

Министерство науки и высшего образования Российской Федерации  
Санкт-Петербургский политехнический университет Петра Великого  
Высшая школа прикладной математики и вычислительной физики

Работа допущена к защите  
Директор ВШПМиВФ  
\_\_\_\_\_ Уткин Л.В.  
«\_\_\_\_\_» \_\_\_\_\_ 2020 г.

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**  
**РАБОТА БАКАЛАВРА**  
**ПОИСК ПЕРЕСЕЧЕНИЙ ЛИНИЙ НА ИЗОБРАЖЕНИИ**  
по направлению подготовки 01.03.02 Прикладная математика и информатика  
Направленность (профиль) 01.03.02\_02 Системное программирование

Выполнил  
студент гр. 3630102/60201

Туников Д.А.

Руководитель  
доцент,  
к.т.н.

Шубников В.Г.

Консультант по нормоконтролю

Арефьева Л.А.

Санкт-Петербург  
2020

## **РЕФЕРАТ**

59 с., 67 рисунков, 4 таблицы, 0 приложений.

**КЛЮЧЕВЫЕ СЛОВА:** ПРЕОБРАЗОВАНИЕ ХАФА, МАШИНА ОПОРНЫХ ВЕКТОРОВ, ГИСТОГРАММА ОРИЕНТИРОВАННЫХ ГРАДИЕНТОВ, ПОИСК ОБЪЕКТОВ НА ИЗОБРАЖЕНИИ, ОБНАРУЖЕНИЕ ЖЕЛЕЗНОДОРОЖНЫХ СТРЕЛОК

Тема выпускной квалификационной работы: «Поиск пересечений линий на изображении». Цель работы - реализовать алгоритм для поиска железнодорожных стрелок на изображениях, сделанных с локомотива поезда. В ходе работы были решены следующие проблемы:

1. Реализация двух алгоритмов для поиска ж/д стрелок: на основе преобразования Хафа для прямых, и алгоритм с обучением машины опорных векторов.
2. Создание датасета из 1357 размеченных ж/д стрелок. Данный датасет использовался для проверки качества работы обоих алгоритмов и обучения машины опорных векторов.
3. Сравнение результатов работы разработанных алгоритмов.

Алгоритм на основе преобразования Хафа заключался в том, что изображение разбивалось на некоторое количество горизонтальных блоков, и в каждом блоке применялся алгоритм Хафа для поиска прямых. После чего пересечения найденных линий искались с учетом геометрических особенностей различных видов ж/д стрелок.

В алгоритме с использованием машины опорных векторов был SVM-классификатор, входными векторами которого были гистограммы ориентированных градиентов окрестностей ж/д стрелок, размеченных в датасете.

В результате алгоритм с использованием преобразования Хафа показал точность обнаружения=40% и процент найденных стрелок=70%, а алгоритм с построением SVM-классификатора показал точность=73% и процент найденных стрелок=78%.

Таким образом, можно сделать вывод о том, что алгоритм с применением SVM-классификатора достаточно устойчив к шуму и показывает хорошие результаты для изображений, сделанных в различных погодных условиях, и работает лучше алгоритма на основе преобразования Хафа.

## ABSTRACT

59 p., 67 figures, 4 tables, 0 appendices.

**KEYWORDS:** HOUGH TRANSFORM, SUPPORT VECTOR MACHINE, HISTOGRAM OF ORIENTED GRADIENTS, OBJECT DETECTION, DETECTION OF RAILWAY SWITCHES

The thesis theme is "Lines intersections detection in the image". The purpose of this work is to develop two algorithms for detection railway switches on images taken from a train locomotive. During the work following problems were solved:

1. Developing of two algorithms for railway switches detection: algorithm based on Hough Transform and the algorithm based on support vector machine.
2. Creating dataset of 1357 labeled railway switches. This dataset was used for training SVM classifier and testing both algorithms.
3. Comparison of the results of both algorithms.

In the algorithm based on Hough Transform first step was to divide an image to multiple horizontal blocks and detect lines in each block separately using Hough algorithm. After that intersections of detected lines were find using geometry features of different railboard switches types. Input vectors for SVM classifier were histograms of oriented gradients of railway switches marked in dataset. Several classifiers were trained for different switches types.

As a result, the algorithm using the Hough transform showed the detection accuracy=40% and percentage of switches found=70%, and the algorithm with the construction of the SVM classifier showed the accuracy=73% and recall=78%.

Thus, we can conclude that the algorithm using the SVM classifier is quite stable to noise and shows good results for images taken in various weather conditions. Also the SVM classifier shows better accuracy and recall than Hough based algorithm.

## СОДЕРЖАНИЕ

Введение .....	5
Глава 1. Введение	5
1.1. Постановка задачи .....	6
1.2. Обзор литературы .....	7
1.2.1. Поиск объекта по его геометрическим свойствам .....	7
1.2.2. Поиск объекта по ключевым точкам .....	19
1.2.3. Обучение классификатора на размеченных данных .....	20
Глава 2. Основная часть	25
2.1. Метод основанный на преобразовании Хафа.....	25
2.1.1. Поиск границ на изображении.....	25
2.1.2. Разбиение изображения на горизонтальные блоки .....	25
2.1.3. Поиск рельсов .....	27
2.1.4. Поиск стрелок .....	29
2.1.5. Сложность алгоритма.....	35
2.2. Метод с построением SVM-классификатора.....	35
2.2.1. Разметка изображений .....	35
2.2.2. Обучение классификатора.....	37
2.2.3. Предсказание результаты на входном изображении .....	37
2.2.4. Первый этап обучения .....	38
2.2.5. Второй этап обучения.....	38
2.2.6. Третий этап обучения .....	40
2.2.7. Подтверждение стрелок .....	42
2.2.8. Сложность алгоритма.....	45
Глава 3. Результаты	47
3.1. Описание данных для тестирования .....	47
3.2. Результаты работы алгоритма, основанного на преобразовании Хафа ..	47
3.2.1. Примеры работы алгоритма.....	47
3.2.2. Статистические результаты .....	50
3.3. Результаты работы алгоритма с построением SVM-классификатор .....	51
3.3.1. Примеры работы алгоритма.....	51
3.3.2. Статистические результаты .....	53
Глава 4. Заключение	56
Литература	57

## ГЛАВА 1. ВВЕДЕНИЕ

Проблема поиска пересечений линий на изображении является очень распространенной и её решение может быть использовано в таких задачах компьютерного зрения как:

- A. Автоматическое определение полей во время спортивных трансляций
- B. Детектирование ориентиров при автоматическом управлении роботом
- C. Детектирование железнодорожных стрелок на изображении, сделанном с локомотива поезда

Для автоматизации процесса управления поездом необходимо создать систему, которая позволяла бы стать надежным автоматическим ассистентом/помощником для машиниста локомотива (стать неким driving assistance), но ни в коем случае не заменить действия человека полностью. Одной из задач такой системы является обнаружение железнодорожных стрелок для обеспечения движения поезда по правильному пути. Решение этой задачи позволит обнаруживать разветвления пути заранее (визуально) и анализировать возможные дальнейшие пути движения поезда, выдавая полезную информацию для машиниста локомотива, такую как: "на правой ветке обнаружены вагоны - путь занят!". Это решение значительно повысит безопасность движения на железных дорогах.

**Целью данной работы** является разработка алгоритма детектирования железнодорожных стрелок на изображении, сделанном с локомотива поезда. Таким образом, необходимо решить следующие задачи:

- A. Разметить датасет из ж/д стрелок на изображениях, сделанных с локомотива поезда.
- B. Разработать алгоритм поиска стрелок на основе преобразования Хафа[1] и геометрических особенностей различных видов ж/д стрелок.
- C. Обучить SVM[11]-классификатор, который по входной гистограмме ориентированных градиентов[7] смог бы предсказать - соответствует эта гистограмма изображению железнодорожной стрелки или нет.
- D. Привести результаты работы обоих алгоритмов на размеченном датасете и сравнить алгоритмы.

## 1.1. Постановка задачи

На вход подается изображение — фотография сделанная с головы поезда. Необходимо найти на входном изображении места пересечений железно-дорожных путей. Пересечением будем называть точку, в которой пересекаются рельсы одной ж/д колеи с другой. На выходе алгоритма должен быть набор точек, в которых было обнаружено пересечение.



Рис.1.1. Пример входного изображения

Замечания:

- A. Точность поиска стрелок - окрестность от 20x20 до 50x50 пикселей(в зависимости от дальности стрелки)
- B. Необходимо правильно находить стрелки в пределах 20-30 метрах перед поездом. Нет задачи детектировать стрелки, которые находятся слишком далеко от поезда(40 метров и далее).

## **1.2. Обзор литературы**

Существует несколько глобальных подходов к поиску объектов на изображении:

- A. Сопоставление границ, градиентов и чёрно-белых пикселей изображения. Этот подход устойчив к изменению света, но не устойчив к изменению положения объекта на изображении(вращение, угол наклона).[9].
- B. Поиск объекта по геометрическим свойствам. Если форма искомого объекта может быть задача аналитически, то такие объекты можно искать с помощью преобразования Хафа [1].
- C. Подход, основанный на сопоставлении признаков ключевых точек изображения с ключевыми точками искомого объекта. Данный подход реализован в следующих методах: SIFT[8], SURF[10].
- D. Обучение классификатора на размеченных данных. В роли классификатора может быть: SVM классификатор[11], дерево решений [12], KNN классификатор [13]. В качестве входных векторов для классификаторов могут быть использованы различные виды признаков, полученные из размеченных изображений. Например, гистограмма ориентированных градиентов(HOG)[7], SURF[8]/SIFT[10] дескрипторы и так далее.

Рассмотрим данные подходы подробнее.

### ***1.2.1. Поиск объекта по его геометрическим свойствам***

Для поиска объектов, геометрическая форма которых может быть задана некоторым уравнением(н-р прямая, круг, эллипс) используется следующий подход:

- A. Из изображения извлекаются границы. Это можно сделать с помощью алгоритма Canny [2].

Пример применения алгоритма Canny к входному изображению:

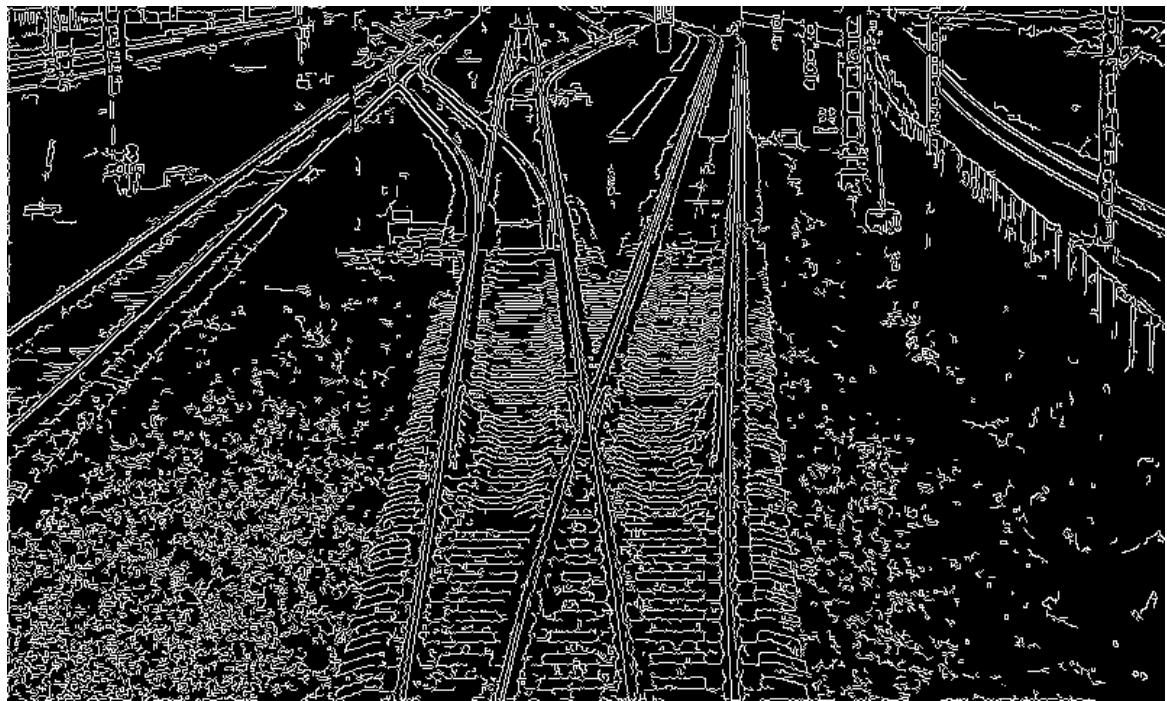


Рис.1.2. Алгоритм Canny

В. После чего к полученным границам применяется алгоритма Хафа [1]. Если целью является поиск прямых, то алгоритм Хафа даст следующий результат(красным обозначены найденные линии):

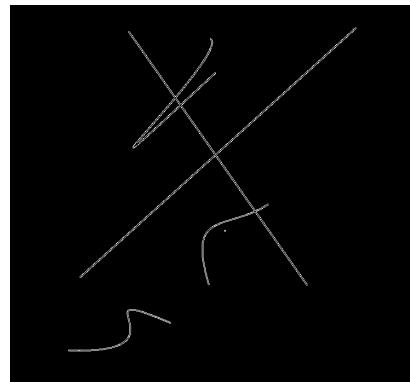


Рис.1.3. Алгоритма Canny

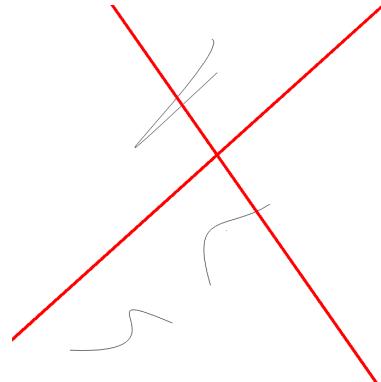


Рис.1.4. Поиск прямых

Таким образом, если требуется искать пересечения прямых линий на изображении, то для этого отлично подойдет следующий алгоритм:

- Поиск границ [2]
  - Поиск линий [1]
  - Поиск пересечений линий, найденных в предыдущем шаге. Это можно сделать аналитически. Для каждой пары найденных прямых составляем систему уравнений, тогда если эти прямые не параллельны, то решением системы будет точка пересечения прямых.
- $$\begin{cases} a_1x + b_1y = c_1 \\ a_2x + b_2y = c_2 \end{cases}$$

## Предыдущие работы, посвященные поиску ж/д путей и их пересечений с помощью применения геометрически ориентированного подхода

**Vision based rail track and switch recognition for self-localization of trains in a rail network [6].** Подход, описанный в данной работе основан на анализе изображения с камеры, стоящей в голове поезда. Условиями для корректного определения стрелок на железной дороге является то, что расстояние между параллельными рельсами всегда одинаковое, а также радиус кривизны рельсов достаточно большой.

В данной работе съёмка ж/д путей происходит достаточно часто, и к каждому вновь сделанному снимку применяется следующий алгоритм:

1. Из изображения вырезается близкая к поезду полоса.



Рис.1.5. Горизонтальный блок

2. К выделенному участку применяется алгоритм поиска ребер:
  - А. Рассчитывается вектор градиента в каждой точке изображения
  - Б. Далее рассматриваются окрестности по 3x3 пикселя. И для каждого пикселя с величиной градиента большей, чем у половины соседей в окрестности выполняется следующая процедура: Строится матрица ковариаций данного пикселя с соседями в окрестности 3x3.  $S = \begin{pmatrix} Cov_{xx} & Cov_{xy} \\ Cov_{yx} & Cov_{yy} \end{pmatrix}$ ,  $Cov_{ab} = \frac{1}{N} \sum_1^N (g_a, g_b)$ , так как среднее значение градиента для черно-белого изображения равняется нулю.

C. Вычисляются собственные числа этой матрицы, и если выполнены неравенства:  $\begin{cases} \lambda_1 \geq t_l \\ \frac{\lambda_2}{\lambda_1} \geq m_l \end{cases}$ , где  $t_l, m_l$  вычисленные эмперическим путём константы, то данный пиксель считается краевым. Также из матрицы ковариации получаются два собственных вектора:  $e_1, e_2$ , больший из которых перпендикулярен краю, а меньший параллелен краю в данной точке.

В итоге границы найденные границы выделенного блока выглядят следующим образом:



Рис.1.6. Границы в блоке

### 3. Поиск прямых.

Строится массив аккумулятора  $A(\theta, x)$ (по аналогии с алгоритмом Хафа[1]). И для каждого краевого пикселя(К) вычисляется угол  $\theta$  между большим собственным вектором  $e_2$  и осью ОY. Также необходимо вычислить значение  $x_c$ . Строится дополнительная прямая  $y_c = \frac{H}{2}$ , где  $H$  - высота рассматриваемого изображения. Тогда  $x_c$  будет координатой  $x$  точки пересечения построенной прямой  $y_c$  и продолжения собственного вектора  $e_2$ . Увеличиваем значения аккумулятора  $A(\theta, x_c) = A(\theta, x_c) + 1$ . В итоге максимумы аккумулятора будут соответствовать параметрам прямых линий на изображении. На рисунке найденные линии обозначены черными точками.

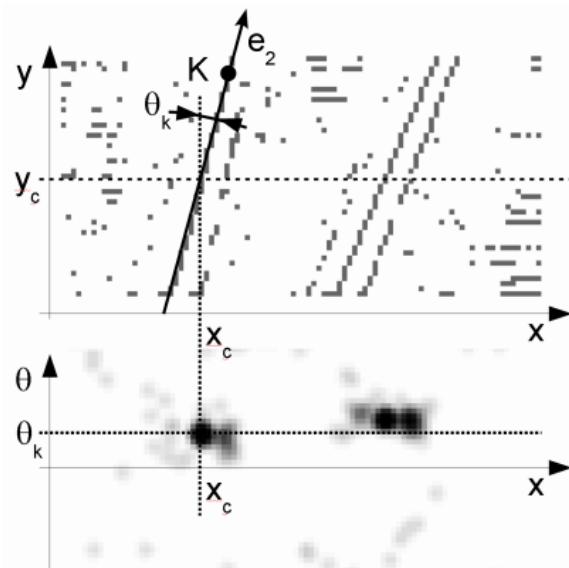


Рис.1.7. Поиск линий

Данный алгоритм работает значительно быстрее преобразования Хафа(ускорение достигается за счет сохранения времени, которое в алгоритме Хафа тратится на перебор  $\theta \in [0, \pi]$ ). Угол  $\theta$  в каждой краевой точке вычисляется с помощью направления собственного вектора матрицы ковариаций в данной точке( $\theta = \angle(e_2, OY)$ ). При этом алгоритм качественно находит прямые линии, проходящие через центральную вертикальную ось изображения.

Существуют и другие алгоритмы, позволяющие ускорить оригинальный алгоритм Хафа. Например [23]. 4. Выделение ж/д колеи по найденным прямым.

В данном алгоритме предполагается, что ширина железнодорожной колеи заранее зафиксирована и приходит на вход алгоритму.

Поэтому чтобы найти пары рельсов, принадлежащих к одной колее нужно выполнить перебор по всем найденным рельсам, сравнивая при этом расстояние между ними с эталонным расстоянием между рельсами(с некоторой погрешностью  $\delta R$ ).

Но нельзя забывать про то, что изображение делается с камеры и нужно учитывать перспективную проекцию [4] точек изображения на точки в реальном мире, чтобы корректно сравнивать расстояние между рельсами на изображении с расстоянием в мировой системе координат. В итоге результат поиска ж/д колеи следующий(на рисунке выделен горизонтальный блок и линии, соответствующие центрам найденных пар рельсов):



Рис.1.8. Центр колеи

## 5. Определение стрелок.

При поиске пересечений найденных рельсов учтем, что нам известен минимальный радиус кривой, по которой может двигаться поезд(является параметром алгоритма). Из данного радиуса можно вычислить максимальное боковое отклонение  $s_1$ , на которое должен отклониться центр колеи, чтобы считаться пройденной колеёй.

Таким образом, мы ожидаем, что одна из наблюдаемых нами колея всегда имеет боковое отклонение от центра поезда принадлежащее интервалу  $[-s_1, s_1]$ . Если центр колеи становится  $< -s_1$ , то это означается, что данная колея является левой веткой колеи, по которой поезд движется в данный момент времени. В обратном случае правой веткой. В момент времени, когда происходит переход данной границы мы можем отметить, что поезд прошел стрелку. На рисунке 1.9 обозначены границы  $s_1$ (теоретическая граница) и  $s_2$ (экспериментальная граница), где  $s_2 = s_1 + \delta s$ (подобрана экспериментально). Пересечение фиксируется в момент времени, когда центр колеи переходит границу  $|s_2|$ (это место показано стрелками на рисунке).

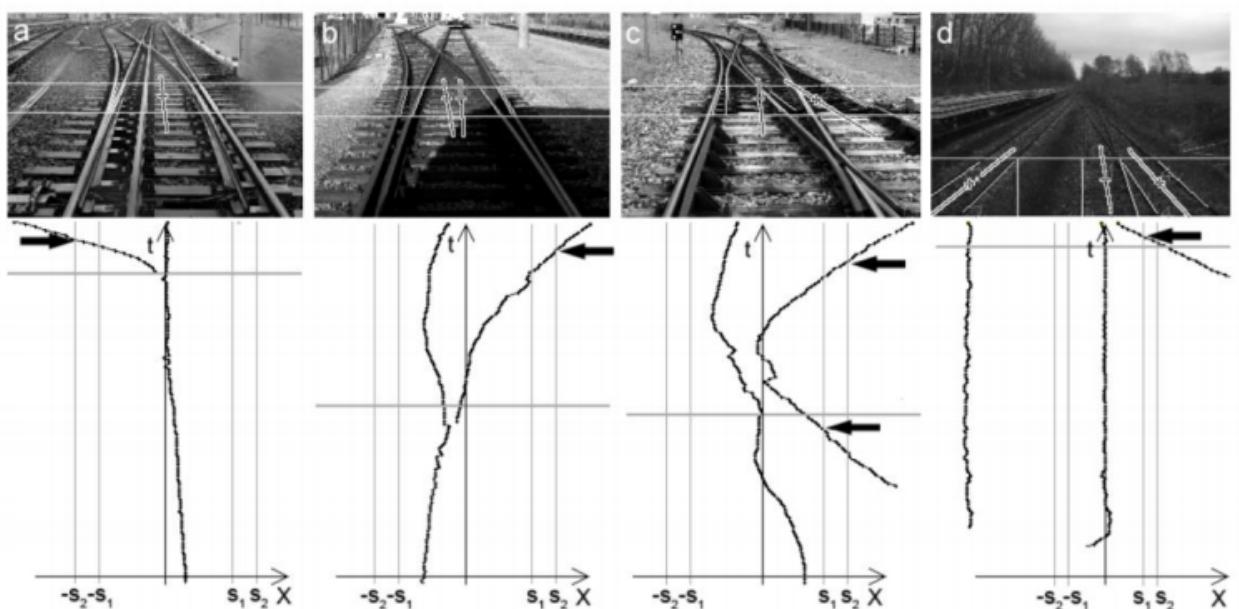


Рис.1.9. Перемещение X центра колеи

На втором слева рисунке видно, что сначала есть отметка только от колеи, идущей прямо. Потом появляется отметка от колеи отклоняющейся налево. После появления отклоняющейся колеи, отметка от прямой колеи отклоняется направо, а отметка от отклоняющейся колеи остается в центре - следовательно поезд пошёл по отклоняющейся колее, так как теперь её координата центра находится ближе

всего к середине поезда. В момент, когда отметка от центра прямого пути проходит порог  $s_2$  отмечается стрелка.

На последнем рисунке сначала есть отметка от двух параллельных путей. Потом появляется отметка от присоединяющегося справа пути. И в момент, когда координата центра присоединяющегося пути проходит порог  $s_1$  отмечается стрелка.

### **Efficient railway tracks detection and turnouts recognition method using HOG features [5]**

В данной работе описывается метод поиска железнодорожных путей с использованием Histogram of oriented gradients[7](гистограмма ориентированных градиентов).

Алгоритм следующий: 1. Входное изображение разбивается на сетку, с уменьшением размера блока в зависимости от координаты Y на изображении(это сделано чтобы имитировать перспективную проекцию).

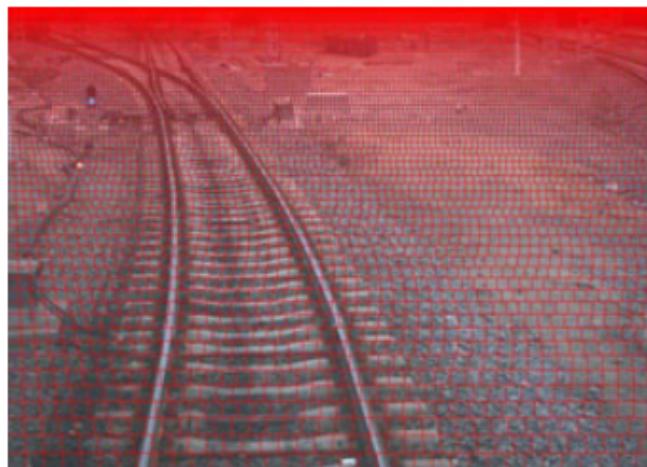


Рис.1.10. Разбиение на сетку

2. Начиная с нижней строки сетки применяется алгоритм growing up, который на основе похожести HOG соседних верхних клеток расширяет множество схожих кусочков изображения. В итоге на выходе алгоритма growing up имеем множество рельсов(на рисунке обозначены разными цветами).

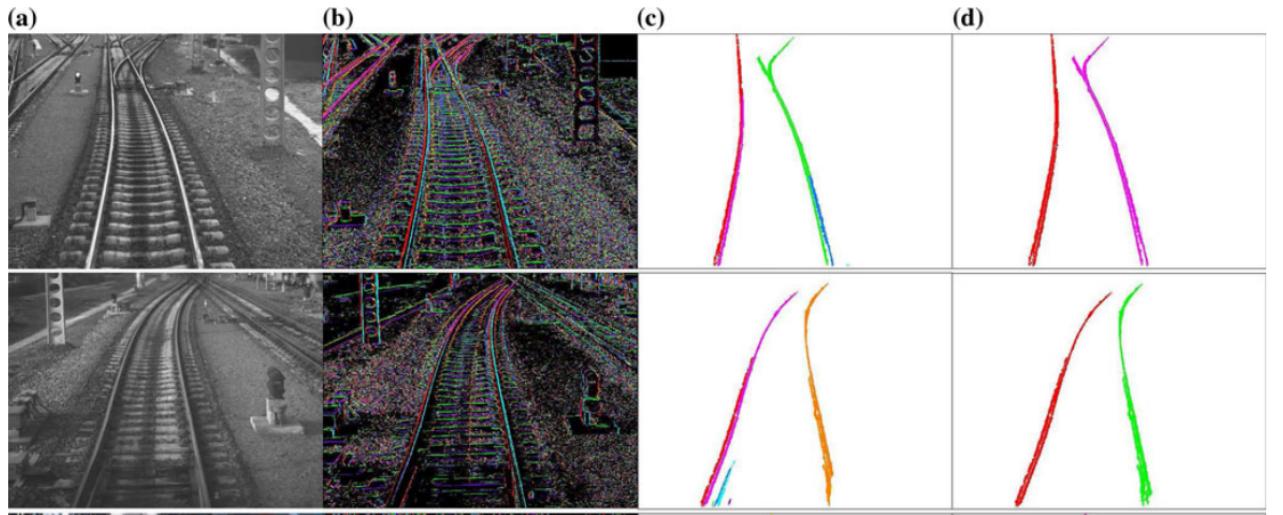


Рис.1.11. Результат поиска рельсов

**Поиск полосы движения автомобиля [15]**. Для решения задачи поиска стрелок полезно уметь находить сами ж/д пути. Задача поиска ж/д пути тесно связана с задачей поиска полосы движения автомобиля(lane detection). Рассмотрим несколько работ, посвященных этой теме.

**Robust lane detection and tracking for lane departure warning** В работе [15] предложен следующий алгоритм для поиска полосы движения автомобиля:

Поиск границ на изображении с помощью алгоритма Canny 1.12(b). После чего удаляются все горизонтальные линии 1.12(c). И далее удаляются одиночные пиксели, которые тоже являются шумом 1.12(d).

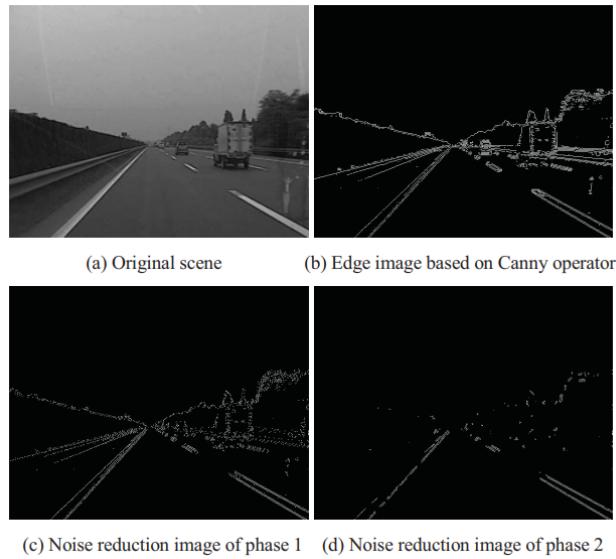


Рис.1.12. Избавление от шума

Далее рассматривается нижняя четверть полученного изображения. И проводится "сканирующая линия на которой считаются интенсивности пикселей и строится график интенсивности в зависимости от горизонтально позиции пикселя на линии 1.13(b).

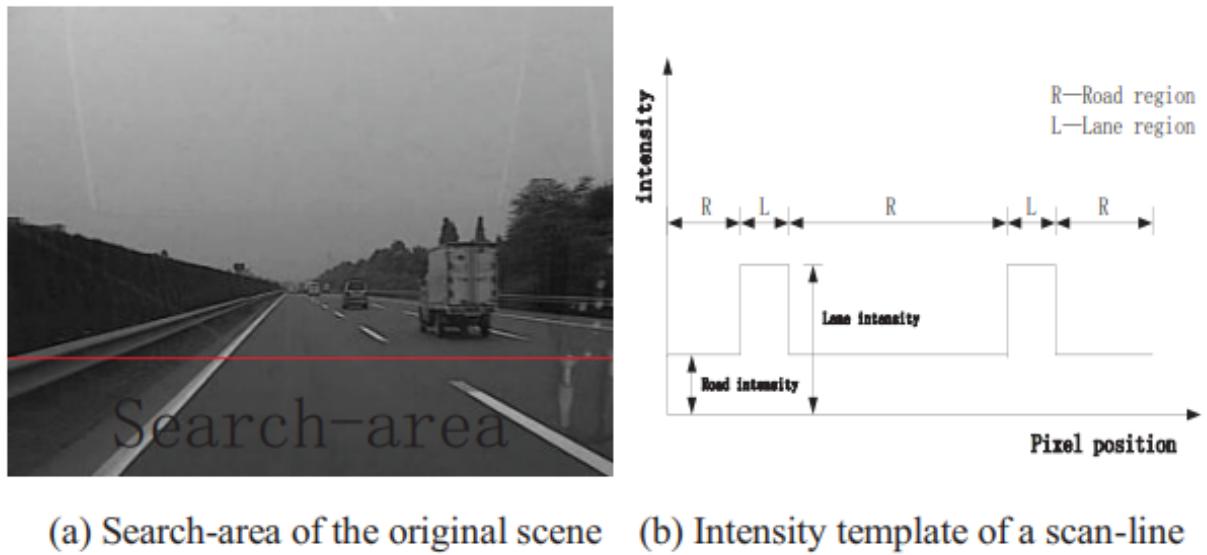


Рис.1.13. График интенсивности

Видно, что интенсивность в районе разметки сильно изменяется относительно дороги. Применяя такую процедуру на каждом кадре можно детектировать полосу в непосредственной близости перед автомобилем. Итоговый результат детектирования полосы:



Рис.1.14. Результат поиска полосы

**Double Lane Line Edge Detection Method Based on Constraint Conditions Hough Transform** В работе [16] предложен алгоритм на основе преобразования

Хафа[1]: Сначала учитывается то, что R и G компоненты цвета для полосы движения значительно выше, чем у самой дороги(высокие значение R и G говорят о желтом оттенке). В связи с этим цвета пикселей изображения преобразуются для удаления лишних шумов:

$$IM(i, j) = \begin{cases} 255, & R(i, j) \geq (0.2R_{\min} + 0.8R_{\max}) \& \& \\ & G(i, j) \geq (0.2G_{\min} + 0.8G_{\max}) \\ 0, & others \end{cases}$$

Рис.1.15

После чего изображение разделяется на три горизонтальный блока: близкий, средний и дальний.

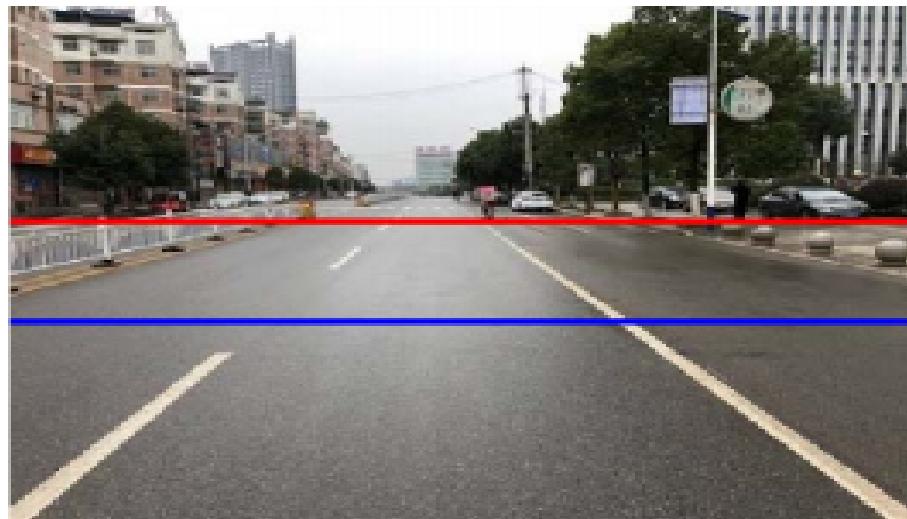


Рис.1.16. Разделение на блоки

Далее применяется алгоритм Canny[2] для поиска границ. И ищутся прямые линии с помощью преобразования Хафа [1].

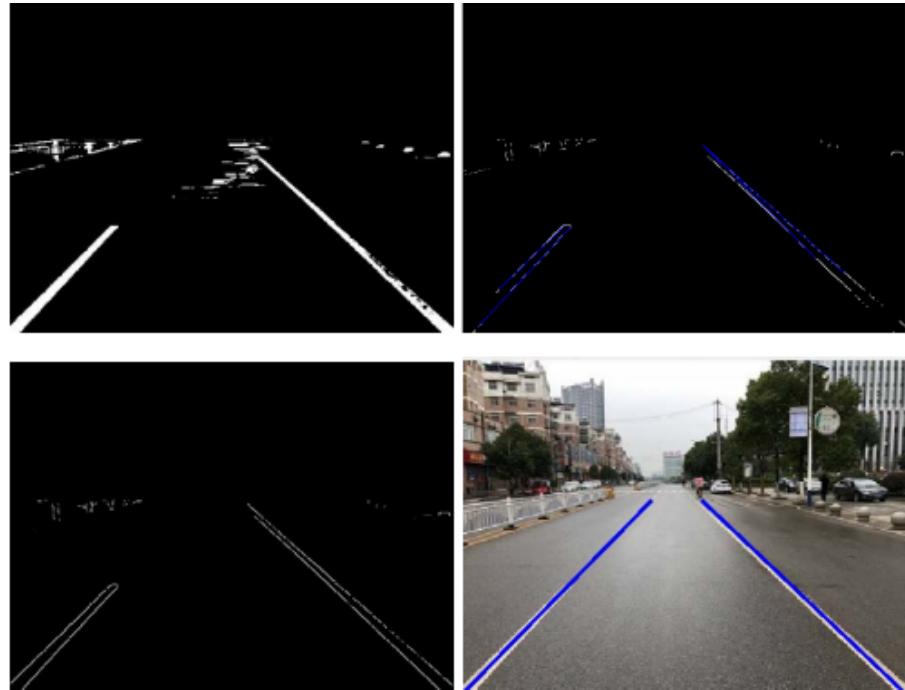


Рис.1.17. Применение алгоритма Хафа

В случае, когда дорога имеет поворот применяется следующая процедура. Считается, что поворот можно описать параболической линией. Поэтому в ближней части прямые ищутся, как и прежде, через преобразование Хафа [1], а в средней части как парабола( $y_m$  - координата прямой разделяющей ближнюю и среднюю части изображения, а  $a$  и  $b$  - параметры прямой>):

$$x = a + by + cy^2 \text{ - уравнение параболы}$$

$$x = \begin{cases} a + by, & y \leq y_m \\ c + dy + ey^2, & y > y_m \end{cases}$$

Теперь для средней зоны ищется продолжение найденных в ближней зоне прямых. Это делается просто проходом с разными шагами по координате X и поиска последовательности точек, которая удовлетворяла бы уравнению параболы. В итоге результат получается следующий:

**A New Lane Line Segmentation and Detection Method based on Inverse Perspective Mapping[22]** В данной работе описан подход для поиска полосы движения автомобиля, основанный на технике "Inverse Mapping". Точки проецируются из 2D картинки на 3D объект с помощью закона перспективного проецирования[4], после чего точка 3D объекта проецируется ещё раз на 2D плоскость таким образом,



Рис.1.18. Результат поиска дороги

чтобы избежать эффекта перспективы на 2D изображении. В итоге получается следующий результат:

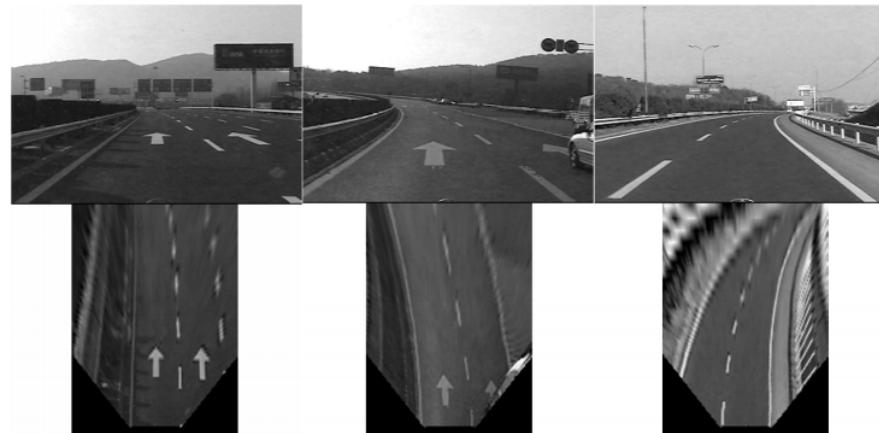


Рис.1.19. Inverse Perspective Mapping

После чего на полученном изображении параллельные прямые, соответствующие линиям разметки, становятся действительно близкими к параллельным и можно применить алгоритма Хафа для поиска прямых[1] и получить следующий результат детектирования полосы движения:

Технику "Inverse Mapping" можно применять и для задачи поиска стрелок на изображении. Действительно, если сделать "Inverse Mapping" изображения, сделанного с локомотива поезда можно было бы получить изображение без эффекта перспективы и на нём прямые линии будет найти проще, чем на исходном изображении. Однако, в нашей задаче параметры камеры, которая установлена на локомотиве неизвестные, поэтому сделать правильный "Inverse Mapping" не удастся, что будет приводить к непредсказуемому итоговому результату.

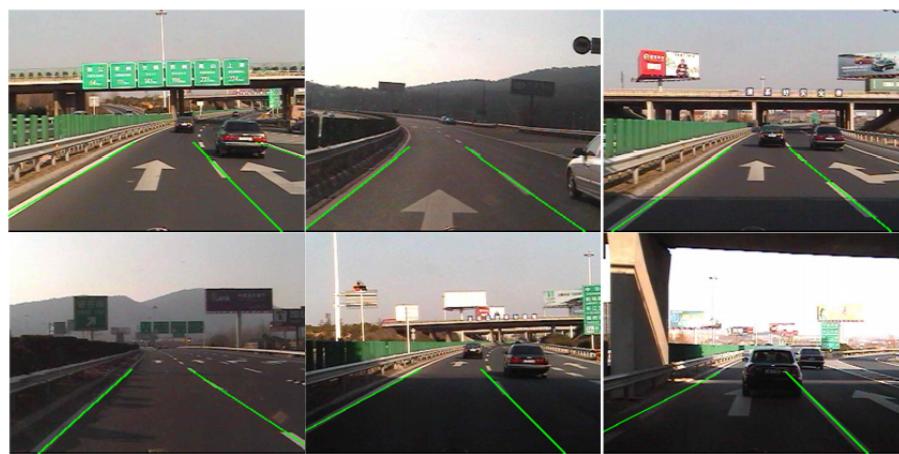


Рис.1.20. Результат детектирования полосы

### 1.2.2. Поиск объекта по ключевым точкам

Одним из методов поиска объектов на изображении является построение SIFT[8] дескриптора и сопоставление этого дескриптора с SIFT дескриптором искомого объекта.

Сначала изображение преобразуется в большой набор векторов признаков(SIFT дескриптор), каждый из которых инвариантен относительно параллельного переноса изображения, масштабирования и вращения, частично инвариантен изменению освещения и устойчив к локальным геометрическим искажениям.

После чего SIFT дескриптор текущего изображения и искомого объекта сопоставляются и на выходе можно увидеть соответствие ключевых точек на изображении и ключевых точек искомого объекта:

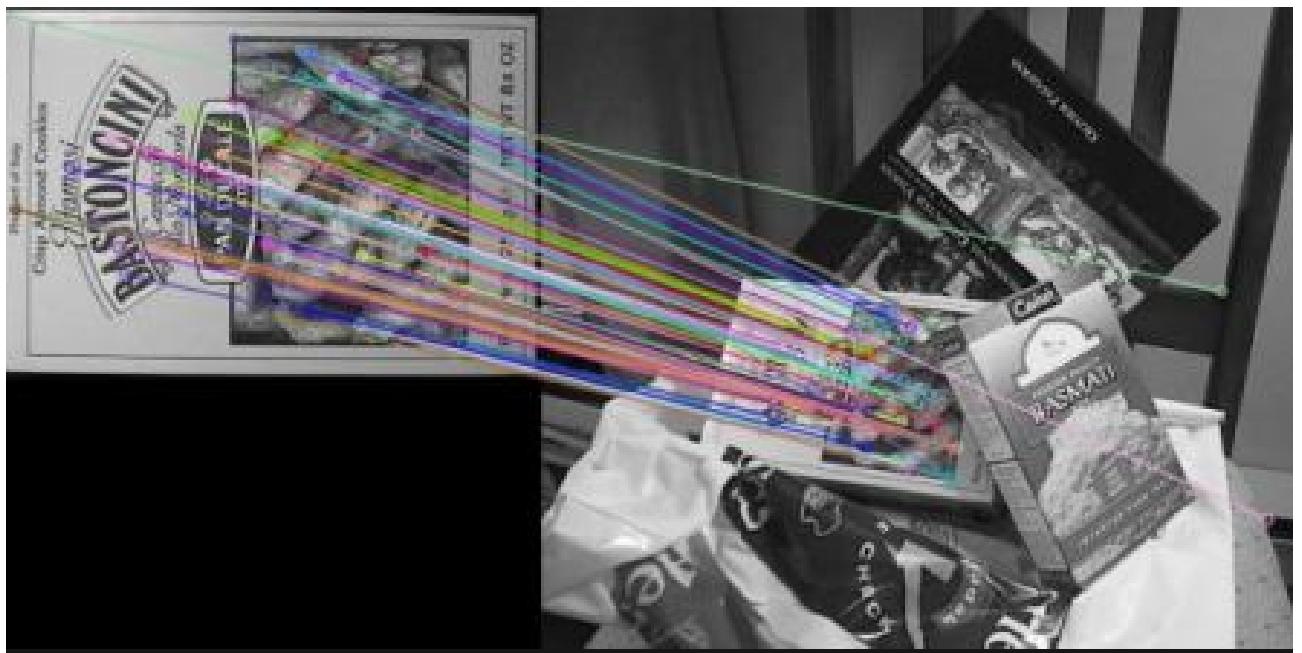


Рис.1.21. SIFT пример

Если говорить о применении техники SIFT для поиска пересечений ж/д путей на изображении. То можно создать некоторую базу с различными видами стрелок и их SIFT дескрипторами.

Тогда алгоритм для поиска пересечений на входном изображении будет следующий:

- A. Вычислять SIFT дескриптор входного изображения
- B. Сопоставить его с каждым из SIFT дескрипторов, находящихся в базе данных.
- C. Если было найдено достаточно точное сопоставление - выполнить уточнение места пересечения и добавить его в выходной результат

### *1.2.3. Обучение классификатора на размеченных данных*

Распространенным способом поиска объекта на изображении является обучение классификатора на размеченных данных. Рассмотри несколько видов классификаторов.

#### *1.2.3.1. Support vector machine*

Рассмотрим классификацию с помощью SVM(машина опорных векторов)[11].

Постановка задачи:

Необходимо классифицировать данные, каждый объект которых представляется как точка  $x^p$  в р-мерном пространстве. Каждая из точек принадлежит одному из двух классов. Необходим построить гиперплоскость размерности  $p - 1$ , которая линейно разделяла бы входные точки. Таких гиперплоскостей может быть много, но необходимо найти ту, расстояние до которой от любого класса был бы максимальным. Что эквивалентно тому, что сумма расстояний от крайних точек классов до плоскости максимальна. Если такая гиперплоскость существует, она называется оптимальной разделяющей гиперплоскостью, а соответствующий ей линейный классификатор называется оптимально разделяющим классификатором.

На вход алгоритма подается набор пар:  $[(x_1, c_1), (x_1, c_1), \dots, (x_n, c_n)]$

Уравнение искомой гиперплоскости плоскости имеет вид:  $wx - b = 0$ , где  $w$  - перпендикуляр к разделяющей плоскости, а  $b$  - расстояние от плоскости до начала координат.

$wx - b = 1$  и  $wx - b = -1$ , соответствуют плоскостям, проходящим через крайние точки классов:

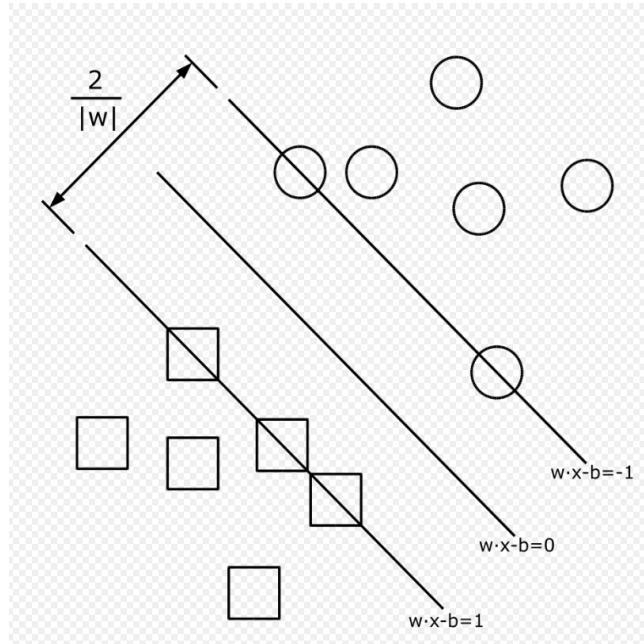


Рис.1.22. SVM гиперплоскость

Для полного разделения классов необходимо чтобы для всех  $i \in [1, n]$  были выполнены неравенства:

$$\begin{cases} wx_i - b \geq 1, c_i == 1 \\ wx_i - b \leq -1, c_i == -1 \end{cases} \quad (1.1)$$

Тогда для того, чтобы найти оптимальную гиперплоскость нужно минимизировать  $\|w\|$ , при условиях 1.1, что соответствует такой задаче минимизации:

$$\begin{cases} \|w\|^2 > \min \\ c_i(wx_i - b) \geq 1, i \in [1, n] \end{cases} \quad (1.2)$$

В случае, если классы не разделимы линейно, вводится дополнительный параметр  $\varepsilon_i \geq 0$ , характеризующий ошибку на объектах  $x_i$ . Таким образом, смягчим ограничение во втором уравнении системы 1.2 и введём штраф за ошибку  $C$  - данный параметр позволяет регулировать соотношение между максимизацией ширины разделяющей полосы и минимизацией ошибки:

$$\begin{cases} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^n \varepsilon_i > \min_{wb\varepsilon_i} \\ c_i(wx_i - b) \geq 1 - \varepsilon_i, i \in [1, n] \\ \varepsilon_i \geq 0 \end{cases} \quad (1.3)$$

Теперь, понимая математический аппарат работы SVM рассмотрим процесс построения классификатора для поиска объекта на изображении.

Входными векторами для SVM могут выступать различные виды признаков искомого объекта. Например, гистограмма ориентированных градиентов(HOG)[7], SIFT[8]/SURF[10] дескрипторы. Таким образом для решения задачи поиска объекта с помощью SVM необходимо выполнить следующие шаги:

- A. Разметить некоторое количество искомых объектов на изображении. Такие объекты будем относить к первому классу.
- B. Сгенерировать(или разметить) некоторое количество отрицательных примеров, которые точно не являются искомым объектом. Такие объекты отнесем ко второму классу.
- C. Рассчитать вектор признаков по которым мы будем обучать классификатор в каждой из размеченных(сгенерированных) точках изображения.
- D. Построить разделяющую объекты разных классов гиперплоскость, используя посчитанные в предыдущем пункте вектора признаков объектов.
- E. Для нахождения объекта на входном изображении, используя технику "скользящее окно"[14] рассчитать вектор признаков во всевозможных окнах и определить по какую сторону от гиперплоскости находится вычисленный вектор. Таким образом в итоге получится некоторый набор окон, в которых был найден искомый объект:

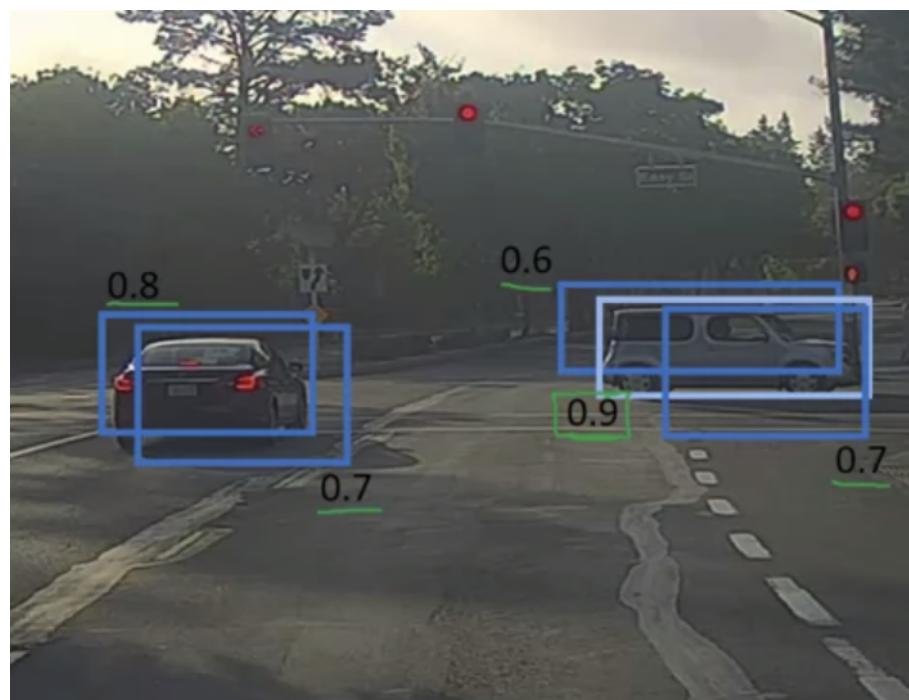


Рис.1.23. Окна с объектами

- F. Из найденных окон выбрать наиболее точно описывающее искомый объект. Это можно сделать выбрав окно с максимальной вероятностью нахождения

искомого объекта. Например, на рисунке 1.23 для правой машины это будет окно с вероятностью 0,9.

#### 1.2.3.2. Haar classifier

Одним из методов поиска рельсов на изображении также может быть обучение классификатора на основе признаков Хаара. Можно заметить, что область изображения, в которой находится рельс, явно выделяется на фоне окрестностей справа и слева от рельса:

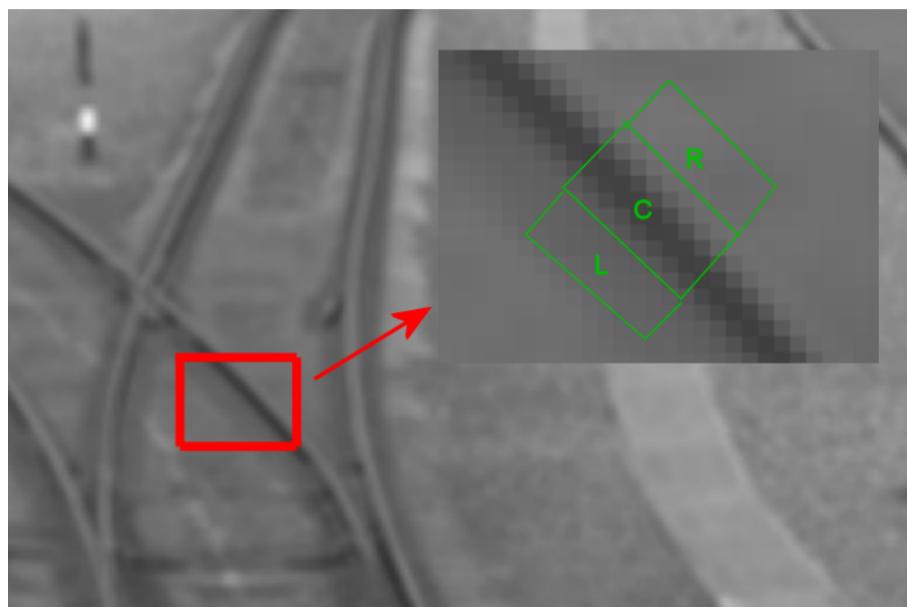


Рис.1.24

Таким образом, с использованием большого датасета размеченных рельсов[24] можно обучить классификатор, который по разности интенсивностей пикселей в черной и белой областях(рис.1.25) смог бы предсказать, является ли выделенная область рельсом или нет. Причем рельсы могут идти под наклоном и тогда только вертикальные признаки Хаара не подойдут, поэтому нужно использовать наклонные признаки Хаара:

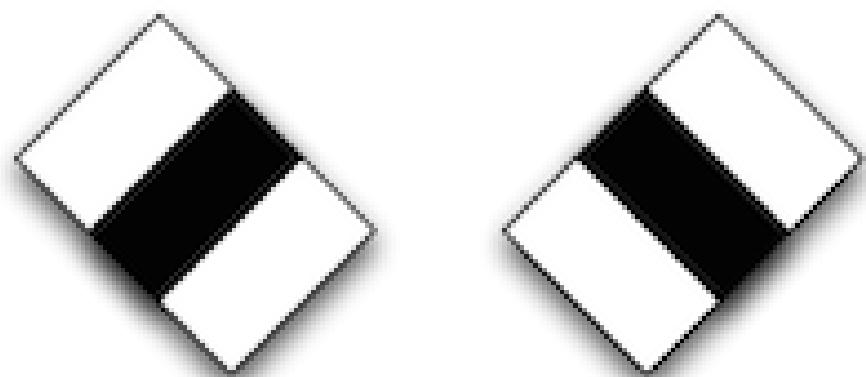


Рис.1.25. Наклонные признаки Хаара

## ГЛАВА 2. ОСНОВНАЯ ЧАСТЬ

### 2.1. Метод основанный на преобразовании Хафа

#### 2.1.1. Поиск границ на изображении

Первым шагом метода является поиск границ на входном изображении. Это делается с помощью алгоритма Canny [2]:

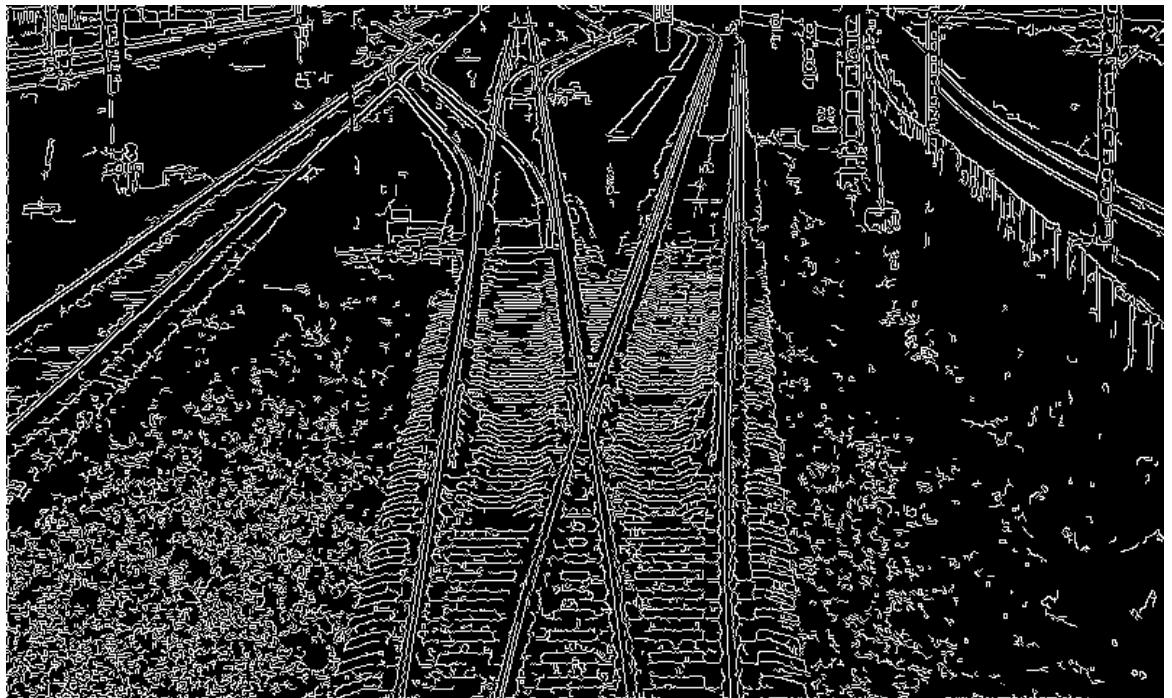


Рис.2.1. Применение алгоритма Canny

#### 2.1.2. Разбиение изображения на горизонтальные блоки

Разбиваем изображение на горизонтальные блоки 2.3. Размер блоков уменьшается в зависимости от координаты Y на изображении, это сделано для учета перспективной проекции объектов из реального мира на изображении - чем дальше объект, тем меньше он на изображении. В идеале размер блока должен меняться по закону, показанному зеленой линией, но мы не знаем параметры камеры и поэтому используем простейшую линейную интерполяцию( $y$  - координата на изображении отсчитывая от нижней точки,  $h$  - высота блока):

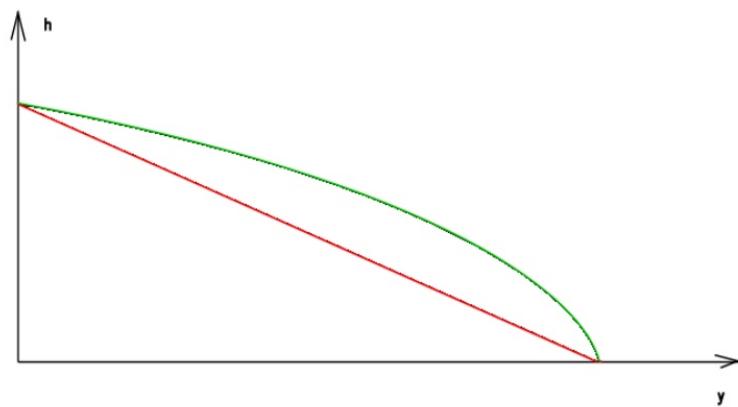


Рис.2.2. Размер горизонтального блока

Формула для расчёта размера блока в зависимости от координаты y:

$$\text{blockSize} = \text{minBlockSize} + \frac{\text{maxBlockSize} - \text{minBlockSize}}{\text{imageHeight} - \text{maxBlockSize}} y, \text{ где } \text{minBlockSize} = 7\text{pix}, \text{maxBlockSize} = 45\text{pix}.$$

Данные значения были выбраны из соображений минимизации процента ошибочных предсказаний стрелок и максимизации процента предсказания реальных стрелок. В идеале размер горизонтального блока нужно рассчитывать с использованием правил перспективного проецирования[4], но для этого необходимо знать параметры камеры(угол наклона, расстояния до земли). В нашем случае эти параметры не известны, поэтому blockSize рассчитывается просто с помощью линейной интерполяции в зависимости от координаты Y.

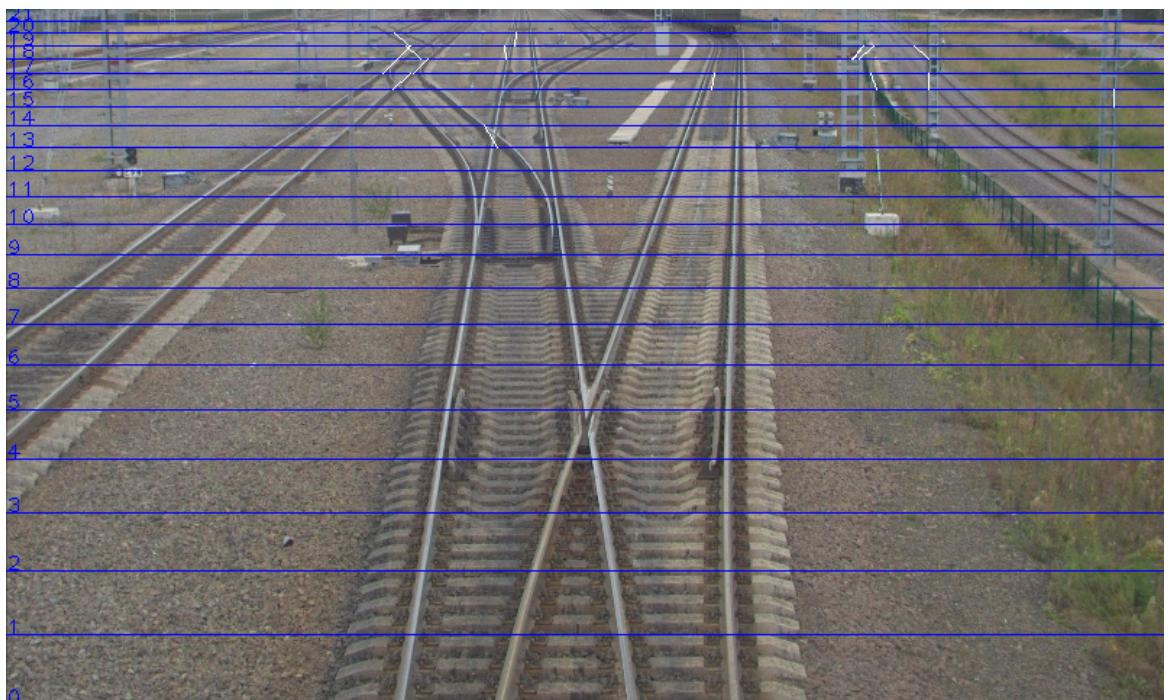


Рис.2.3. Горизонтальные блоки

### 2.1.3. Поиск рельсов

Теперь рассмотрим процесс поиска рельсов на изображении, полученном после применения алгоритма Canny2.1.

После разбиения 2.3 в каждом из горизонтальных блоков ищутся прямые линии с помощью алгоритма Хафа [1]. Можно заметить, что прямые близкие к горизонтальным можно отсекать, т.к. они не могут относиться к рельсам. Таким образом, будем искать прямые с помощью модифицированной алгоритма Хафа, в котором  $\theta \in [0; \frac{\pi}{3}] \cup [\frac{2\pi}{3}; \pi]$ . В качестве порогового значения аккумулятора будем использовать:  $\frac{2 \cdot blockSize}{3}$ , это нужно для отсечения маленьких прямых.

Результат применения алгоритма(красным выделены найденные прямые линии в каждом блоке):

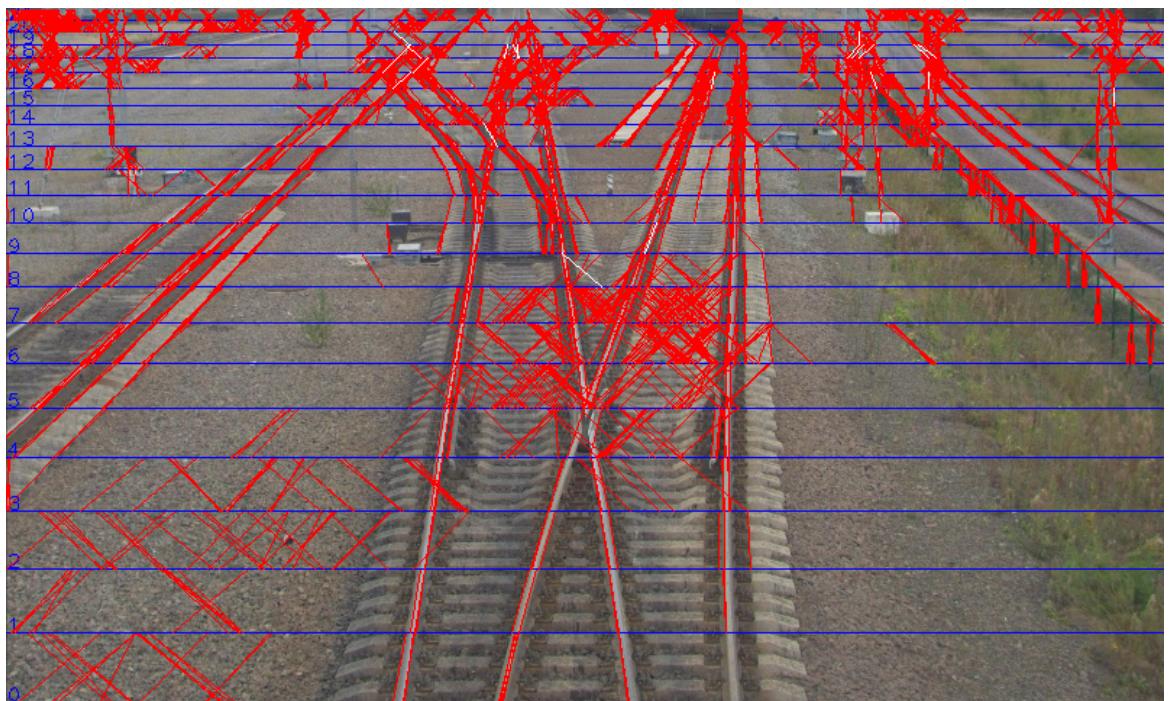


Рис.2.4. Найденные прямые

Можно заметить много прямых образованных на шпалах(между рельсами). Это связано с тем, что на шпалах находится много граничных точек<sup>2.1</sup>. Чтобы минимизировать влияние таких прямых произведем простое удаление горизонтальных рёбер: из изображения, полученного с помощью преобразования Canny удалим такие горизонтальные участки, на которых ребро встречается 3 и более пикселей подряд. В результате получим такую картинку ребер на изображении:

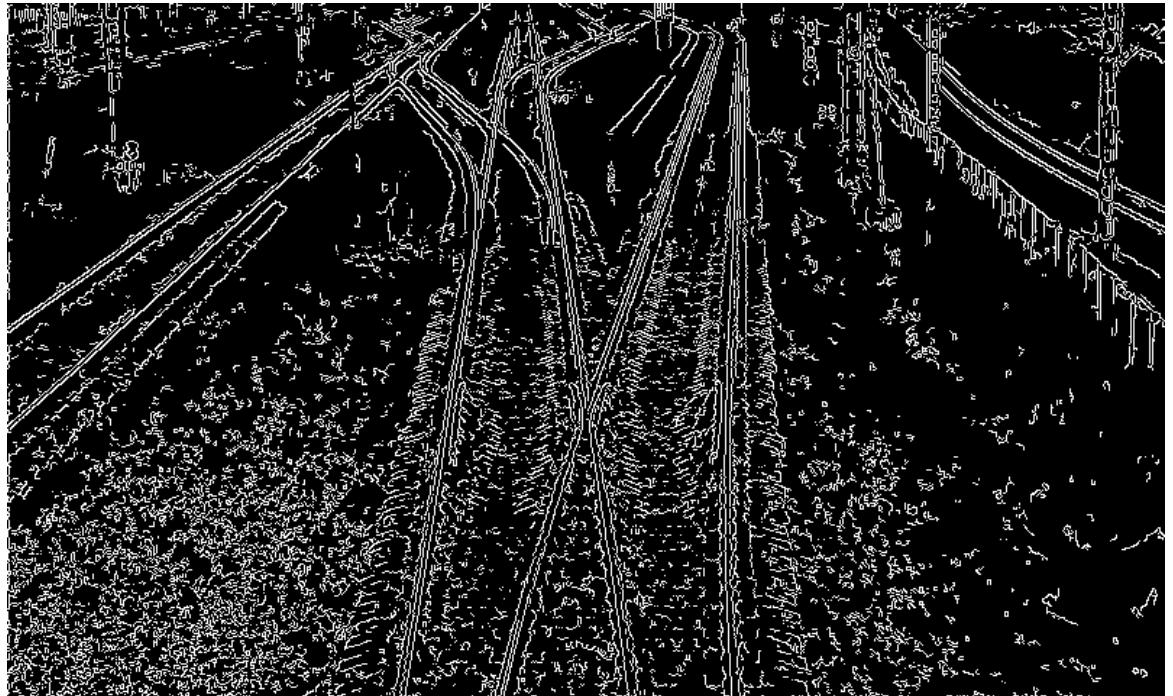


Рис.2.5. Удаление горизонтальных ребер

Также видно, что от одного рельса может появляться сразу несколько прямых. Чтобы этого избежать применим аппроксимацию близких прямых одной средней прямой(координаты такой прямой будут считаться, как центр масс всех близких прямых). Прямые будут считаться близкими, если расстояние между верхними и нижними точками соответствующих прямых меньше заданного:

$$\text{LinesEps} = \text{minLinesEps} + \frac{\text{maxLinesEps} - \text{minLinesEps}}{\text{imageHeight} - \text{maxLinesEps}} y,$$

где  $\text{minLinesEps} = 10$  и  $\text{maxLinesEps} = 20$  - значения полученные с учетом подсчета средней ширины рельса на изображении в зависимости от координаты Y.

Тогда после описанных выше преобразований получим следующий результат:

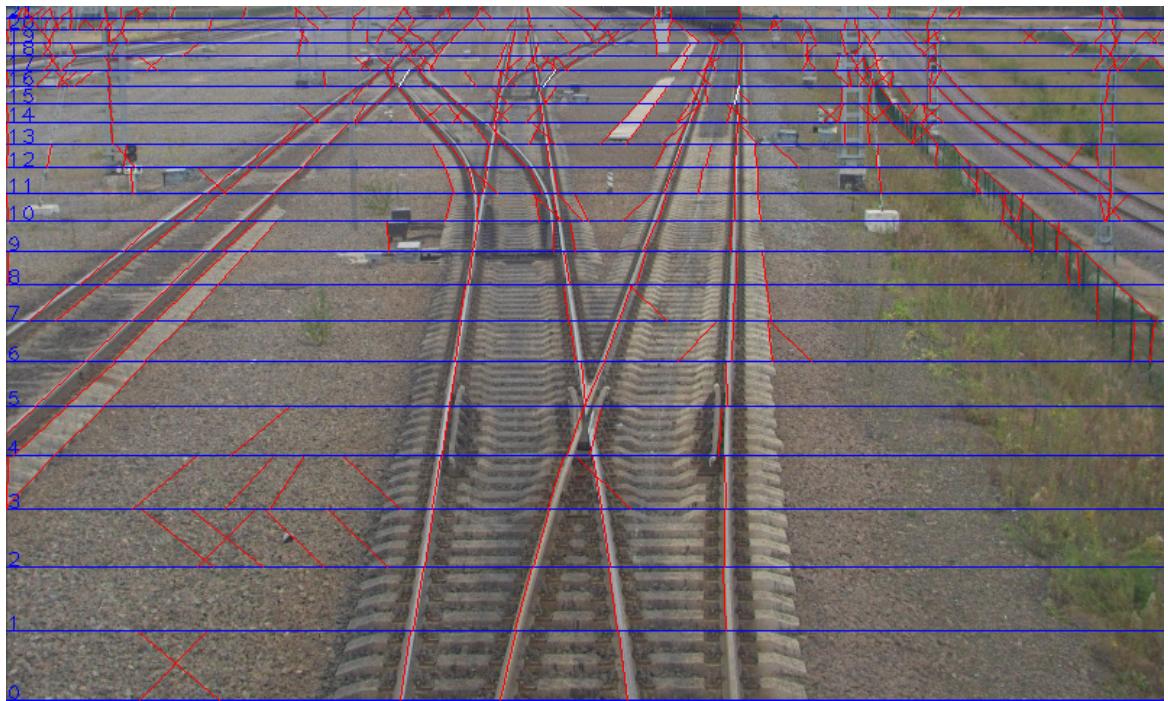


Рис.2.6. Результат поиска рельсов

#### 2.1.4. Поиск стрелок

##### 2.1.4.1. Построение графа прямых

Построим граф, вершинами которого будут найденные прямые. Каждая вершина будет иметь структуру:

$(p1, p2, high\_neighs, low\_neighs)$ , где  $p1$  - нижняя точка прямой,  $p2$  - верхняя точка прямой,  $high\_neighs$  - верхние соседи(прямые, которые выходят из прямой, соответствующей текущей вершине),  $low\_neighs$  - нижние соседи(прямые, которые входят в прямую, соответствующую текущей вершине).

Ребра в графе будут строиться по следующему принципу: начиная с нижнего горизонтального блока, для каждой прямой(*curLine*) ищем верхних соседей в следующем блоке. Прямая(*nextLine*) будет считать верхним соседом, если:

$|nextLine.p1.x - curLine.p2.x| < LinesEps$ , где *LinesEps* - считается также как в параграфе 2.1.3.

Найденные вершины добавляются, как верхние соседи, в текущую вершину. А также текущая вершина добавляется, как нижний сосед, в каждого из верхних соседей.

Например, на рисунке 2.7:

А. прямые 1, 2 являются верхними соседями для прямой 4. А прямая 4 является нижним соседом для прямых 1 и 2.

- B. Прямая 3 лежит дальше, чем blockEps от прямой 4, поэтому между ними нет связи в графе.
- C. прямая 4 является верхним соседом для прямых 5 и 6, но не является верхним соседом для прямой 7. А прямые 5 и 6 в свою очередь являются нижними соседями для прямой 4.

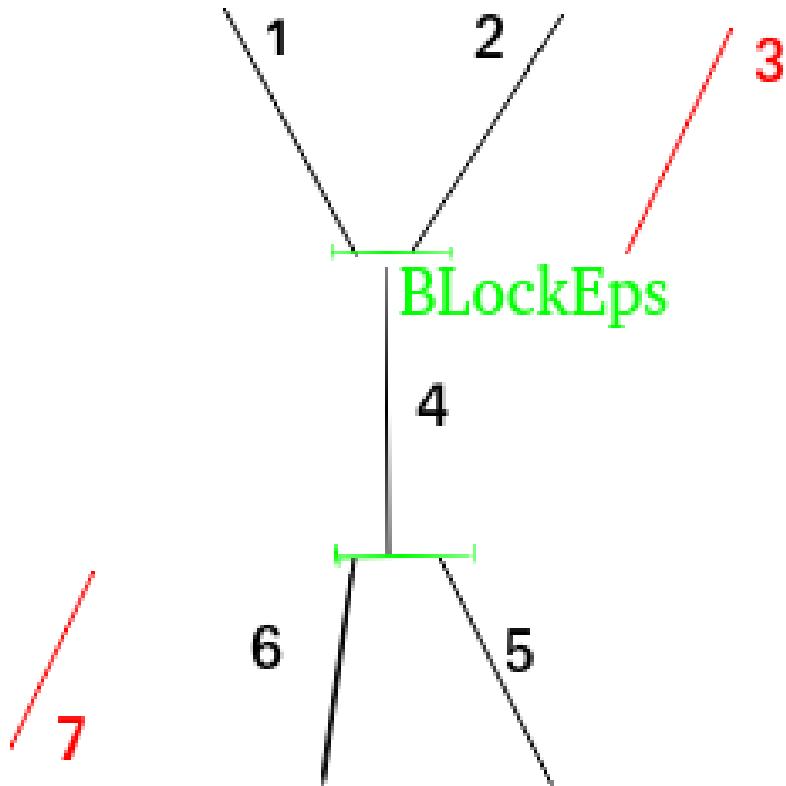


Рис.2.7. Создание ребер

#### 2.1.4.2. Поиск стрелок в графе

В этом параграфе будет рассмотрен алгоритм поиска стрелок в построенном на предыдущем шаге графе.

Алгоритм поиска пересекающихся вершин в графе:

В цикле для каждой вершины графа производим следующие действия:

- A. Если вершина не имеет параллельного нижнего соседа - переходим к следующей вершине
- B. Для каждой пары верхних соседей текущей вершины проверяем, пересекаются ли они(2.1.4.4), если да - верхнюю точку текущей вершины добавляем в итоговый результат найденных стрелок. Таким образом, будут найдены Y и X пересечения.

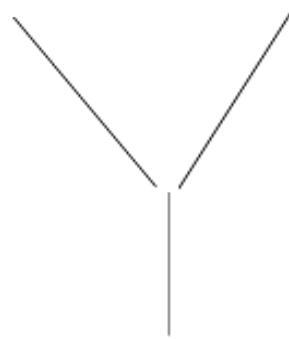


Рис.2.8. Y - образное

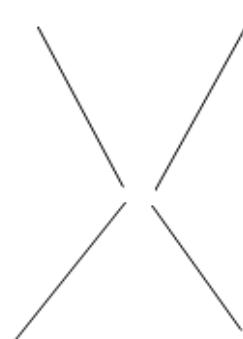


Рис.2.9. X - образное

- C. Пункт 2) повторяется для нижних соседей текущей вершины. Таким образом, будут найдены Y - обратное и X обратные пересечения.



Рис.2.10. Y - обратное

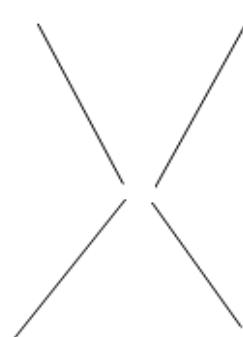


Рис.2.11. X - образное

#### 2.1.4.3. Алгоритм проверки параллельности прямых

Рассмотрим функцию проверки параллельности прямых:

- A. Вычисляем косинусы между прямыми и осью абсцисс.

```

    0x = (1, 0)
    cos1 = cos(line1, 0x)
    cos2 = cos(line2, 0x)
```

- B. Вычисляем разность косинусов

```
    cos_diff = abs(cos1, cos2)
```

- C. Вычисляем насколько близко должны быть прямые, чтобы считать их параллельными. Тут в очередной раз применяется линейная интерполяция. Так как чем ближе рельсы к камере, тем меньше угол должен быть между ними, чтобы посчитать их параллельными. Параметры max\_parallel\_cos и min\_parallel\_cos были получены экспериментально, рассматривая результаты работы алгоритма для крайних случаев реальных параллельных прямых на разных уровнях высоты.

```

    max_parallel_cos = 0.4
    min_parallel_cos = 0.05
    parallel_cos_eps = max_parallel_cos - (
        max_parallel_cos -
        min_parallel_cos) / image_height * current_y
```

- D. Если разность косинусов, вычисленная ранее, меньше parallel\_cos\_eps, прямые считаются параллельными.

```
    return cos_diff < parallel_cos_eps
```

#### 2.1.4.4. Алгоритм проверки пересечения вершин

На вход поступают v1, v2 -вершины, для которых нужно определить пересекаются ли они. А также параметры: глубина проверки пересечений(intersection\_depth) и глубина проверки соседей(check\_neighs\_depth).

Алгоритм:

- A. Если глубина проверки пересечений достигла 0 - возвращаем True - вершины пересекаются.

```

    if intersection_depth == 0:
        return True
```

B. Если прямые, соответствующие вершинам НЕ параллельны - выбираем у каждой прямой параллельного верхнего соседа. Если параллельные соседи существуют, рекурсивно вызываем функцию `is_intersection` с уменьшенной на единицу глубиной поиска пересечений.

```

5   if not is_parallel(v1, v2):
        parallel_1 = v1.getParallelNeigh()
        parallel_2 = v2.getParallelNeigh()
        if parallel_1 and parallel_2:
            return is_intersection(parallel_1, parallel_2,
                                   intersection_depth - 1, check_neighs_depth)

```

C. Если глубина проверки соседей ещё не достигла нуля И у текущих вершин `v1`, `v2` существуют параллельные им верхние соседи, то рекурсивно вызываем `is_intersection` для найденных параллельных верхних соседей с уменьшенной на единицу глубиной проверки соседей.

```

5   if check_neighs_depth > 0:
        parallel_1 = v1.getParallelNeigh()
        parallel_2 = v2.getParallelNeigh()
        if parallel_1 and parallel_2:
            return is_intersection(parallel_1, parallel_2,
                                   intersection_depth, check_neighs_depth - 1)

```

D. Если не было выполнено ни одно из первых трех условий - вершины не являются пересекающимися - вернем `False`.

Описанный выше алгоритм применяется как для проверки пересечения между верхними соседями, так и между нижними соседями.

Рассмотрим алгоритм на примере(синими линиями изображены границы горизонтальных блоков):

В данном примере рассматривается прямая 1.

На вход функции `is_intersection` поступают соседние прямые 2 и 3, и параметры `intersection_depth = 2`, `check_neighs_depth = 1`.

Если прямые 2 и 3 НЕ параллельны, то запускаем `is_intersection` для прямых 4 и 5 и `intersection_depth = 1`, `check_neighs_depth = 1`.

Если прямые 4 и 5 НЕ параллельны, то запускаем `is_intersection` для прямых 6 и 7 и `intersection_depth = 0`, `check_neighs_depth = 1`. Теперь функция возвращает `True` и верхняя точка прямой 1 добавляется, как стрелка в результат.

В случае пересечений между нижними соседями пример будет зеркально отражен по оси Y.

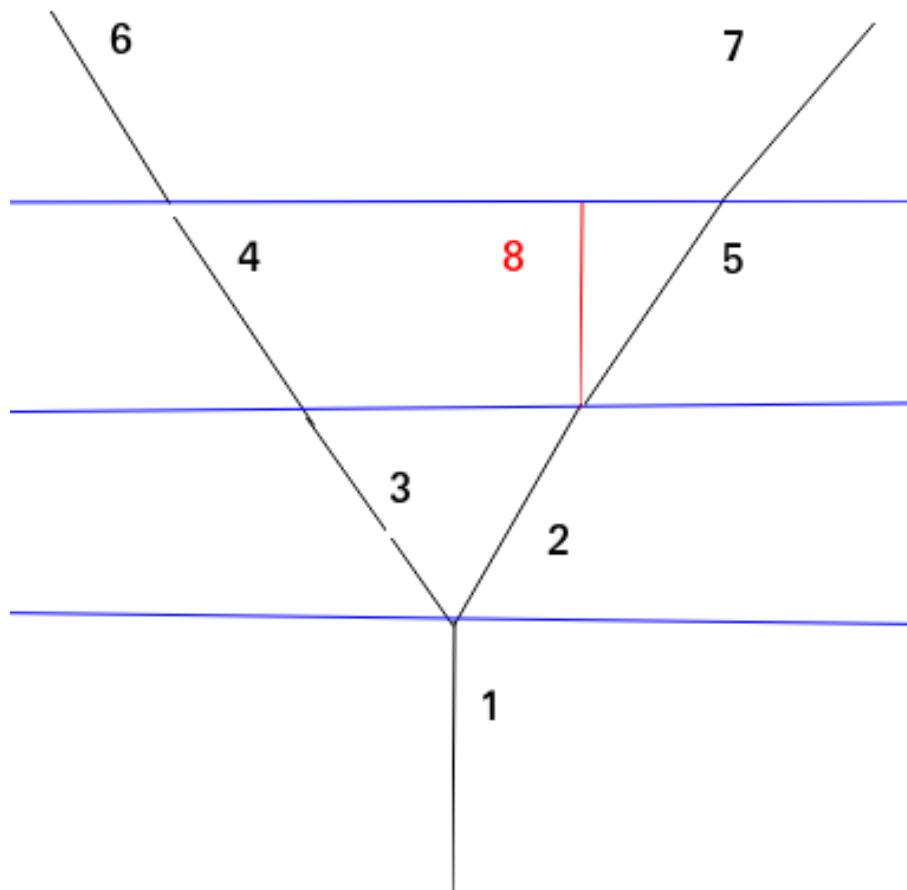


Рис.2.12. Пример

Параметр `check_neighs_depth` нужен в случае, когда сами соседи являются параллельными(прямые 2 и 3), но их продолжения(прямые 4 и 5) уже НЕ являются параллельными. Такая ситуация характерна для Y пересечений, которые находятся близко к камере.

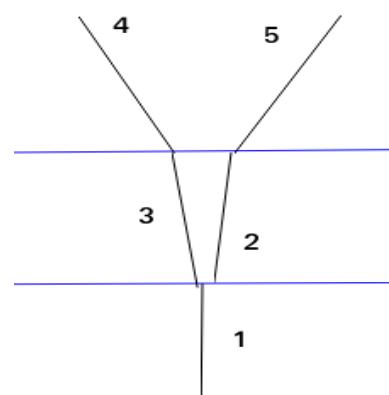


Рис.2.13. Рекурсивная проверка соседей

#### 2.1.4.5. Аппроксимация близких вершин

На последнем этапе близкие по расстоянию стрелки аппроксимируются их центрами масс. Эта процедура повторяется рекурсивно, пока в результирующем массиве все стрелки не станут полностью отделимыми друг от друга.

#### 2.1.5. Сложность алгоритма

1. Применение алгоритма Canny:  $O(H * W * \log(H * W))$
2. Поиск прямых в каждом горизонтальном блоке:  $O(blocksCount * houghComplexity) = O(blocksCount * edgePointsCount^{m-2})$ , где  $m$  - количество параметров перебора(в нашем случае только угол, т.е.  $m = 1$ ). Тогда итоговая сложность операции поиска прямых  $= O(blocksCount * edgePointsCount)$
3. Построение графа соседей и поиск пересекающихся соседей:  $O(blocksCount * linesPerBlock^2)$ .

Итоговая сложность алгоритма:  $O(H * W * \log(H * W)) + O(blocksCount * edgePointsCount) + O(blocksCount * linesPerBlock^2)$ . Таким образом, видно, что в алгоритме нет дорогих по времени операций и поэтому он работает очень быстро. Время работы приведено в таблице 3.1.

## 2.2. Метод с построением SVM-классификатора

Широко известна техника поиска объектов на изображении с использованием обученного SVM[11] - классификатора. Применим данный подход, используя в качестве входных векторов - гистограмму ориентированных градиентов[7] в окрестности размеченных стрелок.

#### 2.2.1. Разметка изображений

Для обучения классификатора необходимо подготовить входной датасет, который должен включать в себя различные виды стрелок(близкие/дальние X-образные, близкие/дальние Y-образные):



Рис.2.14. X - образная  
стрелка



Рис.2.15. Y - образная  
стрелка

Разметка датасета проводилась с помощью утилиты supervisely. Видно, что изображения X-образных стрелок сильно отличается от Y-образных стрелок. Поэтому во время разметки датасета для каждой стрелки указывался её тип(X/Y-образная). Также на первом этапе "отрицательные" примеры, которые тоже необходимы для обучения SVM тоже размечались вручную. Пример разметки изображения(зеленым показаны положительные пример, красным - отрицательные):

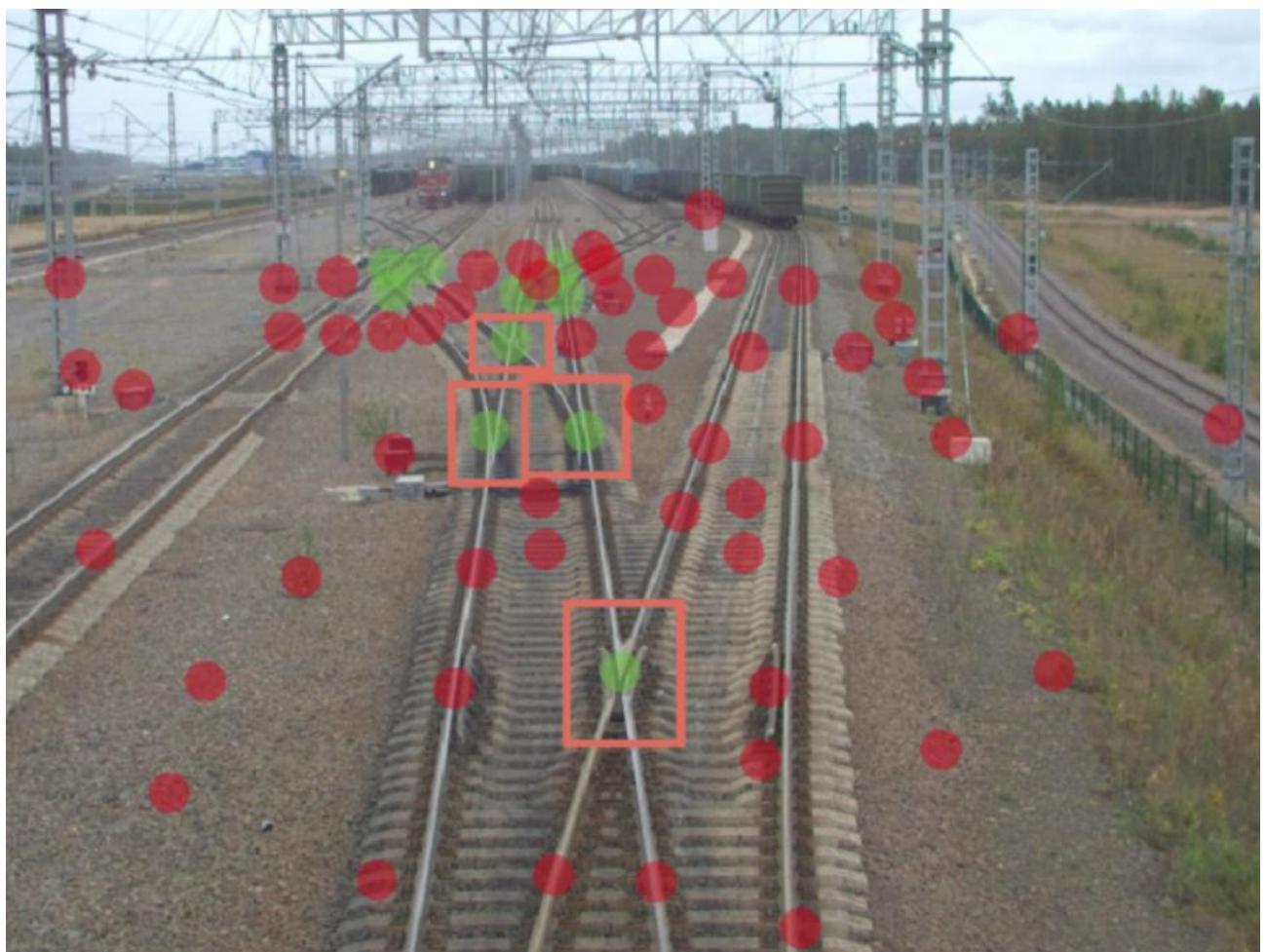


Рис.2.16. Разметка

### 2.2.2. Обучение классификатора

Как было сказано ранее, X и Y образные стрелки сильно отличаются друг от друга. Поэтому было решено делать отдельные классификатора для разных видов стрелок. Также можно заметить, что изображения стрелок вблизи камеры и вдали от камеры отличаются(за счет более детализированной картины близи поезда):



Рис.2.17. Стрелка вблизи камеры



Рис.2.18. Стрелка вдали от камеры

Поэтому были выбраны два размера окна(большое - 48x80pix, маленькое - 12x16pix), и в зависимости от координаты Y размеченной стрелки, окно для подсчета гистограммы ориентированных градиентов данной стрелки масштабировалось к размеру большого либо маленького окна. Если  $y > \frac{1}{5}imageHeight$ , то окно масштабировалось к большему окну, иначе к меньшему. Причём размер окна до масштабирования линейно зависит от координаты y:

$$\begin{cases} winH = minWinH + \frac{maxWinH - minWinH}{imageHeight - maxWinH}y \\ winW = 0.7WinH \end{cases} \quad (2.1)$$

Таким образом, обучались по 2 классификатора(X/Y - образные стрелки) для каждого из двух размеров окна. В итоге были обучены 4 SVM - классификатора: X\_big, X\_small, Y\_big, Y\_small.

### 2.2.3. Предсказание результаты на входном изображении

Поиск стрелок на входном изображении происходил по следующему алгоритму:

```

curWinStep = minWinStep
res = []
for (y = minWinH; y < imageHeight; y += curWinStep)
    curWinStep = minWinStep + (maxWinStep - minWinStep)/(
        imageHeight - maxWinStep)y
    winH = minWinH + (maxWinH - minWinH)/(imageHeight - maxWinH)
    y
    winW = 0.7WinH

```

```

10   for (x = 0.7WinH; x < imageWidth; x += curWinStep)
      feature = HOG(image[x - winW: x][y - winH: y])
      if (svmClassifier.predict(feature))
          res.add((x, y))

```

- A. Начиная с точки с координатой (0.7minWinH, minWinH) вычислялся текущий размер окна<sup>2.1</sup> в зависимости от координаты Y, и вычислялся признак HOG в текущем окне.
- B. После вычисления признака определялось по какую сторону от гиперплоскости, полученной в итоге обучения SVM, лежит значение признака в данной точке и принималось решение можно ли считать данную точку стрелкой.
- C. Далее окно перемещается на текущий шаг - curWinStep.
- D. Алгоритм повторяется для всех точек (x, y) с шагом curStep.

#### ***2.2.4. Первый этап обучения***

На первом этапе положительные и отрицательные примеры были размечены вручную и параметры датасета были следующие:

Object	Count
Y_small	90
Y_big	130
X_small	61
X_big	89
Negative_small	224
Negative_big	394

Таблица 2.1

Параметры датасета

В результате получалось слишком много False Positive предсказаний классификатора на дальних стрелках:

#### ***2.2.5. Второй этап обучения***

Следующим этапом было уменьшение False Positive предсказаний за счет увеличения негативных примеров в обучающей выборке. На этот раз негативные примеры генерировались автоматически на каждом из изображений обучающей выборке. Генерация происходила в виде сетки(для верхней части изображения шаг

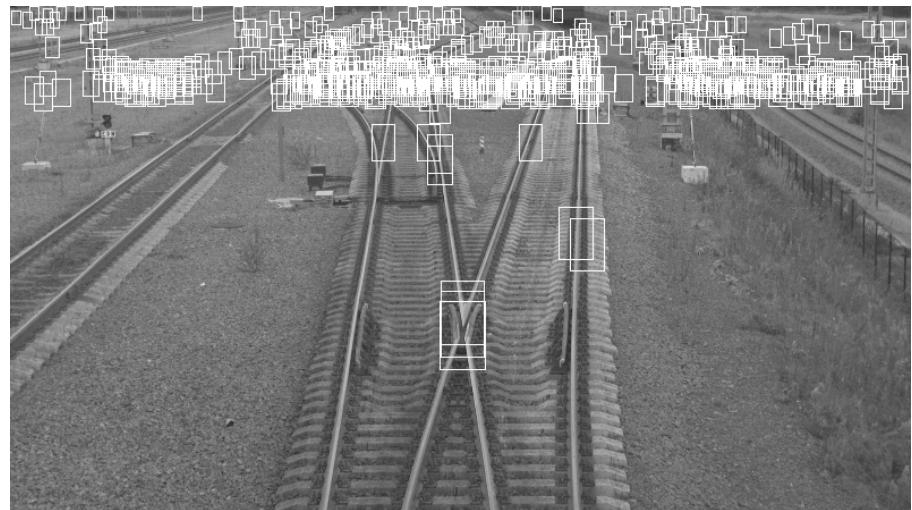


Рис.2.19. Плохой результат

генерации меньше, так как именно там было зафиксировано очень много False Positive предсказаний):

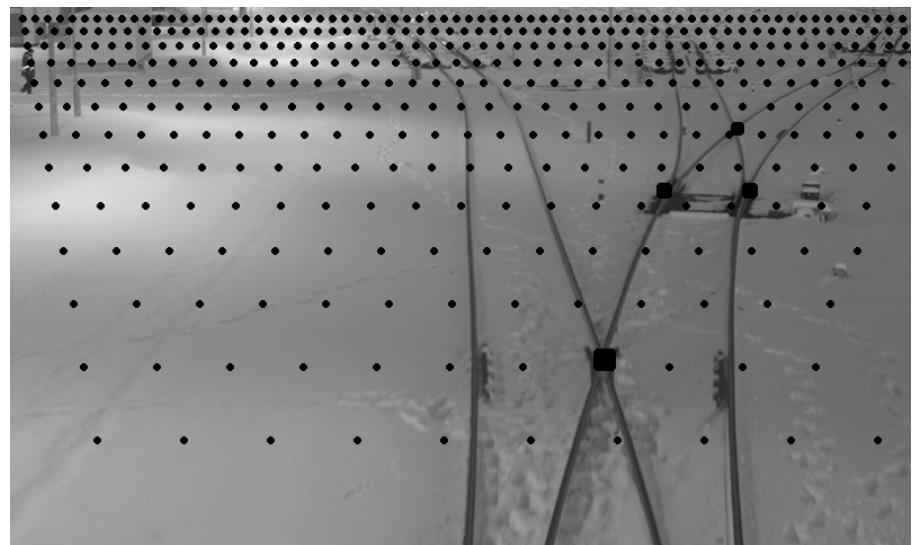


Рис.2.20. Разметка

Также в связи с увеличением кол-ва негативных примеров нужно увеличить кол-во позитивных примеров. Для каждого позитивного примера все точки в некоторой его окрестности тоже добавлялись в обучающую выборку(размер окрестности линейно зависит от координаты Y стрелки):



Рис.2.21. Разметка

Параметры датасета получились следующие:

Object	Count
Y_small	856
Y_big	2724
X_small	384
X_big	1972
Negative_small	75500
Negative_big	146200

Таблица 2.2

Параметры датасета

Результат работы классификатора:

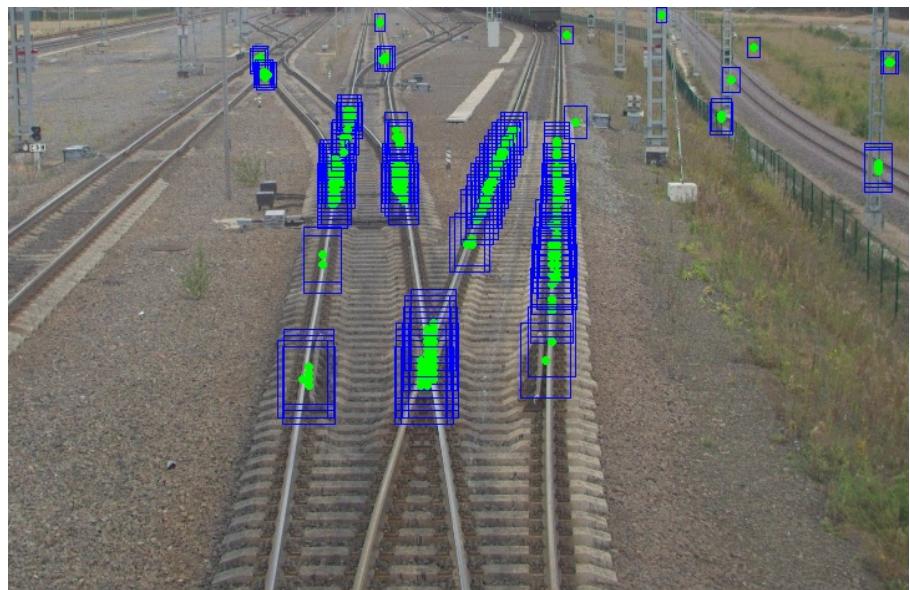


Рис.2.22. Результата классификатора

### 2.2.6. Третий этап обучения

На предыдущем этапе мы избавились от большего числа False Positive предсказаний в дальней части изображения, но при этом теперь рельсы плохо отличаются от стрелок в ближней части изображения. Чтобы этого избежать было решено добавить в обучающую выборку точки, принадлежащие рельсам(но НЕ принадлежащие стрелкам), как негативные примеры. Для этого был разработан алгоритм поиска рельсов на основе метода предложенного в статье[5]. Краткий обзор этой статьи приведен выше в главе 1.2. Результат работы алгоритма поиска рельсов следующий:

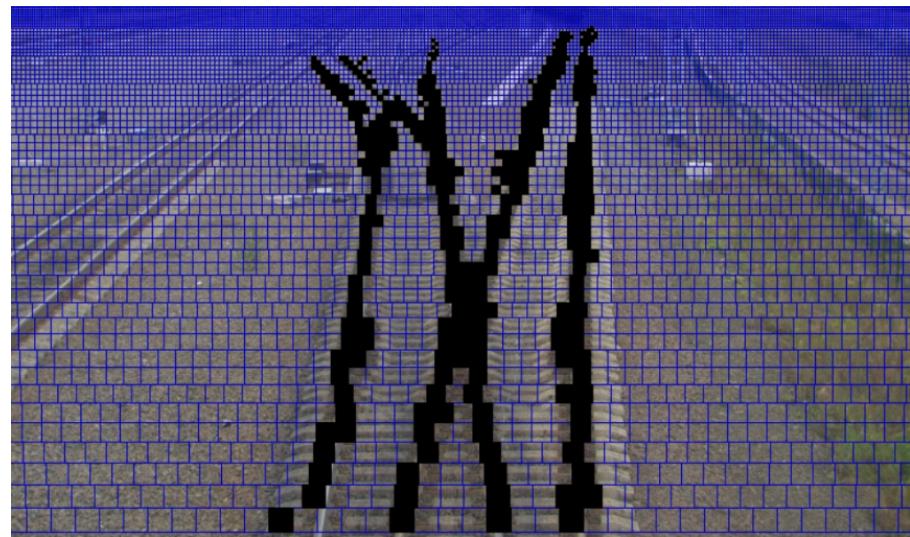


Рис.2.23. Поиск рельсов

Применив данный алгоритм ко всем изображениям из обучающей выборки, и добавив к негативным примерам полученные рельсы получаем следующую картинку разметки каждого изображения:

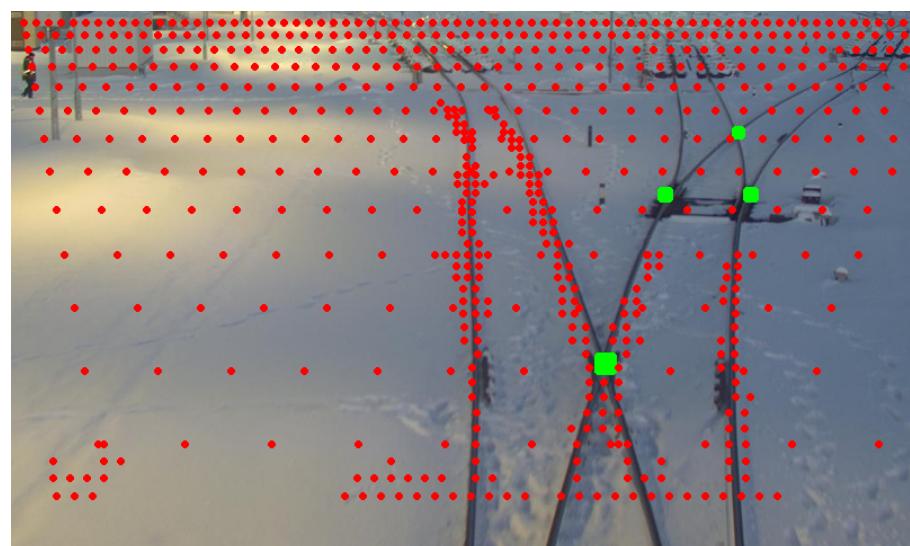


Рис.2.24. Полная разметка

Параметры обучающего датасета:

Результаты предсказания:

Object	Count
Y_small	4106
Y_big	20258
X_small	2260
X_big	12320
Negative_small	132888
Negative_big	133350

Таблица 2.3

Параметры датасета

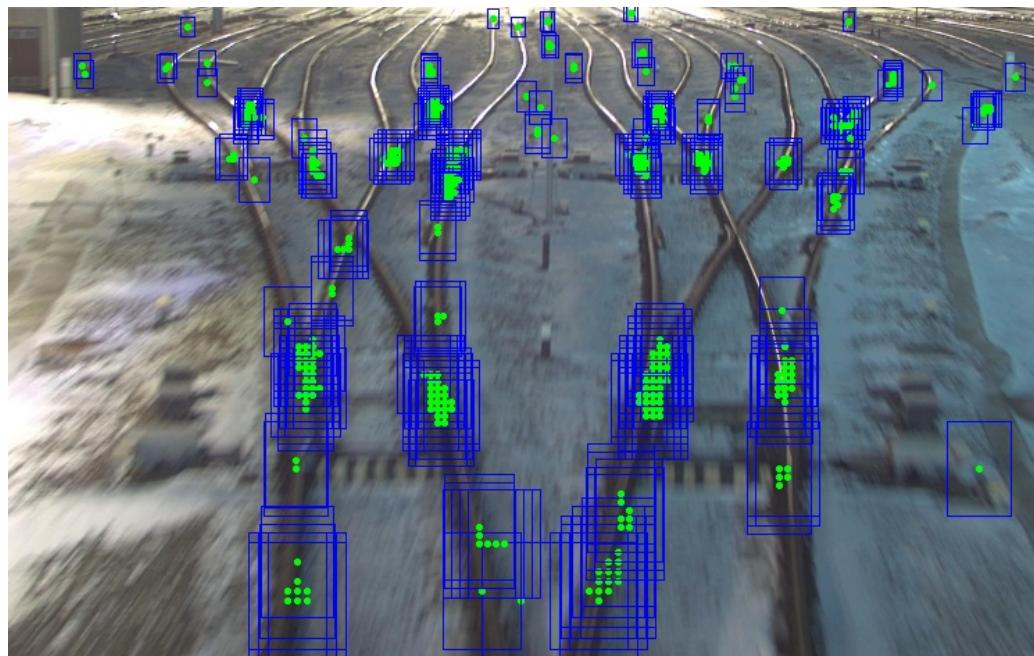


Рис.2.25. Пример результата 1

### 2.2.7. Подтверждение стрелок

Видно, что теперь в большинстве случаев именно в окрестности стрелок высокая концентрация позитивных предсказаний. Теперь предстоит решить задачу выбора таких мест концентрации позитивных предсказаний, в которых действительно находится стрелка.

Рассмотри следующий алгоритм для подтверждения стрелок:

Для каждого изображения из тестовой выборки запускаем алгоритм предсказания стрелок. На каждом из изображений известны ожидаемые положения стрелок. Можно рассчитать, сколько в среднем приходится предсказаний на одну ожидаемую стрелку(сколько предсказанных стрелок находятся в некоторой окрестности ожидаемой стрелки). После применения такой процедуры к каждой ожидаемой

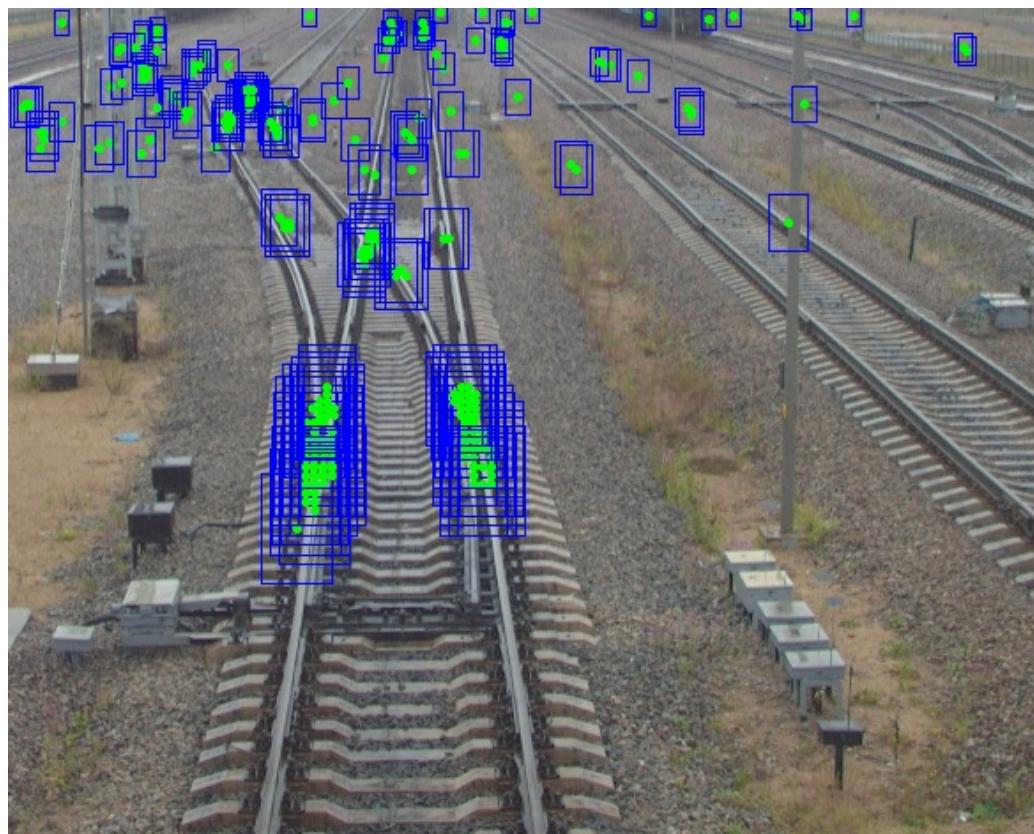


Рис.2.26. Пример результата 2

стрелке, получим среднее число предсказанных стрелок приходящихся на одну ожидаемую стрелку. Назовем это число packageNumber.

Теперь можно применить уже описанный ранее алгоритм кластеризации близких предсказанных стрелок 2.1.4.5. Но уже с подсчетом кол-ва стрелок, которые аппроксимируют одну итоговую стрелку. После чего необходимо отсечь те итоговые стрелки, которые аппроксимируют меньше чем packageNumber предсказанных стрелок. На рисунке ниже зеленым выделены все предсказанные стрелки, а красным - итоговые стрелки, полученные после аппроксимации и отсечения по порогу:

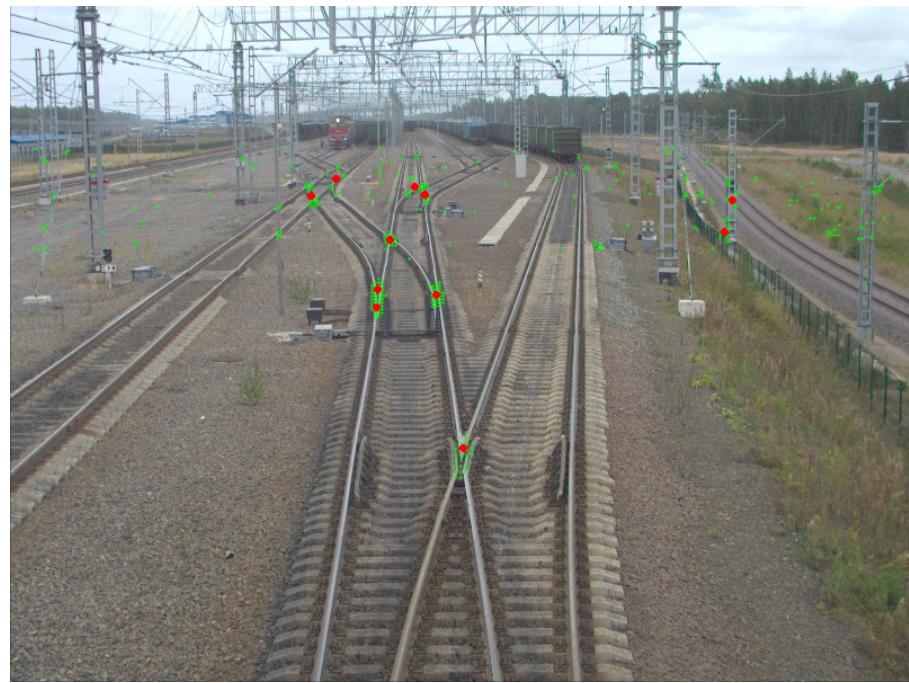


Рис.2.27. Аппроксимация центрами масс

Также для аппроксимации групп близко расположенных стрелок и отсечения выбросов был использован известный алгоритм кластеризации DBSCAN[17].

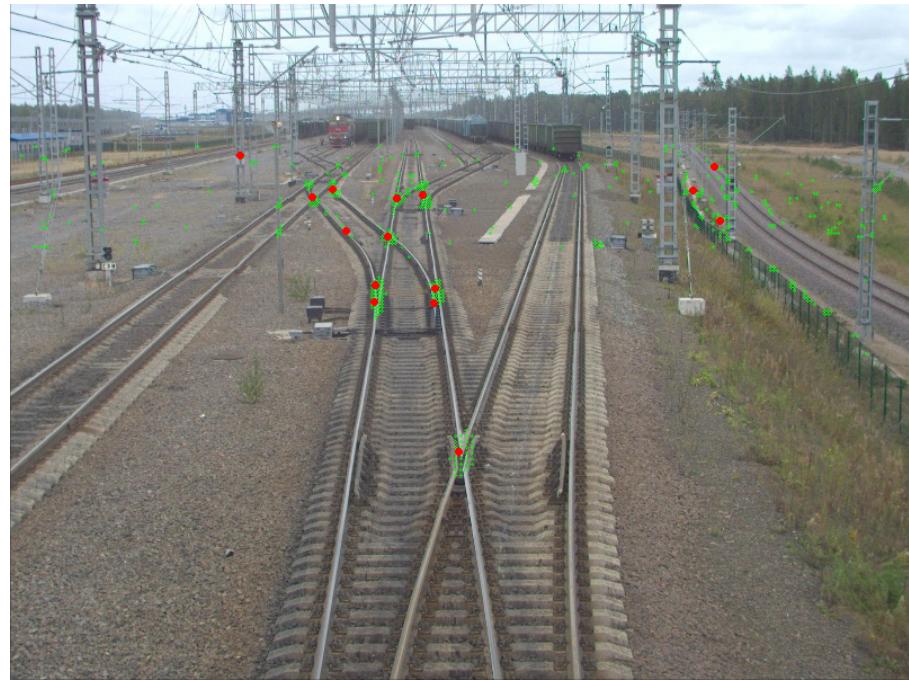


Рис.2.28. DBSCAN

Видно, что оба алгоритма работают приемлемо, в главе 3.3.2 будут рассмотрены статистические результаты работы для обоих алгоритмов кластеризации.

### 2.2.8. Сложность алгоритма

Рассмотрим сложность работы функции предсказания обученного SVM-классификатора.

В каждом окне нужно вычислить `hog=HOG()` и вызвать функцию `Classifier.predict(hog)`. Всего окон получается:

$$windowsCount = \frac{H}{windowSize(y)} * \frac{W}{windowSize(y)} \quad (2.2)$$

, здесь `windowSize(y)` - размер окна в зависимости от координаты Y на изображении.

Рассчитаем размерность вектора HOG для большого и маленького окна. Большое окно имеет размер 80x48pix. Размер блоков, на которые разбивается окно для расчёта HOG - 8x8pix. Для нормализации блоки разбиваются на группы по 2x2 блока(будем называть такие группы большими блоками).

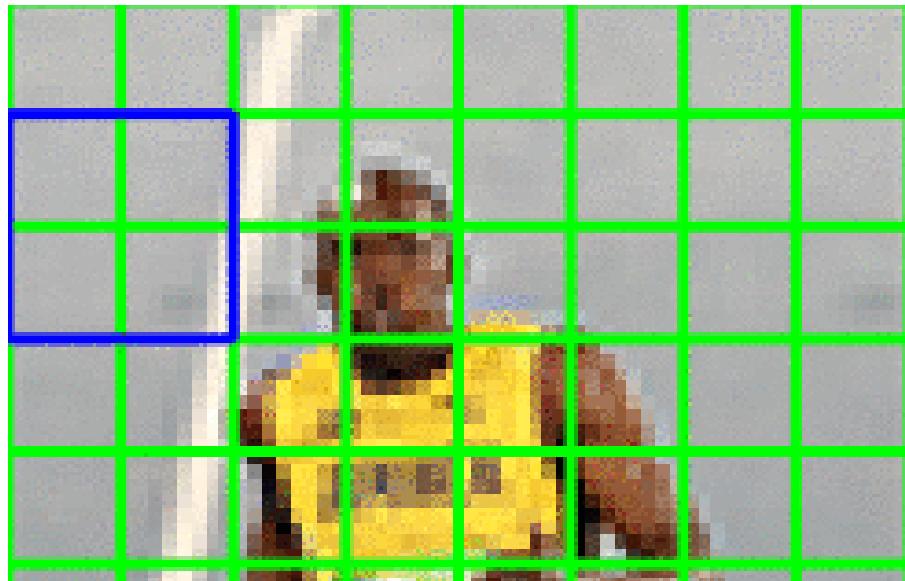


Рис.2.29

Больших блоков всего будет:  $(\frac{80}{8} - 1) * (\frac{48}{8} - 1) = 45$ . Размерность HOG в одном маленьком блоке 8x8pix равна 9(всего 9 направлений). Таким образом в каждом большом блоке гистограмма имеет размерность:  $9 * (2 * 2) = 36$ . Тогда размер вектора HOG для всего большого окна будет:  $45 * 36 = 1620$ . Аналогично можно рассчитать размерность HOG для маленького окна и получить 216(в маленьком окне размерность каждого блока бралась 4x4pix).

Самая дорогая по времени операция это вычисление функции `Classifier.predict(hog)`. Данная операция вызывается для каждого сканирующего окна(их количество

описано выше 2.2). Функция предиктор в случае SVC классификатора выглядит следующим образом:  $\sum_{i \in SV} y_i \alpha_i K(x_i, x) + b$ , где SV - множество опорных векторов ( $x_i$  - i-ый опорный вектор,  $y_i$  - номер класса i-го вектора,  $\alpha_i$  - вес i-ого вектора), полученных в процессе обучения классификатора, K - ядерная функция(в нашем случае используется ядро Гаусса  $\exp^{-\gamma*(x-x_*)^2}$ ). Таким образом, сложность вычисления функции Classifier.predict будет  $O(\text{NumOfSV} * d)$ , где d - размерность вектора x(в нашем случае это размерность HOG), а NumOfSV - кол-во опорных векторов.

Рассмотрим кол-во опорных векторов для нашего обученного классификатора:

- A.  $SV(Y\_big) = 10509$
- B.  $SV(X\_big) = 5163$
- C.  $SV(Y\_small) = 6887$
- D.  $SV(X\_small) = 6381$

Количество 10509, 5163, 6887, 6381 объяснить аналитически нельзя, так как опорные вектора выбираются так, чтобы гиперплоскость, которая строится в процессе обучения SVM проходила максимально оптимально(решается задача оптимизации и заранее нельзя сказать сколько будет опорных векторов, это тесно связано с конкретным датасетом, с тем, как расположились обучающие вектора в пространстве).

Таким образом, видно, что количество опорных векторов и размерность вектора X для SVM-классификатора достаточно большие, что приводит к долгому времени работы алгоритма. Для самой большой модели Y\_big необходимо выполнить  $10509 * 1620 = 17 * 10^6$  операций.

По результатам тестирования среднее время работы классификатора на изображении составило: от 16 до 42 секунд. Тестирование проводилось на процессоре Intel Core i7, частота 800 MHz - 3.50 GHz на операционных системах Windows/Ubuntu.

## ГЛАВА 3. РЕЗУЛЬТАТЫ

### 3.1. Описание данных для тестирования

Для тестирования алгоритма было размечено 192 фотографии ж/д путей, сделанных с локомотива. Часть фотографий была сделана при дневном свете 3.1, а часть при искусственном 3.3. Также имеются изображения, сделанные в летний 3.1 и в зимний период 3.4.

Параметры датасета:

- A. Всего стрелок размечено: 1357
- B. Среднее количество стрелок на одном изображении: 7
- C. Около 70% изображений сделаны в летний период. Около 30% в зимний.

### 3.2. Результаты работы алгоритма, основанного на преобразовании Хафа

#### *3.2.1. Примеры работы алгоритма*

Результаты работы на изображении сделанном при дневном:



Рис.3.1. Пример 1

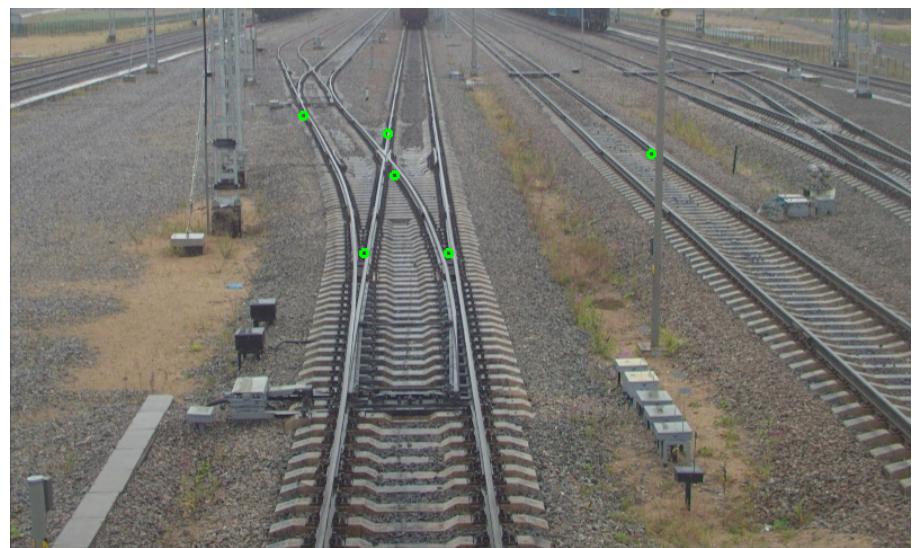


Рис.3.2. Пример 2

Истинное положение стрелок (маркеры) на рисунках 3.1, 3.2, 3.3, 3.4 - не обозначены, так как их положения очевидны

Видно, что есть большие стрелки - те, которые находятся достаточно близко к камере, алгоритм распознает очень хорошо. С маленькими стрелками есть небольшие проблемы, например на изображении 3.2 самая дальняя маленькая стрелка слева не распозналась.

Также видно, что достаточно много false positive стрелок получившихся от столбов и заборов. Один из способов их подавления - использовать алгоритм Region growing up([5]) для сегментации изображения снизу вверх.

Теперь рассмотрим результат работы алгоритма при других погодных условиях и искусственном свете. Видно, что алгоритм хорошо справляется и с такими изображениями.

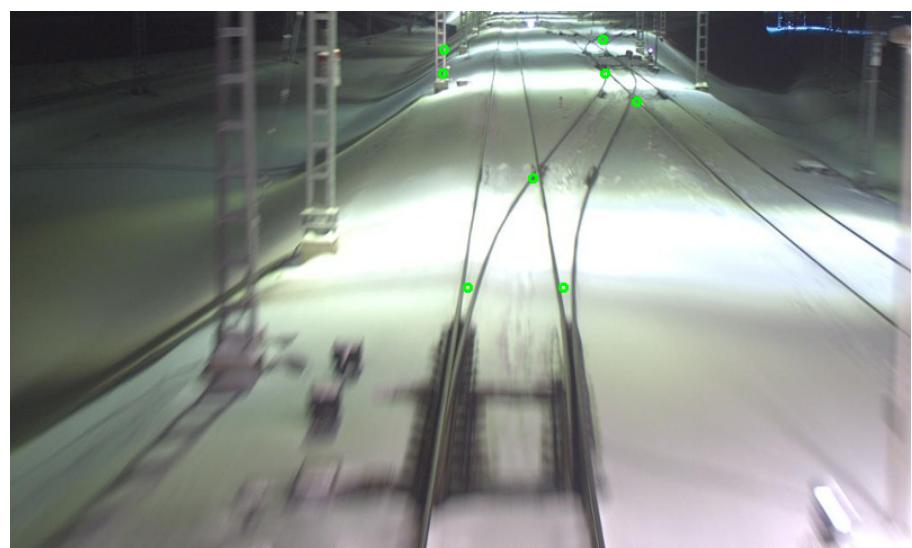


Рис.3.3. Искусственный свет

Теперь посмотрим на пример изображения, на котором алгоритм выдал очень плохой результат: Видно, что на изображении есть ярко освещенные солнцем



Рис.3.4

участки, и темная тень от поезда, в связи с этим алгоритм Отцу [3] выдал достаточно высокое значение пороговых фильтров и рельсы, находящиеся в тени не были восприняты как рёбра алгоритмом Canny[2]. В следствии чего соответствующие рельсам линии не были найдены алгоритмом Хафа [1].

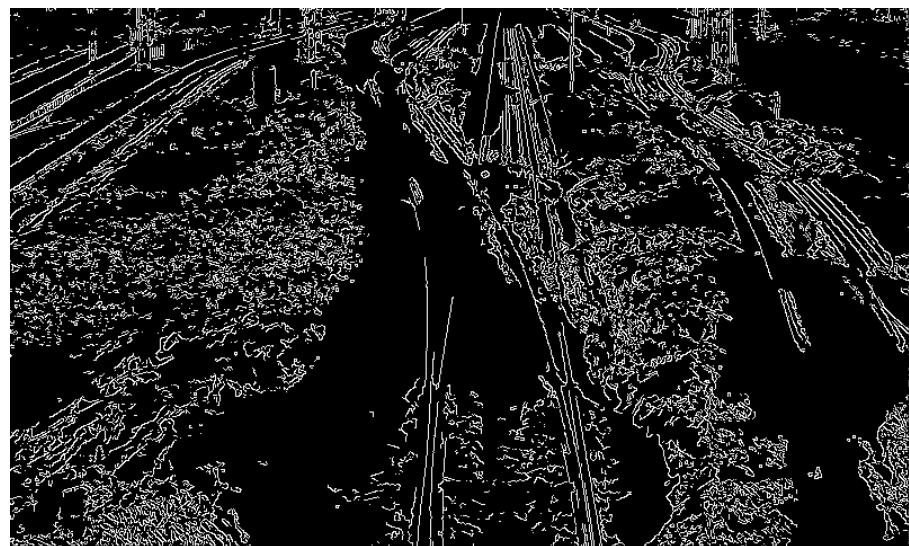


Рис.3.5. Плохое нахождение ребер

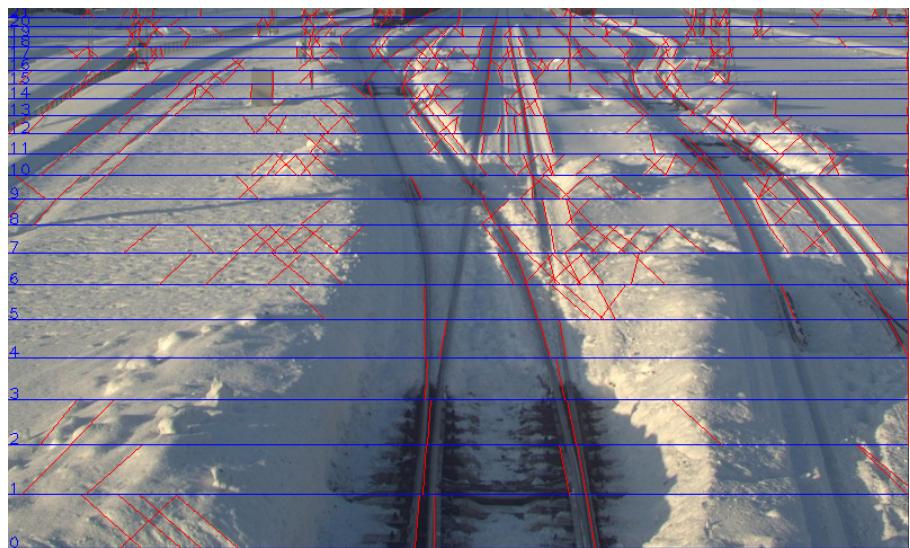


Рис.3.6. Плохое обнаружение линий

### 3.2.2. Статистические результаты

Вспомним о том, что мы не знаем параметры камеры и поэтому ширину горизонтальных блоков выбираем с помощью линейной интерполяции 2.1.4.1. Интерполяция происходит между значениями maxBlockSize и minBlockSize. Рассмотрим зависимость результатов обнаружения от выбора этих параметров(Тестирование проводилось на процессоре Intel Core i7, частота 800 MHz - 3.50 GHz.):

Таблица 3.1

Результат алгоритма на основе преобразования Хафа

<b>minBlockEps</b>	<b>maxBlockEps</b>	<b>time(s)</b>	<b>Precision</b>	<b>Recall</b>
7	45	0.27	0.43	0.76
7	30	0.03	0.34	0.83
20	45	0.02	0.64	0.3

$Precision = \frac{TruePositive}{TruePositive + FalsePositive}$  - процент корректных предсказаний.

$Recall = \frac{TruePositive}{TruePositive + FalseNegative}$  - мера, определяющая как хорошо алгоритм находит позитивные примеры(стрелки)

Видно, что при уменьшении максимального размера блока, уменьшается процент корректных предсказаний. И увеличивается количество найденных положительных примеров.

Также если увеличивать минимальный размер блока, то растет точность, но падает процент найденных стрелок.

При большой нижней границе minBlockEps далекие стрелки вообще не находятся, зато хорошо ищутся близкие стрелки, вот пример для minBlockEps = 20pix:



Рис.3.7. Большие горизонтальные блоки

### 3.3. Результаты работы алгоритма с построением SVM-классификатор

#### 3.3.1. Примеры работы алгоритма

Летнее время года:



Рис.3.8. Лето

Зимнее время года:



Рис.3.9. Зима

Искусственный свет:



Рис.3.10

Видно, что в целом алгоритм устойчив к различным условиям съёмки и находит большинство стрелок на изображении. Рассмотрим подробнее статистические результаты.

### 3.3.2. Статистические результаты

Рассмотрим зависимость точности результата от выбора алгоритма кластеризации и параметров этого алгоритма.

Алгоритм 2.2.7 показал, что в среднем на одну ожидаемую стрелку приходится 13 предсказанных стрелок в окрестности. Но если использовать эту границу при отсечении, мы потеряем многие стрелки, которые были предсказаны меньшим числом голосов. Поэтому необходимо выбрать такую границу, при которой результат работы алгоритма был бы лучшим. Поэтому было проведено исследование зависимости результата(Precision/Recall) от порога предсказанных в окрестности стрелок.

Результаты при применении алгоритма кластеризации основанного на аппроксимации центрами масс:

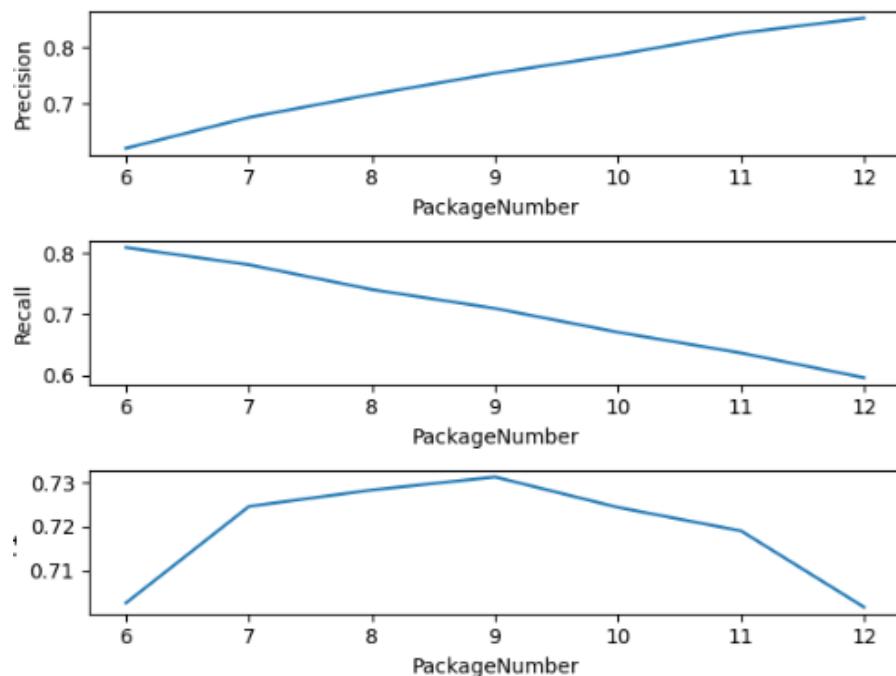


Рис.3.11. Центры масс

Результаты при применении алгоритма кластеризации DBSCAN[17](он тоже принимает на вход минимальное кол-во точек в кластере):

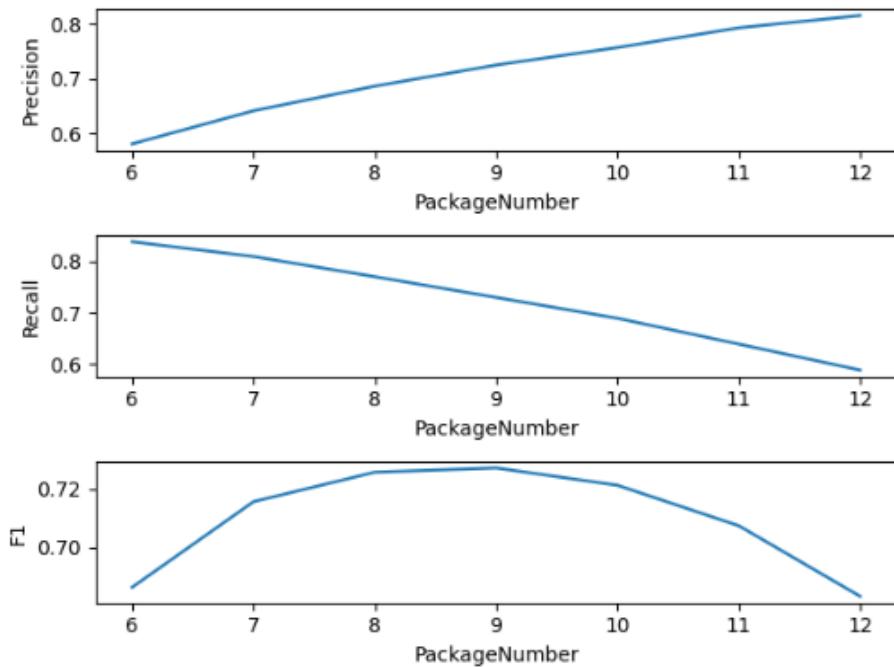


Рис.3.12. DBSCAN

$$F1 = \frac{precision * recall}{precision + recall} \quad (3.1)$$

- среднее гармоническое между precision и recall. Эта метрика дает возможность судить о балансе между precision и recall. В лучшем случае  $F1 = 1$  - идеальный баланс между precision и recall.

Видно, что при увеличении требуемого для подтверждения кол-ва предсказанных стрелок в окрестности(PackageNumber) растет точность предсказаний(Precision) и падает процент найденных стрелок(Recall).

Также по графику зависимости  $F1(PackageNumber)$  видно, что наибольшее значения  $F1$  достигает при  $PackageNumber = 9$ . Если взять  $PackageNumber = 9$ , то получим:

Precision(the percentage of predictions are correct): 0.76

Recall(measures how good you find all the positives): 0.71

F1(perfect precision and recall): 0.73

Именно такие параметры являются лучшими с точки зрения метрики F1.

Причём видно, что оба алгоритма кластеризации дают приблизительно одинаковые результат(F1=[0.72,0.73]).

Однако есть один существенный недостаток этого алгоритма. Если шаг, с которым окно перемещается в процессе предсказания результата очень мал, то количество необходимых вызовов функции предиктора у SVM-классификатора очень большое, что приводит к долгому времени работы.

Поэтому было решено сделать шаг скользящего окна большим и при этом уменьшить порог находящихся рядом предсказанных стрелок(PackageNumber) до 2(это значение было вычислено аналогично с предыдущим исследованием, исходя из оптимального значения метрики F1). Таким образом, удалось уменьшить время работы в 6.5 раз и при этом не потерять и даже немного улучшить показатель метрики F1 для обоих алгоритмов кластеризации предсказанных точек.

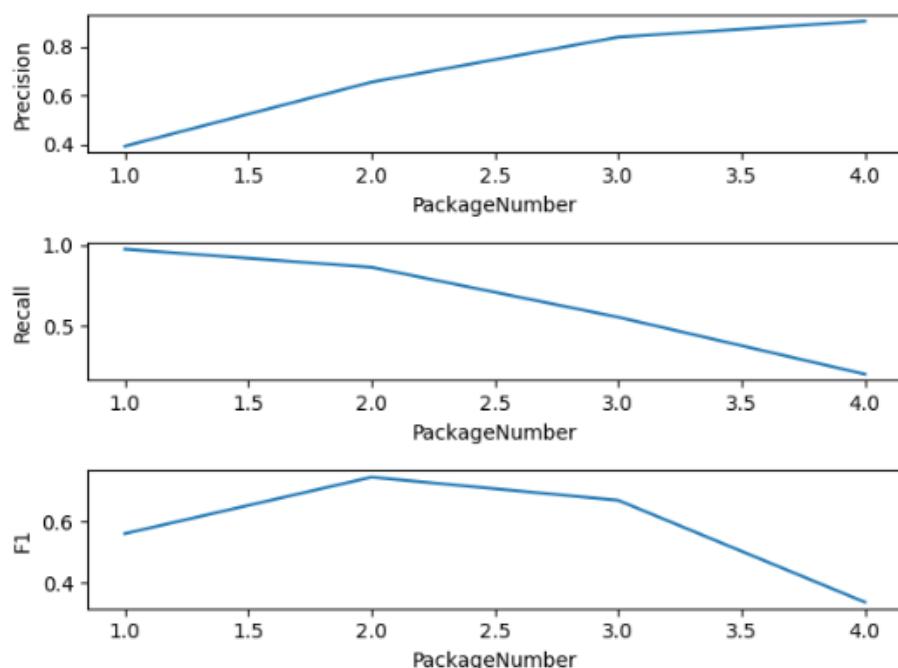


Рис.3.13. Результаты для большого шага окна

Наилучшие показатели достигаются при PackageNumber=2:

Precision: 0.65

Recall: 0.86

F1: 0.74

## ГЛАВА 4. ЗАКЛЮЧЕНИЕ

В ходе работы был размечен датасет из 1357 ж/д стрелок на изображениях, сделанных с локомотива поезда в разных погодных условиях.

Был разработан алгоритм на основе преобразования Хафа2.1, который показал достаточно большую способность к обнаружению стрелок( $\text{Recall}=0.76$ ), но при этом точность предсказаний оказалась очень низкой( $\text{Precision}=0.43$ ) из-за большего количества False Positive обнаружений, которые возникают из-за шума на изображении.

Также был обучен SVM-классификатор2.2, который показал достаточно хорошую способность к обнаружению стрелок( $\text{Recall}=0.86$ ) и при этом хорошую точность предсказаний( $\text{Precision}=0.65$ ).

Алгоритм на основе преобразования Хафа работает в разы быстрее алгоритма на основе SVM-классификатора, но точность предсказания у SVM-классификатора значительно выше.

Таким образом, можно сделать вывод о том, что для задачи поиска ж/д стрелок на изображении метод с обучением показал большую точность и устойчивость к шумам. В дальнейших планах попробовать использовать другие входные вектора для обучения SVM-классификатора и увеличить размер обучающего датасета для повышения вариативности исходных данных.

## ЛИТЕРАТУРА

- [1] **Преобразование Хафа.** Википедия. [Электронный ресурс] URL - [https://en.wikipedia.org/wiki/Hough\\_transform](https://en.wikipedia.org/wiki/Hough_transform)
- [2] **Алгоритм Canny.** [Электронный ресурс] URL - <https://habr.com/ru/post/114589/>
- [3] **Алгоритм Отцу.** [Электронный ресурс] URL - <https://docs.opencv.org/2.4/doc/tutorials/imgproc/threshold/threshold.html>
- [4] **Перспективная проекция.** [Электронный ресурс] URL - <http://stratum.ac.ru/education/textbooks/kgrafic/lection04.html>
- [5] **Efficient railway tracks detection and turnouts recognition method using HOG features.** [Электронный ресурс] URL - <https://link.springer.com/article/10.1007/s00521-012-0846-0>
- [6] **Vision based rail track and switch recognition for self-localization of trains in a rail network.** [Электронный ресурс] URL - <https://ieeexplore.ieee.org/document/5940466?denied=>
- [7] **Histogram of oriented gradients.** Википедия. [Электронный ресурс] URL - [https://en.wikipedia.org/wiki/Histogram\\_of\\_oriented\\_gradients](https://en.wikipedia.org/wiki/Histogram_of_oriented_gradients)
- [8] **Scale-invariant feature transform.** [Электронный ресурс] URL - [https://en.wikipedia.org/wiki/Scale-invariant\\_feature\\_transform](https://en.wikipedia.org/wiki/Scale-invariant_feature_transform)
- [9] **Using Partial Edge Contour Matches for Efficient Object Category Localization**, Hayko Riemenschneider, Michael Donoser and Horst Bischof Proceedings of European Conference on Computer Vision. [Электронный ресурс] URL - <http://www.icg.tugraz.at/Members/hayko/partial-contour-efficient-matching>
- [10] **Speeded up robust features.** [Электронный ресурс] URL - [https://en.wikipedia.org/wiki/Speeded\\_up\\_robust\\_features](https://en.wikipedia.org/wiki/Speeded_up_robust_features)
- [11] **A Library for Support Vector Machines.** [Электронный ресурс] URL - <https://www.csie.ntu.edu.tw/~cjlin/papers/libsvm.pdf>

- [12] **Decision tree.** [Электронный ресурс] URL -  
[https://en.wikipedia.org/wiki/Decision\\_tree](https://en.wikipedia.org/wiki/Decision_tree)
- [13] **K-nearest neighbors algorithm.** [Электронный ресурс] URL -  
[https://en.wikipedia.org/wiki/K-nearest\\_neighbors\\_algorithm](https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm)
- [14] **Sliding window.** [Электронный ресурс] URL -  
<https://www.pyimagesearch.com/2015/03/23/sliding-windows-for-object-detection-with-py>
- [15] **Robust lane detection and tracking for lane departure warning.** [Электронный ресурс] URL - <https://ieeexplore.ieee.org/document/6384266>
- [16] **Double Lane Line Edge Detection Method Based on Constraint Conditions Hough Transform.** [Электронный ресурс] URL -  
<https://ieeexplore.ieee.org/document/8572535>
- [17] **DBSCAN cluster.** [Электронный ресурс] URL -  
<https://ru.wikipedia.org/wiki/DBSCAN>
- [18] **Цифровая обработка изображений.** Гонсалес Р., Вудс Р. Издание 3-е, исправленное и дополненное Москва: Техносфера, 2012. – 1104 с.
- [19] **Цифровая обработка изображений в информационных системах: Учебное пособие.** Грузман И.С., Киричук В.С., Косых В.П., Перетягин Г.И., Спектор А.А. - Новосибирск: Изд-во НГТУ, 2002. - 352 с.
- [20] **Distribution-dependent Vapnik chervonenkis bounds.** V. Nicolas , A. Robert, European Conference on Computational Learning Theory, pp 230- 240, November 1999.
- [21] **F1 score.** [Электронный ресурс] URL - [https://en.wikipedia.org/wiki/F1\\_score](https://en.wikipedia.org/wiki/F1_score)
- [22] **A New Lane Line Segmentation and Detection Method based on Inverse Perspective Mapping.** [Электронный ресурс] URL -  
<https://pdfs.semanticscholar.org/fb7a/76f7994cfc06783e65c58537876ee7c6af7b.pdf>
- [23] **A FAST DECISION TECHNIQUE FOR HIERARCHICAL HOUGH TRANSFORM FOR LINE DETECTION.** [Электронный ресурс] URL -  
<https://arxiv.org/ftp/arxiv/papers/1007/1007.0547.pdf>

- [24] **A dataset for rail scene understanding.** [Электронный ресурс] URL - [http://openaccess.thecvf.com/content\\_CVPRW\\_2019/papers/Autonomous%20Driving/Z](http://openaccess.thecvf.com/content_CVPRW_2019/papers/Autonomous%20Driving/Z)