Lecture 9. Extending the Applicability of HMM Models

1. Higher order HMMS

We saw last time how an HMM can be used to model a bigram probabilistic model. HMMs can be used to model more complex probabilistic models as well. The trick is to define a more complex notion of state that mirror the probabilistic model. Typically this involves expanding the number of nodes, sometimes significantly. For example, consider the trigram approximation for tagging, in which we use the approximation

$$P(C_{1,T}) \cong \prod_i P(c_i \mid c_{i-2}\, c_{i-1})$$

We cannot capture these probabilities within the HMM structure shown in last chapter where each state represents a single c_i. Rather, we need to construct a model where the states represent the context for the conditional probability, i.e., we need pairs of values in the sequence $<c_{i-1}, c_i>$. The interpretation of such a state is that we are generating an output of type c_i having just generated an output of type c_{i-1}. The transition probability between the state $<c_{i-2}, c_{i-1}>$ and $<c_{i-1}, c_i>$ should be the probability of generating an output of type c_i given that we just generated an output og type c_{i-1} and before that an output of type c_{i-2}. This is the trigram probability $P(c_i \mid c_{i-2}\, c_{i-1})$ used in the above formula. The general idea is that we build states that capture the conditional context of the probabilistic model, and the transition produces the new context given the current output and what is needed from the previous state.

With such a model, we have some options for richer output probabilities as well, In general, we have the approximation

$$P(W_{1,T}) \cong \prod_i P(w_i \mid c_{i-1}\, c_i)$$

When people talk of trigram tagging models, however, they usually retain the simplist output probability distribution, approximating each term $P(w_i \mid c_{i-1}\, c_i)$ with $P(w_i \mid c_i)$, just as in the bigram model.

Part of the HMM structure for a trigram model of our 4 tag example is show in Figure 1. This shows all most of the states that involve N in some way. The full network would include 21 states (16 to cover every possible pair from {N, V , P, ART} and 5 involving the special start states. Note that it is not fully connected. For instance, from <ART,N>, we only have non-zero transitions to states that begin with N (<N,N>, <N,P>, etc.). In

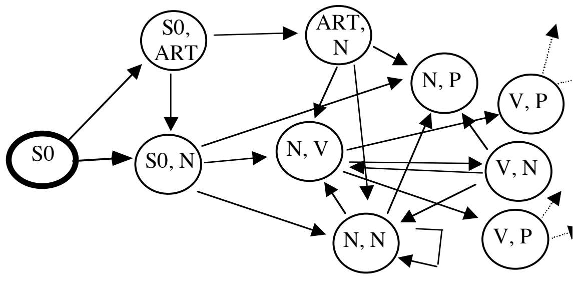

Figure 1: Part of the HMM for a trigram model

genergal, we oinly have N transitions from any one node rather than the N^2 we might expect.

2. Algorithms for Large HMM Models

Even though people correctly say that for any fixed size network, the Viterbi algorithm is linear in the size of the input, it more accurate to say that the complexity of Viterbi algorithm is proportional to N^2*T, where N is the size of the network and T is the length of the input. When N gets large, the full algorithm rapidly becomes impractical. For instance, consider a relatively simple trigram part-of-speech model with 40 different tags. Each state represents a pair of tags, so we have 40^2 (=1600 states). Thus, to fully explore every possible next state in the Viterbi algorithm we would need to look at 40^3 (64,000) possibilities for each term in the input.

Beam Search: Handling Large Networks

The simplest idea is to rank all the alternatives produced so far by their probabilities, and always just expand the most likely sequences. This is called a **beam search**. We continue to do this at each time point until we have an interpretation that covers the entire output. This is a simple modification to the basic Viterbi algorithm. Remember that the original algorithm was driven by a loop that iterated through each position in the input, namely

$$\delta_t(i) = \mathrm{MAX}_{j=1,N}(\delta_{t-1}(j) * P(L_i \mid L_j)) * P(w_t \mid L_i)$$

For each input position, we looped through all N states (the i's), and did a Max calculation over all N predecessor states (the j's), i.e., N^2 operations. It is this double loop that kills the algorithm. A beam search prunes out the unpromising states at every step, leaving only the best to be used in future calculations. We can decide how many to keep using several different methods. We could, for instance, decide on a fixed number to keep at each step, say B nodes. Alternately, we could decide on some factor K and delete all states that have a probability less that m*K, where m is the maximum probability for a state at the current time. For instance, if K=.5, we would only keep hypotheses that have a score greater than half the maximum score at that time.
Let's consider this approach in more detail. We decide to keep all nodes within k percent of the maximum at each step. We now create an array called *beam,* each element of which will store the set indices of states above threshold at time t. The innermost loop is then replaced with an iteration over $beam_t$ and then the step continues as before. Thus we have replaced the line above with

$$\delta_t(i) = (\mathrm{MAX}_{j \text{ in } beam(t-1)}\ \delta_{t-1}(j) * P(L_i \mid L_j)) * P(w_t \mid L_i)$$

We then compute *beam(t)* by finding the maximum value M_t at time t, and deleting those states below the threshold of M_t*k. With this change, we have reduced the number of operations in the inner loop to be B*N, where B is the average number of states that are above threshold at each time. We can set k to make this a manageable number.

More graphically, we can think of the Viterbi algorithm building a trellis of states. Without pruning, the Trellis would have N states at every time point, and store the backpointer as a link between the time points. Figure 1 shows the trellis after 4 time segments with the full Viterbi search, whereas Figure 2 shows the trellis of the same search with a beam search (assuming a small number of states are above threshold).

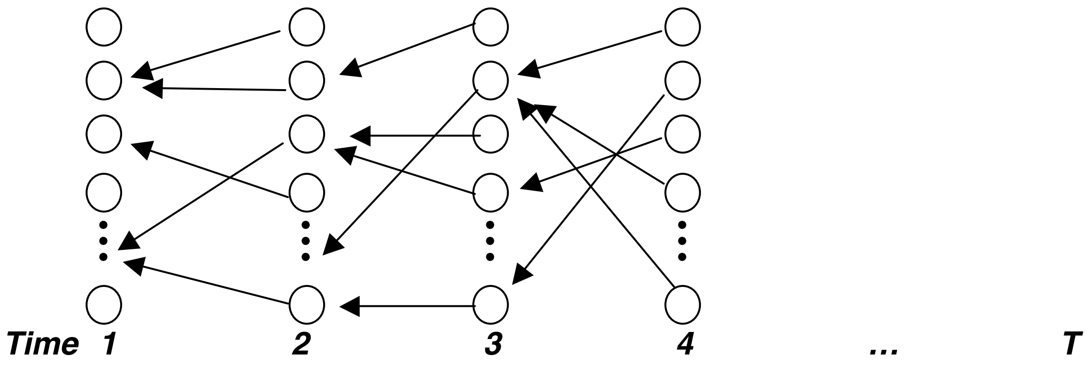

Figure 1: The Trellis after time 4

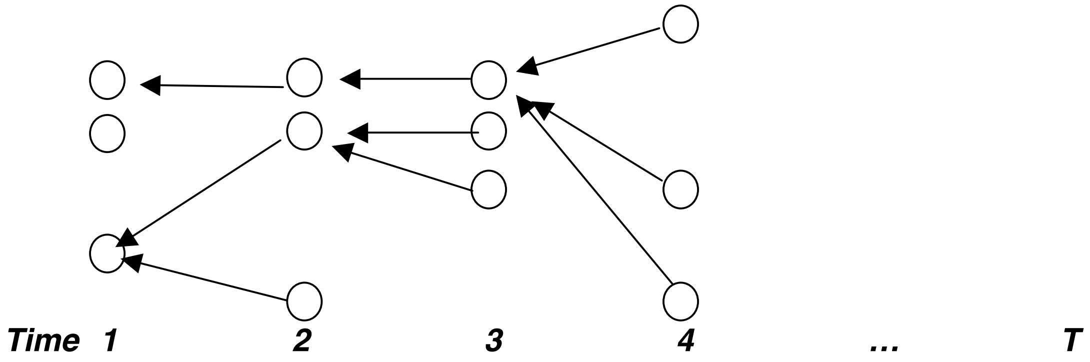

Figure 2: The Same Trellis with a Beam search

Consider the small part of speech tagging example from last time operating on the sentence *flies like a flower*. The result of the Viterbi algorithm on this sentence is shown in Figure 3. As expected, there are 4 nodes expanded at each stage, with each considering 4 possible paths, and 4 stages, meaning we consider 4^3 possibilities. Figure 4 shows the nodes expanded if we did a beam-search Viterbi just keeping entries that are within a factor of .01 of the best. Here three of the steps retain just one entries, and one has 3, so the algorithm considers 5*4 = 20 possibilities during the search. Figure 5 shows a beam-search Viterbi with a fixed beam width of 2 (i.e., we keep the two best at each point). Here we have 2 entries at each stage, so the algorithm consider 8*4 =32 possibilities.

	ART	N	V	P
Flies	7.1e-5	**7.22e-3**	7.59e-6	1.00e-8
Like	4.93e-10	1.13e-5	**3.10e-4**	2.16e-4
A	**7.25e-5**	9.76e-8	4.84e-10	4.95e-10
Flower	7.25e-13	**4.56e-6**	2.10e-9	3.27e-9

Figure 3: The full Viterbi search for *flies like a flower*

	ART	N	V	P
Flies		**7.22e-3**		
Like		1.13e-5	**3.10e-4**	2.16e-4
A	**7.25e-5**			
Flower		**4.56e-6**		

Figure 4: A Beam Viterbi search using a factor K=.01 for *flies like a flower*

	ART	N	V	P
Flies	7.1e-5	**7.22e-3**		
Like			**3.10e-4**	2.16e-4
A	**7.25e-5**	9.76e-8		
Flower		**4.56e-6**		3.27e-9

Figure 5: A Fixed Beam Viterbi search with N=2 for *flies like a flower*

3. Best First Search Algorithms (Basic Stack Decoding)

So far we have considered algorithms that move step by step through the observation sequence in a systematic manner. Another alternative is to allow the search to progress by always expanding the best path so far. Each time we expand a path, we look at all its extensions and add them to a structure called the **agenda**, which is an ordered queue.

We capture each path hypothesis on the queue by a tuple that indicate the path so far and the probability. For example (N V ART, 7.25e-5) says the path N V ART has probability 7.25e-5. Using the numbers in the full search as a resource, we can trace how a best first search would go on the sentence *Flies like a flower*. We start with an agenda in Figure 6(a), which has all the starting positions in descending order of their probability. We remove (N, 7.2e-3) off the agenda and generate all extensions and insert them into the agenda as shown in 6 (b). Note that two of the extensions are now the highest ranked hypotheses, one (N N) is in the middle and the fourth (N ART) is at the bottom. We now extend the (N V) hypothesis to get the agenda shown in 6 (c), but don't show the (N V V)

(N, 7.2e-3)
(ART, 7.1e-5)
(V, 7.6e-6)
(P, 1.0e-8)

(a)

(N V, 3.1e-4)
(N P, 2.2e-4)
(ART, 7.1e-5)
(N N, 1.1e-5)
(V, 7.59e-6)
(P, 1,00e-8)
(N ART, 7.2e-11

(b)

(N P, 2.2e-4)
(N V ART, 7.3e-5)
(ART, 7.1e-5)
(N N, 1.1e-5)
(V, 7.59e-6)
(N V N, 9.8e-8)
(P, 1,00e-8)

(c)

(N V ART, 7.3e-5)
(ART, 7.1e-5)
(N N, 1.1e-5)
(V, 7.59e-6)
(N P ART, 5.7e-6)
(N V N, 9.8e-8)
(N P N, 5.1e-8)
(P, 1,00e-8)

(d)

Figure 6: The agenda for the first four steps of the Best-first search

and (N V P) hypotheses as they are so low (3.1e-12). After we extend the (N P) hypothesis in a similar way we get the agenda shown in 6 (d). Note that the agenda contains two paths of length three ending in ART on the agenda. In the Viterbi algorithm, we would have just kept the best path and dropped the other. The lower probability hypothesis could never be part of the best path, since the probability of the remaining paths depend only on the last state, ART, which is identical between the two hypotheses. This points out a key difference between Viterbi algorithms that use dynamic programming and the best-first algorithm. It suggests an optimization to the best-first search in which we drop any hypotheses for which a better ranked hypothesis of the same length and ending in the same state in already on the agenda. To make this an effective enhancement, we would need to develop a fast way of checking for such supplicates (say using a table like the one we used for the Viterbi algorithm).

We now expand (N V ART), and the best expansion is (N V ART N) with a probability of 4.6e-6. Although we have a complete path now, we don't know it is the best one until this hypothesis becomes the top one on the agenda. Thus we have to expand (ART), (N N) and (V), and any extensions of these three that have a probability greater than 4.6e-6 before (N V ART N) comes to the top and we stop.

Because probabilities always decrease as we extend a hypothesis, the best-first algorithm is guaranteed to find the best overall hypothesis. It will be the first complete path that rises to the top of the agenda.

The fact that probabilities decrease as the paths get longer often creates a problem with this algorithm, because shorter paths will tend to be favored over longer ones. Thus if the differences between the probabilities are less dramatic than in our constructed example, the best-first algorithm starts to resemble a Viterbi algorithm again, considering most of the paths of length n before considering paths of length n+1. A standard AI technique for dealing with this problem is the A* search discussed in the next section.

4. A Search*

A best-first search is an A* search if the score is computed in the following way:

$$\text{Score}(h_{1,j}) = P(h_{1,j}) * H(h_{1,j})$$

Where $h_{1,j}$ is the path, $P(h_{1,j})$ is the probability of the path (as computed above), and $H(h_{1,j})$ is a heuristic estimate of the probability of the best extension of $h_{1,i}$ to the end of the sequence. Note that when i = T (the end of the observation sequence $H(h_{1,j})$ is zero and Score($w_{1,i}$, t) equals the probability of $h_{1,t}$ producing the entire observed sequence. In order to guarantee that the A* algorithm finds the best solution, we must ensure that H(h) doesn't underestimate the probability of being part of a full solution[1]. In this case, we say the heuristic function is **admissible**. Note that if we set $H(h_{1,j})$ to 1 for any hypothesis,

[1] Note, the A* algorithm is usually presented in terms of cost rather than probability. To convert to a cost basis, we could simply take the log of the probabilities. Then Score(h) = C(h) + H(h), where C is the cost of the hypothesis and H(h) would be the estimated cost to complete the search. H(h) is then admissible if it doesn't overestimate the cost of the rest of the solution.

```
The Best-First Search
    Hypotheses are sequences of word hypotheses of form (word, end pos'n)
    Initialize Agenda a null hypothesis: ø from 0 to 0, score 1.
    While Agenda is not empty do
        If top element of agenda covers the entire observation sequence, stop.
        Otherwise, Remove top element [($w_i$, $p_i$)] from agenda
        Generate new hypotheses, each by adding a likely new word to sequence,
        and add to agenda according to its score.
```

Figure 7: The best-first search algorithm

then the score used in the search would just be P($h_{1,j}$) and we would have the best-first search described in the last section.

One common approach in designing the heuristic function is be to derive some estimate of the probability of the rest of the sequence by using some average probability *p* per observation (which we could compute from a corpus). In this case, H($h_{1,j}$) would be *p* * (T - t), where t is the end position of the hypothesis, T is the length of the utterance.