МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Сыктывкарский государственный университет имени Питирима Сорокина»
(ФГБОУ ВО «СГУ им. Питирима Сорокина»)

Институт точных наук и информационных технологий
Кафедра прикладной математики и информационных технологий в образовании

Допустить к защите
Зав. кафедрой прикладной математики,
к. ф.-м. н.,
_____ А. В. Ермоленко
«____»_____ 2017 года

Дипломная работа

Оптимизация алгоритмов распознавания автомобильных номеров для работы с видеопотоком

Научный руководитель,
к.ф.-м.н., доц.
_____ Н. К. Попова
«____»_____ 2017 года

Исполнитель, студент 145 группы
_____ Е. А. Белых
«____»_____ 2017 года

Сыктывкар, 2017 г.

Содержание

1. Введение

Данная работа посвящена задаче распознавания номеров. Суть этой задачи состоит в следующем: имеется изображение — требуется найти на нем пластину с автомобильным номером и вывести символы, находящиеся на ней, в виде текста.

Несмотря на то, что компьютерные программы, решающие данную задачу уже существуют, некоторые проблемы по-прежнему имеются. Главная из них состоит в том, что качественных материалов, посвященных решению этой проблемы крайне мало. В большинстве статей авторы либо используют OpenCV (фреймворк, содержащий готовые реализации алгоритмов, необходимых для решения), либо утаивают важные детали реализации, без которых проблему решить невозможно, либо просто демонстрируют полное непонимание используемых ими методов.

Первая цель этой работы — написать программу, способную распознавать автомобильные номера, не используя каких-либо сторонних библиотек, кроме стандартных библиотек языка Си и библиотеки для открытия разных форматов изображений. Вторая цель — это попытаться описать используемые методы доступным языком, так как, как уже было сказано выше, при их описании часто упускаются из виду важные детали.

Задача распознавания автомобильных номеров состоит из нескольких этапов: первый — это нахождение пластины с номером на изображении, второй — приведение его к стандартному виду (исправление перспективных искажений, выравнивание освещения, и т. д.), а третий — распознавание символов, находящихся на пластине.

Для решения задачи было выбрано машинное обучение. Метод, с помощью которого, имея большое число эмпирических данных об исследуемой проблеме (в случае с автомобильными номерами — это фотографии, на которых пластины с номером находятся в естественных для них условях), можно найти способ её решения.

В первом разделе описывается, что такое машинное обучение, а также дается значение терминов, используемых в этой работе. Остальные разделы описывают применяемые методы, их реализацию, а также результаты их работы.

2. Необходимая терминология

2.1. Машинное обучение

Машинное обучение — это выявление каких-либо закономерностей по набору эмпирических данных. Т.е. имеется какое-либо явление или объект, а также множество данных, полученных непосредственно с него (это множество называют *обучающей выборкой*), требуется на основе этого множества выявить взаимосвязи, присутствующие в данном объекте или явлении.

Машинное обучение можно разделить на *обучение с учителем* и *обучение без учителя*. В первом случае к каждому примеру из обучающей выборки прилагается ответ, который должна дать обучаемая система, исследуя его, а во втором — системе дается только обучающая выборка.

Распространенным примером обучения с учителем является задача классификации, о которой пойдет речь в следующем разделе. А как пример обучения без учителя, можно привести задачу *обнаружения выбросов*: поиск в обучающей выборке небольшого числа значений, сильно отличащихся от остальных.

2.2. Классификация

Классификация — это задача, суть которой состоит в следующем: имеется множество объектов (обучающая выборка), разделенное на определенные группы (классы), по какому-либо признаку, требуется найти способ, которым можно определить принадлежность к одному из этих классов для произвольного объекта (классифицировать объект) того же типа, что и объекты из обучающей выборки. По сути, задача классификации является разновидностью машинного обучения.

Как пример можно привести распознавание текста, где в качестве классов выступают буквы, цифры, знаки препинания и все остальное (отдельный класс), а изображения с ними — в качестве обучающей выборки. Тогда, чтобы классифицировать изображение, надо определить, изображены ли на ней буква, цифра или знак препинания, если да, то какие.

Еще одним интересным примером является проверка на наличие определенного объекта на изображении, например, лица. В этом случае класса всего два: изображение лица, изображение не лица. Тогда обучающая выборка будет состоять из разных изображений, где на одних лица есть, а на других — нет. Классификация изображения будет состоять в том, чтобы определить, изображено ли на нем лицо.

3. Каскад Хаара

Каскад Хаара — способ обнаружения объектов на изображении, основанный на машинном обучении, идея которого была была предложена в статье за авторством *Пола Виолы (Paul Viola)* и *Майкла Джонса (Michael Jones)*.

Обученный каскад Хаара, принимая на вход изображение, определяет, есть ли на нем искомый объект, т.е. выполняет задачу классификации, разделяя входные данные на два класса (есть искомый объект, нет искомого объекта).

Правильно обученный каскад Хаара имеет хорошую скорость выполнения классификации, а также неплохую устойчивость к разного рода отклонениям. Изначально данный способ был предназначен для обнаружения лиц, однако его можно использовать и для обнаружения других объектов, например, пластины с автомобильным номером.

3.1. Признаки Хаара

Признак Хаара является набором прямоугольных областей изображения, примыкающих друг к другу и разделенных на две группы. Возможных признаков Хаара огромное множество (разнообразные комбинации областей разной ширины и высоты с разными позициями на изображении). Первоначальный набор признаков зависит от реализации и конкретной задачи.

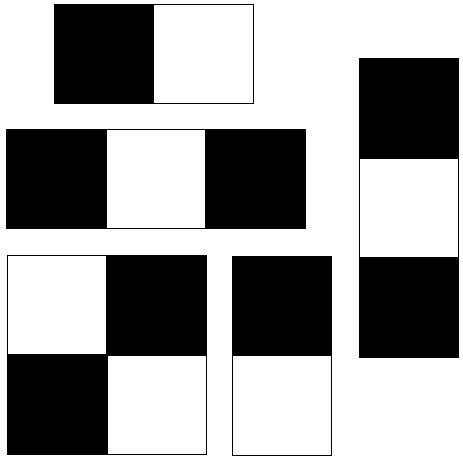

Рис. 1. Комбинации прямоугольных областей, которые использовались при написании этой статьи.

Чтобы вычислить значение конкретного признака Хаара для какого-либо изображения, надо сложить яркости пикселей изображения в первой и второй группах прямоугольных областей по отдельности, а затем вычесть из первой полученной суммы вторую. Полученная разность и есть значение конкретного признака Хаара для данного изображения.

Рис. 2. Признак Хаара на изображении.

Рис. 2 изображает пример признака Хаара на изображении: белые прямоугольники — первая группа группа областей, а черный — вторая. Значение признака Хаара — это разность сумм яркостей пикселей первой и второй группы. В математической форме это будет выглядеть так:

$$a_i = \sum_{i=y_{a_i}}^{y_{a_i}+h_{a_i}-1} \sum_{j=x_{a_i}}^{x_{a_i}+w_{a_i}-1} v_{ij} \qquad b_i = \sum_{i=y_{b_i}}^{y_{b_i}+h_{b_i}-1} \sum_{j=x_{b_i}}^{x_{b_i}+w_{b_i}-1} v_{ij}$$

$$h = \sum_{i=1}^{N_a} a_i - \sum_{i=1}^{N_b} b_i \,,$$

где v_{jk} — яркость пискеля с координатами $[j, k]$, a_i — сумма яркостей пикселей в i-й области первой группы, b_i — сумма яркостей пикселей в i-й области второй группы, h — значение признака Хаара для этого изображения, h_{a_i}, w_{a_i}, h_{b_i} и w_{b_i} — высота и ширина i-х областей первой и второй групп соответственно, y_{a_i} и x_{a_i}, y_{b_i} и x_{b_i} — смещения по оси y и x i-х областей первой и второй групп, а N_a и N_b — количество областей в первой и второй группы.

3.2. Нормализация изображения

Прежде чем классифицировать изображение или использовать его как пример для обучения, это изображение следует нормализовать, т. е. привести к стандартному виду. При использовании каскадов Хаара это означает, что надо перевести изображение из текущей цветовой схемы в черно-белую, а также нормализовать (сделать равным единице) его среднеквадратичное отклонение.

Следует начать с перевода изображения в черно-белую цветовую схему. Так как обычно изображения изначально загружаются в цветовой схеме *RGB*, будет приведена формула только для нее:

$$i[x; y] = \frac{I_r[x; y] + I_g[x; y] + I_b[x; y]}{3} \,,$$

где $i[x; y]$ — пиксель черно-белого изображения, а $I_r[x; y]$, $I_g[x; y]$ и $I_b[x; y]$ — соотвественно красная, синяя или зеленая компоненты изображения.

После этого надо нормализовать среднеквадратическое отклонение. Это необходимо для того, чтобы разница в освещенности на фотографиях оказывала минимум влияния на результат классификации. Для этого следует вычислить среднее арифметическое изображения по следующей формуле:

$$a = \frac{1}{wh} \sum_{i=1}^{w} \sum_{j=1}^{h} x_{ij} \,,$$

а потом найти среднеквадратическое отклонение по формуле:

$$\sigma = \left(\frac{1}{hw} \sum_{i=1}^{h} \sum_{j=1}^{w} (x_{ij} - a)^2 \right)^{\frac{1}{2}}$$

После того, как среднеквадратическое отклонение было найдено, можно его нормализовать, поделив на него значение каждого пикселя в черно-белом изображении:

$$x_{ij} = \frac{x_{ji}}{\sigma}$$

Теперь полученное изображение можно использовать для обучения каскада Хаара. Также эти операции следует выполнять с изображением перед его классификацией.

3.3. Интегральная форма представления изображения

Вычисление значения каждого признака Хаара для изображения требует много операций сложения. Однако, если использовать интегральную форму изображения, количество вычислений можно сильно уменьшить.

В интегральной форме изображения значение каждого пикселя является суммой яркостей этого пикселя и всех пикселей, что находятся выше и левее него (если пиксель с координатами [1;1] находится в верхнем левом углу изображения):

$$S_{yx} = \sum_{i=1}^{y} \sum_{j=1}^{x} x_{ij} ,$$

где S_{yx} — значение пикселя интегральной формы изображения с координатами $[y; x]$, а x_{ij} — значение пикселя исходного изображения с координатами $[i; j]$.

Переведя изображение в интегральную форму, можно вычислять значения признаков Хаара для него, не выполняя суммирование всех требуемых значений яркостей каждый раз по новой. Достаточно посчитать сумму яркостей для каждой прямоугольной области, используя свойства интегральной формы изображения:

$$\begin{cases} a_i = S[y_{a_i} + h_{a_i}; x_{a_i} + w_{a_i}] & , y = 1, x = 1 \\ a_i = S[y_{a_i} + h_{a_i}; x_{a_i} + w_{a_i}] - S[y_{a_i} + h_{a_i}; x_{a_i}] & , y > 1, x = 1 \\ a_i = S[y_{a_i} + h_{a_i}; x_{a_i} + w_{a_i}] - S[y_{a_i}; x_{a_i} + w_{a_i}] & , y = 1, x > 1 \\ a_i = S[y_{a_i} + h_{a_i}; x_{a_i} + w_{a_i}] - S[y_{a_i}; x_{a_i} + w_{a_i}] - S[y_{a_i} + h_{a_i}; x_{a_i}] + S[y_{a_i}; x_{a_i}], & y > 1, x > 1 \end{cases}$$

$$\begin{cases} b_i = S[y_{b_i} + h_{b_i}; x_{b_i} + w_{b_i}] & , y = 1, x = 1 \\ b_i = S[y_{b_i} + h_{b_i}; x_{b_i} + w_{b_i}] - S[y_{b_i} + h_{b_i}; x_{b_i}] & , y > 1, x = 1 \\ b_i = S[y_{b_i} + h_{b_i}; x_{b_i} + w_{b_i}] - S[y_{b_i}; x_{b_i} + w_{b_i}] & , y = 1, x > 1 \\ b_i = S[y_{b_i} + h_{b_i}; x_{b_i} + w_{b_i}] - S[y_{b_i}; x_{b_i} + w_{b_i}] - S[y_{b_i} + h_{b_i}; x_{b_i}] + S[y_{b_i}; x_{b_i}], & y > 1, x > 1 \end{cases}$$

где $S[y; x]$ — значение пикселя интегральной формы изображения. Далее можно вычислить значение признака Хаара как обычно:

$$h = \sum_{i=1}^{N_a} a_i - \sum_{i=1}^{N_b} b_i ,$$

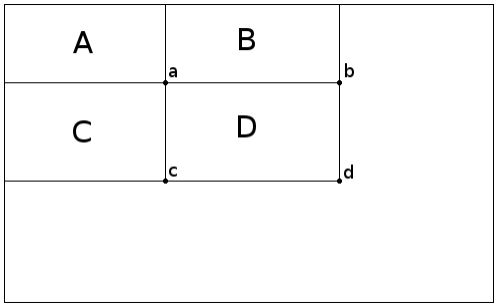

Рис. 3. Вычисление с помощью интегральной формы изображения.

Рис. 3 иллюстрирует, что для того, чтобы вычислить сумму значений пикселей в прямоугольнике D, можно использовать интегральную форму изображения: в ней точка a будет являться суммой значений пикселей в прямоугольнике A, точка b — суммой значений пикселей в прямоугольниках A и B, точка c — суммой значений пикселей в прямоугольниках A и C, а точка d — суммой значений пикселей в прямоугольниках A, B, C и D. Тогда сумма значений пикселей в прямоугольнике D будет равна $d - c - b + a$.

3.4. Алгоритм AdaBoost

Для выбора признаков, лучше всего классифицирующих изображения используется алгоритм *AdaBoost* (adaptive boosting). Этот алгоритм построен на идее, что из большого числа простых способов классификации (называемых *слабыми классификаторами*) можно составить новый способ, выполняющий эту задачу намного эффективнее.

В данном случае слабый классификатор — это функция, которая принимает на вход изображение, вычисляет значение соответствующего ей признака Хаара для этого изображения и сравнивет это значение с порогом, возвращая либо 0, либо 1.

$$h_i(x) = \begin{cases} 1, p_i f_i(x) < p_i \theta_i \\ 0, \text{иначе} \end{cases},$$

где θ_i — порог, x — входное изображение, $f_i(x)$ — значение соотвествующего признака Хаара для изображения x, p_i — направление знака неравенства[1], а $h_i(x)$ — слабый классификатор.

Данный алгоритм перебирает все возможные слабые классификаторы и выбирает те, которые допускают меньше всего ошибок. Важная особенность алгоритма в том, что каждому изображению из обучающей выборки соответсвует определенный вес, и после выбора очередного слабого классификатора веса перераспределяется так, что неверно классифицированные изображения начинают сильнее влиять на значение ошибки. Ниже приведен полный алгоритм:

Входные данные:

h_i — i-й признак Хаара (из всех возможных).
E_i — i-й обучающий пример.
y_i — 0, если i-й обучающий пример отрицательный, и 1, если i-й обучающий пример положительный.
n — количество обучающих примеров.

[1] Принимает одно из двух значений: 1 или −1. Если оно равно 1, ничего не меняется, если же оно равно −1, знак неравенства начинает работать в обратную сторону. Такая запись используется из-за своей краткости, а так же потому, что при реализации на некоторых архитектурах операция умножения работает намного быстрее, чем условные операторы.

C — требуемое число признаков Хаара.

Переменные:

w_i — вес, соответствующий i-му обучающему примеру.
θ_i и p_i — порог и направление знака неравенства для i-го признака Хаара, дающие наименьшую ошибку.
$\varepsilon(h_i)$ — ошибка i-го слабого классификатора.
$\dot{h}_i$ — слабый классификатор, выбранный на i-й итерации.
$\dot{\varepsilon}$ — минимальное значение ошибки слабого классификатора на текущей итерации.
β_i — минимальное значение ошибки слабого классификатора на i-й итерации, представленное в другой форме (для оптимизации вычислений и экономии места на бумаге).

1. Инициализировать веса обучающих примеров:

$$\begin{cases} w_i = \frac{1}{2m}, \text{если } y_i = 0 \\ w_i = \frac{1}{2l}, \text{если } y_i = 1 \end{cases},$$

где m — число отрицательных примеров, а l — число положительных примеров.

2. Для t от 1 до C:
   a. Нормализовать веса обучающих примеров:

$$w_i = \frac{w_i}{\sum_{j=1}^{n} w_j}$$

   b. Найти для каждого признака Хаара, порог и направление знака неравенства такие, чтобы слабый классификатор, составленный из них дал наименьшую ошибку:

$$\dot{h}(x) = \begin{cases} 1, p_j h_j(x) < p_j \theta_j \\ 0, \text{иначе} \end{cases}, \quad \varepsilon(h_j) = \sum_{k}^{n} w_k \left| \dot{h}(E_k) - y_k \right| \text{ — минимальный}$$

   c. Найти классификатор с наименьшей ошибкой:

$$\varepsilon(\dot{h}_t) = \min \varepsilon(h_j)$$

$$\dot{h}_t = \begin{cases} 1, p_n h_n(x) < p_n \theta_n \\ 0, \text{иначе} \end{cases},$$

   где n — номер слабого классификатора, дающего наименьшую ошибку.

   d. Обновить веса обучающих примеров:

$$\beta_t = \frac{\varepsilon(\dot{h}_t)}{1 - \varepsilon(\dot{h}_t)}$$

$$w_i = w_i \beta_t^{1 - \left| \dot{h}_t(E_i) - y_i \right|}$$

3. Итоговый сильный классификатор:

$$H(x) = \begin{cases} 1, \sum_{t=1}^{C} \log(\frac{1}{\beta_t}) \dot{h}_t(E_i) \geq \frac{1}{2} \sum_{t=1}^{C} \log(\frac{1}{\beta_t}) \\ 0, \text{иначе} \end{cases}$$

Стоит отметить, что под "всеми возможными признаками Хаара" подразумеваются признаки Хаара, имеющие заранее определенное взаимное расположение прямоугольников, а также их версии, маштабированные по ширине и высоте так, чтобы те не выходили за границы окна с заданными шириной и высотой. Все обучающие примеры должны иметь такую же ширину и высоту, как и окно.

Полученный сильный классификатор уже способен выполнять задачу классификации, хоть и очень медленно. Значение $a_i = \log \frac{1}{\beta_i}$ далее будет рассматриваться как "коэффициент i-го слабого классификатора".

3.5. Быстрое вычисление наилучшего порога для слабого классификатора

Перебор всех возможных признаков Хаара выполняется за приемлимое время. Однако нахождение наилучшего порога и направления знака неравенства таким же способом требует такого количества времени, что применение на практике каскадов Хаара становится затруднительным (так как порог — число вещественое, количество его возможных значений ограничено лишь особенностями представления вещественных чисел на конкретной машине).

Однако, если обратить больше внимания на то, какую задачу должен выполнять порог в слабом классификаторе, можно заметить, что перебирать все возможные его значения нет нужды.

Если предположить, что веса всех обучающих примеров равны, то порог должен разделять плохие и хорошие примеры по значению признака Хаара таким образом, чтобы с одной стороны было как можно больше хороших, а с другой — как можно больше плохих (какие примеры должны быть с каждой стороны, определяет направление знака неравенства). Но так как веса обычно разные, то задачу надо изменить так: сумма весов примеров, находящихся не по ту сторону порога, по которую им следует находиться, должна быть минимальной.

Так как значение признака Хаара для изображения есть обычное вещественное число, то это можно изобразить так:

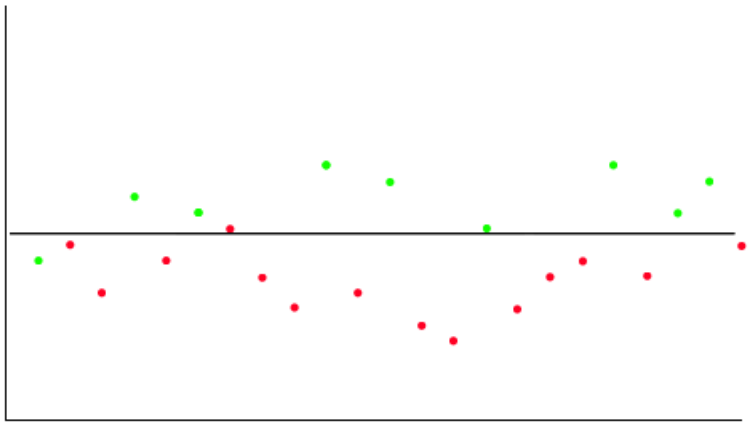

Рис. 4

На рис. 4: по горизонтальной оси расположены номера обучающих примеров, по вертикальной — значения признаков Хаара для соответствующих обучающих примеров. Зеленые точки — значение признака Хаара для хороших примеров, красные точки — для плохих. Горизонтальная линия на графике — один из возможных порогов.

Отсортировав обучающие примеры по их значению признака Хаара, можно получить такую картинку:

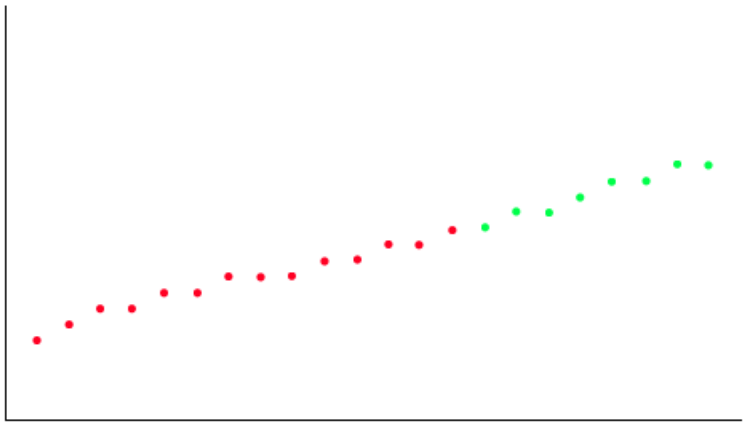

Рис. 5

На рис. 5 по горизонтальной оси расположены номера обучающих примеров, по вертикальной — значения признаков Хаара для соответствующих обучающих примеров. Зеленые точки — значение признака Хаара для хороших примеров, красные точки — для плохих.

Теперь можно заметить, что все пороги, находящиеся между двумя соседними точками на отсортированном графике, дают одинаковые результаты. Следовательно, чтобы получить наименьшую возможную ошибку, достаточно проверить только $n-1$ порогов, где n — количество обучающих примеров, беря как значение порога, например, среднее между двумя соседними отсортированными значениями признака Хаара.

Однако, даже после этого, перебор признаков Хаара занимает очень много времени. Одной из причин является вычисление ошибки слабого классификатора, требующее большое количество операций суммирования и выполняющееся огромное количество раз.

К счастью, есть способ вычислять ошибку слабого классификатора, не выполняя столько операций суммирования. Для этого можно использовать трюк, очень похожий на интегральную форму представления изображения:

h — признак Хаара, для которого ищется порог.
E_i — i-й обучающий пример.
y_i — 0, если i-й обучающий пример отрицательный, и 1, если i-й обучающий пример положительный.
n — количество обучающих примеров.
w_i — вес, соответствующий i-му обучающему примеру.
w^+, w^- — дополнительные массивы.
$\dot{\theta}$ — порог проверяемый на текущей итерации.
$\dot{\varepsilon}$ — ошибка на текущей итерации.
ε — наименьшая полученная ошибка, изначально равна 1.
θ и p — значение порога и направления знака неравенства, при которых была получена наименьшая ошибка.

1. Отсортировать обучающие примеры. Веса и элементы массива y расположить в таком же порядке, как и отсортированные примеры (т.е. чтобы каждому примеру после сортировки соответствовал тот же вес и элемент из y, что и до сортировки).
2. Сделать два дополнительных массива, в первом массиве i-й элемент содержит сумму весов хороших примеров до i-го значения, а во втором — плохих:

$$w^+_i = \sum_{j=1}^{i} \begin{cases} w_j, \text{ если } y_j = 1 \\ 0, \text{ если } y_j = 0 \end{cases}$$

$$w^-_i = \sum_{j=1}^{i} \begin{cases} w_j, \text{ если } y_j = 0 \\ 0, \text{ если } y_j = 1 \end{cases}$$

3. Для всех i от 2 до n:
   a. Вычислить значение проверяемого порога:

$$\dot{\theta} = \frac{h(E_{i-1}) + h(E_i)}{2}$$

   b. Посчитать ошибку для $p = 1$:

$$\dot{\varepsilon} = w^-_{i-1} + w^+_n - w^+_{i-1}$$

   c. Если ошибка проверяемого порога меньше текущей минимальной ошибки, запомнить этот порог, ошибку и направление знака неравенства:

$$\text{если } \dot{\varepsilon} < \varepsilon, \text{тогда } \varepsilon = \dot{\varepsilon}, \theta = \dot{\theta}, p = 1$$

   d. Посчитать ошибку для $p = -1$:

$$\dot{\varepsilon} = w^+_{i-1} + w^-_n - w^-_{i-1}$$

   e. Если ошибка проверяемого порога меньше текущей минимальной ошибки, запомнить этот порог, ошибку и направление знака неравенства:

$$\text{если } \dot{\varepsilon} < \varepsilon, \text{тогда } \varepsilon = \dot{\varepsilon}, \theta = \dot{\theta}, p = -1$$

3.6. Каскад классификаторов

Если уменьшать значение порога *сильного* классификатора, уменьшается количество ложных негативных срабатываний (неверно классифицированных хороших примеров) и увеличивается количество ложных позитивных (неверно классифицированных плохих примеров). Таким образом, даже если сильный классификатор состоит из малого количества слабых классификаторов, понижая порог, ценой большего числа ложных позитивных срабатываний можно свести к минимуму количество ложных негативных.

Используя это свойство, из найденных слабых классификаторов можно составить *каскад*, являющийся набором сильных классификаторов, называемых стадиями, через которые последовательно проходит проверяемое изображение. Каждая стадия содержит больше слабых классификаторов, чем предыдущие. После применения каждой последующей стадии, вероятность ложного позитивного срабатывания снижается, а вероятность ложного негативного остается маленькой.

Проверяемое изображение классифицируется положительно, только если оно дало положительный результат на каждой стадии. Если изображение дало отрицательный результат хотя бы на одной стадии, оно классифицируется отрицательно. Так как на практике проверяется большое количество изображений, лишь малая часть которых содержит искомый объект, большинство отсеивается на ранних стадиях, состоящих из меньшего количества слабых классификаторов.

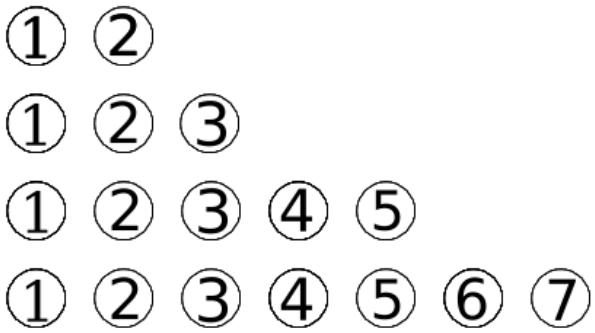

Рис. 6. Иллюстрация структуры каскада.

Рис. 6 иллюстрирует структуру каскада: круги — это слабые классификаторы (цифры в кругах показывают их порядковый номер), каждый ряд кругов — это стадия каскада.

Построить каскад классификаторов можно добавляя к текущей стадии слабые классификаторы и задавая нужный порог, пока доля ложных позитивных срабатываний (отношение количества неверно классифицированных отрицательных примеров к общему их количеству) на этой стадии больше заданного значения. Как только доля ложных позитивных срабатываний стала меньше этого значения, надо перейти к следующей стадии и точно также добавлять к ней слабые классификаторы. Полный алгоритм будет выглядеть так:

N_i — число слабых классификаторов на i-й стадии.
Θ_i — порог на i-й стадии.
n — номер текущей стадии.
ϕ — заданное наперед значение.
p_i — доля ложных позитивных срабатываний i-й стадии.
n_i — доля ложных негативных срабатываний i-й стадии.
a_i — коэффициент i-го слабого классификатора.

1. Установить текущей стадией первую, сделав число слабых классификаторов на ней равным 1:

$$n = 1$$

$$N_n = 1$$

2. Установить для текущей стадии такой порог, чтобы все положительные примеры классифицировались верно:

$$\text{установить } \Theta_n, \text{ такой, что } n_n = 0$$

3. Если доля ложных позитивных срабатываний текущей стадии больше ϕ, увеличить число слабых классификаторов на ней, в противном случае сделать текущей следующую стадию, установив в ней число слабых классификаторов равным числу слабых классификаторов предыдущей, увеличенному на 1:

$$\text{если } p_n > \phi, \text{ тогда } N_n = N_n + 1$$
$$\text{иначе } n = n + 1, N_n = N_{n-1} + 1$$

4. Если число слабых классификаторов на текущей стадии меньше числа доступных слабых классификаторов или доля ложных положительных срабатываний равна 0, перейти к пункту 2.
5. Вернуть стандартный порог последней стадии, если выход из цикла произошел из-за того, что закончились доступные слабые классификаторы:

$$\text{если } p_n > 0, \text{ тогда } \Theta_n = \frac{1}{2}\sum_{i=1}^{N_n} a_i$$

Стоит заметить, что не следует использовать для построения каскада те же хорошие примеры, что и для нахождения слабых классификаторов. Так как слабые классификаторы слишком хорошо обучены для этих примеров, требуемая доля ложных позитивных срабатываний будет достигнута слишком быстро. В этом случае полученный каскад будет содержать мало стадий и начнет допускать большое количество ошибок.

3.7. Оптимизация алгоритма построения каскада классификаторов

Во втором пункте указанного выше алгоритма подразумевается, что требуемый порог ищется перебором, что на практике не очень эффективно. Как и в случае с порогами слабых классификаторов, причиной этому являются большие затраты времени и потери точности.

Однако, с помощью упрощенного варианта оптимизации, использовавшейся для нахождения порогов слабых классификаторов, можно свести затраты времени на построение каскада к минимуму.

Для того, чтобы применить этот метод, надо изменить условия, при которых порог считается наилучшим. Также следует учесть, что у порога сильного классификатора фиксированное направление знака неравенства (>).

Порог слабого классификатора считался наилучшим, если сумма весов неверно классифицированных примеров была минимальной. В сильных же классификаторах наилучший порог тот, при котором верно классифицируются все хорошие примеры, а также максимальное число отрицательных.

Значение, сравниваемое с порогом сильного классификатора, является суммой коэффициентов тех классификаторов, которые верно классифицировали входное изображение:

$$\sum_{i=1}^{C} a_i \dot{h}_i(E),$$

где C — число доступных слабых классификаторов, a_i — коэффициент i-го слабого классификатора, $\dot{h}_i$ — i-й слабый классификатор, а E — входное изображение.

Следовательно, чтобы все хорошие примеры были классифицированы верно, достаточно, чтобы порог сильного классификатора был меньше такой суммы для каждого из них. Или, если выражаться математически, порог должен быть меньше наименьшей среди всех таких сумм для хороших примеров.

Учитывая, что чем порог ниже, тем больше плохих примеров будет классифицировано неверно, условие стоит изменить так: порог должен быть меньше наименьшей среди всех таких сумм для хороших примеров на наименьшее возможное значение (в теории — бесконечно малое, а на практике — зависит от реализации чисел с плавающей точкой на конкретной машине).

При таком условии, в отличие от порогов слабых классификаторов, не требуется ни сортировки, ни проверки выбранного порога. Теперь можно оптимизировать вычислениие сумм для хороших примеров. Для этого следует обратить внимание на тот факт, что при каждой новой итерации число слабых классификаторов на текущей стадии на 1 больше, чем на предыдущей. Тогда вместо того, чтобы каждый раз вычислять сумму заново, можно сделать дополнительный массив (для каждого хорошего примера — соответствующий элемент массива) и при каждой итерации добавлять туда необходимые слагаемые. Ниже приведен алгоритм с учетом всех описанных оптимизаций:

C — число доступных слабых классификаторов.
$\dot{h}_i$ — i-й слабый классификатор.
C_i — число слабых классификаторов на i-й стадии.
Θ_i — порог на i-й стадии.
n — номер текущей стадии.
ϕ — заданное наперед значение.
P_i — доля ложных позитивных срабатываний i-й стадии.
g_i — i-й хороший пример.
a_i — коэффициент i-го слабого классификатора.
S_i — i-я ячейка дополнительного массива, соответствующая i-му хорошему примеру.
δ — миинимальное возможное число, которое можно вычесть из уменьшаемого.

1. Задать каждому элементу дополнительного массива значение 0:

$$S_i = 0$$

2. Установить текущей стадией первую, сделав число слабых классификаторов на ней равным 1:

$$n = 1$$

$$C_n = 1$$

3. Для всех хороших примеров: если последний классификатор текущей стадии классифицировал пример верно, прибавить коэффициент этого классификатора к значению в соответствующей примеру ячейке дополнительного массива:

$$S_i = S_i + \dot{h}_{C_n}(g_i)$$

4. Задать порогу текущей стадии такое значение, меньшее наименьшего элемента дополнительного массива на минимально возможное число:

$$\Theta_i = \min(S_i) - \delta$$

5. Если доля ложных позитивных срабатываний текущей стадии больше ϕ, увеличить число слабых классификаторов на ней, в противном случае сделать текущей следующую стадию, установив в ней число слабых классификаторов равным числу слабых классификаторов предыдущей, увеличенному на 1:

$$\text{если } P_n > \phi, \text{тогда } C_n = C_n + 1$$
$$\text{иначе } n = n + 1, C_n = C_{n-1} + 1$$

6. Если число слабых классификаторов на текущей стадии меньше числа доступных слабых классификаторов или доля ложных положительных срабатываний равна 0, перейти к пункту 2.
7. Вернуть стандартный порог последней стадии, если выход из цикла произошел из-за того, что закончились доступные слабые классификаторы:

$$\text{если } P_n > 0, \text{тогда } \theta_n = \frac{1}{2}\sum_{i=1}^{C_n} a_i$$

3.8. Поиск объектов на большом изображении

Обученный каскад Хаара может классифицировать только изображения того же размера, что и обучающие примеры. Для того чтобы искать объекты на большом изображении, надо по отдельности классифицировать части этого изображения. Для этого используется окно, двигаемое по изображению, размеры которого соотвествуют рабочим размерам каскада. Если какая-либо часть изображения была классифицирована положительно, значит в ней изображен искомый объект.

Также следует построить несколько маштабированных вариантов изображения, называемых пирамидой изображений, и тоже пройтись по ним окном. Это необходимо потому, что изображение искомого объекта совсем необязательно будет того же размера, что и примеры из обучающей выборки.

Рис. 7. Пример пирамиды изображений: размер сторон каждой следующей версии изображения равняется 0.75 размера сторон предыдущего.

3.9. Многопоточное сканирование пирамиды изображений

Сканирование изображения каскдом Хаара, даже с учетом всех оптимизаций — достаточно затратная для процессора задача. Тем не менее, чтобы обнаружать автомобильные номера в реальном времени, оно должно выполняться достаточно быстро — как минимум, несколько раз в секунду. Этого можно достичь благодаря одной важной особенности современных процессоров, а именно, многопоточности.

Под многопоточностью подразумевается способность процессора выполнять несколько задач одновременно. Процессор, обладающий такой особенностью, называется *многоядерным*. Такой процессор, по сути представляет из себя несколько обычных процессоров, называемых ядрами, которые соединены вместе. Каждое ядро выполняет свою задачу независимо от других.

Чтобы многоядерный процессор мог задействовать максимум ресурсов, поставленную ему задачу необходимо разделить на несколько более простых заданий, которые можно выполнять независимо друг от друга. Такие задания иногда называют *потоками* или *нитями*.

С разделением задачи на потоки (*распараллеливанием*) связано множество разнообразных проблем, которые могут поставить под вопрос целесообразность распараллеливания задачи.

Во-первых, после разделения алгоритм может сильно усложниться, поэтому, оно должно выполняться только после того, как первоначальный алгоритм был доведен до рабочего состояния.

Во-вторых, создание потоков и обработка результатов их работы тоже требуют ресурсов и могут свести на "нет" весь получаемый прирост производительности. Этого можно избежать, если распараллеливать задачу так, чтобы потоки как можно дольше могли выполняться независимо.

В-третьих, выполнение задачи будет окончено только тогда, когда будут завершены все потоки. Поэтому, если какие-либо потоки завершат свою работу раньше других, часть ресурсов процессора будет простаивать. По этой причине следует разделять задачу так, чтобы все потоки выполняли свою работу примерно за одно время.

И в-четвертых, может потребоваться, чтобы потоки совместно использовали какие-либо данные, к которым одновременно получить доступ может только один поток. Из-за этого возникает проблема, когда потоки подолгу ждут освобождения доступа к каким либо данным. Также, как и предыдущая проблема, это может сделать распараллеливание бессмысленным. Что бы избежать подобной проблемы, следует разделять задачу так, чтобы как можно больше данных могло использоваться независимо.

К счастью, при распараллеливании алгоритма сканирования пирамиды изображений, большей части этих проблем можно избежать. Для этого следует обратить внимание на структуру пирамиды:

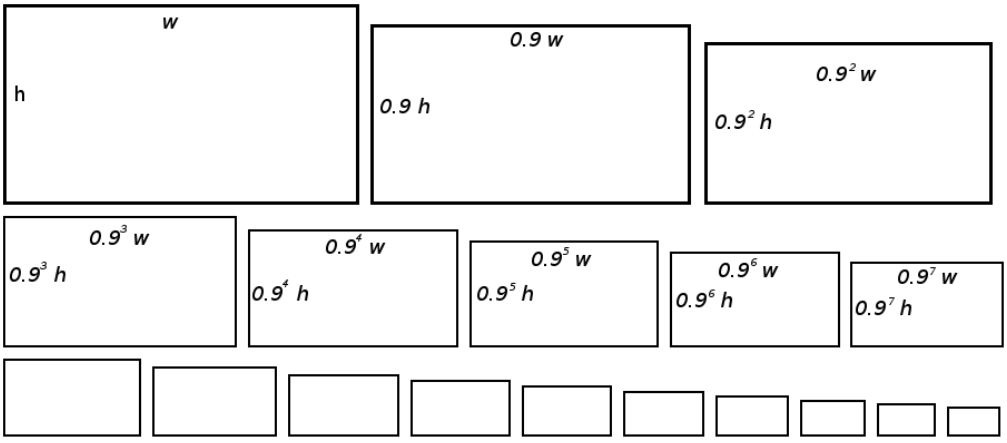

Рис. 8. Пирамида изображений, где ширина и высота каждого изображения составляют 0.9 ширины и высоты предыдущего изображения.

Можно заметить, что каждый масштабированный вариант оригинала, по сути, является отдельным изображением, которое можно сканировать независимо от остальных. Исходя из этого, задачу сканирования пирамиды изображений можно разделить на определенное количество потоков, каждый из которых сканирует свою группу уменьшенных копий.

Также, видно, что для того, что бы построить какое-либо изображение пирамиды, не обязательно строить предыдущее. Это следует из того, что масштабирование является умножением ширины и высоты изображения на число, меньшее единицы, называемым коэффициентом масштабирования, с последующей интерполяцией. Так как в данном случае это число не меняется (на рис. 8 — это 0.9), ширину и высоту любого изображения пирамиды можно получить с помощью формулы:

$$\begin{aligned} w_n &= d^n w \\ h_n &= d^n h \end{aligned},$$

где w_n и h_n — ширина и высота n-го изображения пирамиды, n — номер изображения в пирамиде (0 — это оригинальное изображение), d — коэффициент масштабирования, а w и h — ширина и высота оригинального изображения.

Для интерполяции, о которой речь пойдет позже, требуется только ширина и высота входного и выходного изображений. В нашем случае входное изображение — это оригинал, а выходное — искомая масштабированная копия. Иначе говоря, для интерполяции, так же как и для вычисления ширины и высоты, не требуется предыдущих изображений, за исключением оригинального.

Так как ширина и высота сканируемого изображения не могут быть меньше ширины и высоты окна, то из предыдущей форумулы можно определить номер уменьшенной копии минимально возможного размера:

$$n = \left\lfloor \log_d \max\left(\frac{w_w}{w}, \frac{h_w}{h} \right) \right\rfloor,$$

где n — номер наименьшего изображения из пирамиды, d — коэффициент масштабирования, w_w и h_w — ширина и высота окна, а w и h — ширина и высота оригинального изображения. Квадратные скобки обозначают взятие целой части.

Используя написанные выше формулы можно разделить изображения из пирамиды на группы так, чтобы сканирование всех групп требовало примерно одинакового количества ресурсов процессора и времени. Информацию о том, к какой группе принадлежит каждое изображение, можно хранить в таблице (двумерном массиве), где каждой группе соответствует своя строка, а в ячейках хранятся номера изображений в пирамиде. Далее такая таблица будет называться *картой потоков*.

Так как объем вычислений при сканировании зависит от количества пикселей в изображении, то это количество можно использовать как оценку необходимых для сканирования ресурсов (оценка сложности изображения):

$$\delta_n = w_n h_n ,$$

где δ_n — оценка сложности n-го изображения, а w_n и h_n — ширина и высота n-го изображения пирамиды.

Однако, так как ширина и высота всех изображений одинакого зависят от ширины и высоты оригинального изображения, оценку можно записать так:

$$\delta_n = d^{2n}$$

где δ_n — оценка сложности сканирования n-го изображения, а d — коэффициент масштабирования.

Для краткости назовем сумму оценок сложности изображений в группе *оценкой сложности группы*. Теперь задачу разделения изображений пирамиды на группы можно сформулировать так: сумма оценок сложности всех групп должна быть примерно одинаковой. Так как

оптимальное решение этой задачи, скорее всего, будет достаточно сложным, лучше найти решение, близкое к оптимальному, используя особенности пирамиды изображений.

Для начала нужно найти "идеальную" оценку сложности группы. Т.е. оценку в том случае, если бы она была одинакова для всех групп. Получить ее можно разделив общую сумму оценок сложности изображений на требуемое количество потоков.

$$\delta = \frac{\sum_{i=0}^{n} d^{2i}}{T}$$

где δ — "идеальная" оценка сложности группы, d — коэффициент масштабирования, а n — общее количество изображений в пирамиде.

Теперь можно по очереди заполнять каждую группу. Для этого в заполняемую группу по очереди добавляются изображения так, чтобы её оценка не стала выше "идеальной". Понятно, что вероятность получить оптимальное решение таким способом очень маленькая. Однако можно выбрать такой порядок добавления изображений, чтобы получить приемлимое решение.

Оценки сложности изображений можно представить в виде отрезков, длина которых соответствует их значениям:

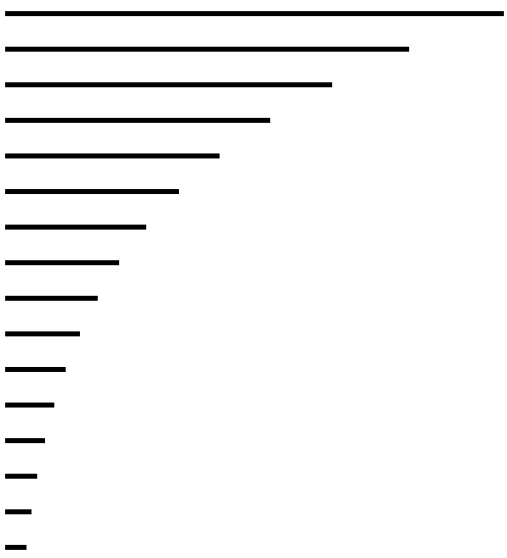

Рис. 9. Оценки сложности изображений в виде отрезков.

Тогда составленные из них большие отрезки будут соответствовать группам изображений. В таком виде задача будет выглядеть так: составить из данных отрезков требуемое количество больших отрезков таким образом, чтобы длина этих больших отрезков была примерно одинаковой. Длина "идеального" отрезка в таком случае будет являться суммой длин всех данных отрезков, делённой на требуемое количество больших.

Можно заметить что более длинные отрезки, стоящие в начале, удобно дополнять более короткими, стоящими в конце, так чтобы их длина стала как можно ближе к длине "идеального" отрезка, при этом не превысив её.

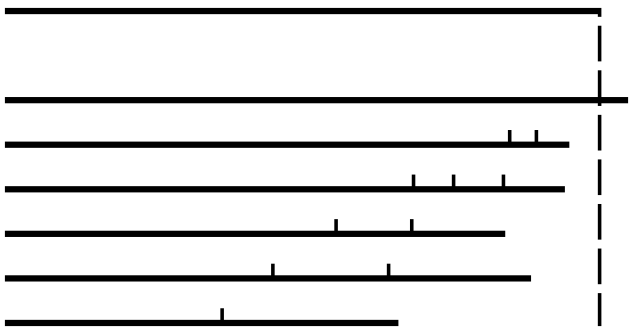

Рис. 10. Пример разделения изображений на 8 групп. Первым изображен идеальный отрезок. Ниже изображены большие отрезки, на которых отмечены границы малых отрезков, из которых они составлены.

На основе этого можно вывести способ добавления изображений в группу: берется первое неиспользованное изображение из начала, а затем к нему последовательно добавляются неиспользованные изображения с конца, так, чтобы оценка группы оставалась меньше "идеальной" оценки. Полный алгоритм можно записать следующим образом:

n — номер последнего значения в пирамиде
δ — "идеальная" оценка
b — индекс начала пирамиды
e — индекс конца пирамиды
T — требуемое количество количество групп изображений
t — количество построенных групп изображений
c — текущее количество изображений в группе
$M[i][j]$ — номер j-го изображения в i-й группе
S — вспомагательная сумма

1. Вычислить номер последнего значения в пирамиде:

$$n = \left\lceil \log_d \max\left(\frac{w_w}{w}, \frac{h_w}{h} \right) \right\rceil$$

2. Вычислить "идеальную" оценку:

$$\delta = \frac{1 - d^{2i}}{T(1 - d)}$$

3. Инициализировать текущие индексы начала и конца пирамиды, а также текущее количество групп:

$$b = 0,\ e = n$$
$$t = 0$$

4. Пока $b < e$:
   a. Инициализировать текущее количество изображений в группе:

$$c = 0$$

   b. Добавить первое неиспользованное изображение в карту потоков:

$$M[t][c] = b$$
$$c = c + 1$$
$$b = b + 1$$

   c. Инициализировать вспомогательную сумму:

$$S = d^b$$

d. Пока $S + d^e < \delta$ и $b < e$:

I. Добавить изображение с конца пирамиды к карте:

$$M[t][c] = e$$
$$c = c + 1$$
$$e = e - 1$$

II. Прибавить оценку добавленного изображения к вспомагательной сумме:

$$S = S + d^{e-1}$$

e. Обозначить конец группы изображений:

$$M[t][c] = -1$$
$$c = c + 1$$

f. Увеличить количество групп изображений:

$$t = t + 1$$

Используя полученную карту можно сканировать изображение в несколько потоков:

M_i — строка карты потоков, обрабатываемая i-м потоком.
N — количество строк в карте потоков.
r_i — массив координат прямоугольных областей, в которых i-м потоком был обнаружен искомый объект.
R — массив, являющийся объединением всех r_i.
I_i — текущая масштабированная копия в i-м потоке.

В i-м потоке:

1. Инициализировать j, сделав его равным 0
2. Пока $M_i[j] \geq 0$:
   a. Масштабировать оригинальное изображение с коэффициентом $M_i[j]$, сохранив результат в I_i.
   b. Сканировать I_i, помещая координаты прямоугольных областей, где был обнаружен искомый объект в массив r_i.
   c. Увеличить j на 1.

Запуск потоков и обработка результатов:

1. Запустить N потоков. Каждый поток обрабатывает свою строку карты потоков.
2. Ждать завершения работы всех потоков.
3. Объединить все массивы r_i в один массив R.
4. Выполнить объединение пересекающихся прямоугольников в R.

3.10. Масштабирование изображений

При построении пирамиды изображений строятся масштабированные копии оригинального изображения. Это делается с помощью билинейной интерполяции, которая является вариантом линейной интерполяции для функции с двумя переменными.

Линейная интерполяция — это способ нахождения приближенного значения функции в точке, находящейся между двумя другими точками, в которых значение этой функции известно. Приближение выполняется с помощью линейной функции, откуда и следует название. Выглядит это следующим образом:

$$f(x) = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0),$$

где x — точка, в которой ищется значение функции, $f(x)$ — функция, значение которой ищется в точке x, x_0 — точка слева от x, x_1 — точка справа от x.

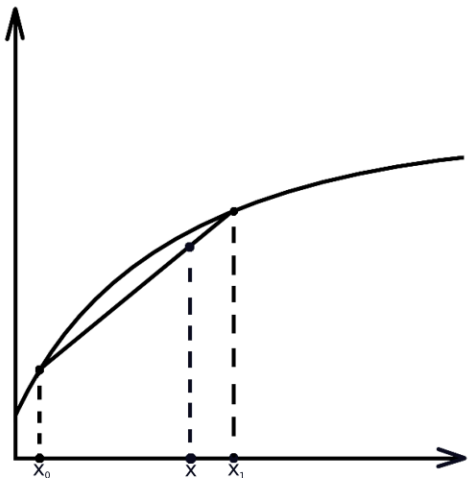

Рис. 11. Линейная интерполяция.

Таким образом, по сути, строится прямая, проходящая через точки на плоскости $(x_0; f(x_0))$ и $(x_1; f(x_1))$, а затем с помощью известной x-координаты находится y-координата соответствующей точки, лежащей на этой прямой. Эта y-координата и будет являться искомым значением функции в точке.

Понятно, что такое приближение может быть неточным, однако, когда точки с известными значениями функции находятся достаточно близко, ошибка становится незначительной. В случае масштабирования изображения для дальнейшего распознавания компьютером, эта ошибка находится в допустимых пределах.

При билинейной интерполяции требуется найти значение функции в точке на плоскости при известных значениях этой функции в четырех других точках на плоскости, являющихся углами прямоугольника. Для этого интерполяция выполняется три раза: сначала два раза по оси x, и затем один раз по оси y, где в качестве известных значений функции уже используются результаты первых двух интерполяций:

$$f(x; y_0) = f(x_0; y_0) + \frac{f(x_1; y_0) - f(x_0; y_0)}{x_1 - x_0}(x - x_0)$$

$$f(x; y_1) = f(x_0; y_1) + \frac{f(x_1; y_1) - f(x_0; y_1)}{x_1 - x_0}(x - x_0)$$

$$f(x; y) = f(x; y_0) + \frac{f(x; y_1) - f(x; y_0)}{y - y_0}(y - y_0),$$

где $(x; y)$ — точка, значение функции в которой ищется, $f(x; y)$ — искомое значение функции в точке $(x; y)$, а $(x_0; y_0)$, $(x_1; y_0)$, $(x_0; y_1)$, $(x_1; y_1)$ — координаты точек, в которых значения функции известны.

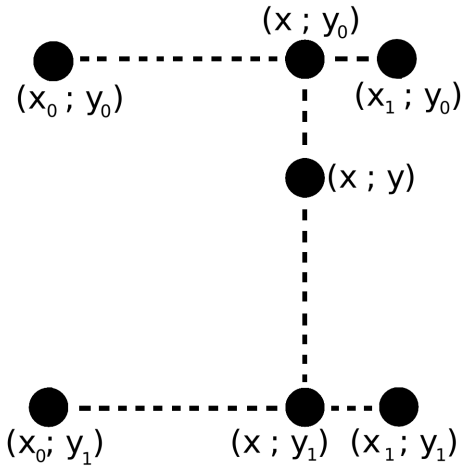

Рис. 12. Билинейная интерполяция. Для удобства поверхность функции двух переменных
изображена "сверху".

Чтобы построить масштабированную копию изображения, сначала вычисляется её ширина и высота. Затем для каждого пикселя этой копии вычисляются его координаты на оригинальном изображении. Далее с помощью этих координат находится яркость этого пикселя (в случае, когда компонент цвета несколько — значение каждой компоненты ищется отдельно).

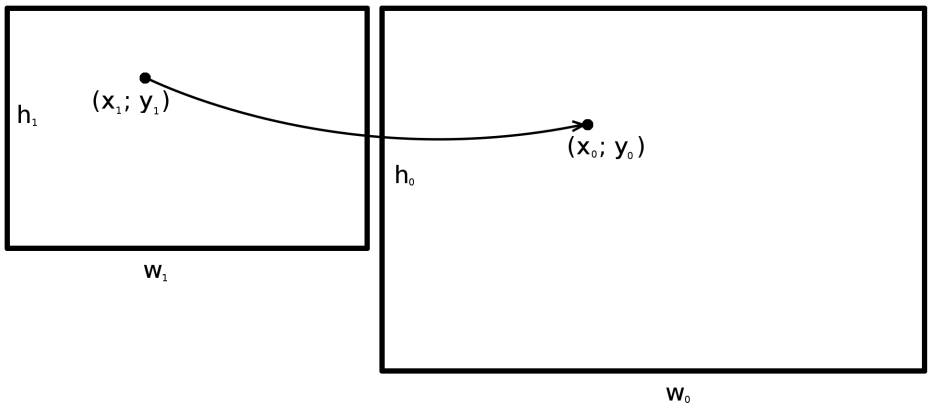

Рис. 13. Координаты пикселя копии на оригинальном изображении. Слева — копия, справа — оригинальное изображение.

Так как нет гарантии, что вычисленные координаты будут целыми, и таким образом будут соответствовать конкретному пикселю, выполняется билинейная интерполяция между четырьмя ближайшими к этим координатам пикселями оригинального изображения. В качестве значения функции выступает яркость. Точка, где это значение ищется — это координаты пикселя копии на оригинальном изображении. А в качестве точек, где это значение известно выступают четыре ближайших к этим координатам пикселя оригинального изображения.

При интерполяции координаты этих четырех пикселей полагаются равными $(0;0)$, $(1;0)$, $(0;1)$ и $(1;1)$, а координаты пикселя копии на оригинальном изображении заменяются своей дробной частью. Это упрощает формулы, использующиеся при билинейной интерполяции, при этом не меняя результата:

$$f(x;0) = f(0;0) + (f(1;0) - f(0;0))x$$

$$f(x;1) = f(0;1) + (f(1;1) - f(0;1))x$$

$$f(x, y) = f(x;0) + (f(x;1) - f(x;0))y \ ,$$

где $(x; y)$ — точка, значение функции в которой ищется, $f(x; y)$ — точка в которой ищется значение функции, а $(0;0)$, $(1;0)$, $(0;1)$, $(1;1)$ — координаты точек, в которых значения функции известны.

Полностью алгоритм масштабирования изображения с одной компонентой цвета будет выглядеть так:

w_0 и h_0 — ширина и высота оригинального изображения.
w_1 и h_1 — ширина и высота оригинального изображения.
d_x и d_y — коэффициенты масштабирования по осям x и y.
$V_0[x; y]$ — яркость пикселя оригинального изображения с координатами $[x; y]$
$V_0[x; y]$ — яркость пикселя масштабированной копии с координатами $[x; y]$.
x_0 и y_0 — координаты пикселя копии на оригинальном изображении.
p_0 и p_1 — результаты интерполяции по оси x.

1. Вычислить ширину и высоту масштабированной копии:

$$w_1 = \text{ceil}(d_x w_0)$$

$$h_1 = \text{ceil}(d_y h_0)$$

2. Для всех y_1 от 0 до h_1:
   Для всех x_1 от 0 до w_1:
   a. Вычислить координаты пикселя копии на оригинальном изображении:

$$x_0 = \frac{x_1}{d_x} \ , \ y_0 = \frac{y_1}{d_y}$$

   b. Выполнить интерполяцию по оси x:

$$p_0 = (V_0[\text{floor}(x_0); \text{floor}(y_0)] - V_0[\text{ceil}(x_0); \text{floor}(y_0)])(x_1 - \text{floor}(x_1)) + V_0[\text{floor}(x_0); \text{floor}(y_0)]$$

$$p_0 = (V_0[\text{floor}(x_0); \text{ceil}(y_0)] - V_0[\text{ceil}(x_0); \text{ceil}(y_0)])(x_1 - \text{floor}(x_1)) + V_0[\text{floor}(x_0); \text{ceil}(y_0)]$$

   c. Выполнить интерполяцию по оси y:

$$V_1[x_1; y_1] = (p_1 - p_0)(y - \text{floor}(y)) + p_0$$

Функция floor — округление в меньшую сторону, ceil — в большую, а frac — взятие дробной части. Такие обозначения приняты для удобства. Чтобы масштабировать изображение, где больше одной цветовой компоненты, необходимо выполнить билинейную интерполяцию для каждой цветовой компоненты пикселя по отдельности.

3.11. Объединение пересекающихся прямоугольников

Так как каскад Хаара имеет достаточно неплохую устойчивость к маштабированию и смещению, после полного прохода по изображению и его маштабированным вариантам искомый объект будет выделен окном несколько в соседних позициях и маштабах.

Рис. 14. Пример выделения объекта после полного прохода окном по изображению: зелеными прямоугольниками выделены области, изображение в которых было классифицировано положительно.

Чтобы исправить эту проблему, был выбран следующий подход: все прямоугольники разбиваются на группы. Прямоугольник, принадлежащий какой-либо группе пересекается хотябы с одним из других прямоугольников этой же группы. После этого каждая группа заменяется одним прямоугольником, являющимся усереднением всех прямоугольников этой группы.

Задачу объединения прямоугольников в группы можно хорошо описать и решить с помощью понятий из теории графов. Прямоугольники можно считать вершинами графа. Если они пересекаются, можно считать что они соединены ребром. Тогда если разбить граф на компоненты связности, вершины принадлежащие одной и той же компоненте и будут представлять из себя искомые группы.

Сначала надо представить набор прямоугольников в виде графа. Для этого прямоугольники нумерются. Также каждому прямоугольнику в соотвествие ставится список, содержащий номера других прямоугольников, пресекающих его. Полученные списки и будут преставлять из себя граф. Построить эти списки можно по следующему алгоритму:

r_i — i —й прямоугольник.
N — количество имеющихся прямоугольников.
e_{ij} — номер (в r_i) j-го прямоугольника из тех, которые пересекаются с i-м прямоугольником.
N_i — количество прямоугольников, пресекающихся с i-м прямоугольником. Изначально равно нулю.

Для всех i и j от 0 до N:
Если прямоугольники r_i и r_j пересекаются, добавить в список, соответствующий вершине r_i значение j:

$$\text{если } r_i, r_j \text{ пересекаются, тогда } e_{iN_i} = j$$

$$N_i = N_i + 1$$

Проверка пересечения двух прямоугольников выполняется с помощью проекций на оси. То есть, проверяются на пересечение их проекций на ось X и на ось Y по отдельности. Если пересечение имеется при проекций на обе оси, то прямоугольники пересекаются. Проверка пересечений проекций выполняется следующим образом:

A_{x0}, A_{y0} — наименьшие координаты угла первого прямоугольника.
A_{x1}, A_{y1} — наибольшие координаты угла первого прямоугольника.
B_{x0}, B_{y0} — наименьшие координаты угла второго прямоугольника.

B_{x1}, B_{y1} — наибольшие координаты угла второго прямоугольника.

если $A_{x0} > B_{x1}$ или $A_{x1} < B_{x0,}$ то проекции на ось X пересекаются

если $A_{y0} > B_{y1}$ или $A_{y1} < B_{y0,}$ то проекции на ось Y пересекаются

После этого требуется разбить полученный граф на компоненты связности. Если точнее, требуется узнать, к какой компоненте относится каждая вершина. Информация о ребрах внутри этих компонент, не требуется. Исходя из этого можно использовать следующий алгоритм:

r_i — соответствует i-й вершине. Если равен 1, то эта вершина была достигнута из выбранной начальной вершины.
R_i — соответствует i-й вершине. Если равен 1, то эта вершина уже была достигнута из другой вершины. Изначально равен 0.
V_i — количество вершин, в i-й компоненте связности. Изначально равно нулю.
c_{ij} — номер j-й вершины в i-й компоненте связности.
C — количество компонент связности. Изначально равно нулю.
e_{ij} — номер j-й вершины из тех, в которые можно попасть из i-й вершины.
E_i — количество вершин, в которые можно попасть из i-й вершины.
n — номер вершины, с которой начинается выделение компоненты связности. Изначально равен нулю.
v — i-я вершина.
N — общее количество вершин.

Пока $n < N$:

1. Обнулить r_i:

$$\text{для всех } i \text{ от } 0 \text{ до } N-1\text{: } r_i = 0$$

2. Отметить все вершины, которые можно достигнуть из текущей вершины:

$$\text{если из вершины } v_n \text{ можно достигнуть вершину } v_i\text{, то } r_i = 1,\ R_i = 1$$

3. Добавить все отмеченные на текущем шаге вершины в новую компоненту связности:

$$\text{для всех } i \text{ от } 0 \text{ до } N-1\text{: если } r_i = 1\text{, тогда } c_{CV_C} = i,\ V_C = V_C + 1$$

$$C = C + 1$$

4. Найти вершину с которой начнется выделение следующей компоненты связности:

$$\text{пока } R_n = 1 \text{ и } n < N\text{: } n = n + 1,$$

Нахождение всех вершин, которые можно достигнуть из текущей выполняется простым проходом по графу и отметкой уже пройденных вершин. В развернутом виде пункт 2. будет выглядеть так:

1. Отметить вершину с номером n:

$$r_n = 1,\ R_n = 1$$

2. Перейти по всем ребрам вершины с номером n:

$$\text{для всех } j \text{ от } 0 \text{ до } E_i - 1\text{: если } r_{e_{ij}} \neq 1\text{, перейти к пункту 1. сделав } n = e_{ij}$$

После того, как было определенно, к какой компоненте связности относится каждая вершина, любую группу пересекающихся прямоугольников можно заменить одним усередненным прямоугольником.

В качестве усередненния используется прямоугольник, чьи наименьшие и наибольшие координаты угла находятся как среднее арифметическое наименьших и наибольших координат угла всех прямоугольников группы:

C — количество компонент связности.
c_i — количество прямоугольников в i-й компоненте связности.
r_{ij} — номер j-й вершины в i-й компоненте связности.
$x0_i$, $y0_i$, $x1_i$, $y1_i$ — минимальные и максимальные x и y координаты углов усередненного прямоугольника, соответствующего i-й компоненте связности.
$x0_{ij}$, $y0_{ij}$, $x1_{ij}$, $y1_{ij}$ — минимальные и максимальные x и y координаты углов прямоугольника, соответствующего j-у прямоугольнику в i-й компоненте связности.

Для всех i от 0 до n:

1. Установить координаты углов усередненного прямоугольника, соответствующего i-й компоненте связности в 0:

$$x0_i = 0,\ y0_i = 0,\ x1_i = 0,\ y1_i = 0$$

2. Для всех j от 0 до c_i:

   Прибавить к координатам углов усередненного прямоугольника, соответствующего i-й компоненте связности, координаты углов j-го прямоугольника этой компоненты связности:

$$x0_i = x0_i + x0_{ij}$$
$$y0_i = y0_i + y0_{ij}$$
$$x1_i = x1_i + x1_{ij}$$
$$y1_i = y1_i + y1_{ij}$$

3. Поделить координаты углов усередненного прямоугольника, на количество прямоугольников в i компоненте связности:

$$x0_i = \frac{x0_i}{c_i},\ y0_i = \frac{y0_i}{c_i}$$
$$x1_i = \frac{x1_i}{c_i},\ y1_i = \frac{y1_i}{c_i}$$

4. Исправление искажений перспективы

После того, как область, в которой находится пластина с автомобильным номером, была выделена, ее следует привести к виду, пригодному для распознаванию отдельных символов на этой пластине. Одна из основных проблем, мешающих корректному распознаванию символом — это искажения перспективы, возникающие когда номер снимается не под прямым углом.

Для решения этой проблемы используется комбинация нескольких методов. Сначала вычисляется градиент изображения, на котором лучше видны резкие перепады яркости. С его помощью на этом изображении обнаружаются контуры объектов. Среди найденных контуров выделяются только те, которые являются прямыми линиями. Далее из них выбирается четыре линии, с наибольшей вероятностью являющиеся контурами пластины с номером. После этого, с помощью этих четырех линий, находятся углы пластины с номером и вычисляется, каким образом нужно преобразовать изображение, что бы исправить искажения перспективы.

Рис. 15. Пример пластины с номером, обнаруженной на фотографии. Зелеными точками обозначены углы номерной пластины.

Рис. 16. Пластина с номером после исправления искажений перспективы. В таком виде номер пригоден для дальнейшего распознавания.

4.1. Градиент изображения. Оператор Собеля

Градиент изображения — это набор векторов, в котором каждый элемент (вектор) соответствует своему пикселю изображения. Направление вектора указывает из соответствующего ему пикселя в ту сторону, двигаясь в которую, значения пикселей будут возрастать быстрее всего, а модуль его показывает скорость возрастания этих значений.

Если рассматривать непрерывную функцию, ее градиент является суммой произведений производных по x, y и векторов $i = (1, 0)$, $\mathrm{j} = (0, 1)$ (такие вектора называются базисными), соответственно:

$$\nabla_f = \frac{\delta f}{\delta x} i + \frac{\delta f}{\delta y} j,$$

где f — непрерывная функция, для которой ищется градиент, а ∇_f — искомый градиент функции f.

Так как изображение, по сути, является дискретной функцией, т.е. функцией, изменяющейся скачками, вычисление вектора градиента в каждой точке требует прохода по всему изображению. Поэтому, градиент изоображения обычно вычисляется приближенно. Для этого используется *Оператор Собеля*, представляющий из себя две матрицы 3×3, использующихся для нахождения приближенных производных по x и по y по отдельности.

Данные матрицы выглядят следующим образом:

$$S_x = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}, \quad S_y = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}.$$

Для приближенного вычисления проиводных используется операция свёртки:

$$G_x = S_x * I = \sum_{i=1}^{3} \sum_{j=1}^{3} I[x-i, y-j]\ S_x[i, j]$$

$$G_y = S_y * I = \sum_{i=1}^{3} \sum_{j=1}^{3} I[x-i, y-j]\ S_y[i, j]\ ,$$

где $I[x, y]$ — пиксель с координатами [x, y], а $S_y[i, j]$ и $S_x[i, j]$ — элемент матрицы S_x или S_y, соответственно, находящийся в i-й столбце и j-й колонке. После этого, с помощью найденных производных по x и y находятся направление и модуль градиента:

$$G = (G_x{}^2 + G_y{}^2)^{\frac{1}{2}}$$

$$\theta = \arctan \frac{G_y}{G_x}\ ,$$

где G — модуль градиента в точке, θ — направление градиента в точке[1], а G_x и G_y — значения частных производных по x и y в этой точке.

4.2. Детектор границ Канни

Обнаружениие границ выполняется с помощью *детектора границ Канни*. Детектор границ Канни является алгоритмом *бинаризации*, т.е. он разделяет пиксели на две группы: в первой группе находятся пиксели контура, а во второй — все остальные.

Данный алгоритм использует вычисленный с помощью оператора Собеля или любым другим способом градиент и состоит из нескольких этапов. Сначала выполняется *подавлениие немаксимумов* среди значений модуля градиента, потом выполняется *двойная пороговая фильтрация*. После этого обрабатываются пиксели, принадлежность контуру которых не была опредена на предыдущем этапе.

Результат возвращается в виде изображения (маски), где каждый пиксель соответствует пикселю исходного изображения с такими же координатами. Пиксели маски принимают одно из двух значений: 0 — если соответствующая точка исходного изображения не принадлежит границе, и любого другое значение, если принадлежит.

Подавление немаксимумов требуется для того, чтобы выделенная граница была мининальной толщины, иначе говоря, чтобы она была четкой настолько, наколько это возможно. Для этого выполняется проход по всем точкам градиента. Если значение градиента в точке меньше, чем значение градиента в двух соседних точках, находящихся слева и справа от

[1] Важно учитывать, что если и G_x, и G_y в точке отрицательны, то минусы "уничтожают" друг друга. Этот случай следует обрабатывать отдельно, так как иначе будет получено неверное значение.

вектора направления градиента, то ему присваивается значение 0.

Для упрощения реализации угол направления градиента дискретизируется, т.е., вместо вещественных чисел представляется фиксированным количеством значений. На основе дискретизированного значения угла уже определяется, с какими соседними значениями градиента сравнивать значение градиента в текущей точке. Ниже приведен полный алгоритм подавления немаксимумов:

$G[x, y]$ — значение градиента в точке с координатами [x, y].
$\theta[x, y]$ — угол градиента с осью X в точке с координатами [x, y].
w — ширина исходного изображения.
h — высота исходного изображения.
T — дискретизированное значение угла.
L_x — координата x точки слева от вектора направления градиента.
L_y — координата y точки слева от вектора направления градиента.
R_x — координата x точки справа от вектора направления градиента.
R_y — координата y точки справа от вектора направления градиента.

Для всех x от 1 до w:
Для всех y от 1 до h:

1. Дискретизировать[1] угол направления градиента в точке $[x, y]$:

$$T = \frac{8\left(\theta[x, y] + \frac{\pi}{8}\right)}{2\pi} \bmod 8 = \left[\frac{4\theta[x, y]}{\pi} + \frac{1}{2}\right] \bmod 8$$

2. Определить, с какими из соседних точек следует сравнивать значение градиента:

если $T = 0$ или $T = 4$: тогда $L_x = x,\ L_y = y+1,\ R_x = x,\ R_y = y-1$
если $T = 2$ или $T = 6$: тогда $L_x = x-1,\ L_y = y,\ R_x = x+1,\ R_y = y$
если $T = 1$ или $T = 5$: тогда $L_x = x-1,\ L_y = y-1,\ R_x = x+1,\ R_y = y+1$
если $T = 3$ или $T = 7$: тогда $L_x = x-1,\ L_y = y+1,\ R_x = x+1,\ R_y = y-1$

3. Сравнить значение градиента со значениями градиента в выбранных соседних точках. Если оно меньше любого из них, заменить его значение нулем:

если $G[x, y] < G[L_x, L_y]$ или $G[x, y] < G[R_x, R_y]$: тогда $G[x, y] = 0$

После подавления немаксимумов выполняется двойная пороговая фильтрация. Выбирается два значения, называемых порогами. Они задаются либо вручную, либо определяются динамически, например, *методом Оцу*. Точки, в которых значение градиента больше наибольшего из двух порогов, отмечаются как точки границы. Те точки, значения которых лежат между двумя порогами, отмечаются как неопределенные. Остальные точки отмечаются как фоновые:

$G[x, y]$ — значение градиента в точке с координатами [x, y].
θ_1 — наименьший из двух порогов
θ_2 — наибольший из двух порогов
w — ширина исходного изображения.
h — высота исходного изображения.
$m[x, y]$ — точка $[x, y]$ выходной маски. Если ее значение равно 0, то точка $[x, y]$ изображения — фоновая, если 2 — то это точка контура, если 1 — ее принадлежность контуру пока не определена.

[1] Здесь предполагается, что $\theta[x, y]$ лежит в интервале $[0; 2\pi]$. Если это не так, пред этим его следует привести к этому интервалу.

Для всех x от 1 до w:
Для всех y от 1 до h:
Сравнить значение градиента в точке $[x, y]$ с порогами θ_1 и θ_2:

если $G[x, y] < \theta_1$: тогда $m[x, y] = 0$
если $G[x, y] > \theta_2$: тогда $m[x, y] = 2$
если $G[x, y] > \theta_1$ и $G[x, y] < \theta_2$: тогда $m[x, y] = 1$

Далее обрабатываются точки, отмеченые как неопределенные. Это делается следующим образом: если принадлежность точки контуру не была определена на предыдущем этапе, но при этом хотябы одна из 8 соседних для нее точек принадлежит контуру, то она тоже помечается как точка контура:

$G[x, y]$ — значение градиента в точке с координатами [x, y].
w — ширина исходного изображения.
h — высота исходного изображения.
$m[x, y]$ — точка $[x, y]$ выходной маски. Если ее значение равно 0, то точка $[x, y]$ изображения — фоновая, если 2 — то это точка контура, если 1 — ее принадлежность контуру пока не определена.

Для всех x от 1 до w:
Для всех y от 1 до h:
Сравнить значение градиента в точке $[x, y]$ с порогами θ_1 и θ_2:

если $m[x, y] = 1$:
если $m[x-1, y-1] = 2$ или если $m[x-1, y] = 2$ или если $m[x-1, y+1] = 2$
или если $m[x, y-1] = 2$ или если $m[x, y] = 2$ или если $m[x, y+1] = 2$
или если $m[x+1, y-1] = 2$ или если $m[x+1, y] = 2$
или если $m[x+1, y+1] = 2$:
тогда $m[x, y] = 2$,
иначе $m[x+1, y+1] = 0$

Рис. 17. Пример работы детектора границ Канни: сверху находится исходное изображение, а снизу выходное изображение детектора.

4.3. Метод Оцу

Как было сказано ранее, при использовании детектора границ Канни необходимо задать два порога. Для нахождения верхнего порога в данной работе используется *Метод Оцу*. Величина нижнего порога определяется как половина величины верхнего.

Метод Оцу является алгоритмом бинаризации. Однако, в отличие от детектора границ Канни, он не выделяет границы, а просто подбирает порог, разделяющий пиксели так, чтобы значения их яркостей внутри обоих групп были как можно ближе друг к другу.

Один из способов выразить понятие "как можно ближе друг к другу" математическим языком — это минимизация дисперсии. Т. е., чем ближе яркости пикселей друг к другу, тем меньше дисперсия. Дисперсия внутри какой-либо группы величин (в данном случае это яркости пикселей) называется внутриклассовой дисперсией.

Для подбора порога, который разделит пиксели изображения так, чтобы дисперсия в полученных группах была минимальна, можно использовать *гистограмму*. Гистограмма — это график, на котором каждому диапазону значений исследуемых величин соответствует количество величин, попавших в этот диапазон. Говоря математическим языком, гистограмма является дискретным вариантом функции плотности распределения.

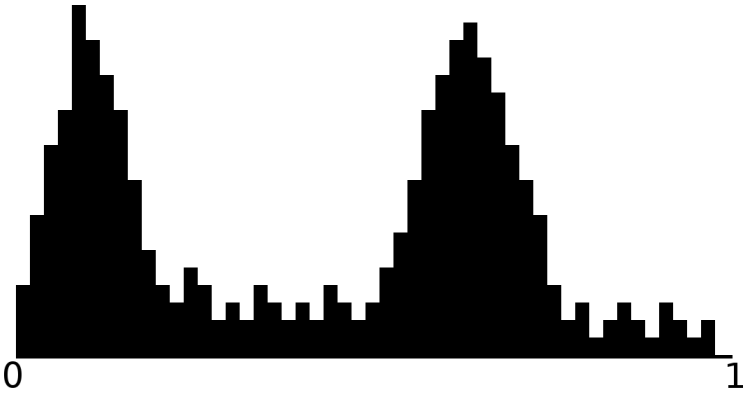

Рис. 18. Пример гистограммы. По горизонтальной оси расположены диапазоны значений, а по вертикальной количество попаданий в соответствующий диапазон.

Далее, используя тот факт, что максимизация *межклассовой дисперсии* эквивалентна минимизации внутриклассовой дисперсии, можно сильно сократить количество необходимых вычислений. Под межклассовой дисперсией подразумевается взвешенная дисперсия набора, состоящего из двух величин — средних арифметических первой и второй группы. В роли весов выступают доли от общего количества пикселей, входящие в первую и вторую группу соответственно:

$$\begin{gathered} w_0 + w_1 = 1 \\ \sigma = w_0(a-E)^2 + w_1(b-E)^2 = w_0 w_1 (a-b)^2 \,, \\ E = w_0 a + w_1 b \end{gathered}$$

где σ — межклассовая дисперсия, $w0$ — доля количества пикселей входящая в первую группу, $w1$ — доля количества пикселей входящая во вторую группу, a и b — среднее арифметическое первой и второй групп соответственно, а E — общее математическое ожидание (среднее арифметическое яркостей всех пикселей).

Для вычисления межклассовой дисперсии требуется знать только средние арифметические групп и доли входящих в них пикселей. Можно использовать гистограмму вместе двумя дополнительными величинами — суммой яркостей всех пикселей и количеством пикселей.

После того, как эти величины вычислены, выполняется проход по всем значениям гистограммы. На каждом шаге прохода с помощью вспомогательных сумм и дополнительных величин вычисляется межклассовая дисперсия. Если на каком-либо шаге эта дисперсия больше максимальной из найденных (которая изначально равна −1), то максимальной становится она, а как искомый порог (который изначально равен 0) задается положение текущего значения в гистограмме. Полученный после полного прохода порог и будет разделять группы так, чтобы их внутренняя дисперсия была максимальной.

В качестве вспомогательных сумм используются сумма яркостей пикселей и количество пикселей в первой группе. Изначально они равны нулю. На каждом шаге к первой сумме добавляется текущее значение гистограммы, домноженное на его номер, а ко второй сумме

просто текущее значение гистограммы. А так как группы разделяются непосредственно в этом текущем значении гистограммы, то для вычисления межклассовой дисперсии достаточно знать только эти вспомогательные величины, общее количество пикселей, а также сумму яркостей всех пикселей.

Стоит заметить, что гистограмма строится для значений от 0 до максимального значения яркости пикселя (все возможные значения). Также, важно, что количество значений в гистограмме, т.е. на сколько диапазонов разбиваются все возможные значения, задается заранее. Поэтому, после нахождения необходимого порога, его необходимо преобразовать, учитывая эти изменения, так, чтобы он соотвествовал яркостям пикселей изображения, а не значениям гистограммы. Ниже приведен полный алгоритм:

$v_{\max}$ — максимальное значение яркости среди всех пикселей.
$h[i]$ — i-е значение из гистограммы.
$I[i, j]$ — значение яркости пикселя с координатами $[i, j]$.
W — ширина изображения.
H — высота изображения.
N_h — число значений в гистограмме.
N — общее количество пикселей.
V — сумма яркостей всех пикселей.
$\sigma_{\max}$ — максимальная найденная на данный момент межклассовая дисперсия.
T — порог, соответствующий максимальной найденной на данный момент межклассовой дисперсии.
v и n — вспомогательные суммы.
w_0 и w_1 — доли общего количества пикселей, входящие в первую и вторую группы соответственно.
a — среднее арифметическое пикселей, чьи яркости ниже текущего порога (первая группа).
σ — текущая межклассвоая дисперсия.

1. Найти максимальное значение яркости пикселя:

$$v_{\max} = \max I[i, j]$$

2. Для i от 0 до H:
   Для j от 0 до W:
   Увеличить на 1 значение гистограммы, соответствующее текущему значению яркости:

$$m = \left[\frac{N_h I[i, j]}{V_{\max}} \right]$$
$$h[m] = h[m] + 1$$

3. Найти общее количество пикселей:

$$N = WH$$

4. Найти сумму всех яркостей пикселей с помощью гисторамммы:

$$\sum_{i=0}^{N_h} i\, h[i]$$

5. Инициализировать вспомогательные суммы:

$$v = 0$$
$$n = 0$$

6. Для каждого t от 0 до $N_h - 1$:
   a. Увеличить вспомогательные суммы:

$$v = v + t\,h[t]$$
$$n = n + h[t]$$

   b. Вычислить долю пикселей, входящих в первую группу:

$$w_0 = \frac{n}{N}$$

   c. С помощью вспомогательных сумм вычислить текущую межклассовую дисперсию:

$$a = \frac{v}{n} - \frac{V - v}{N - n}$$
$$\sigma = w_0\,(1 - w_0)\,a^2$$

7. Масштабировать порог так, чтобы он соответствовал яркостям пикселей избражения:

$$T = \frac{T\,V_{\max}}{N_h}$$

4.4. Преобразование Хафа

После того, как контуры были выделены детектором границ Канни, среди них ищутся прямые линии. Для этого используется преобразование Хафа. Оно, как и детектор границ Канни, состоит из нескольких этапов. Сначала выполняется поиск прямых. Далее выполняется подавление немаксимумов. А после этого линии, оставшиеся после предыщего этапа сортируются по количеству голосов, полученных на первом этапе.

Для поиска прямых в используется *аккумулирующий массив*. Каждый элемент массива соотвествуют одной из всех возможных прямых. Он является целым числом, позывающим сколько точек, принадлежащих границе, лежит на этой прямой.

Изначально всем элементам аккумулирующего массива присваивается значение 0, далее выполняется проход по всем точкам, принадлежащим контуру. Для каждой такой точки определяются все прямые, которые могут через нее проходить и значения элементов аккумулирующего массива, соответствующие этим прямым, увеличиваются на 1.

Для поиска всех прямых, на которых может лежать точка (а также для определения количества элементов аккумулирующего массива) используется уравнение прямой, записанное в специальной форме:

$$r = x\cos\theta + y\sin\theta\,,$$

где θ — угол нормали прямой с осью X, а r — минимальное расстояние от точки $[0, 0]$ до прямой.

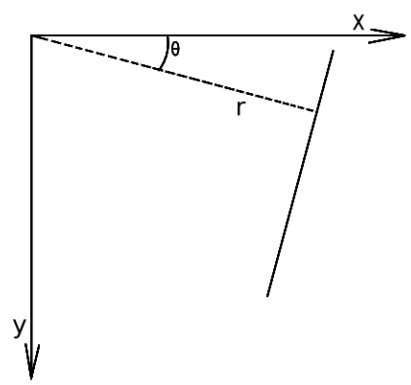

Рис. 19. Представление прямой через угол нормали с осью X и длину перпендиикуляра.

Так как параметры θ и r — вещественные числа, возможных прямых бесконечное множество. Следовательно, полный их перебор невозможен. Поэтому θ и r дискретизируются с заранее заданным шагом. Т.е. расстояние между двумя соседними значениями параметра в дискретизированным виде равно фиксированному значению. В данной работе параметр θ дискретизируется с шагом $\frac{\pi}{180}$ (вместо радиан используются целые углы), а параметр r с шагом 1 (просто округляется до ближайщего целого).

Все прямые, проходящие через точку находятся с помощью приведенного выше уравнения прямой. Выполняется проход для всех возможных значений θ, на каждом шаге текущее значения θ и координат точки подставляются в уравнение прямой, и вычисляется значение параметра r. Так как при фиксированном значении параметра θ, через конкретную точку может проходить только одна прямая, таким образом можно найти все искомые прямые.

Для того чтобы быстро определять, какой элемент аккумулирующего массива соответствует найденной прямой, аккумулирующий массив следует представить в виде изображения. Для этого надо расположить его элементы в несколько строк так, чтобы все элементы одной строки соответствовали прямым с одинаковым значением r (равным *номеру* строки) и разными значениями θ (равными *номеру* элемента в строке). Такое представление иногда называют пространством Хафа. Теперь, для того, чтобы определить, какой элемент аккумулирующего массива соответствует прямой со значениями параметров θ и r, достаточно найти точку в пространстве Хафа с координатами $[\theta, r]$.

Важно отметить, что при представлении изображения в компьютере значения координат точки (пикселя) обычно являются положительными целыми числами, а значения параметров прямой не обязательно являются целыми и могут быть отрицательными. Эта проблема решается в помощью смещения и маштабирования, т.е. прямой со значениями параметров θ и r соотвествует элемент аккумуллирующего массива с координатами $[a_\theta\theta + b_\theta, a_r r + b_r]$, где a_θ и a_r — маштабирующие коэффициенты параметров прямой, а b_θ и b_r — их смещения.

Ниже приведен полный алгоритм нахождения прямых:

$A[a, d]$ — элемент аккумулирующего массива с координатами $[a, d]$.
w — ширина входного изображения.
h — высота входного изображения.
T — ширина аккумулирующего массива.
R — высота аккумулирующего массива.
$d\theta$ — шаг дискретизации параметра θ.
dr — шаг дискретизации параметра r.
θ — значение параметра θ текущей прямой.
r — значение параметра r текущей прямой.

Для всех x от 1 до w:
    Для всех y от 1 до h:

Если значение пикселя входного изображения с координатами $[x, y]$ не равно 0:
Для всех a от нуля до T:

1. Вычислить значение θ для текущей прямой:

$$\theta = a \cdot d\theta$$

2. Вычислить значение r для текущей прямой:

$$r = x \cos\theta + y \sin\theta$$

3. Увеличить значение элемента аккумулирующего массива, соответствующего текущей прямой:

$$A[a, d + \frac{R}{2}] = A[a, d + \frac{R}{2}] + 1$$

После заполнения аккумулирующего массива из него следует выбрать элементы, являющееся максимумами и посторить из них список. При этом используется метод, очень похожий на подавление немаксимумов в детекторе границ Канни. Только сравнениие выполняется с четырьмя соседними элементами вместо двух:

$A[a, d]$ — элемент аккумулирующего массива с координатами $[a, d]$.
T — ширина аккумулирующего массива.
R — высота аккумулирующего массива.
$l_\theta[i]$ — параметр θ i-й прямой в заполняемом списке.
$l_r[i]$ — параметр r i-й прямой в заполняемом списке.
$l_v[i]$ — количество голосов, отданных за i-ю прямую в заполняемом списке.
L — количество прямых в заполняемом списке. Изначально равно нулю.
$d\theta$ — шаг дискретизации параметра θ.
dr — шаг дискретизации параметра r.

Для всех a от 1 до T:
Для всех d от 1 до R:
Если значение элемента аккумулирующего массива с координатами $[a, d]$ не равно нулю:
Если значение элемента аккумулирующего массива с координатами $[a, d]$ больше всех четырех соседних элементов, добавить значение параметров θ и r прямой, которой он соответствует, в список:

если $A[a, d] > A[a, d-1]$ и $A[a, d] > A[a, d+1]$
и $A[a, d] > A[a-1, d]$ и $A[a, d] > A[a+1, d]$:
тогда $l_\theta[L] = a \cdot d\theta$, $l_r[L] = d - \frac{R}{2}$, $l_v[L] = A[a, d]$, $L = L + 1$

После этого прямые в полученном списке сортируются по убыванию количества голосов любым способом, например, *быстрой сортировкой*. На выход подается отсортированный список, содержащий значения параметров θ и r найденых прямых.

4.5. Поиск границ пластины с автомобильным номером

Данный раздел раздел посвящен тому, каким образом комбинируются описанные выше методы для того, чтобы находить границы номера.

Основой поиска границ пластины с номером в данной работе является выбор самых ярких прямых, найденных преобразованием Хафа. Однако, границы пластины с номером не всегда являются самыми яркими прямыми.

Так как заранее примерно известно, в какой части изображения можно найти каждую из четырех границ, то первое, что можно сделать, чтобы это исправить — это разбить изображение на части и в каждой части искать отдельную границу.

Для поиска горизонтальных границ изображение разбивается горизонтальным сечением на две части: верхнюю и нижнюю половины. В верхней половине ищется верхняя граница номера, а в нижней — нижняя. Для поиска вертикальных границ изображение разбивается вертикальным сечением на несколько равных частей (в данной работе — на пять). В самой левой части ищется левая граница номера, а в самой прямой — правая.

Рис. 20. Исходное изображение пластины с номером.

Рис. 21. Изображение пластины с номером, разбитое на верхнюю и нижнюю половины.

Рис. 22. Изображение пластины с номером, разбитое на пять частей вертикальным сечением.

Для того, чтобы вертикальные границы пластины с номером (так как они менее яркие, чем горизонтальные) лучше обнаруживались с помощью преобразования Хафа, части изображения, в которых они ищутся, растягиваются по высоте (в данной работе они растягиваются так, чтобы их высота была равна ширине всего изображения с пластиной). Таким образом все вертикальные прямые на них при преобразовании Хафа получат больше голосов.

Рис. 23. Растянутые изображения, в которых ищутся вертикальные границы.

Также, можно использовать разные дополнительные условия, чтобы отсеять некоторые яркие прямые (найденные с помощью преобразования Хафа), не являющиеся границами пластины с номером. В данной работе используется два дополнительных условия.

Первое из них заключается в проверке угла наклона прямых. Если ищется верхняя или нижняя граница номера, то прямые, параметр θ которых меньше $\frac{\pi}{4}$ или больше $\frac{3\pi}{4}$ (т.е. прямые, не являющиеся горизонтальными) не рассматриваются. Аналогично, если ищется левая или правая граница: не рассматриваются прямые, чей параметр θ больше $\frac{\pi}{4}$ и меньше $\frac{3\pi}{4}$ (прямые, не являющиеся вертикальными).

Второе дополнительное условие основанно на том факте, пластина с автомобильным номером обычно белого цвета и имеет черные края. Поэтому второе условие заключается в сравненнии сумм яркостей в двух полосах (область изображения, ограниченная двумя прямыми). Первая полоса, лежит непосредственно на обнаруженной прямой. Вторая полоса лежит вплотную к первой по ту сторому от нее, по которую должна находиться пластина с номером. Если сумма яркостей пикселей, находящихся в первой полосе меньше, чем такая же сумма во второй полосе, то обнаруженная прямая считается границей пластины с номером, иначе прямая больше не рассматривается.

Второе условие применяется только при поиске вертикальных границ пластины с номером (горизонтальные прямые, для которых выполняется первое условие, сразу считаются границами пластины с номером), так как на практике результат этой проверки для горизонтальных границ был неудовлетворительным. Это следствие того, что горизонтальные границы пластины итак обычно являются самыми яркими, и, следовательно добавление этой проверки при их поиске несет лишь дополнительные ошибки распознавания.

Ширина полосы выбирается как доля от ширины или высоты. Т.е. ширина горизонтальных полос равняется высоте изображения, умноженной на заранее определенную константу (которая больше нуля и меньше единицы), а ширина вертикальных полос — равняется ширине изображения, умноженной на другую заранее определенную константу (которая также больше нуля и меньше единицы).

Каждая полоса представляется в виде двух прямых (границ полосы), чьи параметры θ равны, а параметры r отличаются на ширину полосы (т.е. модуль их разности равен ширине полосы). Для того, чтобы проверить, лежит ли точка в полосе, ищется прямая, проходящая через эту точку, с таким же параметром θ, как и границы полосы. Если значение параметра r этой прямой лежит между значениями параметров r границ полосы, то точка лежит в полосе.

Нахождение параметра r такой прямой, по сути, является нахождением проекции точки $[x, y]$ на перпендикуляр к границам полосы. Эту проекцию можно найти как длину катета, прилежащего к углу между прямой, проведенной из точки $[0, 0]$ в точку $[x, y]$ и перпендикуляром к границам полосы.

Алгоритм суммирования яркостей всех пикселей, лежащих в полосе выглядит так:

$I[x, y]$ — пиксель изображения (содержащего яркости), на котором ищется прямая.
w — ширина изображения, на котором ищется прямая.
h — высота изображения, на котором ищется прямая.
θ — параметр θ обоих прямых, составляющих полосу.
r_0, r_1 — параметр r первой и второй прямых составляющих полосу. r_0 должен быть меньше, чем r_1.
ϕ — угол прямой, проведенной из точки $[0, 0]$ в точку $[x, y]$ с осью X.
d — длина прямой, проведенной из точки $[0, 0]$ в точку $[x, y]$.
ρ — проекция точки $[x, y]$ на перпендикуляр к границам полосы.
S — сумма пикселей в полосе. Изначально равна нулю.

Для всех a от 1 до w:
Для всех d от 1 до h:

1. Вычислить длину и угол с осью X прямой, проходящей из точки $[0, 0]$ в точку $[x, y]$:

$$\phi = \arctan \frac{y}{x}$$
$$d = \sqrt{x^2 + y^2}$$

2. Вычислить значение проекции точки $[x, y]$ на перпендикуляр к границам полосы:

$$\rho = d \cos(\phi - \theta)$$

3. Если проекция точки $[x, y]$ лежит между параметрами r границ полосы, то она лежит в полосе и, следовательно, значение яркости пикселя с этими координатами надо прибавить к сумме:

$$\text{если } \rho > r_0 \text{ и } \rho < r_1\text{: тогда } s = s + I[x, y]$$

После того, как граница пластины с номером была найдена, она переводится из представления через перпендикуляр и его угол с осью X в представление через две точки, лежащие на ней:

$$\begin{array}{ll} \text{если } \theta > \varepsilon: & p_{0x} = 0,\ \ p_{0y} = \dfrac{r}{\sin\theta},\ \ p_{1x} = w,\ \ p_{1y} = r - w\dfrac{\cos\theta}{\sin\theta}, \\ \text{иначе} & p_{0x} = r,\ \ p_{0y} = 0,\ \ p_{1x} = r,\ \ p_{1y} = h \end{array},$$

где θ и r — параметры прямой в представлении через перпендикуляр и его угол с осью X, w и h — ширина и высота изображения с пластиной, p_{0x} и p_{0y} — координаты первой точки в новом представлении линии, а p_{1x} и p_{1y} — координаты второй точки в новом представлении линии.

После того, как были найдены все четыре границы пластины, составляющие контур номера, ищутся координаты её углов. Для этого находятся точки пересечения левой и верхней (левый верхний угол пластины), левой и нижней (левый нижний угол пластины), правой и верхней (правый верхний угол пластины), правой и нижней (правый нижний угол пластины) границ.

Далее, с помощью найденных координат углов выполняется перспективное преобразование изображения с пластиной, и после этого выполняется распознавание отдельных символов на пластине.

4.6. Перспективные преобразования

Как уже говорилось, после того, как все четыре угла пластины с номером найдены, выполняется перспективное преобразование, изменяющее избражение пластины с номером так, чтобы найденные углы совпадали с углами выбранного заранее четырёхугольника (Рис. 15, Рис. 16). Иначе говоря, пластина на изображении должна быть расположена таким образом, будто съемка выполнялась под прямым углом без каких-либо поворотов.

Такое преобразование можно получить с помощью линейной алгебры. В этом случае, оно будет представлено в виде матрицы. Умножение координат пикселя итогового изображения на эту матрицу даст координаты соответствующей им точки на оригинальном изображении. Далее с помощью этой матрицы можно выполнить проход по всем пикселям итогового изображения, на каждом шаге находя соответствующую текущему пикселю точку оригинального изображения и выполняя билинейную интерполяцию.

Как можно заметить, используется способ, похожий на тот, который использовался при масштабировании. Отличие заключается только в том, что для нахождения соответствующей точки оригинального изображения вместо деления используется матрица. На самом деле построить матрицу можно и для масштабирования. Однако это не будет соответствовать сложности задачи, поэтому был выбран более простой метод, использующий деление.

Матрица для перспективного преобразования состоит из двух других матриц, "комбинируемых" с помощью матричного умножения. Первая матрица преобразует координаты пикселя итогового изображения в промежуточные точки, а вторая преобразует полученные промежуточные точки в искомые координаты на оригинальном изображении.

Обе матрицы будут искаться в виде квадратных матриц 3×3. Поэтому, чтобы выполнять с их помощью преобразования, координаты будут представляться в виде вектора из трех компонент. Первые две компоненты вектора — это x и y координаты, а третья равняется единице.

Сначала ищется вторая матрица, преобразующая промежуточные точки в координаты на оригинальном изображении. Для этого достаточно, чтобы она преобразовывала четыре промежуточные точки в координаты углов выделенного на оригинальном изображении четырёхугольника. В качесте этих точек используются точки $(1,0,0)$, $(0,1,0)$, $(0,0,1)$ и $(1,1,1)$.

Матрица для отображения первых трех промежуточных точек в первые три угла будет выглядеть следующим образом:

$$\begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{bmatrix},$$

где x_1, x_2, x_3, y_1, y_2 и y_3 — x-координаты и y-координаты углов четырёхугольника на оригинальном изображении.

Чтобы четвертая промежуточная также преобразовывалась в координаты четвертого угла, столбцы матрицы умножаются на такие подбираемые значения, чтобы выполнялось равенство:

$$\begin{bmatrix} ax_1 & ax_2 & ax_3 \\ by_1 & by_1 & by_1 \\ c & c & c \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} x_4 \\ y_4 \\ 1 \end{bmatrix},$$

где x_1, x_2, x_3, x_4, y_1, y_2, y_4 и y_4 — x-координаты и y-координаты углов четырёхугольника на оригинальном изображении, а a, b и c — подбираемые значения. Это равенство можно записать другим образом:

$$\begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_1 & y_1 \\ 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} x_4 \\ y_4 \\ 1 \end{bmatrix},$$

Такое равенство, по сути, является системой линейных уравнений, где в роли неизвестных выступают значения a, b и c. Такую систему можно решить с помощью одного из численных методов, например, *методом Гаусса-Жордана*, который будет описан в следующием разделе.

В итоге будет получена матрица A:

$$A = \begin{bmatrix} ax_1 & ax_2 & ax_3 \\ by_1 & by_1 & by_1 \\ c & c & c \end{bmatrix},$$

которая преобразует $(1,0,0)$ в (ax_1, ay_1, a), $(0,1,0)$ в (bx_2, by_2, b), $(0,0,1)$ в (cx_3, cy_3, c), а $(1,1,1)$ в $(x_4, y_4, 1)$. При преобразовании первых трех координат появляются лишние множители, однако их можно убрать, если после умножения каждую компоненту результата делить на его z-компоненту.

Описанный выше способ также используется при получении первой матрицы, преобразующей координаты пикселя в итоговом изображении в промежуточные точки. Сначала с помощью него строится матрица B, преобразующая промежуточные точки $(1,0,0)$, $(0,1,0)$, $(0,0,1)$ и $(1,1,1)$ в координаты углов четырёхугольника, выделенного на итоговом изображении. После этого с помощью метода Гаусса-Жордана находится матрица B^{-1}, обратная к полученной. Как и в случае со второй матрицей, после умножения на матрицу B^{-1} каждую компоненту результата нужно разделить на его z-компоненту.

Итоговая матрица C, используемая для преобразования координат пикселя итогового изображения в координаты соответствующей ему точки на оригинальном изображении, будет результатом умножения матрицы A на матрицу B^{-1}:

$$C = A \times B^{-1} \quad .$$

Преобразование выполняется с помощью умножения координат пикселя итогового изображения на матрицу C. После этого необходимо разделить все компоненты полученного в итоге вектора на его z-компоненту, чтобы убрать лишние множители, которые добавляются матрицами составляющих C.

Ниже приведен полный алгоритм получения матрицы для перспективного преобразования:

x_1, x_2, x_3, x_4, y_1, y_2, y_3 и y_4 — x-координаты и y-координаты углов четырёхугольника, выделенного на оригинальном изображении.
$\tilde{x_1}$, $\tilde{x_2}$, $\tilde{x_3}$, $\tilde{x_4}$, $\tilde{y_1}$, $\tilde{y_2}$, $\tilde{y_3}$ и $\tilde{y_4}$ — x-координаты и y-координаты углов четырёхугольника, выделенного на итоговом изображении.
A — матрица, преобразующая промежуточные точки в координаты на оригинальном изображении.
B — матрица, преобразующая промежуточные точки в координаты на итоговом изображении.
B^{-1} — матрица, преобразующая координаты пикселя итогового изображения в промежуточные точки.
C — итоговая матрица, преобразующая координаты пикселя итогового изображения в соответствующие координаты на оригинальном изображении.
a, b и c — подбираемые значения для матрицы A.
$\tilde{a}$, $\tilde{b}$ и $\tilde{c}$ — подбираемые значения для матрицы B.

1. Найти значения a, b и c, решив систему линейных уравнений:

$$\begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_1 & y_1 \\ 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} x_4 \\ y_4 \\ 1 \end{bmatrix}$$

2. Построить матрицу A:

$$A = \begin{bmatrix} ax_1 & ax_2 & ax_3 \\ by_1 & by_1 & by_1 \\ c & c & c \end{bmatrix}$$

3. Найти значения $\tilde{a}$, $\tilde{b}$ и $\tilde{c}$, решив систему линейных уравнений:

$$\begin{bmatrix} \tilde{x_1} & \tilde{x_2} & \tilde{x_3} \\ \tilde{y_1} & \tilde{y_1} & \tilde{y_1} \\ 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} \tilde{a} \\ \tilde{b} \\ \tilde{c} \end{bmatrix} = \begin{bmatrix} \tilde{x_4} \\ \tilde{y_4} \\ 1 \end{bmatrix}$$

4. Построить матрицу B:

$$B = \begin{bmatrix} \tilde{a}\tilde{x_1} & \tilde{a}\tilde{x_2} & \tilde{a}\tilde{x_3} \\ \tilde{b}\tilde{y_1} & \tilde{b}\tilde{y_1} & \tilde{b}\tilde{y_1} \\ \tilde{c} & \tilde{c} & \tilde{c} \end{bmatrix}$$

5. Найти матрицу, обратную для матрицы B:

$$B^{-1} = \begin{bmatrix} \tilde{a}\tilde{x_1} & \tilde{a}\tilde{x_2} & \tilde{a}\tilde{x_3} \\ \tilde{b}\tilde{y_1} & \tilde{b}\tilde{y_1} & \tilde{b}\tilde{y_1} \\ \tilde{c} & \tilde{c} & \tilde{c} \end{bmatrix}^{-1}$$

6. Построить матрицу C:

$$C = A \times B^{-1}$$

4.7. Решение системы линейных уравнений методом Гаусса—Жордана

Для нахождении коэффициентов в матрицах, составляющих матрицу перспективного преобразования, требуется решить систему линейных уравнений. Это делается с помощью метода Гаусса-Жордана, являющегося модификацией метода Гаусса.

Оба метода используют тот факт, что элементарные преобразования матрицы, не меняют множества решений системы линейных уравнений, которую эта матрица представляет. Под элементарными преобразованиями матрицы подразумеваются:

- Перестановка строк матрицы:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \to \begin{bmatrix} a_{21} & a_{22} & a_{23} \\ a_{11} & a_{12} & a_{13} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

- Умножение строки матрицы на ненулевую константу:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \to \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ \beta a_{21} & \beta a_{22} & \beta a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

- Прибавление одной строки матрицы, умноженной на константу, к другой:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \to \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} + \beta a_{11} & a_{32} + \beta a_{12} & a_{33} + \beta a_{13} \end{bmatrix}$$

Метод Гаусса-Жордана используется для решения системы линейных уравнений, представленной в следующем виде:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Данную систему можно записать следующим образом:

$$\left[\begin{array}{ccc|c} a_{11}x_1 & a_{12}x_2 & a_{13}x_3 & b_{13} \\ a_{21}x_1 & a_{22}x_2 & a_{23}x_3 & b_{23} \\ a_{31}x_1 & a_{32}x_2 & a_{33}x_3 & b_{33} \end{array}\right]$$

Теперь можно прибавить первую строку к остальным строкам. При каждом сложении умножая её на отрицание отношения первого элемента строки, к которой она прибавляется, к её первому элементу. После этого, все значения в первом столбце, кроме первого, станут равны нулю:

$$\left[\begin{array}{ccc|c} a_{11}x_1 & a_{12}x_2 & a_{13}x_3 & b_1 \\ a_{21}x_1 & a_{22}x_2 & a_{23}x_3 & b_2 \\ a_{31}x_1 & a_{32}x_2 & a_{33}x_3 & b_3 \end{array}\right] \to \left[\begin{array}{ccc|c} a_{11}x_1 & a_{12}x_2 & a_{13}x_3 & b_1 \\ 0 & \dot{a}_{22}x_2 & \dot{a}_{23}x_3 & \dot{b}_2 \\ 0 & \dot{a}_{32}x_2 & \dot{a}_{33}x_3 & \dot{b}_3 \end{array}\right],$$

$$\dot{a}_{22} = a_{22} - \frac{a_{21}}{a_{11}} a_{12}\,,\ \ \dot{a}_{23} = a_{23} - \frac{a_{21}}{a_{11}} a_{13}\,,\ \ \dot{b}_2 = b_2 - \frac{a_{21}}{a_{11}} b_1\,,$$

$$\dot{a}_{32} = a_{32} - \frac{a_{31}}{a_{11}} a_{12}\,,\ \ \dot{a}_{33} = a_{33} - \frac{a_{31}}{a_{11}} a_{13}\,,\ \ \dot{b}_3 = b_3 - \frac{a_{31}}{a_{11}} b_1\,,$$

Таким же образом, прибавляя вторую строку к остальным, умножая её на отношение второго элемента строки, к которой она прибавляется к её второму элементу, можно обнулить весь второй столбец за исключением одного элемента:

$$\left[\begin{array}{ccc|c} a_{11}x_1 & a_{12}x_2 & a_{13}x_3 & b_1 \\ 0 & \dot{a}_{22}x_2 & \dot{a}_{23}x_3 & \dot{b}_2 \\ 0 & \dot{a}_{32}x_2 & \dot{a}_{33}x_3 & \dot{b}_3 \end{array}\right] \to \left[\begin{array}{ccc|c} a_{11}x_1 & 0 & \ddot{a}_{13}x_3 & \ddot{b}_1 \\ 0 & \dot{a}_{22}x_2 & \dot{a}_{23}x_3 & \dot{b}_2 \\ 0 & 0 & \ddot{a}_{33}x_3 & \ddot{b}_3 \end{array}\right],$$

$$\ddot{a}_{13} = a_{13} - \frac{a_{12}}{\dot{a}_{22}}\dot{a}_{23}\ ,\ \ \ddot{b}_1 = b_1 - \frac{a_{12}}{\dot{a}_{22}}\dot{b}_2$$

$$\ddot{a}_{33} = \dot{a}_{33} - \frac{\dot{a}_{32}}{\dot{a}_{22}}\dot{a}_{23}\ ,\ \ \ddot{b}_3 = \dot{b}_3 - \frac{\dot{a}_{32}}{\dot{a}_{22}}\dot{b}_2$$

Данные действия можно повторять для оставшихся строк, пока все значения в левой части матрицы, кроме тех, что лежат на главной диагонали, не станут равны нулю. В матрице, приведенной в примере, осталась только третья строка:

$$\left[\begin{array}{ccc|c} a_{11}x_1 & 0 & \ddot{a}_{13}x_3 & \ddot{b}_1 \\ 0 & \dot{a}_{22}x_2 & \dot{a}_{23}x_3 & \dot{b}_2 \\ 0 & 0 & \ddot{a}_{33}x_3 & \ddot{b}_3 \end{array}\right] \to \left[\begin{array}{ccc|c} a_{11}x_1 & 0 & 0 & \tilde{b}_1 \\ 0 & \dot{a}_{22}x_2 & 0 & \tilde{b}_2 \\ 0 & 0 & \ddot{a}_{33}x_3 & \ddot{b}_3 \end{array}\right],$$

$$\tilde{b}_1 = \ddot{b}_1 - \frac{\ddot{a}_{13}}{\ddot{a}_{33}}\ddot{b}_3\ ,\ \ \tilde{b}_2 = \dot{b}_2 - \frac{\dot{a}_{23}}{\ddot{a}_{33}}\ddot{b}_3$$

После этого, если разделить каждую строку на оставшийся элемент, лежащий на главной диагонали, то левая часть будет содержать только переменные, а правая часть будет являться искомым решением.

$$\left[\begin{array}{ccc|c} a_{11}x_1 & 0 & 0 & \tilde{b}_1 \\ 0 & \dot{a}_{22}x_2 & 0 & \tilde{b}_2 \\ 0 & 0 & \ddot{a}_{33}x_3 & \ddot{b}_3 \end{array}\right] \to \left[\begin{array}{ccc|c} x_1 & 0 & 0 & \hat{b}_1 \\ 0 & x_2 & 0 & \hat{b}_2 \\ 0 & 0 & x_3 & \hat{b}_3 \end{array}\right] \to \begin{cases} x_1 = \hat{b}_1 \\ x_2 = \hat{b}_2 \\ x_3 = \hat{b}_3 \end{cases},$$

$$\hat{b}_1 = \frac{\tilde{b}_1}{a_{11}}\ ,\ \ \hat{b}_2 = \frac{\tilde{b}_2}{\ddot{a}_{22}}\ ,\ \ \hat{b}_3 = \frac{\ddot{b}_3}{\ddot{a}_{33}}$$

В общем виде алгоритм будет выглядеть следующим образом:

a_{ij} — коэффициент перед x_j, стоящий на j-й позиции в i-й строке.
b_i — элемент правой части, стоящий в i-й строке.

1. Для каждого i от 0 до $n-1$:
   Для каждого j от 0 до $n-1$:
   Обнулить все элементы i-го столбца левой части, кроме того, что стоит на главной диагонали:

$$a_{jk} = a_{jk} - \frac{a_{ji}}{a_{ii}}a_{ik}\ ,\ \ k \in [0; n-1];\ \ b_j = b_j - \frac{a_{ji}}{a_{ii}}b_i$$

2. Для каждого i от 0 до $n-1$:
   Для каждого j от 0 до $n-1$:

$$a_{ij} = \frac{a_{ij}}{a_{ii}}\ ,\ b_j = \frac{b_j}{a_{ii}}$$

Важное замечание: перед тем как решать систему линейных уравнений методом Гаусса-Жордана, необходимо убедиться, что на главной диагонали матрицы, представляющей эту систему нет нулей. А если нули есть, то можно попробовать переставить строки так, чтобы они не находились на главной диагонали. В случае, если этого сделать нельзя или матрица вообще не является квадратной, единственного решения не существует.

4.8. Нахождение обратной матрицы методом Гаусса—Жордана

Метод Гаусса—Жордана также можно использовать для нахождения обратной матрицы. Так как матрица, обратная к матрице A — это матрица, которая при перемножении с матрицей A дает единичную матрицу, задачу о нахождении обратной матрицы можно записать так:

$$\begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{21} & b_{31} \\ b_{12} & b_{22} & b_{32} \\ b_{13} & b_{23} & b_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Каждый столбец результата умножения матрицы A на матрицу B, является результатом умножения матрицы A на соответствующий столбец матрицы B. Следовательно, нахождение каждого столбца обратной матрицы можно сформулировать как отдельную задачу:

$$\begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} \\ b_{12} \\ b_{13} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}; \begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{21} \\ b_{22} \\ b_{23} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix};$$

$$\begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{31} \\ b_{32} \\ b_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}.$$

Теперь видно, что для нахождения каждого столбца обратной матрицы необходимо решить систему линейных уравнений. Причем, для всех столбцов коэффициенты перед неизвестными одинаковые. А так как при решении системы линейных уравнений методом Гаусса—Жордана в левой части изменяются только эти коэффициенты, достаточно выполнить все необходимые преобразования один раз, дублируя их для каждой правой части.

Таким образом можно использовать компактную запись, похожую на ту запись, что была использована при решении обычных систем линейных уравнений:

$$\left[\begin{array}{ccc|ccc} a_{11} & a_{21} & a_{31} & 1 & 0 & 0 \\ a_{12} & a_{22} & a_{32} & 0 & 1 & 0 \\ a_{13} & a_{23} & a_{33} & 0 & 0 & 1 \end{array}\right].$$

Далее, также как и при использовании обычного метода Гаусса—Жордана, левая часть сводится к диагональной матрице, при этом все преобразования выполняются и над правой частью. Сами неизвестные не пишутся, поэтому вместо них в левой части в итоге должны быть единицы. Проще говоря, левая часть сводится к единичной матрице:

$$\left[\begin{array}{ccc|ccc} a_{11} & a_{12} & a_{13} & 1 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 1 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & 1 \end{array}\right] \rightarrow \left[\begin{array}{ccc|ccc} a_{11} & a_{12} & a_{13} & 1 & 0 & 0 \\ 0 & \dot{a}_{22} & \dot{a}_{23} & \dot{b}_{21} & 1 & 0 \\ 0 & \dot{a}_{32} & \dot{a}_{33} & \dot{b}_{31} & 0 & 1 \end{array}\right],$$

$$\dot{a}_{22} = a_{22} - \frac{a_{21}}{a_{11}} a_{12}, \ \dot{a}_{23} = a_{23} - \frac{a_{21}}{a_{11}} a_{13}, \ \dot{b}_{21} = -\frac{a_{21}}{a_{11}}, \rightarrow$$

$$\dot{a}_{32} = a_{32} - \frac{a_{31}}{a_{11}} a_{12}, \ \dot{a}_{33} = a_{33} - \frac{a_{31}}{a_{11}} a_{13}, \ \dot{b}_{31} = -\frac{a_{31}}{a_{11}}.$$

$$\rightarrow \left[\begin{array}{ccc|ccc} a_{11} & 0 & \ddot{a}_{13} & \ddot{b}_{11} & \ddot{b}_{12} & 0 \\ 0 & \dot{a}_{22} & \dot{a}_{23} & \dot{b}_{21} & 1 & 0 \\ 0 & 0 & \ddot{a}_{33} & \ddot{b}_{31} & \ddot{b}_{32} & 1 \end{array}\right],$$

$$\ddot{a}_{13} = a_{13} - \frac{a_{12}}{\dot{a}_{22}} \dot{a}_{23}, \ \ddot{b}_{11} = 1 - \frac{a_{12}}{\dot{a}_{22}} \dot{b}_{21}, \ \ddot{b}_{12} = -\frac{a_{12}}{\dot{a}_{22}} \rightarrow$$

$$\ddot{a}_{33} = \dot{a}_{33} - \frac{\dot{a}_{32}}{\dot{a}_{22}} \dot{a}_{23}, \ \ddot{b}_{31} = \dot{b}_{31} - \frac{\dot{a}_{32}}{\dot{a}_{22}} \dot{b}_{21}, \ \ddot{b}_{32} = -\frac{\dot{a}_{32}}{\dot{a}_{22}}$$

$$\left[\begin{array}{ccc|ccc} a_{11} & 0 & 0 & \tilde{b}_{11} & \tilde{b}_{12} & \tilde{b}_{13} \\ 0 & \dot{a}_{22} & 0 & \tilde{b}_{21} & \tilde{b}_{22} & \tilde{b}_{23} \\ 0 & 0 & \ddot{a}_{33} & \ddot{b}_{31} & \ddot{b}_{32} & 1 \end{array}\right],$$

$$\rightarrow \begin{array}{l} \tilde{b}_{11} = \ddot{b}_{11} - \frac{\ddot{a}_{13}}{\ddot{a}_{33}}\ddot{b}_{31}\,,\ \tilde{b}_{11} = \ddot{b}_{12} - \frac{\ddot{a}_{13}}{\ddot{a}_{33}}\ddot{b}_{32}\,,\ \tilde{b}_{13} = -\frac{\ddot{a}_{13}}{\ddot{a}_{33}} \\ \tilde{b}_{21} = \dot{b}_{21} - \frac{\dot{a}_{23}}{\ddot{a}_{33}}\ddot{b}_{31}\,,\ \tilde{b}_{22} = 1 - \frac{\dot{a}_{23}}{\ddot{a}_{33}}\ddot{b}_{32}\,,\ \tilde{b}_{23} = -\frac{\dot{a}_{23}}{\ddot{a}_{33}} \end{array} \rightarrow$$

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & \hat{b}_{11} & \hat{b}_{12} & \hat{b}_{13} \\ 0 & 1 & 0 & \hat{b}_{21} & \hat{b}_{22} & \hat{b}_{23} \\ 0 & 0 & 1 & \hat{b}_{31} & \hat{b}_{32} & \hat{b}_{33} \end{array}\right],$$

$$\begin{array}{l} \hat{b}_{11} = \frac{\tilde{b}_{11}}{a_{11}}\,,\ \hat{b}_{12} = \frac{\tilde{b}_{12}}{a_{11}}\,,\ \hat{b}_{13} = \frac{\tilde{b}_{13}}{a_{11}} \\ \hat{b}_{21} = \frac{\tilde{b}_{21}}{\dot{a}_{22}}\,,\ \hat{b}_{22} = \frac{\tilde{b}_{22}}{\dot{a}_{22}}\,,\ \hat{b}_{23} = \frac{\tilde{b}_{23}}{\dot{a}_{22}} \\ \hat{b}_{31} = \frac{\ddot{b}_{31}}{\ddot{a}_{33}}\,,\ \hat{b}_{32} = \frac{\ddot{b}_{32}}{\ddot{a}_{33}}\,,\ \hat{b}_{33} = \frac{1}{\ddot{a}_{33}} \end{array}$$

После этого, записав полученную матрицу в виде нескольких систем линейных уравнений, можно получить значение каждой ячейки обратной матрицы:

$$\begin{cases} b_{11} = \hat{b}_{11} \\ b_{21} = \hat{b}_{21} \\ b_{31} = \hat{b}_{31} \end{cases}, \begin{cases} b_{12} = \hat{b}_{12} \\ b_{22} = \hat{b}_{22} \\ b_{32} = \hat{b}_{32} \end{cases}, \begin{cases} b_{13} = \hat{b}_{13} \\ b_{23} = \hat{b}_{23} \\ b_{33} = \hat{b}_{33} \end{cases}$$

Полностью алгоритм нахождения обратной матрицы можно записать так:

a_{ij} — элемент левой части, стоящий на j-й позиции в i-й строке.
b_{ij} — элемент правой части, стоящий на j-й позиции в i-й строке.

1. Для каждого i от 0 до $n-1$:
   Для каждого j от 0 до $n-1$:
   Обнулить все элементы i-го столбца левой части, кроме того, что стоит на главной диагонали:

$$a_{jk} = a_{jk} - \frac{a_{ji}}{a_{ii}}a_{ik}\,,\ b_{jk} = b_{jk} - \frac{a_{ji}}{a_{ii}}b_{ik}\,,\ k \in [0; n-1];$$

2. Для каждого i от 0 до $n-1$:
   Для каждого j от 0 до $n-1$:

$$a_{ij} = \frac{a_{ij}}{a_{ii}}\,,\ b_{ij} = \frac{b_{ij}}{a_{ii}}$$

5. Работа с видеопотоком

Видеопоток является цифровым представлением видео в форме непрерывно идущих битов. Также как и аналоговое видео видеопоток содержит в себе отдельные *кадры*, которые должны идти с определенной частотой, например, 24 кадра в секунду. Каждый кадр также представляет из себя набор битов. Если видеопоток не закодирован, то он, по сути, является просто набором последовательно идущих битовых представлений кадров.

Алгоритмы для поиска пластины с номером работают с отдельным изображением. Чтобы сканировать с помощью них видеопоток, следует извлекать из этого видеопотока кадры и уже работать с ними. Так как задача распознавания номеров подразумевает использование видеопотока, поступающего в реальном времени, возникает ряд проблем, которые необходимо решить.

Первая из них — это скорость работы алгоритмов, выполняющих поиск пластины с номером. Так как одновременно можно работать только с одним кадром, то во время его сканирования остальные кадры приходится пропускать. Понятно, что если между обрабатываемым и пропущеными кадрами разница во времени небольшая, то вероятность пропустить что-то важное тоже невелика. Однако, если сканирование будет занимать слишком много времени, есть вероятность, что на пропущеных кадрах может оказаться новая пластина с номером.

Вторая проблема связана с первой. Как уже говорилось, кадры, приходящие во время сканирования придется пропустить. А так как сканирование разных кадров может занимать разное количество времени, необходимо определять, когда эти кадры пропускать, а когда принимать. Данную проблему усугубляет тот факт, что в случае закодированного видеопотока, о котором пойдет речь в следующем разделе, каждый следующий кадр использует информацию из предыдущих. По этой причине, приходится принимать кадры, даже если нет возможности их просканировать.

Первая проблема была подробно рассмотрена во втором разделе. Именно для её решения были выполнены все оптимизации, описанные в разделе, посвященном каскадам Хаара. Вторая проблема решается с помощью выполнения сканирования и декодирования кадров в разных процессах, используя средства межпроцессного взаимодействия. Этому и посвящен данный раздел.

5.1. Закодированный поток изображений

Как уже говорилось, видеопоток в незакодированном виде является просто набором последовательно идущих кадров. Однако, в таком виде для его передачи потребуется канал с огромной пропускной способностью. К примеру, передача видеопотока с разрешением 1280×720, частотой 25 кадров секунду и глубиной цвета 10 бит, где кадры представлены в цветовой схеме *Y'CrCb* с цветовой субдискретизацией $4{:}2{:}2$, потребует скорости как

минимум 54,9 мегабайт в секунду.

Однако, если использовать *кодирование*, можно значительно снизить требования к пропускной способности канала. Под кодированием видеопотока обычно подразумевается его сжатие, т.е. изменение структуры данных в нем таким образом, чтобы снизить количество передаваемой информации этом оставляя неизменным или почти неизменным само видео, представляемое этим потоком.

Существует много разных способов кодирования видео (их иногда называют *кодеками*). Но, в основе большинства из них лежит одна и та же особенность видео — многие пиксели в соседних кадрах имеют одинаковые или похожие значения. Благодаря этому нет необходимости хранить значение каждого пикселя каждого кадра.

К примеру, можно кодировать видео, запоминая значение каждого пикселя в каком либо кадре, называемом ключевым кадром, и то, в скольки кадрах после этого оно повторяется. Таким образом, для последующих кадров, не потребуется передавать значение всех их пикселей. На практике используются более сложные методы, однако, этот пример хорошо описывает принцип работы большинства из них.

Сторона, принимающая закодированный видеопоток, должна *декодировать* его, чтобы извлекать из него кадры. И, как уже было написано выше, даже если нет необходимости обрабатывать каждый кадр, при декодировании все-равно нужно принимать весь видеопоток. Проблема, связанная с этим и её решение будут описаны в разделе, посвященному сканированию непрерывного видеопотока. В данной работе для приема видеопотока и его декодирования с последующим извлечением кадров используется библиотека *LibAV*, которая подробно описана в следующем разделе.

5.2. LibAV

LibAV — библиотека для работы с видеокодеками, которая предоставляет удобный интерфейс на языке Си. В данном разделе описываются функции этой библиобтеки, предназначенные для приема и декодирования видеопотока, а также основные структуры используемые в LibAV.

В LibAV используется множетсво структур. Однако, для приема и декодирования видеопотока достаточно только нескольких из них. Так как эти структуры имеют очень много разных полей полного их описания приводиться не будет, а отдельные поля будут описаны по необходимости.

`struct AVFormatContext` — содержит данные открытого видеопотока.
`struct AVCodec` — содержит данные открытого видеокодека.
`struct AVCodecContext` — контекст, позволяющий работать с открытым кодеком.

Открытие видеотока выполняется с помощью функции

```
int avformat_open_input(AVFormatContext ** ps,
                        const char *       filename,
                        AVInputFormat *    fmt,
                        AVDictionary **    options
                       );
```

, где `ps` — структура, куда будут помещены данные открытого видеопотока, `filename` — имя открываемого видеопотока: может быть как путём к файлу, так и URL, `fmt` — формат в котором следует принимать видеопоток, если задан как `NULL`, то формат выбирается автоматически, а `options` — дополнительные опции, может передаваться как `NULL`.

Видеопоток может также содержать аудиопоток, идущий одновременно с ним. Поэтому необходимо найти в нем поток содержащий именно видеоинформацию. Для это этого сначала необхомо вызвать функцию

```
int avformat_find_stream_info(AVFormatContext * ic,
                              AVDictionary **   options
                             );
```

, где `ic` — открытый видеопоток, а `options` — дополнительные опции, может передаваться как `NULL`.

Эта функция позволяет получить данные о характеристиках открытого видеопотока, считав из него несколько кадров. Полученные данные записываются в соответствующие поля первого аргумента.

После этого можно найти необходимый поток, проверяя поле `codec` в каждом из потоков в структуре `AVFormatContext`.

```
for (sn = 0; sn < s-> nb_streams; ++sn)
        if (s->streams[sn]->codec->codec_type == AVMEDIA_TYPE_VIDEO)
                vstreamid = sn;
```

Поле `streams` является массивом структур `AVStream`, в котором каждый элемент соответствует одному потоку. Полу `codec` является указателем на контекст кодека, связанного с этим потоком, а поле `codec_type` структуры `AVCodecContext` является элементом перечисления `AVMediaType`, показывающим тип информации, с которой работает этот кодек.

После того как открыт сам видеопоток, необходимо открыть соответствующий видеокодек. Для этого кодек сначала ищется в системе функцией

```
AVCodec * avcodec_find_decoder(enum AVCodecID id);
```

, где `id` — идентификатор искомого кодека, содержащийся в структуре `AVStream` открытого видеопотока. После этого из открытого видеопотока копируется контекст. Чтобы это сделать надо выделить для него память функцией

```
AVCodecContext * avcodec_alloc_context3(const AVCodec *codec);
```

, где `codec` — это кодек, для которого создается контекст. А затем копировать значения полей из контекста в открытом видеопотоке функцией

```
int avcodec_copy_context(AVCodecContext *       dest,
                         const AVCodecContext * src
                        );
```

, где `dest` — структура, куда выполняется копирование, а `src` — структура, из которой выполняется копирование. Некоторые форматы устроены так, что в присылаемом пакете может быть неполный кадр. Необходимо проверить, может ли выбранный кодек работать в таком режиме, и, если может, установить соответствующий флаг в его контексте:

```
if (vcodec->capabilities & CODEC_CAP_TRUNCATED)
        vcodecc->flags |= CODEC_FLAG_TRUNCATED;
```

, где `vcodec` — кодек, а vcodecc — связаный с ним контекст. Далее кодек открывается с помощью фукнции

```
avcodec_open2(AVCodecContext * avctx,
              const AVCodec *  codec,
              AVDictionary **  options
             );
```

, где `avctx` — контекст, связанный с открываемым кодеком, `codec` — открываемый кодек, а `options` — дополнительные опции, может передаваться как `NULL`. После этого контекст можно использовать для декодирования принимаемого видеопотока.

После того, как были открыт видеопоток и кодек, связаный с ним, можно принимать и декодировать кадры. Для это используется функция

```
int av_read_frame(AVFormatContext * s,
                  AVPacket *        pkt
                 );
```

, где первый аргумент — открытый видеопоток, а второй агрумент — структура

```
struct AVPacket;
```

в которой, в незакодированном виде сохраняются принятые данные. Если элемент `buf` этой структуры в данном аргументе имеет значение `NULL`, то функция `av_read_frame` сохранит данные во внутреннем буффере LibAV.

Далее выполняется попытка декодировать полученные данные с помощью функции

```
int avcodec_decode_video2(AVCodecContext * avctx,
                          AVFrame *        picture,
                          int *            got_picture_ptr,
                          const AVPacket * avpkt
                         );
```

, где `avctx` — контекст кодека, которым будут декодироваться данные, `picture` - структура

```
struct AVFrame;
```

в которую записывается декодированный кадр, `got_picture_ptr` — адрес объявленного пользователем целого числа, куда будет записан 0, если этих данных недостаточно для декодирования полноценного кадра, или 1, если кадр был успешно декодирован, а `avpkt` — декодируемые данные.

Аргумент `got_picture_ptr` используется именно по той причине, что принятые данные не обязательно будут содержать полный кадр. Поэтому в случае, когда по адресу, указанному в этом аргументе оказывается 0, LibAV, запомнив переданные в аргументе *avpkt* данные, ожидает повторного вызова функции `avcodec_decode_video2` и не возвращает ошибки.

Для инициализации структуры `AVFrame` используется функция

```
AVFrame* av_frame_alloc(void);
```

возвращающая указатель на эту структуру для которой уже выделена память и установлены значения полей по умолчанию. Если эта структура больше не требуется, то вызывается функция

```
void av_frame_free(AVFrame ** frame);
```

освобождающая выделенную для нее память.

Также необходимо учесть, что полученнные данные не обязательно являются видеоданными. Они могут, например, являться данными параллельно идущего с видео аудиопотока. Для того, чтобы узнать к какому потоку относятся данные можно проверить значение поля `stream_index` структуры `AVPacket` сравнив его с найденным при открытии видеопотока номера потока содержащего видео.

После того, как была выполнена попытка декодировать кадр, независмо от успеха, из структуры `AVPacket` удаляется ссылка на буффер, в котором были сохранены данные. Делается это с помощью функции

```
void av_packet_unref(AVPacket * pkt);
```

, `pkt` — структура, содержащая принятые данные.

Для удобства в данной работе написана функция `readframe`, вызывающая внутри себя функции `av_read_frame` и `av_codec_decode_video2`:

```
int readframe(AVFormatContext *s, AVCodecContext *vcodecc,
        int vstreamid, AVFrame *frame, int *isdecoded)
{
        AVPacket pack;
        int ret;

        memset(&pack, 0, sizeof(AVPacket));
        pack.data = NULL;
        pack.buf = NULL;
        pack.size = 0;

        if ((ret = av_read_frame(s, &pack)) < 0) {
                if (ret != AVERROR_EOF) {
                        char buf[255];

                        av_strerror(ret, buf, 255);
                        fprintf(stderr, "%s0, buf);

                        return (-1);
                }

                return 1;
        }

        if (pack.stream_index == vstreamid) {
                int cret;

                if ((cret = avcodec_decode_video2(vcodecc, frame,
                        isdecoded, &pack)) < 0) {
                        char buf[255];

                        av_strerror(cret, buf, 255);
                        fprintf(stderr, "%s0, buf);

                        return (-1);
                }
        }

        av_packet_unref(&pack);

        return 0;
}
```

Аргумент `s` — открытый видеопоток, `vodecc` — кодек, предназначеный для декодирования этого видеопотока, `vstreamid` — номер потока с видео, `frame` — структура, куда будет сохранён полученный кадр, должна быть инициализирована пользователем, а `isdecoded` — аналогична аргументу `got_picture_ptr` функции `av_codec_decode_video2`.

5.3. Сканирование непрерывного видеопотока

При работе с непрерывным видеопотоком сканирование и декодирование выполняется в разных процессах. Так как большинство современных компьютеров способны одновременно

выполнять несколько процессов, это решает проблему связанную с необходимостью непрерывно принимать кадры: пока один процесс сканирует с доступной ему скоростью, другой процесс продолжает приём и декодирование.

Однако, возникает другая проблема — декодирующий процесс должен каким-то образом передавать сканирующему новые кадры. В данной работе для этого используются способы межпроцессного взаимодействия систем семейства Unix. Для передачи декодирующим процессом готовых кадров используется разделяемый сегмент памяти, к которому имеют доступ оба процесса. А для управления порядком доступа к этому сегменту используется неименованный канал.

Для порождения новых процессов используется функция

```
pid_t fork(void);
```

, создающая точную копию вызывающего процесса. Причем, в родительском процессе она возвращает *pid* (идентификатор процесса в системе) дочернего процесса, а в дочернем возвращает 0. Благодаря этому решаются сразу две проблемы. Во-первых, проверяя возвращаемое значение, можно разделить код на тот, что выполняется в дочернем процессе, тот что выполняется в родительском. Во-вторых, идентификаторы — это способ для родительского процесса различать дочерние.

Чтобы создать разделяемую область памяти, используется функция, изначально предназначенная для отображения файлов и устройств в память:

```
void *mmap(void *addr, size_t length, int prot,
           int flags, int fd, off_t offset);
```

, где `addr` — адрес, в который требуется выполнить отображение, если он передаётся как `NULL`, то ядро системы само выбирает адрес, `length` — длина отображения в байтах, `prot` — разрешенные способы доступа, `flags` — определяет, доступно ли данное отображение другим процессам, а также позволяет управлять радом других возможностей, `fd` — дескриптор файла или устройства, которое требуется отобразить, а `offset` — смещение в отображаемом файле или устройстве. Возвращаемое значение — указатель на область памяти, в которую было выполнено отображение.

Данная функция позволяет создавать анонимные отображения, не требующие привязки к какому-либо файлу или устройству. Если сделать такое отображение доступным другим процессам, то после вызова `fork` к этой области памяти смогут обращаться и родительский, и дочерний процесс. Разделяемое аноминое отображение создаётся путем передачи в аргумент `flags` скомбинированных с помощью побитового «или» значений `MAP_ANON` и `MAP_SHARED`.

Для того, чтобы удалить созданное отображение, используется функция

```
int munmap(void *addr, size_t length);
```

, где `addr` — адрес, в который было выполнено отображение, а `length` — длина этого отображения.

Неименованный канал позволяет передавать данные между родственными процессами и представляет из себя связку из двух дескрипторов: дескриптор для записи и дескриптор для чтения. Все данные, записанные в первый дескриптор, можно считать через второй. А так как при вызове функции `fork` открытые дескрипторы родительского процесса копируются, то если один из родственных процессов будет записывать данные в дескриптор для записи, другой сможет их прочитать из дескриптора для чтения. Из-за принципа их работы, неименованные каналы называют трубами (`pipes`).

Для создания неименованных каналов используется функция

```
int pipe(int pipefd[2]);
```

, где `pipefd` — массив из двух целых: после вызова функции его первый элемент будет дескриптором для чтения, а второй — дескриптором для записи. Данная функция возвращает 0, если выполнена успешно и −1 в случае ошибки. Если один созданных дескприпторов

больше не нужен, его, как и любой другой дескриптор, следует закрыть функцией

```
int close(int fd);
```

, где `fd` — закрываемый дескриптор.

Для записи в неименованный канал используется функция

```
ssize_t write(int fd, void *buf, size_t count);
```

, где `fd` — дескриптор, в который выполняется запись, `buf` — буффер, содержимое которого требуется записать в дексриптор, а `count` — количество байт из `buf`, которое необходимо записать.

Для чтения из неименованного канала используется функция:

```
ssize_t write(int fd, void *buf, size_t count);
```

, где `fd` — дескриптор, из которого выполняется чтение, `buf` — буффер, в который будет записаны прочитанные данные, а `count` — количество байт , которое требуется прочитать из дескриптора.

Для того чтобы узнать можно ли считать данные с дескритора, используется функция

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
        fd_set *exeptfds, struct timeval *timeout);
```

, где `nfds` самый большой номер среди номеров всех дескрипторов из наборов, передаваемых тремя следующими аргументами, увеличенный на 1, `readfds` — набор дескрипторов, на которых ожидается операция чтения, `writefds` — набор дескрипторов, на которых ожидается операция записи, `exceptfds` — набор дескрипторов, на которых ожидаются исключения, а `timeout` — структура, задающая время ожидания.

Набор дескрипторов является последовательностью байтов, где каждый *i*-й байт соответствует дескриптору с номером *i*. Если какой-либо бит в наборе установлен, то при передаче этого набора функции `select`, будет ожидаться событие (чтение, запись, исключение) на соответствующем этому биту дескрипторе. Обнуление набора дескрипторов выполняется макросом

```
void FD_ZERO(fd_set *set);
```

, где `set` — обнуляемый набор. Установка бита соответствующего нужному дескриптору выполняется макросом

```
void FD_SET(int fd, fd_set *set);
```

, где `fd` — дескриптор, соответствующий которому бит нужно установить, а `set` — набор, в котором этот бит устанавливается.

Функция `select` имеет широкое применение, однако в данной работе она применяется только для проверки того, можно ли считать байт из дескриптора для чтения неименованного канала.

5.4. Синхронизация процессов с помощью неименованных каналов

Для передачи кадров используется разделяемая память. Если декодирующему процессу требуется передать кадр, он копирует его в заранее созданый участок разделяемой памяти. Если сканирующему процессу требуется получить новый кадр, он читает его из этого же участка разделяемой памяти.

Возникает проблема: сканирующий процесс может начать читать кадр, в то время как декодирующий процесс записывает новый. Решить эту проблему можно добавив механизм,

позволяющий процессу узнать, можно ли в данный момент обращаться к данному участку разделяемой памяти. Иначе говоря, требуется реализовать механизм синхронизации процессов.

Реализованный в данной работе представляет интерфейс из нескольких функций:

```
int nd_psinitprefork();
```

вызывается перед вызовом функции `fork`. Инициализирует механизм синхронизации. Возвращает 0 в случае успешно выполнения и –1 в случае ошибки.

```
int nd_psinitpostfork(int pn);
```

, где `pn` — номер процесса, из которого выполняется вызов: 0 — декодирующий, а 1 — сканирующий. Вызывается после вызова функции `fork` в обоих процессах. Завершает инициализацию для каждого процесса отдельно.

```
int nd_pslock(int pn);
```

, где `pn` — номер процесса, из которого выполняется вызов. Вызывается перед работой с разделяемым участком памяти. Блокирует выполнение программы, пока другой процесс не сообщит о том, что он завершил работу с требуемым участком памяти. Возвращает 0 в случае успешно выполнения и –1 в случае ошибки.

```
int nd_psunlock(int pn);
```

, где `pn` — номер процесса, из которого выполняется вызов. Вызывается после завершения работы с разделяемым участком памяти. Сообщает другому процессу, о том, что работа с данным участком завершена. Возвращает 0 в случае успешно выполнения и –1 в случае ошибки.

```
int nd_pstrylock(int pn);
```

, где `pn` — номер процесса, из которого выполняется вызов. Как и `nd_pslock`, вызывается перед работой с разделяемым участком памяти, но в отличие от него не блокирует выполнение программы. Возвращает 0, требуемый участок памяти занят, 1, если свободен и –1 в случае ошибки.

```
int nd_psunlock(int pn);
```

, где `pn` — номер процесса, из которого выполняется вызов. Вызывается перед завершением работы процесса. Завершает работу механизма синхронизации.

Данные функции реализованы с помощью двух неименованных каналов. Если процессу нужно получить доступ к учаску разделяемой памяти, он пытается считать из одного канала байт, а если он завершил работу с этим участком он записывает байт в другой канал. Первый процесс читает из первого канала и записывает во второй, а другой процесс наоборот, записывает в первый канал и читает из второго.

Дескрипторы для чтения и записи обоих каналов объявлены как статические переменные в том же файле исходного кода, что и остальные функции механизма синхронизации:

```
static int pfdc[2];
static int pfdp[2];
```

Функция `nd_psinitprefork` создает оба неименованных канала, а также вызавает функцию `nd_psunlock`, чтобы разблокировать участок разделяемой памяти для декодирующего процесса:

```
int nd_psinitprefork()
{
        if (pipe(pfdc) < 0) {
                nd_seterrormessage("pipe error", __func__);
                return (-1);
        }

        if (pipe(pfdp) < 0) {
                nd_seterrormessage("pipe error", __func__);
                return (-1);
        }

        if (nd_psunlock(1) < 0) {
                nd_seterrormessage("pipe error", __func__);
                return (-1);
        }

        return 0;
}
```

Функция `nd_initpostfork` закрывает те дескрипторы, среди открытых функцией `nd_psinitprefork`, которые не будут использоваться в конкретном процессе:

```
int nd_psinitpostfork(int pn)
{
        assert(pn <= 1 && pn >= 0);

        if (pn == 0) {
                if (close(pfdc[0]) < 0 || close(pfdp[1]) < 0) {
                        nd_seterrormessage("pipe error", __func__);
                        return (-1);
                }
        }
        else {
                if (close(pfdc[1]) < 0 || close(pfdp[0]) < 0) {
                        nd_seterrormessage("pipe error", __func__);
                        return (-1);
                }
        }

        return 0;
}
```

Функция `nd_pslock` блокирует процесс, пока не будет считан байт с соответстующего неименованного канала:

```
int nd_pslock(int pn)
{
        int c;
        char buf[1024];
        int fd;

        assert(pn <= 1 && pn >= 0);

        fd = (pn == 0) ? pfdp[0] : pfdc[0];

        if ((c = read(fd, buf, 1024)) < 0) {
                nd_seterrormessage("pipe error", __func__);
                return (-1);
        }

        return 0;
}
```

Функция `nd_psunlock` отправляет байт в соответствующий неименованный канал:

```
int nd_psunlock(int pn)
{
        int c;
        int fd;

        assert(pn <= 1 && pn >= 0);

        fd = (pn == 0) ? pfdc[1] : pfdp[1];

        if ((c = write(fd, "u", 1)) < 0) {
                nd_seterrormessage("pipe error", __func__);
                return (-1);
        }

        return 0;
}
```

Функция `nd_pstrylock` проверяет с помощью функция `select`, можно ли считать байт из соответствующего неименнованого канала, и, если это возможно, считывает, иначе, возвращает 0:

```
int nd_pstrylock(int pn)
{
        fd_set readfs;
        struct timeval tv;
        int retval;
        int fd;

        assert(pn <= 1 && pn >= 0);
```

```
        fd = (pn == 0) ? pfdp[0] : pfdc[0];

        FD_ZERO(&readfs);
        FD_SET(fd, &readfs);

        tv.tv_sec = 0;
        tv.tv_usec = 0;

        if ((retval = select(fd + 1, &readfs, NULL, NULL, &tv)) < 0) {
                nd_seterrormessage("pipe error", __func__);
                return (-1);
        }

        if (retval > 0) {
                if (nd_pslock(pn) < 0)
                        return (-1);

                return 1;
        }

        return retval;
}
```

Функция `nd_psclose` закрывает все дескрипторы, используемые для синхронизации:

```
int nd_psclose(int pn)
{
        assert(pn <= 1 && pn >= 0);

        if (pn == 0) {
                if (close(pfdp[0]) < 0 || close(pfdc[1]) < 0) {
                        nd_seterrormessage("pipe error", __func__);
                        return (-1);
                }
        }
        else {
                if (close(pfdp[1]) < 0 || close(pfdc[0]) < 0) {
                        nd_seterrormessage("pipe error", __func__);
                        return (-1);
                }
        }

        return 0;
}
```

5.5. Декодирующий процесс

Декодирующий процесс читает кадры в бесконечном цикле. На каждой итерации, сначала с помощью функции `readframe` из видеопотока считываются данные, пока не будет декодирован один кадр. После этого считаный кадр переводится в цветовую схему RGB, а также создаётся его уменшенная копия, которая будет использоваться при сканировании.

Далее процесс пытается получить доступ к разделяемой памяти. Если она занята, то полученный кадр и его уменьшенная удаляются и процесс переходит к следующей итерации. Если процесс получает доступ к разделяемой памяти, он копирует в нее кадр и его уменьшенную копию и сообщает о завершении работы с разделяемой памятью.

В качестве первого аргумента функция основного цикла декодирующего процесса принимает структуру

```
struct avdata {
        AVFormatContext *s;
        AVCodecContext *vcodecc;
        AVFrame *frame;
        int vstreamid;
};
```

, в которой содержатся все необходимые для работы с видеопотоком структуры. Предполагается, что они инициализированы перед началом основного цикла.

В качестве второго и третьего аргумента принимаются структуры предназначенные для представления изображения в удобной для распознавания форме:

```
struct nd_image {
        int w;
        int h;
        enum ND_PIXELFORMAT format;
        double *data;
};
```

, где `w` и `h` — ширина и высота изображения соответственно, `format` — формат пикселей (ARGB, RGB, Grayscale), а `data` — данные изображения. Данные обоих изображений, передаваемых в функцию основного цикла декодирующего процесса разположены в разделяемом участке памяти и используются для передачи кадров сканирующему процессу.

Ниже приведен исходный код функции основного цикла декодирующего процесса:

```
int decodeloop(struct avdata *av, struct nd_image *img,
        struct nd_image *imgorig)
{
        struct playbackstate pbs;
        AVRational tmpr;

        tmpr = av->s->streams[av->vstreamid]->avg_frame_rate;
        if (pbstateinit(&pbs, (double) tmpr.num / tmpr.den) < 0)
                return 1;

        while (1) {
                int isdecoded;
                uint8_t *rgbdata;
                uint8_t *rgbdataorig;
                int rgblinesize;
                int rgblinesizeorig;
                int retval;

                isdecoded = 0;
```

```
                do {
                        if ((retval = readframe(av->s, av->vcodecc,
                                av->vstreamid, av->frame, &(isdecoded))) < 0)
                                return (-1);

                        if (retval > 0)
                                return 0;
                } while (!isdecoded);

                if (frametorgb(av->frame, img->w, img->h,
                        &rgbdata, &rgblinesize) < 0)
                        return (-1);

                if (frametorgb(av->frame, imgorig->w, imgorig->h,
                        &rgbdataorig, &rgblinesizeorig) < 0)
                        return (-1);

                if (nd_pstrylock(0)) {
                        rgbdatatoimg(rgbdata, rgblinesize, img);
                        rgbdatatoimg(rgbdataorig, rgblinesizeorig, imgorig);

                        if (nd_psunlock(0) < 0) {
                                fprintf(stderr, nd_geterrormessage());
                                return (-1);
                        }
                }

                free(rgbdata);
                free(rgbdataorig);
        }

        fprintf(stderr, "Unexpexted loop quit0);
        return (-1);
}
```

5.6. Сканирующий процесс

Сканирующий процесс загружает из памяти данные заранее обученного каскада Хаара, а затем устанавливает для него стандартную конфигурацию. После этого запускается бесконечный цикл, в котором на каждой итерации процесс ждет пока не освободится участок разделяемой памяти, и как только она освобождается, выполняет необходимые для распознавания действия. Когда сканирование кадра завершено, процесс сообщает о завершении работы с разделяемой памятью.

Основная функция сканирующего процесса принимает первыми двумя аргументами изображения, представленные структурой `struct nd_image`. Как в случае с декодирующем процессом, данные этих изображений расположены в разделяемой памяти и через них сканирующий процесс принимает кадры. Третьим аргументом данная функция принимает путь к данным заранее обученного каскада Хаара. А Четвертый аргумент — это путь к директории куда следует сохранять изображения обнаруженный номеров.

Ниже приведен исходный код основной функции сканирующего процесса:

```
int scanloop(struct nd_image *img, struct nd_image *imgorig,
        const char *hcpath, const char *outputdir)
{
        struct hcdata hcd;

        if (hc_hcascaderead(&(hcd.hc), hcpath) < 0) {
                fprintf(stderr, nd_geterrormessage());
                return (-1);
        }

        hc_confbuild(&(hcd.scanconf),
                hcd.hc.ww, hcd.hc.wh, img->w, img->h,
                0.9, 1, 1, threadcount);

        while (1) {
                struct nd_matrix3 perspmat;
                struct hc_rect *r;
                int rc;

                if (nd_pslock(1) < 0) {
                        fprintf(stderr, nd_geterrormessage());
                        return (-1);
                }

                if (img->data == NULL)
                        return 0;

                if (getperspmat(img->w, img->h, &perspmat) < 0)
                        return (-1);

                if (nd_imgapplytransform(img, &perspmat) < 0) {
                        fprintf(stderr, nd_geterrormessage());
                        return (-1);
                }

                if (hc_imgpyramidscan(&(hcd.hc), img, &r, &rc,
                        &(hcd.scanconf)) < 0) {
                        fprintf(stderr, nd_geterrormessage());
                        return (-1);
                }

                if (rc) {
                        printf("Detected.0);
                        detectedtofile(img, imgorig, &perspmat,
                                r, rc, outputdir);
                        free(r);
                }
```

```
                if (nd_psunlock(1) < 0) {
                        fprintf(stderr, nd_geterrormessage());
                        return (-1);
                }
        }

        fprintf(stderr, "Unexpexted loop quit0);
        return (-1);
}
```

6. Заключение

В результате проделанной работы был написан набор утилит для Unix-подобных систем, предназначенных для распознавания номеров. Их список включает в себя: утилиту для поиска слабых классификаторов каскада Хаара, утилиту для построения каскада Хаара из найденных примитивов, утилиту для поиска объектов на изображении с помощью обученного каскада Хаара (имеется уже обученный под поиск пластины с номером каскад), а также утилиту для исправления перспективных искажений в найденном изображении с пластиной.

Структура каскадов Хаара и алгоритм обучения были реализованы на основе оригинальной статьи. Также была предпринята попытка объяснить их суть доступным языком. Был добавлен ряд оптимизаций, не описаных в оригинальной статье, значительно увеличивающих скорость обучения.

Алгоритмы и фильтры, используемые при исправлении искажений перспективы (оператор Собеля, метод Оцу, детектор границ Канни, преобразование Хафа) были реализованы на основе их описаний в книге "Цифровая обработка изображений" Рафаеля Гонзалеза и Ричарда Вудса. При изучении особенностей фотографий с изображением автомобильных номеров был определен способ совмещения этих алгоритмов и фильтров, дающий описаные ниже результаты, а также были добавлены дополнительные алгоритмы для улучшения качества обнаружения границ.

Данная работа имеет ряд недостатков. Каскад Хаара не может обнаружать объекты, сильно наклоненные относительно того объекта, под который он обучался. Поэтому, автомобильные номера, сфотографированные под нестандартными углами, обычно не обнаружаются. Это не является серьезным недостатком, так как написанные утилиты в основном предназначены для специальных камер, снимающих машины под требуемым углом. Также, эту проблему можно решить, проверяя помимо исходного изображения его копии, повернутые на фиксированный угол.

Детектор границ Канни периодически может давать сбой. Это обычно связано с плохим освещением, грязными номерами, большим уровнем шума, яркими границами у пластины с номером или её физизескими искажениями (например, если пластина с номером загнута). В итоге, для дальнейшего распознавания пригодно только около 80% обнаруженных номеров.

Дальнешую работу планируется направить на исправление описанных выше недостатков. После этого планируется искать способы распознавания отдельных символов на изображении, полученном после исправления перспективных искажений.

7. Список литературы

1. *An Introduction to Practical Neural Networks and Genetic Algorithms For Engineers and Scientists. Christopher MacLeod.* — небольшая книга, посвященная нейронным сетям и генетическим алгоритмам. Также, в ней описаны разные методы, применяемые при машинном обучении.
2. *Rapid Object Detection using a Boosted Cascade of Simple Features. Paul Viola, Michael Jones.* — оригинальная статья, в которой были представлены каскады Хаара, содержит часть информации, необходимой для их реализации.
3. *Digital Image Processing. Rafael C. Gonzalez, Richard E. Woods.* — книга, посвященная цифровой обработке изображений. В ней описывается большая часть методов, применяемых в этой работе при исправлении искажений перспективы.
4. *Без паники! Цифровая обработка сигналов. Юкио Сато.* — введение в цифровую обработку сигналов.
5. *The Scientist and Engineer's Guide to Digital Signal Processing. Steven W. Smith.* — книга, посвященная цифровой обработке сигналов. В ней описано множетсво методов, применяемых улучшении сигналов и их анализе.