cv课设中期进展报告

张扬2020212185

孙泽凯2020212180

系统功能描述

我们选择的是实验六：**实时目标跟踪系统**

- **目标跟踪**则是在视频中**连续跟踪**一个或多个目标对象的过程。它通常在目标检测的基础上进行，即首先在视频的初始帧中检测目标，然后在后续帧中跟踪这些目标的位置和状态。
- 目标跟踪需要处理一些额外的挑战，例如目标的移动，遮挡，姿态变化，照明变化，以及相机的移动等。目标跟踪的结果是**视频中每一帧的目标位置**（通常是**一个边界框**）和可能的状态信息。
- 我们的实验六与实验五的区别是：目标检测关注于“在图像中哪里有什么”，而目标跟踪关注于“**目标如何随时间移动**”。

总的来说我们的算法应该实现的目标：

以摄像头数据或视频数据为输入，前期使用鼠标框选一个物体，开始追踪后追踪框不断追踪该物体的位置

算法调研

目标跟踪是一个复杂的问题，历史上已经提出了许多方法来解决这个问题。我们调研了以下这些经典的目标跟踪算法：

1. **MeanShift/CAMShift (Continuously Adaptive Mean Shift)**：这些算法基于颜色直方图模型进行目标跟踪。MeanShift算法通过找寻给定窗口中的颜色直方图的最大密度位置来确定目标位置。CAMShift算法是MeanShift的扩展，它能适应目标大小的变化和旋转。
2. **Optical Flow**：光流是一种描述图像帧之间像素或特征点运动的模型。它可以被用于估计目标的位置和速度。其中一种经典的光流方法是Lucas-Kanade方法。
3. **Kalman Filter/Extended Kalman Filter**：卡尔曼滤波器是一种预测系统状态的方法，可以用于预测目标的位置和速度。对于非线性系统，可以使用扩展卡尔曼滤波器。
4. **Particle Filter**：粒子滤波器是一种基于蒙特卡洛方法的非线性和非高斯滤波方法。它使用一组粒子来表示目标的可能状态，并通过重采样来适应目标的运动。
5. **Tracking-Learning-Detection (TLD)**：TLD是一种复杂的长期跟踪方法，它结合了跟踪，学习和检测三个组件来处理目标的外观变化和临时的遮挡。
6. **Correlation Filters, including Kernelized Correlation Filters (KCF)**：相关滤波器是一种基于模板匹配的方法，它可以快速地在图像中寻找与模板相似的区域。KCF是相关滤波器的一个变种，它使用核技巧来处理非线性问题。
7. **Deep Learning based methods**：近年来，深度学习在目标跟踪中也取得了显著的成果。例如，Siamese networks（如SiamFC和SiamRPN）使用深度神经网络来学习目标的外观模型，以实现高效的实时跟踪。

实验指导上的主要参考文献为 `《High-Speed Tracking with Kernelized Correlation Filters》` 这是第六点中的KCF算法，KCF是相关滤波器的一个变种。

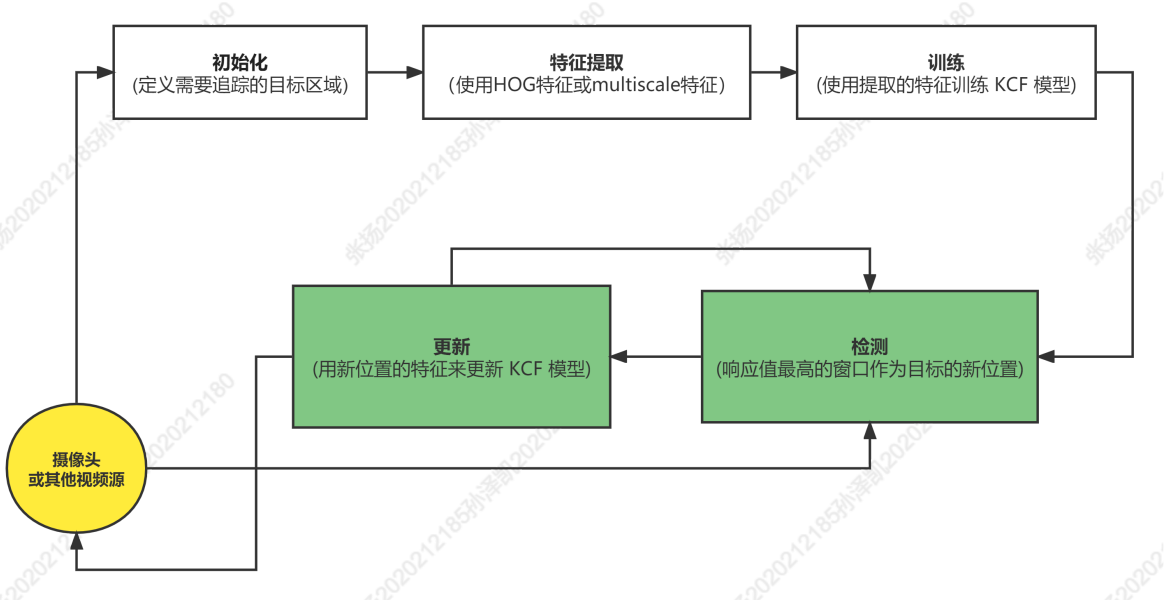
我们打算使用 `numpy` 库手动实现**KCF算法**，结合 `opencv` 库处理图像，实现目标跟踪的任务。

此外，我们还打算尝试使用基于深度学习的方法，并将二者效果进行比较。

系统设计

KCF算法的基本思想是在原始的最小输出和滤波器（Minimum Output Sum of Squared Error, MOSSE）的基础上引入核技巧（Kernel Trick）进行改进，使得算法能够学习复杂的非线性关系，从而提高了追踪精度。

系统框图



简要提纲

1. **初始化**：在视频序列的第一帧中，定义需要追踪的目标区域。然后，从这个区域中提取特征（使用 HOG 特征或multiscale特征）。
2. **训练**：使用提取的特征训练 KCF 模型。在这个过程中，KCF 将使用圆形窗函数（circular window function）计算两个特征之间的自相关（autocorrelation）。然后，将得到的相关矩阵用于训练滤波器。
3. **检测**：在视频的下一帧中，使用训练好的 KCF 模型对整个图像进行滑窗搜索，计算每个窗口位置的响应值。这个响应值可以理解为目标出现在这个窗口位置的概率。然后，选择响应值最高的窗口作为目标的新位置。
4. **更新**：用新位置的特征来更新 KCF 模型。更新时，通常会使用一个学习率参数来平衡新旧数据的重要性。
5. **循环**：重复步骤 3 和步骤 4，直到视频序列结束。

核心算法设计

（理解消化后的自行撰写的算法原理，请勿直接从网上抄或论文直译）

KCF算法的基本思想是通过**计算输入图像和参考图像之间的相关性**，来确定目标物体在图像中的位置。

KCF算法的主要滤波器是使用循环矩阵表示的相关滤波器。相关滤波器在信号处理中有广泛应用，它可以用于检测信号中是否包含某个特定的模式。在KCF中，相关滤波器的使用在于寻找图像中和目标模式相似的区域。

此外，KCF算法还引入了核技巧（`Kernel trick`）。这是一种常见的方法，可以将数据映射到一个更高维的空间，以便更好地处理非线性问题。核技巧在许多机器学习算法中都有使用，如支持向量机（SVM）。

总的来说，KCF算法主要使用了相关滤波器和核技巧这两种技术，前者用于寻找和目标模式相似的区域，后者用于处理非线性问题。

相关滤波器和岭回归

KCF（Kernelized Correlation Filter）是一种用于目标跟踪的算法。KCF 利用了岭回归方法来训练其跟踪器。在深入了解其方法之前，我们首先需要理解基础的岭回归和相关滤波器（Correlation Filter）。

岭回归是一种用于回归分析的技术，它通过对系数的大小施加惩罚，以解决多重共线性问题。基本的岭回归可以表示为：

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

其中，X 是输入数据，y 是目标输出，β 是待求的回归系数，I 是单位矩阵，λ 是控制正则化强度的参数。

在 KCF 中，我们实际上是在训练一个相关滤波器，其目标是找到一个滤波器 f，它能够将一个图像 x 映射到一个目标响应 y。在频域中，此目标可以表示为求解以下最小化问题：

$$\min_f ||\mathcal{F}(f) \odot X - Y||^2 + \lambda ||f||^2$$

其中，X 和 Y 分别是输入图像和目标响应的傅里叶变换，$\odot$ 表示 Hadamard 乘积（对应元素的乘积），$||\cdot||^2$ 表示平方范数，λ 是正则化参数，$\mathcal{F}(f)$ 是滤波器的傅里叶变换。对于这个问题，我们可以直接得到解析解：

$$\mathcal{F}(f) = \frac{Y}{X + \lambda}$$

然后，我们可以将 $\mathcal{F}(f)$ 通过逆傅里叶变换得到滤波器 f。

接下来，我们使用核方法来引入非线性。核方法通过引入一个核函数 k，将输入数据映射到高维空间。这允许我们在高维空间中学习非线性模型，同时只需要计算原始输入数据的核函数。对于 KCF，我们使用高斯核函数：

$$k(x, z) = e^{-\frac{||x-z||^2}{2\sigma^2}}$$

其中，x 和 z 是两个输入数据，σ 是高斯核的宽度参数。利用核方法，我们可以将 KCF 的优化问题重写为：

$$\min_\alpha ||\mathcal{F}(k(x, x)\alpha) \odot X - Y||^2 + \lambda \alpha^T K \alpha$$

其中，$K = k(x, x)$ 是输入数据的核矩阵，α 是要学习的参数。对于这个问题，

我们可以得到解析解：

$$\mathcal{F}(\alpha) = \frac{Y}{K \odot X + \lambda}$$

然后，我们可以通过逆傅里叶变换得到 α，这就是 KCF 的训练过程。在跟踪过程中，

我们可以通过将当前图像 z 映射到训练过的滤波器 α，来预测目标的位置：

$$y = k(z, x)\alpha$$

以上就是 KCF 使用岭回归方法训练跟踪器的大致流程。

核技巧

核技巧（Kernel trick）是在机器学习中常用的一种技术，它使我们可以在高维空间中进行计算，而无需显式地进行高维空间的计算。核技巧通常用于处理非线性问题，它的主要思想是将数据映射到一个高维空间，使得在高维空间中数据变得线性可分，然后在高维空间中进行线性学习。

这里的"核"（Kernel）指的是一个函数，它可以计算在高维空间中两个数据点的内积，而不需要显式地将数据点映射到高维空间。这个函数可以是任何满足 Mercer 定理的函数，常见的核函数包括线性核、多项式核、径向基函数（Radial Basis Function，RBF）核等。

例如，我的理解是：

在原始二维空间中，两个向量 (x_1, x_2) 和 (y_1, y_2) 的内积定义为：

$$(x_1, x_2) \cdot (y_1, y_2) = x_1 y_1 + x_2 y_2$$

如果我们将二维向量映射到一个三维空间，即

$$(x_1, x_2) \to (x_1^2, \sqrt{2} x_1 x_2, x_2^2)$$

$$(y_1, y_2) \to (y_1^2, \sqrt{2} y_1 y_2, y_2^2)$$

那么在这个三维空间中，这两个向量的内积变为：

$$\begin{aligned}(x_1^2, \sqrt{2} x_1 x_2, x_2^2) \cdot (y_1^2, \sqrt{2} y_1 y_2, y_2^2) &= x_1^2 y_1^2 + 2 x_1 x_2 y_1 y_2 + x_2^2 y_2^2 \\ &= (x_1 y_1 + x_2 y_2)^2 \\ &= (x \cdot y)^2\end{aligned}$$

我们可以看到，尽管在高维空间中计算内积可能非常复杂，但如果我们使用适当的核函数（在这个例子中，是$(x \cdot y)^2$），我们可以直接在原始空间中进行计算，无需显式地进行高维映射。这就是核技巧的基本思想。

系统实现

（算法实现、界面实现）

初始化

```
def draw_boundingbox(event, x, y, flags, param):
```

`draw_boundingbox()` 是定义需要追踪的目标区域的函数

用户可以用鼠标在图片上绘制一个矩形框（即“边界框”或 "bounding box"）。这个函数是处理鼠标事件的回调函数，它会被OpenCV的 `cv2.setMouseCallback()` 函数使用。

这个函数中定义了一些全局变量：

- `selectingObject`：布尔值，如果当前正在选择对象则为True。
- `initTracking`：布尔值，如果需要初始化追踪则为True。
- `onTracking`：布尔值，如果当前正在追踪对象则为True。
- `ix, iy`：起始点（鼠标左键按下的位置）的x和y坐标。
- `cx, cy`：当前鼠标位置的x和y坐标。
- `w, h`：矩形框的宽度和高度。

函数处理了四种鼠标事件：

1. `cv2.EVENT_LBUTTONDOWN`：左键按下时，开始选择对象，记录下当前鼠标的位置作为矩形框的起始点。
2. `cv2.EVENT_MOUSEMOVE`：鼠标移动时，记录下当前鼠标的位置。
3. `cv2.EVENT_LBUTTONUP`：左键释放时，完成选择对象。如果鼠标移动距离足够大（大于10像素），那么将计算出矩形框的宽度和高度，同时设定矩形框的起始点为左上角（x和y坐标较小的点）。此时需要初始化追踪。
4. `cv2.EVENT_RBUTTONDOWN`：右键按下时，取消选择对象。如果已经有一个矩形框存在，那么将矩形框的中心设定为当前鼠标的位置，并需要初始化追踪。

这样，在图像中就可以通过鼠标操作来选择和追踪感兴趣的区域。

特征提取

```
def getFeatures(self, image, inithann, scale_adjust=1.0):
```

`getFeatures` 是从图像中提取特征的函数，以下是函数的主要步骤的说明：

1. **初始化**：函数首先初始化了一些值，包括从ROI（感兴趣区域）中提取中心点坐标，并根据是否需要初始化Hanning窗口以及当前模板尺寸和填充大小来计算并设置模板大小和尺度。
2. **ROI提取**：函数计算了提取的ROI的大小和位置，然后使用 `cv2.BORDER_REPLICATE` 边界模式从图像中提取这个区域。如果提取的区域的尺寸不符合模板的尺寸，函数将会调整其尺寸以匹配模板。
3. **特征提取**：这部分代码取决于是否使用HOG特征。如果 `_hogfeatures` 为真，函数将使用 `fhog.getFeatureMaps` 提取HOG特征，并对其进行规范化、截断和PCA降维。如果 `_hogfeatures` 为假，函数将直接使用灰度值或原始像素值作为特征。
4. **特征预处理**：特征矩阵被转化为浮点数，并且从0~1的范围转化为-0.5~0.5的范围。
5. **Hanning窗口**：最后，如果 `inithann` 为真，函数将创建Hanning窗口，并将其应用于特征图。

下面是这些步骤中用到的一些重要概念的解释：

- **ROI**：ROI是"Region of Interest"的缩写，意思是感兴趣的区域，是在图像中需要处理的区域。
- **Hanning窗**：Hanning窗是一种窗函数，用于在做傅立叶变换时减少频谱泄漏。在这里，它被应用于特征图，是为了在计算相关性时减小图像边缘的影响。
- **HOG特征**：HOG是"Histogram of Oriented Gradients"的缩写，意思是方向梯度直方图，是一种常用的图像特征。HOG特征可以很好地捕捉图像的形状信息，对于目标检测和识别任务很有用。
- **PCA**：PCA是"Principal Component Analysis"的缩写，意思是主成分分析，是一种常用的降维方法。通过PCA，可以将高维数据映射到低维空间，同时尽可能保留原始数据的信息。

训练

```
def train(self, x, train_interp_factor):
```

`train` 函数是Kernelized Correlation Filter (KCF) 追踪器中的一个重要部分，它负责计算模型的参数，用于后续的对象定位。

函数的签名是：`void KCFTracker::train(const cv::Mat &im, float interp_factor)`

参数包括：

- `im`：当前帧的图像。
- `interp_factor`：插值因子，用于控制模型参数的更新程度。

函数的主要步骤如下：

1. `getFeatures`：首先，从图像中提取特征。这可能包括灰度值、颜色信息、梯度信息等。在KCF中，通常使用Histogram of Oriented Gradients (HOG) 特征。
2. `gaussianCorrelation`：然后，计算提取出的特征与模型的高斯相关性。这个步骤是用于计算目标位置的得分。
3. `cv::dft`：计算特征的离散傅里叶变换（DFT）。在KCF中，滤波器的训练实际上是在频域（Fourier domain）中进行的。
4. 然后，根据插值因子 `interp_factor` 更新滤波器的参数。如果 `interp_factor` 接近1，那么新的观测会大大影响滤波器；如果 `interp_factor` 接近0，那么滤波器会主要基于过去的观测。这是一个权衡过去和现在观测的方法。

具体的更新公式是：

```
_tmpl = (1.0 - interp_factor) * _tmpl + interp_factor * x;
_prob = (1.0 - interp_factor) * _prob + interp_factor * y;
```

其中 `_tmpl` 是在频域中的模型参数，`_prob` 是模型的目标响应（一个高斯函数）。

这就是 `train` 函数的主要步骤。其目的是通过新的观测来更新模型的参数，以便在下一帧中更好地定位目标。

检测

```
def detect(self, z, x):
```

`detect` 函数是在 KCF (Kernelized Correlation Filters) 跟踪器中进行对象检测的核心功能。KCF 是一种视觉跟踪算法，通过在线学习生成的过滤器，对目标对象进行跟踪。让我们一起看一下这个函数的具体操作。

函数的输入有两个参数，`z` 和 `x`，它们都是在 KCF 中表示图像的特征映射。

1. `k = self.gaussianCorrelation(x, z)`：这一行代码是计算两个特征映射 `x` 和 `z` 之间的高斯核函数。高斯核函数用于度量两个特征映射之间的相似性。
2. `res = real(fftd(complexMultiplication(self._alphaf, fftd(k)), True))`：这一行代码首先通过快速傅里叶变换（FFTD）得到 `k` 的频域表示，然后与 `_alphaf` 进行复数乘法，最后通过逆快速傅里叶变换得到相应的结果。这实际上是在频域中应用 KCF 过滤器。
3. `_, pv, _, pi = cv2.minMaxLoc(res)`：这一行代码找到 `res` 中的最大值（即最大的响应值，表示目标位置）及其位置。`pv` 是最大响应值，`pi` 是最大响应值的位置。

4. `p = [float(pi[0]), float(pi[1])]`：将位置转换为浮点数列表。
5. 如果最大响应值的位置不在 `res` 的边界，那么通过 `subPixelPeak` 函数进行子像素级别的插值，提高检测精度。
6. `p[0] -= res.shape[1] / 2.` 和 `p[1] -= res.shape[0] / 2.`：最后，从位置中减去 `res` 的中心位置，将坐标原点移动到 `res` 的中心。这是因为在 KCF 中，目标对象被假设为在 `res` 的中心。所以，这两行代码的目的是将相对位置转换为绝对位置。

总的来说，`detect` 函数是在 KCF 中检测目标对象位置的核心部分。它首先通过高斯核函数和 KCF 过滤器得到响应图，然后找到响应图中的最大值，通过子像素插值提高精度，最后返回目标对象的位置。

更新

```
def update(self, image):
```

`update` 函数主要用于更新 KCF 跟踪器中的目标对象位置和尺度信息。其输入是当前帧图像。

让我们详细解析这个函数中的各个步骤：

1. 首先，函数对 `_roi`（Region of Interest，即目标对象的边界框）的位置进行了边界检查。如果 `_roi` 的位置超出了图像的边界，那么就将 `_roi` 的位置修正到图像边界上。这是为了确保目标对象的边界框总是在图像内。
2. 然后，计算 `_roi` 的中心坐标 `(cx, cy)`。
3. `loc, peak_value = self.detect(self._tmpl, self.getFeatures(image, 0, 1.0))`：这一行代码是调用 `detect` 函数来检测目标对象的新位置。`self._tmpl` 是目标对象的模板，`self.getFeatures(image, 0, 1.0)` 是从当前图像中提取的特征。
4. 接下来的一段代码是对目标对象进行尺度更新。如果 `scale_step` 不为 1，那么会在较小和较大的尺度上重新检测目标对象的位置。如果在新的尺度上找到了更好的响应，那么就更新目标对象的位置和尺度。
5. `self._roi[0] = cx - self._roi[2] / 2.0 + loc[0] * self.cell_size * self._scale` 和 `self._roi[1] = cy - self._roi[3] / 2.0 + loc[1] * self.cell_size * self._scale`：这两行代码是更新 `_roi` 的位置。`loc[0] * self.cell_size * self._scale` 和 `loc[1] * self.cell_size * self._scale` 是在原有 `_roi` 中心的基础上，根据检测到的新位置 `loc` 做的偏移。
6. 接下来又一次对 `_roi` 的位置进行了边界检查。这是因为在更新 `_roi` 的位置后，有可能超出了图像的边界。
7. `x = self.getFeatures(image, 0, 1.0)`：这一行代码是从当前图像中提取特征。
8. `self.train(x, self.interp_factor)`：这一行代码是调用 `train` 函数来更新 KCF 跟踪器。`self.interp_factor` 是插值因子，用于控制 KCF 过滤器的学习速度。

9. 最后，函数返回更新后的 `_roi`。

总的来说，`update` 函数主要用于更新 KCF 跟踪器中的目标对象位置和尺度。其主要步骤是检测目标对象的新位置，更新目标对象的尺度，更新目标对象的位置，然后重新训练 KCF 跟踪器。

下一阶段任务

1.进一步修改调试代码

2.对算法效果进行全面测试

3.构建前端界面

4.尝试使用深度学习方法进行目标跟踪并将二者进行比较