(Lossless) Video Coding

Let's suppose we have a sequence of images taken with a video camera. A video gives us alot of data. For example, if you are recording 1024×1024 images with 8 bits per pixel at a rate of 30 images (frames) per second, then you are recording 30 MB per second. This is just for the black and white video.

The most straightforward way to encode video data would be to take each frame and compress it, for example, using differential coding. How can we do better than this? e.g. How can we combine data across frames?

To answer this question, let's first consider what a video gives you that a still image doesn't give you, namely, a video gives you *motion*. In a video, the position (and size) of different parts of the image can change over time. There are really two distinct types of motion. The first is due to the *motion of the camera.* If the camera's position is fixed but the orientation changes (e.g. the camera pans or tilts), then image moves since different parts of the scene project to different pixels in the image. Or, if the camera's orientation is fixed, but the position changes, then again image motion occurs, since the relative position of objects in the world and the position of the camera changes. Sometimes both the camera's position and orientation vary.

A second type of image motion arises from *motion of objects in the world.* For example, suppose you are taking a video of me during one of my lectures. You mount the camera on a tripod and just let the video record whatever happens. My position in the image relative to the blackboard would change over time as I pace back and forth. There will also be movement of the other students as they change position in their chairs, turn to talk to their friends, etc.

The most common method for coding image motion across frames is to partition the image into blocks, and then treat the blocks as (possibly) moving from one frame to the next. For each block we can try to represent this motion by a 2D vector which says by how many pixels the block moved in the horizontal and vertical directions from one frame to the next.

As a simple example, think of a camera panning across a landscape. Panning is done by rotating the camera about the vertical axis (y axis). All the pixels "move" with the same image speed (pixels per frame), which is determined by how fast the camera is panning. A second interesting example is that that the camera is not moving (it is mounted on a tripod), but some of objects are moving in the scene. In this case, many but not all of the pixels would have zero motion.

Let's next think of how we could use these observations to compress a video.

Motion Compensation

Suppose we encode frame 1 of the video, using one of the differential coding or transform coding methods discussed last class. Then, assuming we have encoded frame j, we give a method for encoding frame $j+1$.

To encode frame $j+1$, we use a technique called *motion compensation.* The idea is that most parts of frame $j+1$ were also present in frame j, except that the position may have been different. (That is, assuming that a given block may have *moved* from frame j to frame $j+1$.) Rather than encoding the block in frame $j+1$, the encoder estimates where this block was in frame j and also codes the intensity differences between corresponding blocks. This is specified more formally as follows.

Encoder (given a sequence of $n \times n$ image frames)

1. Encode frame $j = 1$ using a single image compression method (last lecture).
2. For each frame $j + 1$, partition image frame $j + 1$ into $\frac{n}{m} \times \frac{n}{m}$ disjoint blocks, each of size $m \times m$ (*e.g.* $m = 16$). For each block, let (k_1, k_2) be the upper-left pixel coordinate of that block (k_1 and k_2 are multiples of m).
   (a) Find [1] an *offset* vector, i.e. *motion vector* (v_1, v_2) that minimizes:
   $$\sum_{i_1=0}^{m-1}\sum_{i_2=0}^{m-1} \{ \ X_{j+1}(k_1 + i_1, k_2 + i_2) - X_j(k_1 + i_1 - v_1, k_2 + i_2 - v_2) \ \}^2$$
   (b) Encode motion vector (v_1, v_2)
   (c) Encode the $m \times m$ *difference image* for that block
   $$X_{j+1}(k_1 + i_1, k_2 + i_2) - X_j(k_1 + i_1 - v_1, k_2 + i_2 - v_2) \ .$$

I have not specified what code to use for the offset vector (v_1, v_2) and difference image. There are many possibilities.

The more important issue is why this achieves compression. As long as the change in intensity are mostly due to a translation of pixel intensities, the intensity difference values will all tend to be near zero. As such, these random variables have low entropy, and we don't need many bits to encode each value. The velocity vectors (v_x, v_y) also tend to be small numbers, since typically there is not much motion from frame to frame.

When does the model fail? One way it fails is if frame j and $j + 1$ are independent. In this case, there is no reason why a good match can be found for each block in the $j + 1$ frame, when searching frame j. So, the intensity differences could be large (implying that lots of bits are needed to encode them), *and* we still would encode a (v_1, v_2) vector.

This worst case is important because it does arise in real videos. When the video "cuts" from one scene to another, we have precisely this independence between neighboring frames. We will see later in this lecture how to avoid this problem.

A few final points. First, the technique of motion compensation is reminiscent of Lempel-Ziv methods in the sense that we are looking for a pattern that we have seen before and encoding an offset to that pattern. The distinction here is that we don't demand an exact match to the pattern (as in LZ), but rather we demand a best match. By "best" we have some norm in mind, e.g. sum of squares of intensity differences.

Second, note that computing the best offset (v_1, v_2) requires computational effort. If you allow yourself to search over a wide range of offsets, then you may get a better match. (Of course, your code would need to allow you to represent a larger range of offsets as well, and this itself tends to increase codeword lengths slightly.) Is it worthwhile to search for the best match ? It depends. For some video applications, the sequence is meant to be encoded once and then shown thousands of times. For these applications, one may be willing to use a computationally complex encoding algorithm if it means using fewer bits. For example, one may be willing to apply an expensive optimization algorithm to find the best (v_1, v_2) for each block. On the other hand, if real time encoding is required, it may be necessary to keep the search for (v_1, v_2) as simple as possible.

[1] There are many *least squares* methods available for solving this minimization problem.

Decoder

1. Decode frame 1.
2. For each frame $j+1$, and for each block (with upper left corner (k_1, k_2)):
   (a) Decode the motion vector (v_1, v_2) for that block.
   (b) Using the decoded intensities in frame j and the motion vector for the particular block, decode a prediction of the intensities for that block. For fixed k_1, k_2 (block) and fixed (v_1, v_2), the prediction is
   $$X_j(k_1 + i_1 - v_1, k_2 + i_2 - v_2)$$
   for (i_1, i_2) in the block.
   (c) Decode the difference image for that block,
   $$X_{j+1}(k_1 + i_1, k_2 + i_2) - X_j(k_1 + i_1 - v_1, k_2 + i_2 - v_2) \ .$$
   (d) Add these to the predicted intensities for that block. This gives the intensities of the block as a function of (i_1, i_2), with (k_1, k_2) fixed, i.e.

$$\begin{aligned} & X_j(k_1 + i_1 - v_1, k_2 + i_2 - v_2) \ + (X_{j+1}(k_1 + i_1, k_2 + i_2) - X_j(k_1 + i_1 - v_1, k_2 + i_2 - v_2) \ ) \\ = \ & X_{j+1}(k_1 + i_1, k_2 + i_2) \end{aligned}$$

I,P,B frames

Another feature that we wish our encoded video to have is the option of fast-forward and fast-rewind, i.e. to view a sparsely sampled subset of the frames. You are all familiar with this ability from your use of DVD players.

Currently we are encoding frame $j+1$ based on frame j. Suppose we were to encode every 10th frame on its own, rather than based on the previous frame. We could then keep a header file saying where these individually encoded frames begin (i.e. their address in the encoded file). This would allow us to jump quickly through these frames only. This fairly obvious idea of using two types of frames (individual frames vs. frames based on previous frames) can be extended slightly further.

Suppose we were to use three types of image frames. The encoded (i.e. compressed) file begins with a header that contains the addresses of the beginning of each coded frame and the type of each frame. The three types of frames are as follows:

- `I` frames: (`I` stands for *intraframe*, that is, "within") These are frames that are encoded independently of other frames. It can use one of the methods from last lecture.

  If you want to fast forward or fast rewind through a video, while viewing only a fraction of the frames, then you can do so using the I frames only (or some fraction of the I frames). The decoder can decode these frames without using any neighboring frames.

- `P` frames: (`P` stands for *previous*.) These are frames that are encoded using motion compensation. Each `P` frame is encoded based on the closest *previous* `I` or `P` frame (whichever it happens to be).

  Better compression is typically (but not always) achieved with `P` frames than with `I` frames.

- `B` frames: (`B` stands for *between*). Each block of a `B` frame is coded using motion compensation, but now using either the next `P` or `I` frame, or the previous `P` or `I` frame[2]. `B` frames are not used to predict other `B` frames.

To predict a block in a `B` frame, we use one of the following two formulas. Let (v_1', v_2') be the computed motion compensation vector if the prediction is from a future frame, and let (v_1, v_2) be the motion compensation vector if the prediction is from a previous frame.
The pixel intensity at $(k_1 + i_1, k_2 + i_2)$ in a block is predicted by one of the two following formulas:

$$X_{predicted}(k_1 + i_1, k_2 + i_2) := X_{next}(k_1 + i_1 + v_1', k_2 + i_2 + v_2')$$

$$X_{predicted}(k_1 + i_1, k_2 + i_2) := X_{prev}(k_1 + i_1 - v_1, k_2 + i_2 - v_2)$$

Note that for each block in each frame, one bit is needed to specify which of the two formulas is being used. For each block, the decoder needs to be told where it should get its predictions from.

Backwards prediction allows objects to suddenly appear in certain frames. This is the case, for example, in the first frame of a new scene. If this first frame is a `B` frame which is followed by an `I` frame, then we can use the `I` frame to backwards-predict the intensities in this `B` frame. (If this first frame of the new scene is followed by a `P` frame, however, then that `P` frame would be predicted by its previous `I` frame, which would come from another scene. In this case, the `P` frame would not be well predicted.)

Display order vs. bit order

Notice that frames are not encoded in the order in which they occur in the original video (which is the order in which they are displayed). Thus need to distinguish two orderings, the *display ordering* (original) and the *bit ordering* (used in the encoding).

An example of the display order might be the following sequence:

I	B	B	P	B	B	P	B	B	P	B	B	I
1	2	3	4	5	6	7	8	9	10	11	12	13

The bit ordering for the same sequence would be:

I	P	B	B	P	B	B	P	B	B	I	B	B
1	4	2	3	7	5	6	10	8	9	13	11	12

Note that we would expect to use fewer bits by allowing for B frames (than if we were to use only I and P frames, say) because for each block we have a larger set of candidate blocks from which the predict the intensities. However, we must keep in mind that there are additional costs to using B frames. First, there needs to be a header file that specifies the labelling of the frames (which frames are I vs. B vs. P). This header has a small cost. Second, for each block of a B frame, we need to specify whether the prediction is coming from previous or from next. Again there is a small cost. The claim is that for a typical video, these small costs are more than compensated for by the compression we achieve.

[2]For example, if you have four consecutive frames, `IBPB`, then the method cannot use the `I` frame to predict the second of the `B` frames.