样本图像，并联构建训练样本X，使得训练出的模型效果更好。

$$X = C(x)$$



根据循环矩阵能够被离散傅里叶矩阵对角化的性质（定理证明），我们可以将训练样本X在傅氏空间内使用离散傅里叶矩阵进行对角化处理：

$$X = C(x) = Fdiag(\hat{x})F^H$$

其中，$\hat{x}$是x的傅里叶变换$\hat{x} = \mathcal{F}(x) = \sqrt{n}Fx$，$F$是离散傅里叶矩阵，上标$H$表示Hermitian转置，即共轭转置。

我们可以使用 `numpy.fft` 中的 `fft2` 将特征x进行傅里叶变换：

In [ ]:
```
from numpy.fft import fft2, fftshift #导入傅里叶变换相关函数
#对特征x进行二维傅里叶变换
x_hat = fft2(x)
```

循环矩阵的计算可以直接把所有的样本都转换为对角矩阵进行处理，因为在循环矩阵对样本进行处理的时候，样本并不是真实存在的样本，存在的只是虚拟的样本，可以直接利用循环矩阵所特有的特性，直接把样本矩阵转换为对角矩阵进行计算，这样可以大幅度加快矩阵之间的计算，因为对角矩阵的运算只需要计算对角线上非零元素的值即可。

2.6 傅氏对角化简化的脊回归

我们可以将w的求解转换到傅里叶空间做点积运算，应用离散傅里叶变换提高运算速度。权重向量 w 的解在复数域中形式可以写作：

$$w = (X^H X + \lambda I)^{-1} X^H y$$

代入X有：

$$\begin{aligned} w &= (F\mathrm{diag}(\hat{x}^*)F^H F diag(\hat{x})F^H + \lambda I)^{-1} X^H y \\ &= (F\mathrm{diag}(\hat{x}^* \odot \hat{x})F^H + \lambda I)^{-1} X^H y \\ &= (F\mathrm{diag}(\hat{x}^* \odot \hat{x})F^H + \lambda F F^H)^{-1} X^H y \\ &= (F(\mathrm{diag}(\hat{x}^* \odot \hat{x}) + \lambda I)F^H)^{-1} X^H y \\ &= (F(\mathrm{diag}(\hat{x}^* \odot \hat{x}) + \mathrm{diag}\lambda)F^H)^{-1} X^H y \\ &= (F\mathrm{diag}(\hat{x}^* \odot \hat{x}) + \lambda)F^H)^{-1} X^H y \\ &= F(\mathrm{diag}(\hat{x}^* \odot \hat{x}) + \lambda)^{-1} F^H X^H y \\ &= F\mathrm{diag}\left(\frac{\hat{x}^*}{\hat{x}^* \odot \hat{x} + \lambda}\right) F^H y \end{aligned}$$

两边同时左乘 F^H 得：

$$F^H w = F^H F \mathrm{diag}\left(\frac{\hat{x}^*}{\hat{x}^* \odot \hat{x} + \lambda}\right) F^H y$$

两边同时取共轭转置得：

$$Fw = \mathrm{diag}\left(\frac{\hat{x}}{\hat{x}^* \odot \hat{x} + \lambda}\right) Fy$$

$$\hat{w} = \mathrm{diag}\left(\frac{\hat{x}^*}{\hat{x}^* \odot \hat{x} + \lambda}\right) \hat{y}$$

最终，w在傅里叶空间的求解可以化简为如下形式。由此可见，引入循环矩阵不仅可以增加样本数量提高跟踪准确率，还能帮助简化求解的计算。

$$\hat{w} = \frac{\hat{x}^* \odot \hat{y}}{\hat{x}^* \odot \hat{x} + \lambda}$$

其中，$\hat{x}^*$是x复数的共轭，我们可以使用 `numpy` 中的 `conj` 计算得到：

In [ ]:
```
from numpy import conj
x_c = conj(x_hat)
```

⊙是哈曼德(Hadamard)点乘，即对形状相同的矩阵进行运算，并产生相同维度的第三个矩阵：

example：

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \odot \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} a_{11}\,b_{11} & a_{12}\,b_{12} & a_{13}\,b_{13} \\ a_{21}\,b_{21} & a_{22}\,b_{22} & a_{23}\,b_{23} \\ a_{31}\,b_{31} & a_{32}\,b_{32} & a_{33}\,b_{33} \end{pmatrix}$$

然后再将解逆变换回真实空间：

$$w = \mathcal{F}^{-1}(\hat{w})$$

可以使用 `numpy.fft` 中的 `ifft2` 将离散傅里叶逆变换转换回真实空间：

In [ ]:
```
from numpy.fft import ifft2
from numpy import real
#对W_hat进行逆傅里叶变换回真实空间，因为傅里叶变换可能产生复数结果，使用real函数取实部
W = real(ifft2(W_hat))
```

2.7 检测阶段

读取下一帧图像，提取搜索区域图像特征，得到待检测样本z，然后使用$f(z)$（响应图）预测目标位置，检测预测值（响应值）越大越有可能是跟踪目标所在区域。可以使用预测值最大的位置作为预测目标的中心位置。

$$f(z) = w^T z = \mathcal{F}^{-1}(\hat{w} \odot \hat{z})$$

2.8 模型更新

对目标的位置进行预测后，根据此位置信息，更新目标区域以及模型权重w，重复上述过程对后续帧目标位置进行预测，直到所有的视频序列检测完成。

3. 算法框架

In [ ]:
```python
if __name__ == '__main__':
    #打开视频文件
    cap = cv2.VideoCapture('car.avi')
    #获取帧率
    fps = cap.get(cv2.CAP_PROP_FPS)
    #设置播放速度
    new_fps = 0.5*fps
    cap.set(cv2.CAP_PROP_FPS,new_fps)
    #读取第一帧
    ok, frame = cap.read()
    if not ok:
        print("error reading video")
        exit(-1)
    roi = cv2.selectROI("tracking", frame, False, False)
    #创建跟踪器对象
    tracker = Tracker()
    #初始化跟踪器
    tracker.init(frame, roi)
    #开始跟踪
    while cap.isOpened():
        ok, frame = cap.read()
        if not ok:
            break
        #更新跟踪器
        x, y, w, h = tracker.update(frame)
        #绘制跟踪结果框
        cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 255), 1)
        cv2.imshow('tracking', frame)
        c = cv2.waitKey(1) & 0xFF
        if c==27 or c==ord('q'):
            break
    cap.release()
    cv2.destroyAllWindows()
```

跟踪器实现：

In [ ]:
```python
import cv2
import numpy as np
from numpy.fft import fft2, ifft2, fftshift
from numpy import conj, real

class Tracker():
    def __init__(self):
        self.max_patch_size = 256
        self.padding = 2.5
        self.lambdar = 0.0001 #岭回归的正则化参数
        self.update_rate = 0.012 #学习率

    def get_feature(self, image, roi):
        """
        处理图像并获取特征
        """
        cx, cy, w, h = roi
        #padding让样本中含有需要学习的背景信息，同时保证样本中目标的完整性
        w = int(w * self.padding) // 2 * 2
        h = int(h * self.padding) // 2 * 2
        x = int(cx - w // 2)
        y = int(cy - h // 2)
        sub_image = image[y:y+h, x:x+w, :]
        resized_image = cv2.resize(sub_image, (self.pw, self.ph))
        #提取HOG特征
        feature = self.hog.get_feature(resized_image)
        fc, fh, fw = feature.shape
        self.scale_h = float(fh) / h
        self.scale_w = float(fw) / w
        #针对提取得到的特征，采用余弦窗进行相乘平滑计算
        #因为移动样本的边缘比较突兀，会干扰训练的结果
        #如果加了余弦窗，图像边缘像素值就都接近0了，循环移位过程中只要目标保持完整那这个样本就是合理的
        hann2t, hann1t = np.ogrid[0:fh, 0:fw]
        hann1t = 0.5 * (1 - np.cos(2*np.pi*hann1t / (fw-1)))
        hann2t = 0.5 * (1 - np.cos(2*np.pi*hann2t / (fh-1)))
        hann2d = hann2t * hann1t
        feature = feature * hann2d

        feature = np.sum(feature,axis=0)
        return feature

    def gaussian_peak(self, w, h):
```

```
        """
        使用高斯函数制作样本标签y
        """
        output_sigma = 0.125
        sigma = np.sqrt(w * h) / self.padding * output_sigma
        syh, sxh = h // 2, w // 2 # 目标框中心点
        y, x = np.mgrid[-syh:-syh+h, -sxh:-sxh+w]
        x = x + (1 - w % 2) / 2.
        y = y + (1 - h % 2) / 2.
        # 生成标签(h,w)，越靠近中心点值越大
        g = 1. / (2. * np.pi * sigma ** 2) * np.exp(-((x**2 + y**2)/(2. * sigma**2)))
        return g

    def train(self, ):
        # TODO: 实现回归求解的代码，返回 W_hat
        raise NotImplementedError("此部分需要同学们自行实现。")


    def detect(self, ):
        # TODO: 实现目标在新帧中检测的代码，返回 f(z)
        raise NotImplementedError("此部分需要同学们自行实现。")

    def init(self, image, roi):
        x1, y1, w, h = roi
        #目标区域的中心坐标
        cx = x1 + w // 2
        cy = y1 + h // 2
        roi = (cx, cy, w, h)

        scale = self.max_patch_size / float(max(w, h))
        self.ph = int(h * scale) // 4 * 4 + 4
        self.pw = int(w * scale) // 4 * 4 + 4
        self.hog = HOG((self.pw, self.ph))
        x = self.get_feature(image, roi)
        y = self.gaussian_peak(x.shape[1], x.shape[0])
        self.wf = self.train(x, y, self.lambdar)#求解权重参数w
        self.x = x
        self.roi = roi

    def update(self, image):
        """
        对给定的图像，重新计算其目标的位置
        """
        cx, cy, w, h = self.roi
        max_response = -1
        # 尝试多个尺度，也就是在目标跟踪的过程中，适应目标大小的变化
        for scale in [0.85, 1.0, 1.02]:
            roi = map(int, (cx, cy, w * scale, h * scale))
            z = self.get_feature(image, roi)
            responses = self.detect(self.wf, z)#检测目标
            height, width = responses.shape
            idx = np.argmax(responses)
            res = np.max(responses)
            #选取检测预测值最大的位置作为目标的新位置
            if res > max_response:
                max_response = res
                dx = int((idx % width - width / 2) / self.scale_w)
                dy = int((idx / width - height / 2) / self.scale_h)
                best_w = int(w * scale)
                best_h = int(h * scale)
                best_z = z
        # 更新,每次训练得到的参数受以往训练得到的参数的影响，有一个加权的过程
        self.roi = (cx + dx, cy + dy, best_w, best_h)
        self.x = self.x * (1 - self.update_rate) + best_z * self.update_rate
        y = self.gaussian_peak(best_z.shape[1], best_z.shape[0])
        new_w = self.train(best_z, y, self.lambdar)
        self.wf = self.wf * (1 - self.update_rate) + new_w * self.update_rate
        cx, cy, w, h = self.roi
        # 返回目标区域的中心坐标和大小
        return (cx - w // 2, cy - h // 2, w, h)
```

4. 算法进阶

上述只是一个非常简单的目标跟踪算法实现，但是对于现实中大部分非线性的模型，我们没办法找到它们的确定形式。 为了捕捉目标在更复杂情况下的运动和变化，提高跟踪的准确性和计算速度，可以通过将低维数据映射到高维，使映射后的样本在高维空间中线性可分。用一个简单的核函数直接做映射就可以很好的完成这个工作，这就是**核岭回归**。

将线性转为非线性：$w=\sum \alpha_i \phi(x_i)$, $\phi(x)$ 把特征映射到了高维空间（核空间）。

$\phi^T(x)\phi(x')$ 为核函数可以替代隐式特征空间的点积，结果存在核矩阵 K 中，类似于核空间变量的求协方差，K是所有训练样本的核相关矩阵：

$$K_{ij}=k(x_i,x_j)$$

需要优化的目标函数变为：

$$\min_{\alpha} ||\phi^T(X)\phi(X)\alpha-y||^2+\lambda||\phi^T(X)\alpha||^2$$

引入核方法，得到非线性回归函数 f：

$$f(z)=w^Tz=\sum_{i=1}^{n}\alpha_i k(z,x_i)$$

核空间的岭回归解为：

$$\alpha=(K+\lambda I)^{-1}y$$

进一步利用循环矩阵傅里叶变换的特性化简：

$$\begin{aligned}\alpha&=(F\mathrm{diag}(\hat{k_{xx}})F^H+\lambda FF^H)^{-1}y\\&=(F\mathrm{diag}(\hat{k_{xx}}+\lambda)F^H)^{-1}y\\&=F\mathrm{diag}\left(\frac{1}{\hat{k_{xx}}+\lambda}\right)F^Hy\\&=\mathcal{C}(\mathcal{F}^{-1}\left(\frac{1}{\hat{k_{xx}}+\lambda}\right)y)\end{aligned}$$

左右两边做傅里叶变换得：

$$\hat{\alpha}=\left(\frac{1}{\hat{k_{xx}}+\lambda}\right)^*\odot\hat{y}$$

最终，求解化简为：

$$\hat{\alpha}=\frac{\hat{y}}{\hat{k_{xx}}+\lambda}$$

每次训练得到的参数受以往训练得到的参数的影响，有一个加权的过程，即

$$\hat{\alpha}_t=(1-\rho)\hat{\alpha}_{t-1}+\rho\hat{\alpha}_t$$

感兴趣的同学可以使用核岭回归实现对第3节的算法进行改进。

5. 动手实践

完成第3节的代码部分，并使用car.avi文件实现对视频中车辆的单目标跟踪。