CS5114 - Project 1 - Decoding Convolutional Codes with the Viterbi Algorithm

Brian Hulette

05/11/2014

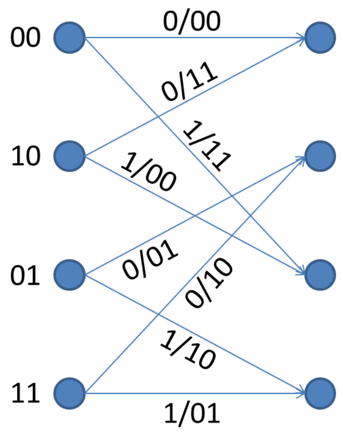
Contents

1 How Convolutional Codes Work

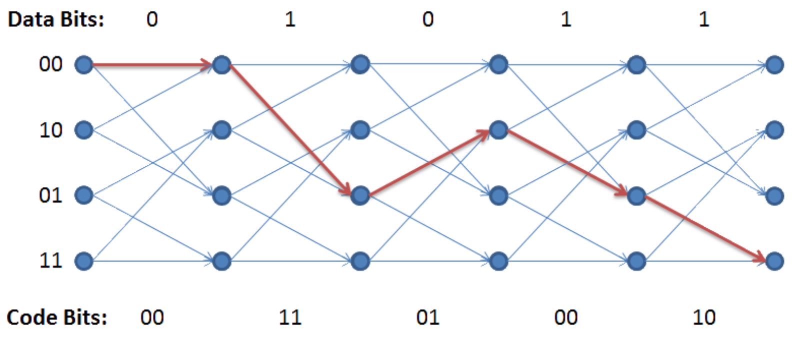
A convolutional code is a type of error correcting code. This is away of introducing redundant data into a data stream in order to allow a receiver to correct any errors that occur in transmission. A very simple code might simply send each bit in a stream multiple times, so instead of sending 10110, a transmitter would send 111 000 111 111 000. Its easy to see that this would make it much easier for a receiver to successfully determine what the intended message was, even if some of the bits were received in error.

A convolutional code works similarly, but it introduces redundancy in a much more complex way to improve preformance. Unfortunately, the additional complexity makes decoding the intended message much more difficult.

The concept of a convolutional code is much easier to understand by looking at a trellis diagram, as shown in Figure 1. The image can be a little daunting at first, but it's not too bad once you break it down. The trellis is simply a way of visualizing a finite state machine over time. The detail view in Figure 1a shows the four possible states on the left, and then all of the possible transitions to the states in the next iteration, shown on the right. The labels on each transition are of the form "input/output". So, for example, if the encoder is currently in state 00 and if the input bit for this step is 0, then it will remain in state 00 in the next iteration and output the code bits 00. The encoder then repeat this process for each of the data bits that comes in.



(a) Trellis Detail - each transiton is marked with I/OO where I is the input bit that leads to that transition, and OO are the code bits that are output when that transition is taken.



(b) Example traversal of a Trellis for encoding. On each iteration the encoder follows the transition which corresponds to the data bit for that iteration, and outputs the two code bits associated with that transition.

Figure 1: Viterbi Trellis

Figure 1b shows what this process looks like when it is repeated for several bits. the "Data Bits" along the top are the input bits - this is the data that we want to encode. The "Code Bits" along the bottom are the output bits - this is the encoded data, which we will transmit. The red transitions show the path that was taken through the trellis to produce these outputs.

So that describes how the encoder works, but that is the easy part. The receiver has a harder job, it needs to take the coded bits that we received (possibly with some bit errors from an imperfect transmission) and decode them to determine the original data bits - this is the problem that the Viterbi algorithm excels at.

2 Decoding Convolutional Codes

Decoding a convolutional code is really just an optimization problem. Given a set of code bits (with some bit errors) the decoder needs to find the sequence of data bits which is the *most* likely to have produced those code bits.

There are a few different approaches to this problem. You could take a brute force approach, which is very easy to understand but also very inefficient. Another option is a slightly more efficient recursive approach. But both of these options are blown away by the Viterbi Algorithm, a Dynamic Programming approach which takes advantage of the problem's optimal substructure. I will discuss each of these approaches in detail in the following sections.

First I want to discuss a little terminology. The length of the data and code bit sequences is given by N_d and N_c, respectively. N_d and N_c are related by the code rate, r, with the relation $N_d = rN_c$. I will use d_{ij} to refer to the vector of data bits, and c_{ij} for the vector of code bits. The subscript is used to refer to a subset of the bit sequence, so $c_{0:N_d}$ refers to the entire code bit sequence, $c_{0:2}$ refers to the first two bits, etc...

2.1 Decoding with Brute Force

A brute force approach is to simply iterate through all 2^{N_d} of the possible sequences of data bits, encode each of them, and compare those code bits to the code bits that were received. The sequence that produces code bits that are the most similar to the received code bits must be the input data.

How do we determine how "similar" a bit sequence is to the actual received code bits? We simply count the number of bits which are different between the two sequences - this is called the **hamming distance**, which I will denote with $d_h(s_1, s_2)$. For example, the hamming distance between the sequences 11011 and 11101 is $d_h(11011, 11101) = 2$, because the third and fourth bits are different.

It's easy to see that this brute force approach is going to be very inefficient, since we are iterating over 2^{N_d} possible sequences, so let's discuss some other options.

2.2 Decoding with a Recursive Approach

A recursive approach actually isn't much more efficient than the brute force approach, but it helps us to see the optimal substructure that we will use in the Viterbi Algorithm.

For the recursive approach, the important thing to realize is that there are exactly two possible transitions leading from each state s, based on whether the data bit for that step is a 1 or a 0, as you can see in Figure 1a. Each one of these transitions has associated with it a certain set of output bits, which I will call o_0 and o_1, and a certain next state, which I will call s'_0 and s'_1.

So if our recursive function starts in state $s = 00$[1], then for each of the possible input bits, $i = 0, 1$, we need to measure the hamming distance between its output bits and the first two received bits, $d_h(o_i, c_{0:2})$. Then we call the recursive function again for each i, but this time start it in state s'_i, and use the code bits $c_{2:N_d}$. We add the hamming distance for this state to that returned

[1]We know to start the decoder in the 00 state, because convolutional encoders always start in 00

by the recursive function call, and whichever input bit i yielded the lowest hamming distance is our decision for this step. The recursive calls stop once we they reach the right side of the trellis and there are no more code bits to process.

It's easy to see that as we progress from left to right in the trellis, these recursive calls will begin to overlap each other, and we will be computing the same thing multiple times. Clearly a dynamic programming approach which remembers these calculations is needed.

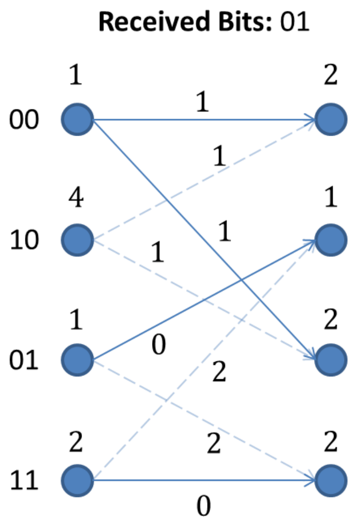
2.3 Decoding with Dynamic Programming - the Viterbi Algorithm

The recursive solution can clearly benefit from a Dynamic Programming approach because of the large amount of overlapping subproblems. To find a DP approach we can follow the Four Steps for developing a DP algorithm.

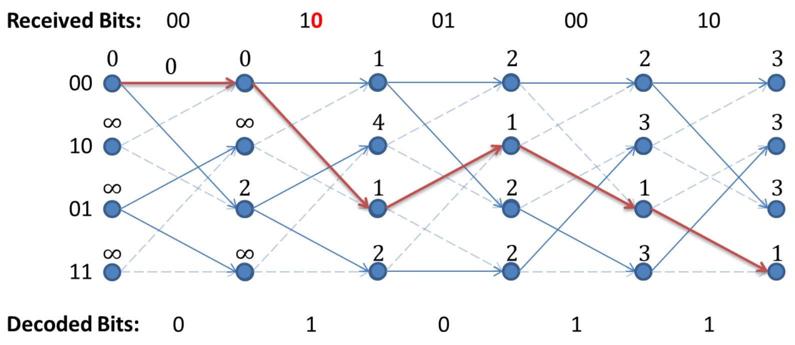
Step 1 is to characterize the sturcture of an optimal solution - in this case an optimal solution is a sequence of data bits, d, that minimizes the hamming distance between the given code bits, c, and the code bits that it generates, c'.

Step 2 is to recursively define the value of an optimal solution - I have already done this in Section 2.2.

Step 3, Computing the Optimal Costs, is where things get interesting. To do this, we need to think about the problem a little bit differently. Consider the fact that on every iteration there are two possible transitions leading into each of the next states. For example, you can see in Figure 1a that to get to the state 00 you could come from state 00 with an input bit of 0, or from state 01 with an input bit of 0. The idea of the Viterbi algorithm is to figure out which of these two transitions is more likely for each state.



(a) Traversal of the Trellis step for iteration 2 to 3. Numbers on the transitions indicate the hamming distance of that path.



(b) Traversal of the entire Trellis. The numbers over each state are the path metrics for that state. The "Loser" branches for each state are faded out. The red line shows the results of the traceback.

Figure 2: The Viterbi algorithm used to decode the example from Figure 1

Ok, so how do we do that? On each iteration, we will assign a metric to each of the states, called the path metric. This metric is the distance between the solution so far and the received code bits, c, so lower is better. I will call this metric $m_{i,j}$, where i is the state, and j is the iteration. For the first iteration we assign 0 to state 00 and infinity to all of the other states (ie $m_{00,0} = 0$ and $m_{01,0} = m_{01,0} = m_{01,0} = \infty$), because we know the encoder started in state 00. On each iteration we will use the $m_{i,j}$ values to compute the $m_{i,j+1}$ values. The new metric is simply one of the old metrics, $m_{i,j}$, plus the hamming distance between that transition's outputs and the received code bits for this iteration. For each of the states of iteration $j+1$ we will pick the transition which yields the lower metric. We can then repeat this process all the way through the trellis.

Figure 2b shows the metrics for every iteration of my example problem, and Figure 2a shows the details of this process for the transitions from iteration 2 to iteration 3.

Step 4 is to Construct an Optimal Solution based on the result of Step 3. To do this we simply have to do a "traceback" of the entire trellis. Figure 2b shows the traceback for this example problem. The traceback starts in the last iteration, on the state with the lowest metric, in this case that is state 11 with a metric of 1. Then we traverse backward through the trellis, choosing the "winner" transitions which we found in Step 3. On each step back we note the input bit for that transition and add it to the optimal solution.

2.4 Top-Down vs. Bottom-Up

The algorithm described in Section 2.3 would be considered a Top-Down DP algorithm. Could we come up with a Bottum-Up algorithm that would be more efficient?

The answer is no. Because of the structure of this problem, there isn't really a distinction between the performance of a Top-Down and Bottom-Up approach. A Bottom-Up approach would look basically the same as the Top-Down aporach except we would traverse from right to left.

In fact, there are actually some disadvantages to a Bottom-Up approach. First of all, we couldn't use the fact that the encoder always starts in state 00 to set our initial conditions. And second, since this algorithm is usually used in communications, we need to process data as it's coming in, in real time. In this case it makes sense to process the data Top-Down, so we can do some of the processing as the bits are coming in.

3 Performance of Brute Force vs. Viterbi

In this section I will determine the runtime of each algorithm as a function of the number of data bits, N_d. You can see pseudocode for each of these algorithms in Appendix A.

3.1 Brute Force

The main loop in BRUTE_FORCE_DECODE on is iterating over every possible N_d-bit sequence, so there are 2^{N_d} iterations. On each of these iterations it must encode the N_d-bit sequence using ENCODE, (not shown here, runs in $\Theta(N_d)$ time), and compute the hamming distance between the encoded sequence and the received bits using HAMMING_DISTANCE, which also runs in $\Theta(N_d)$ time.

Thus the runtime of the brute force approach is $\Theta(N_d 2^{N_d})$. Of course this runtime is completely untenable as a solution to this problem.

3.2 Recursive

Every iteration of the recursive helper must call itself twice, with a data sequence shortened by two bits, which yields $2T(N_c - 2)$ and it must perform an operation which is constant with respect to N_c, or $\Theta(1)$. So we know $T(n) = 2T(n-2) + \Theta(1)$.

Thus the algorithm's recursion tree will have $N_c/2$ levels, and each level will have 2^L calls on it. So the total number of calls is:

$$
\begin{aligned}
T(n) &= \sum_{L=0}^{N_c/2} 2^L = \frac{1 - 2^{N_c/2+1}}{1-2} = 2\left(2^{N_c/2}\right) - 1 = \Theta\left(2^{N_c/2}\right) \\
&= \Theta\left(2^{N_d}\right)
\end{aligned}
$$

Where in the last step I use the fact that $N_d = rN_c = N_c/2$. So the recursive approach is also exponential, but at least we've gotten rid of the linear factor.

3.3 Viterbi

The Viterbi algorithm is clearly much more efficient than both of these algorithms. It simply iterates over the entire trellis, and performs a $\Theta(1)$ operation on each step. The number of iterations is simply the number of data bits, N_d. Thus the viterbi runtime is $\Theta(N_d)$.

4 Other Parameters Affecting Performance

So far in this paper I have restricted myself to talking about just one type of convolutional code, called a rate 1/2, constraint length 3 code. But in reality there are infinitely many ways to configure one of these codes. Each code is defined by a set of "generating polynomials" or "generators" which basically tell the encoder what to do. In my pseudocode in Appendix A you will see that the generators are passed in as an argument, since I wrote the algorithms to work with any of them.

There are a lot of different properties of the code we can determine from the generating polynomials. For example, the code rate r is determined by the polynomials. The number of generating polynomials determines how many bits are output for each input bit - so a set of two polynomials yields a code with rate $r = 1/2$, as in my example.

Another property determined by the polynomials is the constraint length, K, which is just the length of each polynomial. The number of states in the code is given by 2^{K-1}. In my example, the constraint length is $K = 3$, which is why there are $2^{3-1} = 4$ states.

It's easy to see that the Viterbi algorithm actually runs in linear time with respect to the number of states (because on each step it needs to iterate through each state to find its path metric), which means that viterbi actually runs in exponential exponential time with respect to the constraint length: $\Theta(2^K)$. Clearly, the Viterbi algorithm's performance is very sensitive to this parameter. For this reason, code designers will generally use a constraint length less than 10.

A Algorithm Pseudocode

A.1 Brute Force

```
BRUTE_FORCE_DECODE( code_bits, generators )
    r = 1/size(generators, 2)   // Determine the code rate
    N = length(code_bits)*r
    d = infinity
    input_bits = array of N zeros
    for every possibe N-bit sequence, seq
        code_bits = ENCODE(seq, generators)
        this_dist = HAMMING_DISTANCE(code_bits, code_bits)
        if this_dist < d
            d = this_dist
            input_bits = seq
    return input_bits

HAMMING_DISTANCE(a, b)
    return sum(xor(a, b))
```

A.2 Recursive

```
RECURSIVE_DECODE( code_bits, generators )
    // outputs[this_state][input_bit] is the bits that are output when that
    // transition is made
    global outputs = GET_OUTPUTS(generators)
    // next_state_table[this_state][input_bit] is the state that "this_state"
    // will transition to if "input_bit" comes in
    global next_state_table = GET_NEXT_STATES(generators)
    input_bits, metric = RECURSIVE_DECODE_HELPER(input_bits, 0)
    return input_bits

RECURSIVE_DECODE_HELPER( code_bits, curr_state )
    if code_bits is empty
        return [], 0
    end

    best_metric = infinity
    for in_bit = 0,1:
        data_bits, metric = ...
        RECURSIVE_DECODE_HELPER(code_bits[bits_per_step:end], ...
                                next_state_table[current_state][in_bit])
        metric += HAMMING_DISTANCE(code_bits[0:bits_per_step], ...
                                   outputs[current_state][in_bit])
        if metric < best_metric:
```

```
        best_data_bits = in_bit + data_bits

return best_data_bits, best_metric
```

A.3 Viterbi

```
VITERBI_DECODE( code_bits, generators )
K = size(generators, 1);
// n is the number of code bits produced on each iteration, =1/r
n = size(generators, 2);
num_states = 2^(K-1);

// Pre-compute tables
// input_for_next_state[this_state][next_state] is the input bit that will
// make the transition from this to next.  The value is -1 if that
// transition is not allowed
input_for_next_state  = GET_INPUT_FOR_NEXT_STATE(K)
// outputs[this_state][input_bit] is the bits that are output when that
// transition is made
outputs = GET_OUTPUTS(generators)

path_metrics is an array of length num_states, element 0 is 0 others are infinity
next_path_metrics is an array of length num_states, filled with 0s

num_iterations = length(input_bits)/n;
branch_winners = 2D array of length num_states by num_iterations;
for j = 0 to num_iterations - 1
   code_bits = code_bits(j*n:j);

    for next_state = 0 to num_states - 1
        min_path = -1;
        min_path_dist = inf;
        for state = 1:num_states
            data_bit = input_for_next_state[this_state][next_state);
            if data_bit == -1
                continue;

            output_bits = outputs[this_state][input_bit]
            path_dist = HAMMING_DISTANCE(code_bits(j*n-1), output_bits) + path_metrics(thi
            if path_dist < min_path_dist
                min_path = this_state;
                min_path_dist = path_dist;
        branch_winners(next_state, j) = min_path;
        next_path_metrics(next_state) = min_path_dist;
    path_metrics = next_path_metrics;
```

```

// Do the traceback
PERFORM_TRACEBACK(branch_winners, path_metrics);
```