Object Detection I: Classical CV

tags: *Object Detection*

link1, *link2*, *link3*

1. Object Detection (OB) vs Object Recognition (OR)

(?) Что такое *Object Detection*? В чем отличие от *Object Recognition*?

OR - определяет присутствие определенного объекта в кадре, выдает метку и вероятность.

OD - выдает то же что и *OR* + *bounding box* (*x,y,w,h*), указывающий положение объекта в кадре.

Таким образом, в основе любого *OD* алгоритма стоит *OR* алгоритм. Чтобы локализовать объект, мы сперва должны выбрать субрегионы изображения (*patches*) и подать их на *OR*. Координаты объекта определяются по координатам патча, на котором *OR* выдал наибольшую вероятность.

2. Sliding Window

Самый очевидный способ выбирать патчи - использовать скользящее окно (*sliding window approach*). В таком подходе мы проходим окном по изображению, та область изображения, которая в данный момент покрыта окном - считается патчем и подается на вход *OR* алгоритму. Но мы должны пройтись окном не только по всему изображению, ну и также по его разным *масштабам* и *отношением сторон* (обычно моделируется изменением размера и пропорций окна).

(?) Какой недостаток? В итоге мы классифицируем десятки тысяч патчей, что налагает большую вычислительную трудоемкость.

3. Region Proposals

Преодолеть вышеназванные проблемы помогает *Region Proposal* подход. В таком подходе на вход алгоритма подается изображение, а на выходе патчи, которые с наибольшей вероятностью содержат все объекты на изображении (будем называть такие патчи - *регионы-кандидаты* / *RP*s). Эти *RP*s могут быть зашумленными, перекрываться друг с другом, а также вовсе не содержать объекты. Но идя в том, что среди этих *RP*s будут содержаться все корректные регионы, содержащие объекты, которые далее будут классифицированы при помощи *OR* алгоритма.

Как правило, *RP* алгоритмы используют *сегментацию*. Сегментация осуществляется группировкой смежных регионов на основе *метрик сходства* (по цвету, текстуре и тд.). В отличие от *sliding window* подхода, где объекты ищутся на уровне пикселей, *RP* алгоритмы группируют пиксели в небольшое число сегментов и работают на уровне этих сегментов. Поэтому *RP* алгоритмы генерируют на порядок меньшее количество патчей для дальнейшей их классификации *OR*.

Важным свойством Region Proposal подхода является высокая оценка *полноты* (*recall*).

4. Список некоторых RP алгоритмов

1) *Objectness*
2) *Constrained Parametric Min-Cuts for Automatic Object Segmentation*
3) *Category Independent Object Proposals*
4) *Randomized Prim*
5) ***Selective Search***

5. Metrics

5.1 Precision and Recall

(?) Что такое TP/FP/TN/FN?

Precision = TP/(TP+FP) = TP/#ground_truths. Как много обнаруженных объектов действительно являются объектами. *Recall = TP/(TP+FN) = TP/#predictions*. Как много объектов алгоритм вообще смог обнаружить. *F1=2*(Recall*Precision)/(Recall+Precision)*. (см. *рис.1)*

Recall важен для *RP* алгоритмов, т.к. большое количество *FP* хоть и замедляет скорость работы, но не особо влияет на итоговую точность детекции (т.к. *FP* патчи будут отброшены *OR*). В то время как *FN* на точность влияет очень сильно, т.к. приводит к пропуску патчей содержащих объекты.

5.2 Intersection over Union (IoU)

Используется чтобы определить верен ли предсказанный бокс. Площадь пересечения предсказанного бокса с *gt* боксом делится на площадь их объединение (см. *рис.2*). Предсказание считается *TP*, если *IoU* >= заданного порога, иначе *FP*.

5.3 Average Precision (AP)

Рассмотрим пример (*рис. 3*): даны 2 изображения, на них 3 объекта (зеленым), и 4 предсказанных бокса (красным) с *confidence* (оценка уверенности детекции). Составим таблицу (*рис. 4*):

1) Каждая строка описывает предсказанный бокс, строки упорядочены по *confidence*.
2) Считаем *IoU*, и если *IoU* > *0.5*, то бокс *TP*, иначе *FP*. Если для одного объекта предсказано несколько боксов, то бокс с бОльшим *IoU* считается *TP*, а все остальные *FP*.
3) Считаем *Precision:* доля *TP* от общего числа детекций (на текущей строке и выше).
4) Считаем *Recall:* доля *TP* (на текущей строке и выше) от общего числа объектов (всего их 3).
5) Строим *Precision-Recall (PR, рис.5)*. Видно что PR-кривая имеет зигзагообразную форму, при этом *Precision* убывает, а *Recall* монотонно возрастает.
6) Площадь под PR-кривой называется *AP*. Вычисляется так: $\mathrm{AP} = \int_0^1 p(r)dr$.
7) Но перед вычислением AP полезно сгладить зигзагообразную форму кривой. Для этого на каждом значении *Recall* мы заменяем значение *Precision* на максимальное значение правее от него (*рис. 6*). Вычисленное AP на такой кривой менее подвержено небольшим колебаниям в ранге бокса (*confidence*). Математически: $p_{interp(r)} = \max_{\tilde{r}:\tilde{r}\geq r} p(r)$.

5.4 Mean Average Precision (mAP)

mAP это среднее по *AP* для всех классов. Чтобы вычислить mAP сперва нужно вычислить *AP* для каждого класса объектов: $\mathrm{AP} = \frac{1}{\#\text{classes}} \sum_{\text{class} \in \text{classes}} AP[class]$. Но для разных датасетов/соревнований используются немного разные метрики, но наиболее общепринятые это PASCAL VOC и MS COCO.

4.5.1 PASCAL VOC 2007.

Использовалось *аппроксимированное AP (AAP)*: на PR-кривой берется среднее по 11 точкам *Precision*, полученным с равным шагом по оси *Recall*: [0,0.1,…,1] (*рис. 7*). Математически:

$AP = \frac{1}{11} \sum_{r\in(0,0.1,\ldots,1)} p_{interp(r)}$, где $p_{interp(r)} = \max_{\tilde{r}:\tilde{r}\geq r} p(r)$. Однако такой способ вычисления *AP* страдает от недостатков: 1) это менее точно чем *AP*; 2) трудно измерять разницу для методов с низким *AP*.

4.5.2 PASCAL VOC 2010-2012. Area Under Curve (AUC).

Для более поздних VOC *AP* вычисляется по другому (без аппроксимации, т.н. *AUC*, рис. 8). PR-кривая делится на участки в тех точках, где максимальное значение *Precision* падает. Далее *AP* считается как сумма площадей прямоугольников: $\mathrm{AP} = \Sigma\,(r_{n+1} - r_n)\, p_{interp}(r_{n+1})$.

4.5.3 MS COCO mAP

В VOC детекция считалась *TP* только для одного значения *IoU>0.5*. Таким образом боксы с разным *IoU* (например, 0.6 и 0.9) имеют одинаковый вес при подсчете *mAP*, что вносит некоторый

bais в эту метрику. Чтобы избежать этого, в COCO *mAP* считается для нескольких порогов *IoU (AP@[.5:.05:.95])*, после чего берется их среднее (т.е. mAP это среднее от среднего и от среднего). Кроме того, используется аппроксимированное AP с 101 точкой (диапазон [0:.01:1]). (см. рис. 9)

$$mAP_{\text{COCO}} = \frac{mAP_{0.50} + mAP_{0.55} + \ldots + mAP_{0.95}}{10}.$$

Подведем итог (в наши дни используется только MS COCO):

1) PASCAL VOC 2008 использует *AAP* вычисленный на 11 точках.
2) PASCAL VOC 2010-2012 использует *AUC* (т.е. все точки *PR*-кривой).
3) MS COCO использует *AAP* вычисленный на 101 точке и 10 порогах *IoU*.
4) ImageNet используется *AUC*.

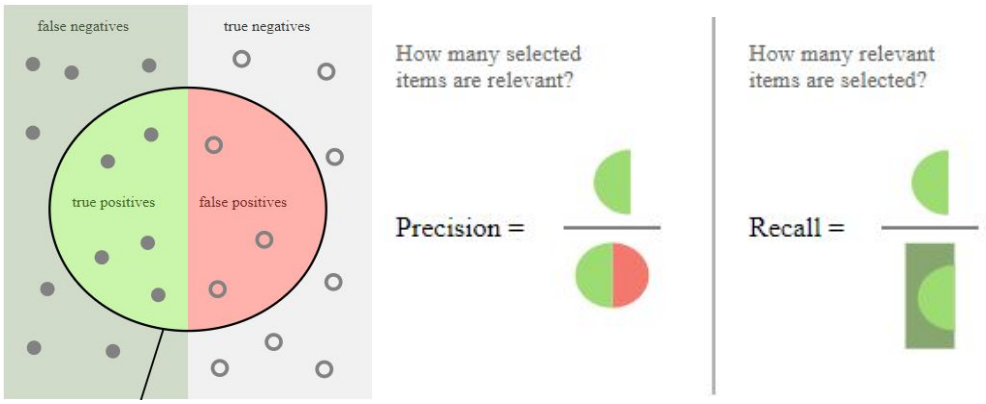


Рис. 1

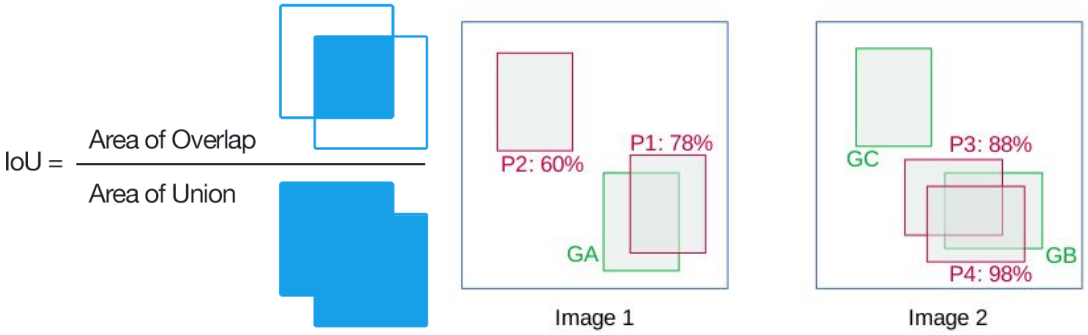


Рис.2 / Рис.3

Image	Detection	Confidence	IoU	Ground Truth	TP/FP	Acc TP	Acc FP	Precision	Recall
Image 2	P4	98%	> 0.5	GB	TP	1	0	1	0.33
Image 2	P3	88%	> 0.5	GB	FP	1	1	0.5	0.33
Image 1	P1	78%	> 0.5	GA	TP	2	1	0.67	0.67
Image 1	P2	60%	< 0.5	-	FP	2	2	0.5	0.67

```
Average Precision (AP):
  AP                % AP at IoU=.50:.05:.95 (primary challenge metric)
  AP^IoU=.50        % AP at IoU=.50 (PASCAL VOC metric)
  AP^IoU=.75        % AP at IoU=.75 (strict metric)
AP Across Scales:
  AP^small          % AP for small objects: area < 32^2
  AP^medium         % AP for medium objects: 32^2 < area < 96^2
  AP^large          % AP for large objects: area > 96^2
```

Рис.4 / Рис. 9

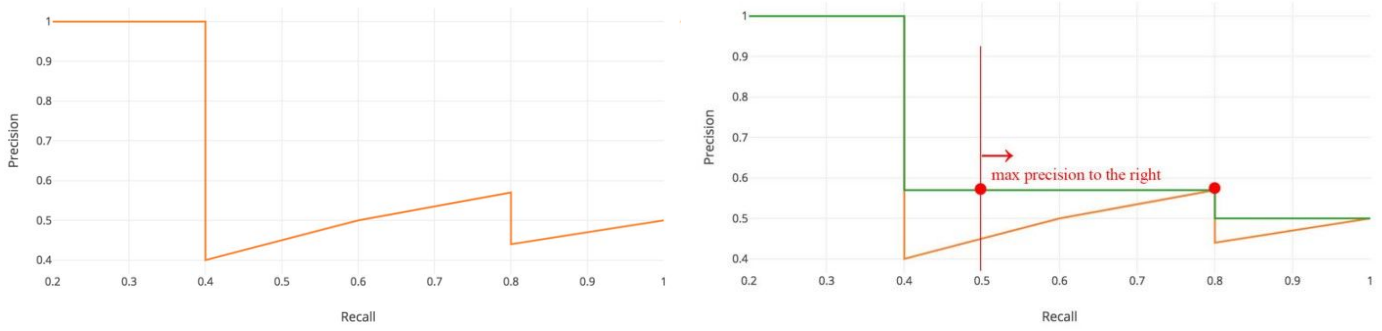
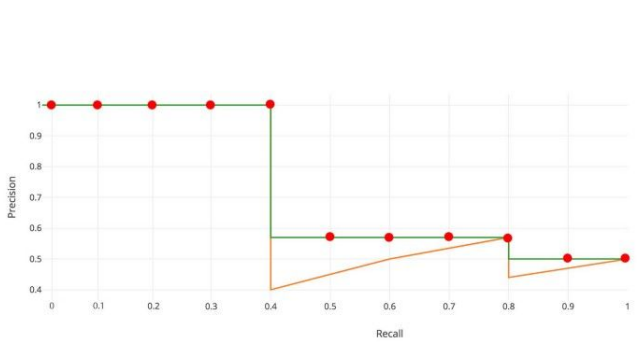


Рис. 5 / Рис. 6



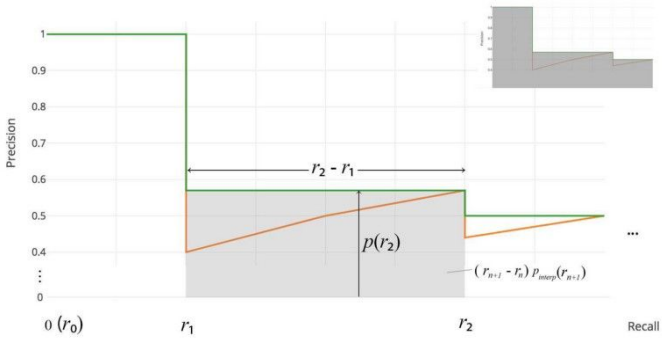


Рис. 7 / Рис. 8

6. Datasets for Object Detection

6.1 MS COCO

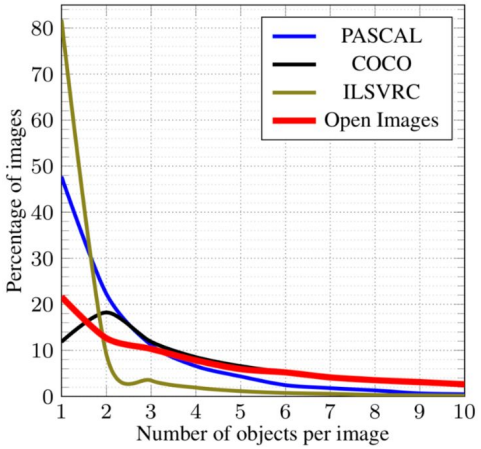
- 330k изображений
- 80 классов объектов
- 1.5m объектов размечено боксами (200k изображений)
- также содержит instance segmentation, keypoints для 250k человек, и описание контекста сцены
- 25 ГБ.

6.2 Open Images Dataset

- 9m изображений
- 600 классов объектов
- 16m объектов размечено боксами (1.7m изображений)
- также содержит instance segmentation, свойства объектов, взаимодействия между парами объектов, а также описание действий человека в сцене.
- 500 ГБ.

6.3 ImageNet

- 14m изображений
- 20k классов объектов (в том числе 120 пород собак)
- На 1.5m изображений объекты размечены боксами (всего 3к классов объектов)
- В наше время используется в основном для предобучения моделей
- 150 ГБ.



Selective Search for Object Detection, 2012

tags: *Selective Search, Object Detection, Region Proposals*

http://www.huppelen.nl/publications/selectiveSearchDraft.pdf

https://www.learnopencv.com/selective-search-for-object-detection-cpp-python/

https://arthurdouillard.com/post/selective-search/

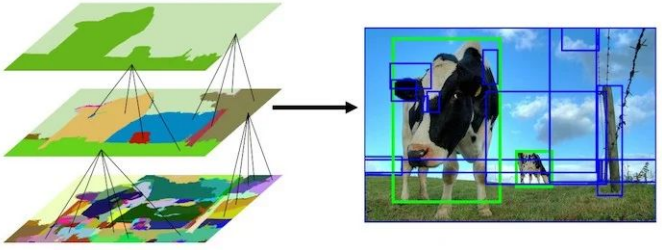
Selective Search (*SS*) это алгоритм семейства *region proposal* (*RP*). Обладает достоинствами: 1) инвариантен к размерам объектов; 2) относительно высокая скорость работы (на тот момент); 3) высокая оценка *полноты* (*recall*).


Алгоритм иерархически итеративно группирует схожие регионы основываясь на метриках сходства по цвету, текстуре, размеру и соответствию форм. На вход алгоритма поступает изображение, которое предварительно сегментировано на основе интенсивности пикселей (алгоритм *Graph Based Image Segmentatio*n, см рис.).

(?) Можем ли мы использовать эту сегментацию как *RP*s? Ответ - нет, по причинам: 1) большинство объектов имеют два и более сегментов; 2) для окклюдированных объектов (например, чашка с налитым кофе, или чашка на блюдце) не могут быть сгенерированы *RP*s. Если попытаемся исправить первую проблему при помощи объединения смежных сегментов, то в конечном итоге можем получить, что один сегмент будет покрывать два объекта. Но наша основная задача - найти как можно больше *RP*s, которые имеют высокое перекрытие с объектами.

1. Selective Search by Hierarchical Grouping

1) Описать *bounding boxes* (*bb*) вокруг каждого сегмента и добавить их в список *RP*s.
2) Сгруппировать смежные сегменты основываясь на метриках сходства.
3) Перейти к п.1.



На каждой итерации формируются бОльшие сегменты из маленьких (поэтому иерархический) и добавляются в список *RP*s (*bottom-up* подход). Алгоритм останавливается когда все изображение превращается в единый регион.

На рисунке изображены начальный, средний и последний шаг работы алгоритма.

Для большей устойчивости метода, этот алгоритм применяется для изображения в разных цветовых пространствах (*RGB, HSV, Lab* - lightness, green-red, blue-yellow).

2. Similarity

SS использует четыре метрики, которые считаются над фичами, вычисленными на основе значений пикселей. Чтобы каждый раз после слияния регионов не пересчитывать метрики, авторы разработали эти фичи так, чтобы их можно было объединять (*propagated* к новым регионам).

2.1. Color Similarity (Сходство по цвету)

Гистограмма строится из 25 бинов/интервалов для каждого цветового канала, затем эти гистограммы конкатенируются, чтобы получить итоговый цветовой дескриптор размерностью *25*3=75*. Цветовое сходство для двух регионов вычисляется на основе пересечения гистограмм (иначе говоря это количество одинаковых значений пикселов):

$$s_{color}(r_i, r_j) = \sum_{k=1}^{n} min(c_i^k, c_j^k)$$, где *c^k* - значение *k*-го бина в дескрипторе.

Слияние двух регионов происходит усреднение их гистограмм с учетом размеров регионов:

$$C_t = \frac{size(r_i) * C_i + size(r_j) * C_J}{size(r_i) + size(r_j)}$$

2.2. Texture Similarity (Сходство по текстуре)

Берутся Гауссовские производные (*Gaussian derivatives, SIFT)* в 8 ориентациях для каждого из каналов. Для каждой ориентации и каждого канала строится гистограмма из 10 бинов. Гистограммы конкатенируются и получается текстурный дескриптор размерностью *8*3*10=240*. Текстурное сходство вычисляется аналогично пересечением гистограмм:

$$s_{texture}(r_i, r_j) = \sum_{k=1}^{n} \min(t_i^k, t_j^k)$$, где *t^k* - значение *k*-го бина в дескрипторе.

Слияние аналогично предыдущему.

2.3. Size Similarity (Сходство по размеру)

Вводится для того, чтобы избежать дисбаланса между регионами разных размеров. Без этого один большой регион поглощали бы все смежные маленькие.

$$s_{size}(r_i, r_j) = 1 - \frac{size(r_i) + size(r_j)}{size(image)}$$

При слиянии просто берется сумма размеров двух регионов.

2.4. Shape Compatibility (Совместимость по форме)

Используется чтобы объединять пересекающиеся регионы. Величина пропорциональна фракции, которую покрывают два региона в общем ограничивающем боксе:

$$s_{fill}(r_i, r_j) = 1 - \frac{size(BB_{ij}) - size(r_i) - size(r_j)}{size(im)}$$, где *BBij* - общий бокс вокруг *ri* и *rj.*

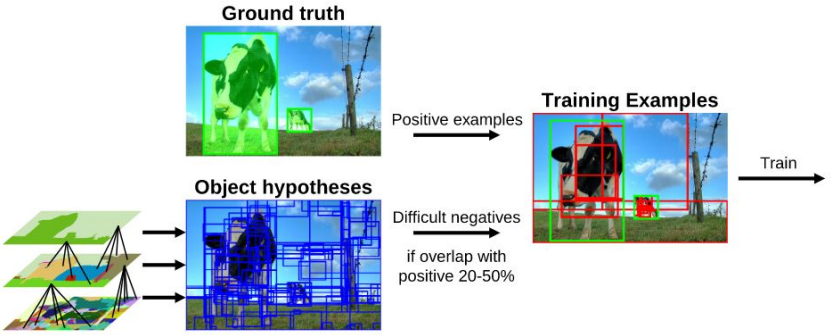
2.5. Final Similarity (Финальная оценка сходства)

Вычисляется как линейная взвешенная комбинация четырех оценок сходства:

$$s(r_i, r_j) = a_1 s_{color}(r_i, r_j) + a_2 s_{texture}(r_i, r_j) + a_3 s_{size}(r_i, r_j) + a_4 s_{fill}(r_i, r_j)$$, где *ak ∈ (0,1)*.

3. Classification

На финальном шаге происходит классификация *RP*s. Для этого используется бинарный *SVM*. Для обучения позитивными примерами считаются *gt*, негативными - боксы сгенерированные SS, которые имеют перекрытие от 20% до 50% с *gt*.



4. Implementation

Реализован в OpenCV. В этой имплементации *SS* выдает тысячи *RP*s упорядоченных в порядке убывания по *objectness*. На практике хватает 1000-2000 штук чтобы покрыть все корректные *RP*s. Есть мнение что выдается рандомно (вероятно либо есть *SVM*, либо нет).

Viola-Jones Object Detection Algorithm, 2001

tags: *Object Detection, OpenCV, Haar*

paper, link2, link3, video

Метод Виолы-Джонса для детекции объектов (*VJ*) был предложен Полом Виолой и Майклом Джонсоном в статье "*Rapid Object Detection using a Boosted Cascade of Simple Features*" в 2001. Основан на каскаде классификаторов с использованием (в оригинале) *признаков Хаара* (*Haar features*). Это ML-подход где т.н. *каскадная функция* обучается на датасете из позитивных (содержащих объект) и негативных (без объекта) изображений.

Признаки Хаара (Haar-like features)

Чтобы обучить классификатор необходимы фичи. Для этого используются *HFs*. *HFs* основаны на *вейвлетах Хаара* (*wavelet* - функция позволяющая анализировать различные частотные компоненты данных). Каждый *HF* состоит из смежных прямоугольных областей, а его значение вычисляется как вычитание суммы интенсивности пикселей под белой прямоугольной областью из суммы под черной областью (*рис.1*) . По сути каждый *HF* это ядро(*kernel*) свертки. Бывают 2/3/4-прямоугольными.

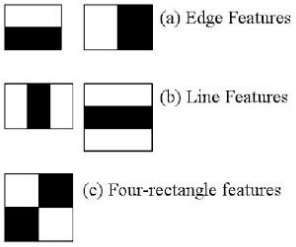


Рис. 1

VJ использует *sliding window (SW)* подход для поиска объектов на изображении. Поэтому вычислять *HFs* (если их сотни или тысячи) для каждой позиции *SW* и для каждого масштаба очень затратно. Поэтому для быстрого вычисления *HFs* был использован метод *integral image*.

Интегральное изображение (integral image)

Интегральное изображение (*II*) это алгоритм для быстрого вычисления *HFs,* что позволяет *VJ* работать в *real-time* даже на CPU. В основе алгоритма лежит *матрица интегрального представления*, в которой каждый элемент вычисляется как сумма элементов выше и левее данного элемента плюс сам элемент: $I(x,y)=\sum_{i=j=1}^{i\le W,\ j\le H} J(i,\ j),$, где *I(x,y)* - элемент *II*, *J(i,j)* - яркость пикселя исх. изобр., *W/H* - его ширина/высота. Для более эффективного расчета используется рекуррентная формула *(рис. 2)*: $I(x,y)=i(x,y)+I(x,y-1)+I(x-1,y)-I(x-1,y-1)$, где *i(,)* - исх. изображение.

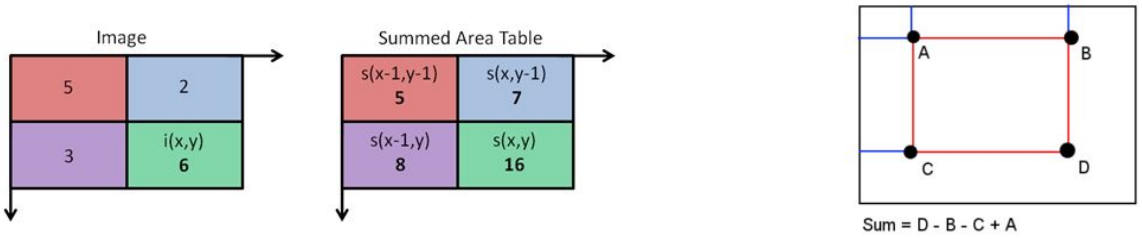


Рис 2. / Рис. 3

II позволяет быстро вычислять *сумму значений* прямоугольной области исходного изображения используя всего 4 значения: $\sum_{\substack{x_0<x\le x_1\\ y_0<y\le y_1}} i(x,y)=I(D)+I(A)-I(B)-I(C)$ (*рис. 3*). Сложность алгоритма *O(1)*, т.е. работает за константное время. Метод реализован в OpenCV как *cv2.integral()*.

Алгоритм Adaboost

И так, у нас есть *фичи* и способ их быстрого вычисления. Под *фичей* здесь подразумевается определенный *HF* примененной к определенной области окна. На *рис.4* представлены два *HFs*, которые образуют хорошие фичи (в рамках *face detection*): 1) регион глаз обычно темнее региона щек и носа; 2) глаза темнее чем переносица. Но если применить эти же *HFs* в другие области окна (например, на уровне щек), то они дадут уже не такие хорошие фичи. И в результате, даже если использовать окно размером 24x24, то применяя *HFs* во все возможные позиции окна, мы получим более сотни тысяч фич! Как же выбрать нужные из них? Это достигается при помощи *Adaboost*:

1) Присвоить одинаковый вес каждому изображению из датасета.
2) Вычислить фичи для каждого изображения из датасета.
3) Для каждой полученной фичи найти порог (*train*), который лучше всего классифицирует изображения на позитивные (есть объект) и негативные (объекта нет).
4) Рассчитать ошибку классификации для каждой фичи (на всех изображениях).
5) Выбрать те фичи, которые дают наименьшую ошибку классификации.
6) Увеличить вес тех изображений, на которых фичи ошибаются чаще всего.
7) Если заданное число фич найдено, или необходимая точность классификации достигнута, то завершить процесс, иначе перейти к п.2.

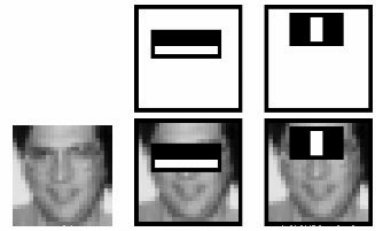


Рис. 4

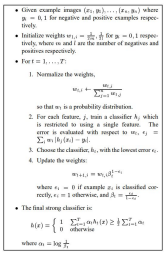


Рис. 5

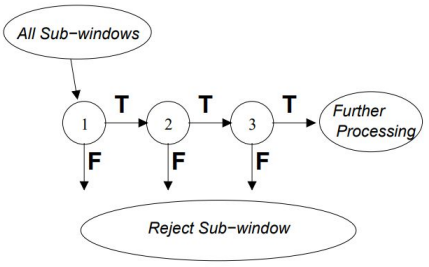


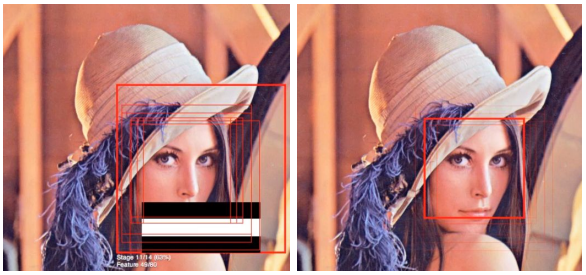
Рис. 6

Каждая отобранная фича образует слабый классификатор (*weak classifier*). Финальная оценка классификации вычисляется как взвешенная сумма слабых классификаторов.

Авторы *VJ* использовали в итоге 6000 фич и *скользящее окно (SW)* размером 24x24 (т.е. это минимальное разрешение объекта, объекты меньших размеров не могут быть классифицированы). Таким образом, для каждой позиции *SW* и для каждого масштаба необходимо вычислять 6000 фич, что все еще достаточно вычислительно затратно.

Каскад классификаторов (Cascade of Classifiers)

На помощь приходит концепции *каскада классификаторов* (*CC*). Зачем нам вычислять все 6000 фич для каждой позиции *SW*, если можно вычислить всего несколько фич и отбросить изображение, если оно классифицируется по этим фичам как негативное. Таким образом, все фичи группируются в несколько стадий (stages). Первые стадии содержат небольшое количество фич и позволяют быстро отбрасывать негативные регионы. Если регион классифицирован как позитивный, он передаются на следующую стадию. Если регион прошел все стадии - считается, что он содержит искомый объект (*рис. 6*). Авторы VJ разбивали фичи на 38 стадий, первые пять стадий содержат 1/10/25/25/50 фич. В среднем, для каждой позиция SW вычисляется 10 фич. (*см.* video).



Local Binary Pattern (LBP)

Придуман в 1996 как экстрактор фич из изображения. Представляет собой бинарный код, описывающий окрестность пикселя. Для каждого пикселя на изображении выбирается *n* его соседей, если сосед больше или равен центральному пикселю, то соответствующему значению бита кода присваивается 1, иначе 0. Разные типы соседства изображены на *рис.7*.

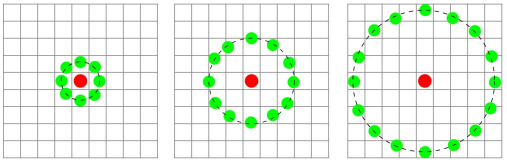


Рис. 7

В OpenCV *VJ* вместо *HFs* могут быть использованы классификаторы, обученные на *LBP* фичах.

Имплементация в OpenCV

Пример кода отсюда link2.