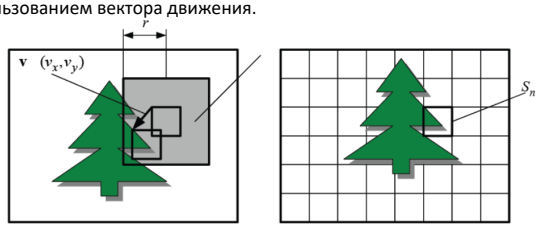
Блоковая оценка движения. Быстрые алгоритмы оценки движения. Кодирование с векторами полупиксельной и четвертьпиксельной точности

5 января 2023 г. 1:30

При кодировании видео учтем, что соседние кадры похожи друг на друга, а кадры - это по сути изображения. То есть нужно к алгоритму кодирования изображений добавить зависимость от времени и схожесть кадров.

Если поделить кадр на блоки и рассмотреть любой из них, можно найти похожий на предыдущем кадре (два блока, разность которых минимальна). Тогда связь между этими двумя похожими блоками можно выразить вектором. Обозначим за "начало координат" левый верхний угол блока, тогда вектор от "начала координат" блока на текущем кадре до "начала координат" блока на предыдущем кадре будет вектором смещения. Это - основа компенсации движения. В таком случае нужно будет как-то закодировать этот вектор без потерь и сам блок с потерями. Если таким образом обработать текущий и предыдущий кадры, т.е. найти разности похожих блоков, то получится кадр после компенсации движения.

Блоковая компенсация движения
Функцию, которую мы пытаемся минимизировать при поиске основного блока, можно представить так сумму разностей значений пикселей на текущем кадре и на предыдущем. Предыдущий выражается из текущего с использованием вектора движения.



Для каждого блока S_n в текущем кадре n выполняется поиск блока S_k минимизирующего

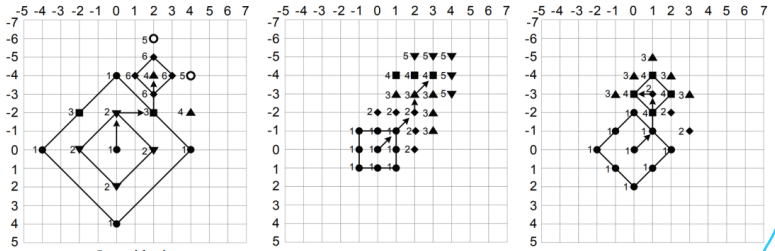
$$J(\mathbf{v}) = \sum_{(x,y)\in S_n} |s_n(x,y) - s_k(x+v_x, y+v_y)|,$$

где $\mathbf{v} = (v_x, v_y)$ – вектор движения, $|v_x| \le r$, $|v_y| \le r$, r – радиус поиска, $s_j(x,y)$ – значение пикселя с координатами (x,y) в кадре j.

Здесь мы кодируем вектор движения и разностный блок. Поскольку имеются две составляющие, может быть такое, что если мы сжимаем на высоком качестве, то вклад этих векторов получается небольшой, на них можно почти не обращать внимания. Если же сжать как можно сильнее (низкое качество), то нужно учитывать битовые затраты.
Если будем выискивать лучший блок полным перебором, алгоритм будет медленным.

Быстрые алгоритмы оценки движения

- Полный поиск требует $(2 \cdot r + 1)^2$ раз вычислить метрику $J(\mathbf{v})$.
- Для $r = 7$ Diamond search проверяет в среднем 15.5 метрик вместо 225 без существенной потери в эффективности кодирования[6].



1. Логарифмический поиск
   Начинаем с координат (0,0) и вычисляем метрику них и для еще четырех (помечены цифрой 1), допустим, при шаге 4
   Если выяснилось, что наилучшая метрика в центре, то уменьшаем фигуру в 2 раза (ромб, помеченный цифрой 1), получаем еще 4 пары координат (ромб, помеченный 2)
   Если точка с наилучшей метрикой не в центре, то перемещаем фигуру (этот ромб) так, чтобы его центр совпал с этой точкой. Нам будут известны значения метрик для центральной и одной из крайних точек, остальные довычисляем
   И т.д., пока центр не стал лучше всех точек вокруг
2. Градиентный поиск
   Каждый раз проверяем 9 точек: центр и 8 вокруг. Считаем метрики, перемещаем центр в точку с наименьшей метрикой, досчитываем неизвестные и т.д., пока центр не стал лучше всех точек вокруг
3. Diamond search
   Похож на предыдущие: как и в логарифмическом, рассматриваем ромбы, как и в градиентном, рассматриваем по 9 точек.

Вычисление векторов с половинной и четвертьпиксельной точностью
Возьмем во внимание, что кадры дискретны, т.е. пиксели друг от друга на каком-то расстоянии, а движение непрерывное. Получается, что движение объектов не кратно целым значениям пикселей. Оно может быть

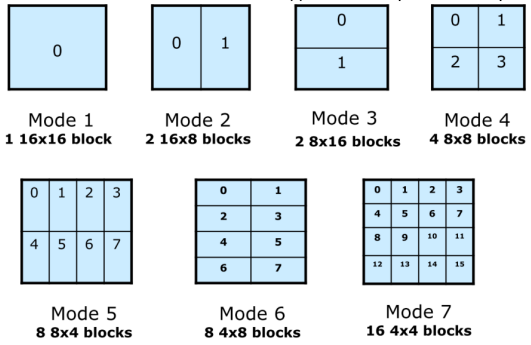
кратно половине пикселя, четверти пикселя. Поэтому, кроме поиска векторов с целочисленными значениями, выполняется поиск с половинной точностью и четвертьпиксельной точностью.
Мы проигрываем в скорости сжатия, поскольку приходится передавать еще какое-то количество чисел после запятой, но при высоком качестве это позволяет передавать картинку более точно.
Мы берем текущий кадр (на картинке пиксели текущего кадра выделены рамкой) и путем интерполяции из пикселей предыдущего кадра создаем еще несколько версий кадра, соответствующих смещению на определенное расстояние (может быть ½, ¼, ⅛ пикселя и т.д. в зависимости от необходимой точности). Затем проверяем произошло ли уменьшение разности между текущим и предыдущим блоком при изменении точности.

Рассмотрим кадр, жирным пометим известные пиксели референсного кадра. Если надо вычислить полупиксельную точность, то, например, при полупиксельной точности, рассматриваем значения по середине между известными пикселями. Т.е. кодер как бы вычисляет вот эти промежуточные значения и получает кадр, сдвинутый на полпикселя. Так можно найти лучшую метрику, чем если бы мы оперировали целыми пикселями.

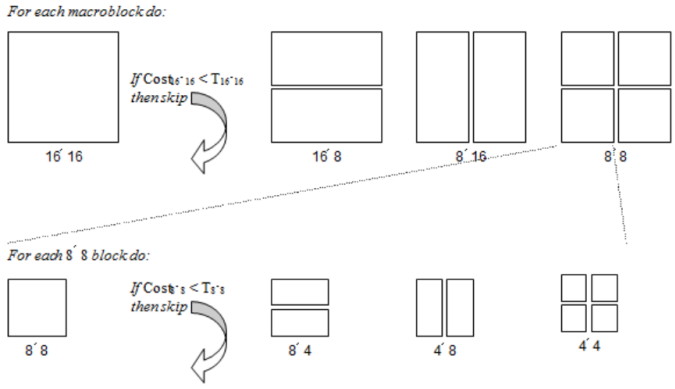
Кроме того, разбивать на одинаковые блоки при компенсации движения - не лучший вариант, т.к. движение очень сложное. Например, схожим образом происходит движение фона, а что-то другое может двигаться каким-нибудь "странным" образом. Поэтому выгодно было бы внести в кодер некую вариабельность, т.е. чтобы можно было использовать как кодирование одним блоком (например, 16x16) с одним вектором, так и делить его на подблоки с соответствующим количеством векторов. Т.е. так можно увеличить число векторов, применяемых к одному блоку.
Частный случай - 1 вектор на 1 пиксель, тогда вообще идеально всё найдем, но это не выгодно, поэтому либо тратим мало бит на вектор и хуже распознаем движение (хотя вариант нормально сработает в ситуации, когда в видео только фон, т.е. все пиксели двигаются однотипно), либо сжимаем хуже, но движение распознаем качественнее

Используется древовидное иерархическое разбиение блоков при компенсации движения.

Это позволяет использовать от 1 до 16 векторов на макроблок.



Алгоритм разбиения макроблоков
Берем блок, например, 16x16. Считаем для него идентичный блок на предыдущем кадре (та функция сверху, разбиваем его несколькими способами: на два подблока 16x8, на два подблока 8x16, на 4 подблока 8x8. Смотрим, как изменяется суммарная стоимость (та же функция сверху). Ищем, где она больше всего уменьшилась, выбираем этот вариант и дальше разбиваем уже его. Этот алгоритм сокращает количество перебираемых разбиений.



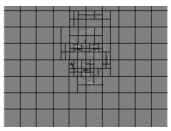
Также поскольку мы векторы кодируем без потерь, то чем на бОльшем шаге квантования работает кодер (т.е. на более сильном сжатии), тем больше доля бит на вектор у нас в битовом потоке. Когда мы сжимаем сильно, нам может оказаться более выгодно хуже искать вектор, лишь бы при сжатии он занимал меньше бит. Чтобы это учесть, можно ввести другую метрику. Она зависит от первой, рассмотренной ранее, а также от числа бит на векторы движения и параметра квантования (там множитель Лагранжа от него зависит)

- Большее число векторов на макроблок в общем случае обеспечивает лучшее предсказание. Однако, большее количество векторов требует большего количества бит. Поэтому, с точки зрения функции скорость-искажение более эффективно выполнять разбиение учитывая биты на векторы движения.
- Вместо $J(\mathbf{v})$ используется

$$\Omega(\mathbf{v} = J(\mathbf{v}) + \lambda(QP) \cdot R(\mathbf{v}),$$

где $R(\mathbf{v})$ – число бит на вектор(ы) движения, $\lambda(QP)$ – множитель Лагранжа, зависящий от параметра квантования QP. Чем больше QP, тем больше $\lambda(QP)$.

Т.е. не только пытаемся минимизировать то, как текущий блок предсказывает, а еще и чтобы была минимизация бит. Чем больше лямбда, тем больше мы учитываем бит, которые тратятся на вектор



Использование нескольких кадров при компенсации движения