Chapter 7

Sequence labeling

The goal of sequence labeling is to assign tags to words, or more generally, to assign discrete labels to discrete elements in a sequence. There are many applications of sequence labeling in natural language processing, and chapter 8 presents an overview. For now, we'll focus on the classic problem of **part-of-speech tagging**, which requires tagging each word by its grammatical category. Coarse-grained grammatical categories include **N**OUNs, which describe things, properties, or ideas, and **V**ERBs, which describe actions and events. Consider a simple input:

(7.1) They can fish.

A dictionary of coarse-grained part-of-speech tags might include NOUN as the only valid tag for *they*, but both NOUN and VERB as potential tags for *can* and *fish*. A accurate sequence labeling algorithm should select the verb tag for both *can* and *fish* in (7.1), but it should select noun for the same two words in the phrase *can of fish*.

7.1 Sequence labeling as classification

One way to solve a tagging problem is to turn it into a classification problem. Let $\boldsymbol{f}((\boldsymbol{w}, m), y)$ indicate the feature function for tag y at position m in the sequence $\boldsymbol{w} = (w_1, w_2, \ldots, w_M)$. A simple tagging model would have a single base feature, the word itself:

$$\boldsymbol{f}((\boldsymbol{w} = \textit{they can fish}, m = 1), \text{N}) = (\textit{they}, \text{N}) \quad [7.1]$$
$$\boldsymbol{f}((\boldsymbol{w} = \textit{they can fish}, m = 2), \text{V}) = (\textit{can}, \text{V}) \quad [7.2]$$
$$\boldsymbol{f}((\boldsymbol{w} = \textit{they can fish}, m = 3), \text{V}) = (\textit{fish}, \text{V}). \quad [7.3]$$

Here the feature function takes three arguments as input: the sentence to be tagged (e.g., *they can fish*), the proposed tag (e.g., N or V), and the index of the token to which this tag

is applied. This simple feature function then returns a single feature: a tuple including the word to be tagged and the tag that has been proposed. If the vocabulary size is V and the number of tags is K, then there are $V \times K$ features. Each of these features must be assigned a weight. These weights can be learned from a labeled dataset using a classification algorithm such as perceptron, but this isn't necessary in this case: it would be equivalent to define the classification weights directly, with $\theta_{w,y} = 1$ for the tag y most frequently associated with word w, and $\theta_{w,y} = 0$ for all other tags.

However, it is easy to see that this simple classification approach cannot correctly tag both *they can fish* and *can of fish*, because *can* and *fish* are grammatically ambiguous. To handle both of these cases, the tagger must rely on context, such as the surrounding words. We can build context into the feature set by incorporating the surrounding words as additional features:

$$\begin{aligned}
\boldsymbol{f}((\boldsymbol{w} = \textit{they can fish}, 1), \text{N}) = \{&(w_m = \textit{they}, y_m = \text{N}), \\
&(w_{m-1} = \square, y_m = \text{N}), \\
&(w_{m+1} = \textit{can}, y_m = \text{N})\} \qquad [7.4] \\
\boldsymbol{f}((\boldsymbol{w} = \textit{they can fish}, 2), \text{V}) = \{&(w_m = \textit{can}, y_m = \text{V}), \\
&(w_{m-1} = \textit{they}, y_m = \text{V}), \\
&(w_{m+1} = \textit{fish}, y_m = \text{V})\} \qquad [7.5] \\
\boldsymbol{f}((\boldsymbol{w} = \textit{they can fish}, 3), \text{V}) = \{&(w_m = \textit{fish}, y_m = \text{V}), \\
&(w_{m-1} = \textit{can}, y_m = \text{V}), \\
&(w_{m+1} = \blacksquare, y_m = \text{V})\}. \qquad [7.6]
\end{aligned}$$

These features contain enough information that a tagger should be able to choose the right tag for the word *fish*: words that come after *can* are likely to be verbs, so the feature $(w_{m-1} = \textit{can}, y_m = \text{V})$ should have a large positive weight.

However, even with this enhanced feature set, it may be difficult to tag some sequences correctly. One reason is that there are often relationships between the tags themselves. For example, in English it is relatively rare for a verb to follow another verb — particularly if we differentiate **M**ODAL verbs like *can* and *should* from more typical verbs, like *give*, *transcend*, and *befuddle*. We would like to incorporate preferences against tag sequences like VERB-VERB, and in favor of tag sequences like NOUN-VERB. The need for such preferences is best illustrated by a **garden path sentence**:

(7.2) The old man the boat.

Grammatically, the word *the* is a **D**ETERMINER. When you read the sentence, what part of speech did you first assign to *old*? Typically, this word is an ADJECTIVE — abbreviated as J — which is a class of words that modify nouns. Similarly, *man* is usually a noun. The resulting sequence of tags is D J N D N. But this is a mistaken "garden path" interpretation, which ends up leading nowhere. It is unlikely that a determiner would directly

follow a noun,[1] and it is particularly unlikely that the entire sentence would lack a verb. The only possible verb in (7.2) is the word *man*, which can refer to the act of maintaining and piloting something — often boats. But if *man* is tagged as a verb, then *old* is seated between a determiner and a verb, and must be a noun. And indeed, adjectives often have a second interpretation as nouns when used in this way (e.g., *the young*, *the restless*). This reasoning, in which the labeling decisions are intertwined, cannot be applied in a setting where each tag is produced by an independent classification decision.

7.2 Sequence labeling as structure prediction

As an alternative, think of the entire sequence of tags as a label itself. For a given sequence of words $\boldsymbol{w} = (w_1, w_2, \ldots, w_M)$, there is a set of possible taggings $\mathcal{Y}(\boldsymbol{w}) = \mathcal{Y}^M$, where $\mathcal{Y} = \{\text{N}, \text{V}, \text{D}, \ldots\}$ refers to the set of individual tags, and $\mathcal{Y}^M$ refers to the set of tag sequences of length M. We can then treat the sequence labeling problem as a classification problem in the label space $\mathcal{Y}(\boldsymbol{w})$,

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})}{\text{argmax}}\, \Psi(\boldsymbol{w}, \boldsymbol{y}), \tag{7.7}$$

where $\boldsymbol{y} = (y_1, y_2, \ldots, y_M)$ is a sequence of M tags, and Ψ is a scoring function on pairs of sequences, $V^M \times \mathcal{Y}^M \rightarrow \mathbb{R}$. Such a function can include features that capture the relationships between tagging decisions, such as the preference that determiners not follow nouns, or that all sentences have verbs.

Given that the label space is exponentially large in the length of the sequence M, can it ever be practical to perform tagging in this way? The problem of making a series of interconnected labeling decisions is known as **inference**. Because natural language is full of interrelated grammatical structures, inference is a crucial aspect of natural language processing. In English, it is not unusual to have sentences of length $M = 20$; part-of-speech tag sets vary in size from 10 to several hundred. Taking the low end of this range, we have $|\mathcal{Y}(\boldsymbol{w}_{1:M})| \approx 10^{20}$, one hundred billion billion possible tag sequences. Enumerating and scoring each of these sequences would require an amount of work that is exponential in the sequence length, so inference is intractable.

However, the situation changes when we restrict the scoring function. Suppose we choose a function that decomposes into a sum of local parts,

$$\Psi(\boldsymbol{w}, \boldsymbol{y}) = \sum_{m=1}^{M+1} \psi(\boldsymbol{w}, y_m, y_{m-1}, m), \tag{7.8}$$

where each $\psi(\cdot)$ scores a local part of the tag sequence. Note that the sum goes up to $M+1$, so that we can include a score for a special end-of-sequence tag, $\psi(\boldsymbol{w}_{1:M}, \blacklozenge, y_M, M+1)$. We also define a special tag to begin the sequence, $y_0 \triangleq \lozenge$.

[1] The main exception occurs with ditransitive verbs, such as *They gave the winner a trophy.*

In a linear model, local scoring function can be defined as a dot product of weights and features,

$$\psi(\boldsymbol{w}_{1:M}, y_m, y_{m-1}, m) = \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m). \quad [7.9]$$

The feature vector $\boldsymbol{f}$ can consider the entire input $\boldsymbol{w}$, and can look at pairs of adjacent tags. This is a step up from per-token classification: the weights can assign low scores to infelicitous tag pairs, such as noun-determiner, and high scores for frequent tag pairs, such as determiner-noun and noun-verb.

In the example *they can fish*, a minimal feature function would include features for word-tag pairs (sometimes called **emission features**) and tag-tag pairs (sometimes called **transition features**):

$$\begin{aligned}
\boldsymbol{f}(\boldsymbol{w} = \textit{they can fish}, \boldsymbol{y} = \text{N V V }) &= \sum_{m=1}^{M+1} \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) && [7.10] \\
&= \boldsymbol{f}(\boldsymbol{w}, \text{N}, \lozenge, 1) \\
&\quad + \boldsymbol{f}(\boldsymbol{w}, \text{V}, \text{N}, 2) \\
&\quad + \boldsymbol{f}(\boldsymbol{w}, \text{V}, \text{V}, 3) \\
&\quad + \boldsymbol{f}(\boldsymbol{w}, \blacklozenge, \text{V}, 4) && [7.11] \\
&= (w_m = \textit{they}, y_m = \text{N}) + (y_m = \text{N}, y_{m-1} = \lozenge) \\
&\quad + (w_m = \textit{can}, y_m = \text{V}) + (y_m = \text{V}, y_{m-1} = \text{N}) \\
&\quad + (w_m = \textit{fish}, y_m = \text{V}) + (y_m = \text{V}, y_{m-1} = \text{V}) \\
&\quad + (y_m = \blacklozenge, y_{m-1} = \text{V}). && [7.12]
\end{aligned}$$

There are seven active features for this example: one for each word-tag pair, and one for each tag-tag pair, including a final tag $y_{M+1} = \blacklozenge$. These features capture the two main sources of information for part-of-speech tagging in English: which tags are appropriate for each word, and which tags tend to follow each other in sequence. Given appropriate weights for these features, taggers can achieve high accuracy, even for difficult cases like *the old man the boat*. We will now discuss how this restricted scoring function enables efficient inference, through the **Viterbi algorithm** (Viterbi, 1967).

7.3 The Viterbi algorithm

By decomposing the scoring function into a sum of local parts, it is possible to rewrite the tagging problem as follows:

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})}{\text{argmax}} \Psi(\boldsymbol{w}, \boldsymbol{y}) \tag{7.13}$$

$$= \underset{\boldsymbol{y}_{1:M}}{\text{argmax}} \sum_{m=1}^{M+1} \psi(\boldsymbol{w}, y_m, y_{m-1}, m) \tag{7.14}$$

$$= \underset{\boldsymbol{y}_{1:M}}{\text{argmax}} \sum_{m=1}^{M+1} s_m(y_m, y_{m-1}), \tag{7.15}$$

where the final line simplifies the notation with the shorthand,

$$s_m(y_m, y_{m-1}) \triangleq \psi(\boldsymbol{w}_{1:M}, y_m, y_{m-1}, m). \tag{7.16}$$

This inference problem can be solved efficiently using **dynamic programming**, an algorithmic technique for reusing work in recurrent computations. We begin by solving an auxiliary problem: rather than finding the best tag sequence, we compute the *score* of the best tag sequence,

$$\max_{\boldsymbol{y}_{1:M}} \Psi(\boldsymbol{w}, \boldsymbol{y}_{1:M}) = \max_{\boldsymbol{y}_{1:M}} \sum_{m=1}^{M+1} s_m(y_m, y_{m-1}). \tag{7.17}$$

This score involves a maximization over all tag sequences of length M, written $\max_{\boldsymbol{y}_{1:M}}$. This maximization can be broken into two pieces,

$$\max_{\boldsymbol{y}_{1:M}} \Psi(\boldsymbol{w}, \boldsymbol{y}_{1:M}) = \max_{y_M} \max_{\boldsymbol{y}_{1:M-1}} \sum_{m=1}^{M+1} s_m(y_m, y_{m-1}). \tag{7.18}$$

Within the sum, only the final term $s_{M+1}(\blacklozenge, y_M)$ depends on y_M, so we can pull this term out of the second maximization,

$$\max_{\boldsymbol{y}_{1:M}} \Psi(\boldsymbol{w}, \boldsymbol{y}_{1:M}) = \left(\max_{y_M} s_{M+1}(\blacklozenge, y_M) \right) + \left(\max_{\boldsymbol{y}_{1:M-1}} \sum_{m=1}^{M} s_m(y_m, y_{m-1}) \right). \tag{7.19}$$

The second term in Equation 7.19 has the same form as our original problem, with M replaced by $M-1$. This indicates that the problem can be reformulated as a recurrence. We do this by defining an auxiliary variable called the **Viterbi variable** $v_m(k)$, representing

Algorithm 11 The Viterbi algorithm. Each $s_m(k, k')$ is a local score for tag $y_m = k$ and $y_{m-1} = k'$.

for $k \in \{0, \ldots K\}$ **do**
  $v_1(k) = s_1(k, \lozenge)$
for $m \in \{2, \ldots, M\}$ **do**
  for $k \in \{0, \ldots, K\}$ **do**
    $v_m(k) = \max_{k'} s_m(k, k') + v_{m-1}(k')$
    $b_m(k) = \text{argmax}_{k'} s_m(k, k') + v_{m-1}(k')$
$y_M = \text{argmax}_k s_{M+1}(\blacklozenge, k) + v_M(k)$
for $m \in \{M-1, \ldots 1\}$ **do**
  $y_m = b_m(y_{m+1})$
return $\boldsymbol{y}_{1:M}$

the score of the best sequence terminating in the tag k:

$$v_m(y_m) \triangleq \max_{\boldsymbol{y}_{1:m-1}} \sum_{n=1}^{m} s_n(y_n, y_{n-1}) \tag{7.20}$$

$$= \max_{y_{m-1}} s_m(y_m, y_{m-1}) + \max_{\boldsymbol{y}_{1:m-2}} \sum_{n=1}^{m-1} s_n(y_n, y_{n-1}) \tag{7.21}$$

$$= \max_{y_{m-1}} s_m(y_m, y_{m-1}) + v_{m-1}(y_{m-1}). \tag{7.22}$$

Each set of Viterbi variables is computed from the local score $s_m(y_m, y_{m-1})$, and from the previous set of Viterbi variables. The initial condition of the recurrence is simply the score for the first tag,

$$v_1(y_1) \triangleq s_1(y_1, \lozenge). \tag{7.23}$$

The maximum overall score for the sequence is then the final Viterbi variable,

$$\max_{\boldsymbol{y}_{1:M}} \Psi(\boldsymbol{w}_{1:M}, \boldsymbol{y}_{1:M}) = v_{M+1}(\blacklozenge). \tag{7.24}$$

Thus, the score of the best labeling for the sequence can be computed in a single forward sweep: first compute all variables $v_1(\cdot)$ from Equation 7.23, and then compute all variables $v_2(\cdot)$ from the recurrence in Equation 7.22, continuing until the final variable $v_{M+1}(\blacklozenge)$.

The Viterbi variables can be arranged in a structure known as a **trellis**, shown in Figure 7.1. Each column indexes a token m in the sequence, and each row indexes a tag in $\mathcal{Y}$; every $v_{m-1}(k)$ is connected to every $v_m(k')$, indicating that $v_m(k')$ is computed from $v_{m-1}(k)$. Special nodes are set aside for the start and end states.

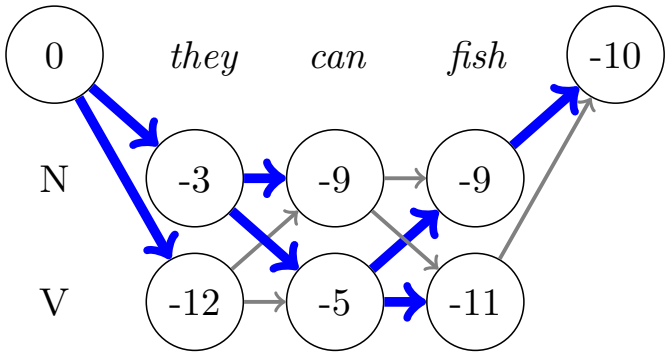

Figure 7.1: The trellis representation of the Viterbi variables, for the example *they can fish*, using the weights shown in Table 7.1.

The original goal was to find the best scoring sequence, not simply to compute its score. But by solving the auxiliary problem, we are almost there. Recall that each $v_m(k)$ represents the score of the best tag sequence ending in that tag k in position m. To compute this, we maximize over possible values of y_{m-1}. By keeping track of the "argmax" tag that maximizes this choice at each step, we can walk backwards from the final tag, and recover the optimal tag sequence. This is indicated in Figure 7.1 by the thick lines, which we trace back from the final position. These backward pointers are written $b_m(k)$, indicating the optimal tag y_{m-1} on the path to $Y_m = k$.

The complete Viterbi algorithm is shown in Algorithm 11. When computing the initial Viterbi variables $v_1(\cdot)$, the special tag $\lozenge$ indicates the start of the sequence. When computing the final tag Y_M, another special tag,$\blacklozenge$ indicates the end of the sequence. These special tags enable the use of transition features for the tags that begin and end the sequence: for example, conjunctions are unlikely to end sentences in English, so we would like a low score for $s_{M+1}(\blacklozenge, \text{CC})$; nouns are relatively likely to appear at the beginning of sentences, so we would like a high score for $s_1(\text{N}, \lozenge)$, assuming the noun tag is compatible with the first word token w_1.

Complexity If there are K tags and M positions in the sequence, then there are $M \times K$ Viterbi variables to compute. Computing each variable requires finding a maximum over K possible predecessor tags. The total time complexity of populating the trellis is therefore $\mathcal{O}(MK^2)$, with an additional factor for the number of active features at each position. After completing the trellis, we simply trace the backwards pointers to the beginning of the sequence, which takes $\mathcal{O}(M)$ operations.

	they	*can*	*fish*
N	-2	-3	-3
V	-10	-1	-3

(a) Weights for emission features.

	N	V	♦
◊	-1	-2	$-\infty$
N	-3	-1	-1
V	-1	-3	-1

(b) Weights for transition features. The "from" tags are on the columns, and the "to" tags are on the rows.

Table 7.1: Feature weights for the example trellis shown in Figure 7.1. Emission weights from ◊ and ♦ are implicitly set to $-\infty$.

7.3.1 Example

Consider the minimal tagset $\{\text{N}, \text{V}\}$, corresponding to nouns and verbs. Even in this tagset, there is considerable ambiguity: for example, the words *can* and *fish* can each take both tags. Of the $2 \times 2 \times 2 = 8$ possible taggings for the sentence *they can fish*, four are possible given these possible tags, and two are grammatical.[2]

The values in the trellis in Figure 7.1 are computed from the feature weights defined in Table 7.1. We begin with $v_1(\text{N})$, which has only one possible predecessor, the start tag ◊. This score is therefore equal to $s_1(\text{N}, ◊) = -2 - 1 = -3$, which is the sum of the scores for the emission and transition features respectively; the backpointer is $b_1(\text{N}) = ◊$. The score for $v_1(\text{V})$ is computed in the same way: $s_1(\text{V}, ◊) = -10 - 2 = -12$, and again $b_1(\text{V}) = ◊$. The backpointers are represented in the figure by thick lines.

Things get more interesting at $m = 2$. The score $v_2(\text{N})$ is computed by maximizing over the two possible predecessors,

$$\begin{aligned} v_2(\text{N}) &= \max(v_1(\text{N}) + s_2(\text{N}, \text{N}), v_1(\text{V}) + s_2(\text{N}, \text{V})) && [7.25] \\ &= \max(-3 - 3 - 3, \quad -12 - 3 - 1) = -9 && [7.26] \\ b_2(\text{N}) &= \text{N}. && [7.27] \end{aligned}$$

This continues until reaching $v_4(♦)$, which is computed as,

$$\begin{aligned} v_4(♦) &= \max(v_3(\text{N}) + s_4(♦, \text{N}), v_3(\text{V}) + s_4(♦, \text{V})) && [7.28] \\ &= \max(-9 + 0 - 1, \quad -11 + 0 - 1) && [7.29] \\ &= -10, && [7.30] \end{aligned}$$

so $b_4(♦) = \text{N}$. As there is no emission w_4, the emission features have scores of zero.

[2] The tagging *they*/N *can*/V *fish*/N corresponds to the scenario of putting fish into cans, or perhaps of firing them.

To compute the optimal tag sequence, we walk backwards from here, next checking $b_3(\text{N}) = \text{V}$, and then $b_2(\text{V}) = \text{N}$, and finally $b_1(\text{N}) = \lozenge$. This yields $\boldsymbol{y} = (\text{N}, \text{V}, \text{N})$, which corresponds to the linguistic interpretation of the fishes being put into cans.

7.3.2 Higher-order features

The Viterbi algorithm was made possible by a restriction of the scoring function to local parts that consider only pairs of adjacent tags. We can think of this as a bigram language model over tags. A natural question is how to generalize Viterbi to tag trigrams, which would involve the following decomposition:

$$\Psi(\boldsymbol{w}, \boldsymbol{y}) = \sum_{m=1}^{M+2} \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, y_{m-2}, m), \quad [7.31]$$

where $y_{-1} = \lozenge$ and $y_{M+2} = \blacklozenge$.

One solution is to create a new tagset $\mathcal{Y}^{(2)}$ from the Cartesian product of the original tagset with itself, $\mathcal{Y}^{(2)} = \mathcal{Y} \times \mathcal{Y}$. The tags in this product space are ordered pairs, representing adjacent tags at the token level: for example, the tag (N, V) would represent a noun followed by a verb. Transitions between such tags must be consistent: we can have a transition from (N, V) to (V, N) (corresponding to the tag sequence N V N), but not from (N, V) to (N, N), which would not correspond to any coherent tag sequence. This constraint can be enforced in feature weights, with $\theta_{((a,b),(c,d))} = -\infty$ if $b \neq c$. The remaining feature weights can encode preferences for and against various tag trigrams.

In the Cartesian product tag space, there are K^2 tags, suggesting that the time complexity will increase to $\mathcal{O}(MK^4)$. However, it is unnecessary to $\max$ over predecessor tag bigrams that are incompatible with the current tag bigram. By exploiting this constraint, it is possible to limit the time complexity to $\mathcal{O}(MK^3)$. The space complexity grows to $\mathcal{O}(MK^2)$, since the trellis must store all possible predecessors of each tag. In general, the time and space complexity of higher-order Viterbi grows exponentially with the order of the tag n-grams that are considered in the feature decomposition.

7.4 Hidden Markov Models

The Viterbi sequence labeling algorithm is built on the scores $s_m(y, y')$. We will now discuss how these scores can be estimated probabilistically. Recall from § 2.2 that the probabilistic Naïve Bayes classifier selects the label y to maximize $\text{p}(y \mid \boldsymbol{x}) \propto \text{p}(y, \boldsymbol{x})$. In probabilistic sequence labeling, our goal is similar: select the tag sequence that maximizes $\text{p}(\boldsymbol{y} \mid \boldsymbol{w}) \propto \text{p}(\boldsymbol{y}, \boldsymbol{w})$. The locality restriction in Equation 7.8 can be viewed as a conditional independence assumption on the random variables $\boldsymbol{y}$.

Algorithm 12 Generative process for the hidden Markov model

$y_0 \leftarrow \lozenge, \quad m \leftarrow 1$
repeat
  $y_m \sim \text{Categorical}(\boldsymbol{\lambda}_{y_{m-1}})$ ▷ sample the current tag
  $w_m \sim \text{Categorical}(\boldsymbol{\phi}_{y_m})$ ▷ sample the current word
until $y_m = \blacklozenge$ ▷ terminate when the stop symbol is generated

Naïve Bayes was introduced as a **generative model** — a probabilistic story that explains the observed data as well as the hidden label. A similar story can be constructed for probabilistic sequence labeling: first, the tags are drawn from a prior distribution; next, the tokens are drawn from a conditional likelihood. However, for inference to be tractable, additional independence assumptions are required. First, the probability of each token depends only on its tag, and not on any other element in the sequence:

$$\text{p}(\boldsymbol{w} \mid \boldsymbol{y}) = \prod_{m=1}^{M} \text{p}(w_m \mid y_m). \quad [7.32]$$

Second, each tag y_m depends only on its predecessor,

$$\text{p}(\boldsymbol{y}) = \prod_{m=1}^{M} \text{p}(y_m \mid y_{m-1}), \quad [7.33]$$

where $y_0 = \lozenge$ in all cases. Due to this **Markov assumption**, probabilistic sequence labeling models are known as **hidden Markov models** (HMMs).

The generative process for the hidden Markov model is shown in Algorithm 12. Given the parameters $\boldsymbol{\lambda}$ and $\boldsymbol{\phi}$, we can compute $\text{p}(\boldsymbol{w}, \boldsymbol{y})$ for any token sequence $\boldsymbol{w}$ and tag sequence $\boldsymbol{y}$. The HMM is often represented as a **graphical model** (Wainwright and Jordan, 2008), as shown in Figure 7.2. This representation makes the independence assumptions explicit: if a variable v_1 is probabilistically conditioned on another variable v_2, then there is an arrow $v_2 \rightarrow v_1$ in the diagram. If there are no arrows between v_1 and v_2, they are **conditionally independent**, given each variable's **Markov blanket**. In the hidden Markov model, the Markov blanket for each tag y_m includes the "parent" y_{m-1}, and the "children" y_{m+1} and w_m.[3]

It is important to reflect on the implications of the HMM independence assumptions. A non-adjacent pair of tags y_m and y_n are conditionally independent; if $m < n$ and we are given y_{n-1}, then y_m offers no additional information about y_n. However, if we are not given any information about the tags in a sequence, then all tags are probabilistically coupled.

[3] In general graphical models, a variable's Markov blanket includes its parents, children, and its children's other parents (Murphy, 2012).

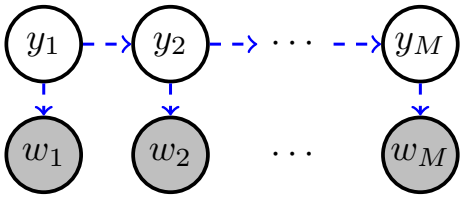

Figure 7.2: Graphical representation of the hidden Markov model. Arrows indicate probabilistic dependencies.

7.4.1 Estimation

The hidden Markov model has two groups of parameters:

Emission probabilities. The probability $\text{p}_e(w_m \mid y_m; \phi)$ is the emission probability, since the words are treated as probabilistically "emitted", conditioned on the tags.

Transition probabilities. The probability $\text{p}_t(y_m \mid y_{m-1}; \lambda)$ is the transition probability, since it assigns probability to each possible tag-to-tag transition.

Both of these groups of parameters are typically computed from smoothed relative frequency estimation on a labeled corpus (see § 6.2 for a review of smoothing). The unsmoothed probabilities are,

$$\phi_{k,i} \triangleq \Pr(W_m = i \mid Y_m = k) = \frac{\text{count}(W_m = i, Y_m = k)}{\text{count}(Y_m = k)}$$
$$\lambda_{k,k'} \triangleq \Pr(Y_m = k' \mid Y_{m-1} = k) = \frac{\text{count}(Y_m = k', Y_{m-1} = k)}{\text{count}(Y_{m-1} = k)}.$$

Smoothing is more important for the emission probability than the transition probability, because the vocabulary is much larger than the number of tags.

7.4.2 Inference

The goal of inference in the hidden Markov model is to find the highest probability tag sequence,

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y}}{\text{argmax}}\, \text{p}(\boldsymbol{y} \mid \boldsymbol{w}). \quad [7.34]$$

As in Naïve Bayes, it is equivalent to find the tag sequence with the highest *log*-probability, since the logarithm is a monotonically increasing function. It is furthermore equivalent to maximize the joint probability $\text{p}(\boldsymbol{y}, \boldsymbol{w}) = \text{p}(\boldsymbol{y} \mid \boldsymbol{w}) \times \text{p}(\boldsymbol{w}) \propto \text{p}(\boldsymbol{y} \mid \boldsymbol{w})$, which is proportional to the conditional probability. Putting these observations together, the inference

problem can be reformulated as,

$$\hat{\boldsymbol{y}} = \operatorname*{argmax}_{\boldsymbol{y}} \log \mathrm{p}(\boldsymbol{y}, \boldsymbol{w}). \tag{7.35}$$

We can now apply the HMM independence assumptions:

$$\log \mathrm{p}(\boldsymbol{y}, \boldsymbol{w}) = \log \mathrm{p}(\boldsymbol{y}) + \log \mathrm{p}(\boldsymbol{w} \mid \boldsymbol{y}) \tag{7.36}$$

$$= \sum_{m=1}^{M+1} \log \mathrm{p}_Y(y_m \mid y_{m-1}) + \log \mathrm{p}_{W|Y}(w_m \mid y_m) \tag{7.37}$$

$$= \sum_{m=1}^{M+1} \log \lambda_{y_m, y_{m-1}} + \log \phi_{y_m, w_m} \tag{7.38}$$

$$= \sum_{m=1}^{M+1} s_m(y_m, y_{m-1}), \tag{7.39}$$

where,

$$s_m(y_m, y_{m-1}) \triangleq \log \lambda_{y_m, y_{m-1}} + \log \phi_{y_m, w_m}, \tag{7.40}$$

and,

$$\phi_{\blacklozenge, w} = \begin{cases} 1, & w = \blacksquare \\ 0, & \text{otherwise}, \end{cases} \tag{7.41}$$

which ensures that the stop tag $\blacklozenge$ can only be applied to the final token $\blacksquare$.

This derivation shows that HMM inference can be viewed as an application of the Viterbi decoding algorithm, given an appropriately defined scoring function. The local score $s_m(y_m, y_{m-1})$ can be interpreted probabilistically,

$$s_m(y_m, y_{m-1}) = \log \mathrm{p}_y(y_m \mid y_{m-1}) + \log \mathrm{p}_{w|y}(w_m \mid y_m) \tag{7.42}$$

$$= \log \mathrm{p}(y_m, w_m \mid y_{m-1}). \tag{7.43}$$

Now recall the definition of the Viterbi variables,

$$v_m(y_m) = \max_{y_{m-1}} s_m(y_m, y_{m-1}) + v_{m-1}(y_{m-1}) \tag{7.44}$$

$$= \max_{y_{m-1}} \log \mathrm{p}(y_m, w_m \mid y_{m-1}) + v_{m-1}(y_{m-1}). \tag{7.45}$$

By setting $v_{m-1}(y_{m-1}) = \max_{\boldsymbol{y}_{1:m-2}} \log \mathrm{p}(\boldsymbol{y}_{1:m-1}, \boldsymbol{w}_{1:m-1})$, we obtain the recurrence,

$$v_m(y_m) = \max_{y_{m-1}} \log \mathrm{p}(y_m, w_m \mid y_{m-1}) + \max_{\boldsymbol{y}_{1:m-2}} \log \mathrm{p}(\boldsymbol{y}_{1:m-1}, \boldsymbol{w}_{1:m-1}) \tag{7.46}$$

$$= \max_{\boldsymbol{y}_{1:m-1}} \log \mathrm{p}(y_m, w_m \mid y_{m-1}) + \log \mathrm{p}(\boldsymbol{y}_{1:m-1}, \boldsymbol{w}_{1:m-1}) \tag{7.47}$$

$$= \max_{\boldsymbol{y}_{1:m-1}} \log \mathrm{p}(\boldsymbol{y}_{1:m}, \boldsymbol{w}_{1:m}). \tag{7.48}$$

In words, the Viterbi variable $v_m(y_m)$ is the log probability of the best tag sequence ending in y_m, joint with the word sequence $\boldsymbol{w}_{1:m}$. The log probability of the best complete tag sequence is therefore,

$$\max_{\boldsymbol{y}_{1:M}} \log \text{p}(\boldsymbol{y}_{1:M+1}, \boldsymbol{w}_{1:M+1}) = v_{M+1}(\blacklozenge) \tag{7.49}$$

***Viterbi as an example of the max-product algorithm** The Viterbi algorithm can also be implemented using probabilities, rather than log-probabilities. In this case, each $v_m(y_m)$ is equal to,

$$\begin{aligned}
v_m(y_m) &= \max_{\boldsymbol{y}_{1:m-1}} \text{p}(\boldsymbol{y}_{1:m-1}, y_m, \boldsymbol{w}_{1:m}) && [7.50]\\
&= \max_{y_{m-1}} \text{p}(y_m, w_m \mid y_{m-1}) \times \max_{\boldsymbol{y}_{1:m-2}} \text{p}(y_{1:m-2}, y_{m-1}, \boldsymbol{w}_{1:m-1}) && [7.51]\\
&= \max_{y_{m-1}} \text{p}(y_m, w_m \mid y_{m-1}) \times v_{m-1}(y_{m-1}) && [7.52]\\
&= \text{p}_{w|y}(w_m \mid y_m) \times \max_{y_{m-1}} \text{p}_y(y_m \mid y_{m-1}) \times v_{m-1}(y_{m-1}). && [7.53]
\end{aligned}$$

Each Viterbi variable is computed by *maximizing* over a set of *products*. Thus, the Viterbi algorithm is a special case of the **max-product algorithm** for inference in graphical models (Wainwright and Jordan, 2008). However, the product of probabilities tends towards zero over long sequences, so the log-probability version of Viterbi is recommended in practical implementations.

7.5 Discriminative sequence labeling with features

Today, hidden Markov models are rarely used for supervised sequence labeling. This is because HMMs are limited to only two phenomena:

- word-tag compatibility, via the emission probability $\text{p}_{W|Y}(w_m \mid y_m)$;
- local context, via the transition probability $\text{p}_Y(y_m \mid y_{m-1})$.

The Viterbi algorithm permits the inclusion of richer information in the local scoring function $\psi(\boldsymbol{w}_{1:M}, y_m, y_{m-1}, m)$, which can be defined as a weighted sum of arbitrary local *features*,

$$\psi(\boldsymbol{w}, y_m, y_{m-1}, m) = \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m), \tag{7.54}$$

where $\boldsymbol{f}$ is a locally-defined feature function, and $\boldsymbol{\theta}$ is a vector of weights.

The local decomposition of the scoring function Ψ is reflected in a corresponding decomposition of the feature function:

$$\Psi(\boldsymbol{w}, \boldsymbol{y}) = \sum_{m=1}^{M+1} \psi(\boldsymbol{w}, y_m, y_{m-1}, m) \quad [7.55]$$

$$= \sum_{m=1}^{M+1} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) \quad [7.56]$$

$$= \boldsymbol{\theta} \cdot \sum_{m=1}^{M+1} \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) \quad [7.57]$$

$$= \boldsymbol{\theta} \cdot \boldsymbol{f}^{(\text{global})}(\boldsymbol{w}, \boldsymbol{y}_{1:M}), \quad [7.58]$$

where $\boldsymbol{f}^{(\text{global})}(\boldsymbol{w}, \boldsymbol{y})$ is a global feature vector, which is a sum of local feature vectors,

$$\boldsymbol{f}^{(\text{global})}(\boldsymbol{w}, \boldsymbol{y}) = \sum_{m=1}^{M+1} \boldsymbol{f}(\boldsymbol{w}_{1:M}, y_m, y_{m-1}, m), \quad [7.59]$$

with $y_{M+1} = \blacklozenge$ and $y_0 = \lozenge$ by construction.

Let's now consider what additional information these features might encode.

Word affix features. Consider the problem of part-of-speech tagging on the first four lines of the poem *Jabberwocky* (Carroll, 1917):

(7.3) 'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe.

Many of these words were made up by the author of the poem, so a corpus would offer no information about their probabilities of being associated with any particular part of speech. Yet it is not so hard to see what their grammatical roles might be in this passage. Context helps: for example, the word *slithy* follows the determiner *the*, so it is probably a noun or adjective. Which do you think is more likely? The suffix *-thy* is found in a number of adjectives, like *frothy, healthy, pithy, worthy*. It is also found in a handful of nouns — e.g., *apathy, sympathy* — but nearly all of these have the longer coda *-pathy*, unlike *slithy*. So the suffix gives some evidence that *slithy* is an adjective, and indeed it is: later in the text we find that it is a combination of the adjectives *lithe* and *slimy*.[4]

[4] **Morphology** is the study of how words are formed from smaller linguistic units. chapter 9 touches on computational approaches to morphological analysis. See Bender (2013) for an overview of the underlying linguistic principles, and Haspelmath and Sims (2013) or Lieber (2015) for a full treatment.

Fine-grained context. The hidden Markov model captures contextual information in the form of part-of-speech tag bigrams. But sometimes, the necessary contextual information is more specific. Consider the noun phrases *this fish* and *these fish*. Many part-of-speech tagsets distinguish between singular and plural nouns, but do not distinguish between singular and plural determiners; for example, the well known **Penn Treebank** tagset follows these conventions. A hidden Markov model would be unable to correctly label *fish* as singular or plural in both of these cases, because it only has access to two features: the preceding tag (determiner in both cases) and the word (*fish* in both cases). The classification-based tagger discussed in § 7.1 had the ability to use preceding and succeeding words as features, and it can also be incorporated into a Viterbi-based sequence labeler as a local feature.

Example. Consider the tagging D J N (determiner, adjective, noun) for the sequence *the slithy toves*, so that

$$\begin{aligned}\boldsymbol{w} &= \textit{the slithy toves} \\ \boldsymbol{y} &= \text{D J N}.\end{aligned}$$

Let's create the feature vector for this example, assuming that we have word-tag features (indicated by W), tag-tag features (indicated by T), and suffix features (indicated by M). You can assume that you have access to a method for extracting the suffix *-thy* from *slithy*, *-es* from *toves*, and $\varnothing$ from *the*, indicating that this word has no suffix.[5] The resulting feature vector is,

$$\begin{aligned}\boldsymbol{f}(\text{the slithy toves, D J N}) =& \boldsymbol{f}(\text{the slithy toves, D}, \lozenge, 1) \\ &+ \boldsymbol{f}(\text{the slithy toves, J, D}, 2) \\ &+ \boldsymbol{f}(\text{the slithy toves, N, J}, 3) \\ &+ \boldsymbol{f}(\text{the slithy toves}, \blacklozenge, \text{N}, 4) \\ =& \{(T : \lozenge, \text{D}), (W : \text{the, D}), (M : \varnothing, \text{D}), \\ & (T : \text{D, J}), (W : \text{slithy, J}), (M : \text{-thy, J}), \\ & (T : \text{J, N}), (W : \text{toves, N}), (M : \text{-es, N}) \\ & (T : \text{N}, \blacklozenge)\}.\end{aligned}$$

These examples show that local features can incorporate information that lies beyond the scope of a hidden Markov model. Because the features are local, it is possible to apply the Viterbi algorithm to identify the optimal sequence of tags. The remaining question

[5] Such a system is called a **morphological segmenter**. The task of morphological segmentation is briefly described in § 9.1.4; a well known segmenter is MORFESSOR (Creutz and Lagus, 2007). In real applications, a typical approach is to include features for all orthographic suffixes up to some maximum number of characters: for *slithy*, we would have suffix features for *-y*, *-hy*, and *-thy*.

is how to estimate the weights on these features. § 2.3 presented three main types of discriminative classifiers: perceptron, support vector machine, and logistic regression. Each of these classifiers has a structured equivalent, enabling it to be trained from labeled sequences rather than individual tokens.

7.5.1 Structured perceptron

The perceptron classifier is trained by increasing the weights for features that are associated with the correct label, and decreasing the weights for features that are associated with incorrectly predicted labels:

$$\hat{y} = \underset{y \in \mathcal{Y}}{\text{argmax}}\, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y) \quad [7.60]$$

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} + \boldsymbol{f}(\boldsymbol{x}, y) - \boldsymbol{f}(\boldsymbol{x}, \hat{y}). \quad [7.61]$$

We can apply exactly the same update in the case of structure prediction,

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})}{\text{argmax}}\, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) \quad [7.62]$$

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} + \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) - \boldsymbol{f}(\boldsymbol{w}, \hat{\boldsymbol{y}}). \quad [7.63]$$

This learning algorithm is called **structured perceptron**, because it learns to predict the structured output $\boldsymbol{y}$. The only difference is that instead of computing $\hat{\boldsymbol{y}}$ by enumerating the entire set $\mathcal{Y}$, the Viterbi algorithm is used to efficiently search the set of possible taggings, $\mathcal{Y}^M$. Structured perceptron can be applied to other structured outputs as long as efficient inference is possible. As in perceptron classification, weight averaging is crucial to get good performance (see § 2.3.2).

Example For the example *they can fish*, suppose that the reference tag sequence is $\boldsymbol{y}^{(i)} =$ N V V, but the tagger incorrectly returns the tag sequence $\hat{\boldsymbol{y}} =$ N V N. Assuming a model with features for emissions (w_m, y_m) and transitions (y_{m-1}, y_m), the corresponding structured perceptron update is:

$$\theta_{(\textit{fish},\text{V})} \leftarrow \theta_{(\textit{fish},\text{V})} + 1, \qquad \theta_{(\textit{fish},\text{N})} \leftarrow \theta_{(\textit{fish},\text{N})} - 1 \quad [7.64]$$

$$\theta_{(\text{V},\text{V})} \leftarrow \theta_{(\text{V},\text{V})} + 1, \qquad \theta_{(\text{V},\text{N})} \leftarrow \theta_{(\text{V},\text{N})} - 1 \quad [7.65]$$

$$\theta_{(\text{V},\blacklozenge)} \leftarrow \theta_{(\text{V},\blacklozenge)} + 1, \qquad \theta_{(\text{N},\blacklozenge)} \leftarrow \theta_{(\text{N},\blacklozenge)} - 1. \quad [7.66]$$

7.5.2 Structured support vector machines

Large-margin classifiers such as the support vector machine improve on the perceptron by pushing the classification boundary away from the training instances. The same idea can

be applied to sequence labeling. A support vector machine in which the output is a structured object, such as a sequence, is called a **structured support vector machine** (Tsochantaridis et al., 2004).[6]

In classification, we formalized the large-margin constraint as,

$$\forall y \neq y^{(i)}, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y^{(i)}) - \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y) \geq 1, \tag{7.67}$$

requiring a margin of at least 1 between the scores for all labels y that are not equal to the correct label $y^{(i)}$. The weights $\boldsymbol{\theta}$ are then learned by constrained optimization (see § 2.4.2).

This idea can be applied to sequence labeling by formulating an equivalent set of constraints for all possible labelings $\mathcal{Y}(\boldsymbol{w})$ for an input $\boldsymbol{w}$. However, there are two problems. First, in sequence labeling, some predictions are more wrong than others: we may miss only one tag out of fifty, or we may get all fifty wrong. We would like our learning algorithm to be sensitive to this difference. Second, the number of constraints is equal to the number of possible labelings, which is exponentially large in the length of the sequence.

The first problem can be addressed by adjusting the constraint to require larger margins for more serious errors. Let $c(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}) \geq 0$ represent the *cost* of predicting label $\hat{\boldsymbol{y}}$ when the true label is $\boldsymbol{y}^{(i)}$. We can then generalize the margin constraint,

$$\forall \boldsymbol{y}, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) - \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}) \geq c(\boldsymbol{y}^{(i)}, \boldsymbol{y}). \tag{7.68}$$

This cost-augmented margin constraint specializes to the constraint in Equation 7.67 if we choose the delta function $c(\boldsymbol{y}^{(i)}, \boldsymbol{y}) = \delta\left(() \boldsymbol{y}^{(i)} \neq \boldsymbol{y}\right)$. A more expressive cost function is the **Hamming cost**,

$$c(\boldsymbol{y}^{(i)}, \boldsymbol{y}) = \sum_{m=1}^{M} \delta(y_m^{(i)} \neq y_m), \tag{7.69}$$

which computes the number of errors in $\boldsymbol{y}$. By incorporating the cost function as the margin constraint, we require that the true labeling be seperated from the alternatives by a margin that is proportional to the number of incorrect tags in each alternative labeling.

The second problem is that the number of constraints is exponential in the length of the sequence. This can be addressed by focusing on the prediction $\hat{\boldsymbol{y}}$ that *maximally* violates the margin constraint. This prediction can be identified by solving the following **cost-augmented decoding** problem:

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y} \neq \boldsymbol{y}^{(i)}}{\operatorname{argmax}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}) - \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) + c(\boldsymbol{y}^{(i)}, \boldsymbol{y}) \tag{7.70}$$

$$= \underset{\boldsymbol{y} \neq \boldsymbol{y}^{(i)}}{\operatorname{argmax}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}) + c(\boldsymbol{y}^{(i)}, \boldsymbol{y}), \tag{7.71}$$

[6] This model is also known as a **max-margin Markov network** (Taskar et al., 2003), emphasizing that the scoring function is constructed from a sum of components, which are Markov independent.

where in the second line we drop the term $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)})$, which is constant in $\boldsymbol{y}$.

We can now reformulate the margin constraint for sequence labeling,

$$\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) - \max_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})} \Big(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}) + c(\boldsymbol{y}^{(i)}, \boldsymbol{y}) \Big) \geq 0. \quad [7.72]$$

If the score for $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)})$ is greater than the cost-augmented score for all alternatives, then the constraint will be met. The name "cost-augmented decoding" is due to the fact that the objective includes the standard decoding problem, $\max_{\hat{\boldsymbol{y}} \in \mathcal{Y}(\boldsymbol{w})} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \hat{\boldsymbol{y}})$, plus an additional term for the cost. Essentially, we want to train against predictions that are strong and wrong: they should score highly according to the model, yet incur a large loss with respect to the ground truth. Training adjusts the weights to reduce the score of these predictions.

For cost-augmented decoding to be tractable, the cost function must decompose into local parts, just as the feature function $\boldsymbol{f}(\cdot)$ does. The Hamming cost, defined above, obeys this property. To perform cost-augmented decoding using the Hamming cost, we need only to add features $f_m(y_m) = \delta(y_m \neq y_m^{(i)})$, and assign a constant weight of 1 to these features. Decoding can then be performed using the Viterbi algorithm.[7]

As with large-margin classifiers, it is possible to formulate the learning problem in an unconstrained form, by combining a regularization term on the weights and a Lagrangian for the constraints:

$$\min_{\boldsymbol{\theta}} \quad \frac{1}{2}||\boldsymbol{\theta}||_2^2 - C \left(\sum_i \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) - \max_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w}^{(i)})} \Big[\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}) + c(\boldsymbol{y}^{(i)}, \boldsymbol{y}) \Big] \right), \quad [7.73]$$

In this formulation, C is a parameter that controls the tradeoff between the regularization term and the margin constraints. A number of optimization algorithms have been proposed for structured support vector machines, some of which are discussed in § 2.4.2. An empirical comparison by Kummerfeld et al. (2015) shows that stochastic subgradient descent — which is essentially a cost-augmented version of the structured perceptron — is highly competitive.

7.5.3 Conditional random fields

The **conditional random field** (CRF; Lafferty et al., 2001) is a conditional probabilistic model for sequence labeling; just as structured perceptron is built on the perceptron classifier, conditional random fields are built on the logistic regression classifier.[8] The basic

[7] Are there cost functions that do not decompose into local parts? Suppose we want to assign a constant loss c to any prediction $\hat{\boldsymbol{y}}$ in which k or more predicted tags are incorrect, and zero loss otherwise. This loss function is combinatorial over the predictions, and thus we cannot decompose it into parts.

[8] The name "conditional random field" is derived from **Markov random fields**, a general class of models in which the probability of a configuration of variables is proportional to a product of scores across pairs (or

probability model is,

$$\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}) = \frac{\exp(\Psi(\boldsymbol{w}, \boldsymbol{y}))}{\sum_{\boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{w})} \exp(\Psi(\boldsymbol{w}, \boldsymbol{y}'))}. \quad [7.74]$$

This is almost identical to logistic regression (§ 2.5), but because the label space is now sequences of tags, we require efficient algorithms for both **decoding** (searching for the best tag sequence given a sequence of words $\boldsymbol{w}$ and a model $\boldsymbol{\theta}$) and for **normalization** (summing over all tag sequences). These algorithms will be based on the usual locality assumption on the scoring function, $\Psi(\boldsymbol{w}, \boldsymbol{y}) = \sum_{m=1}^{M+1} \psi(\boldsymbol{w}, y_m, y_{m-1}, m)$.

Decoding in CRFs

Decoding — finding the tag sequence $\hat{\boldsymbol{y}}$ that maximizes $\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w})$ — is a direct application of the Viterbi algorithm. The key observation is that the decoding problem does not depend on the denominator of $\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w})$,

$$\begin{aligned}
\hat{\boldsymbol{y}} &= \operatorname*{argmax}_{\boldsymbol{y}} \log \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}) \\
&= \operatorname*{argmax}_{\boldsymbol{y}} \Psi(\boldsymbol{y}, \boldsymbol{w}) - \log \sum_{\boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{w})} \exp \Psi(\boldsymbol{y}', \boldsymbol{w}) \\
&= \operatorname*{argmax}_{\boldsymbol{y}} \Psi(\boldsymbol{y}, \boldsymbol{w}) = \operatorname*{argmax}_{\boldsymbol{y}} \sum_{m=1}^{M+1} s_m(y_m, y_{m-1}).
\end{aligned}$$

This is identical to the decoding problem for structured perceptron, so the same Viterbi recurrence as defined in Equation 7.22 can be used.

Learning in CRFs

As with logistic regression, the weights $\boldsymbol{\theta}$ are learned by minimizing the regularized negative log-probability,

$$\ell = \frac{\lambda}{2}||\boldsymbol{\theta}||^2 - \sum_{i=1}^{N} \log \mathrm{p}(\boldsymbol{y}^{(i)} \mid \boldsymbol{w}^{(i)}; \boldsymbol{\theta}) \quad [7.75]$$

$$= \frac{\lambda}{2}||\boldsymbol{\theta}||^2 - \sum_{i=1}^{N} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) + \log \sum_{\boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{w}^{(i)})} \exp\left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}')\right), \quad [7.76]$$

more generally, cliques) of variables in a **factor graph**. In sequence labeling, the pairs of variables include all adjacent tags (y_m, y_{m-1}). The probability is *conditioned* on the words $\boldsymbol{w}$, which are always observed, motivating the term "conditional" in the name.

where λ controls the amount of regularization. The final term in Equation 7.76 is a sum over all possible labelings. This term is the log of the denominator in Equation 7.74, sometimes known as the **partition function**.[9] There are $|\mathcal{Y}|^M$ possible labelings of an input of size M, so we must again exploit the decomposition of the scoring function to compute this sum efficiently.

The sum $\sum_{\boldsymbol{y} \in \mathcal{Y}^{\boldsymbol{w}^{(i)}}} \exp \Psi(\boldsymbol{y}, \boldsymbol{w})$ can be computed efficiently using the **forward recurrence**, which is closely related to the Viterbi recurrence. We first define a set of **forward variables**, $\alpha_m(y_m)$, which is equal to the sum of the scores of all paths leading to tag y_m at position m:

$$\alpha_m(y_m) \triangleq \sum_{\boldsymbol{y}_{1:m-1}} \exp \sum_{n=1}^{m} s_n(y_n, y_{n-1}) \tag{7.77}$$

$$= \sum_{\boldsymbol{y}_{1:m-1}} \prod_{n=1}^{m} \exp s_n(y_n, y_{n-1}). \tag{7.78}$$

Note the similarity to the definition of the Viterbi variable, $v_m(y_m) = \max_{\boldsymbol{y}_{1:m-1}} \sum_{n=1}^{m} s_n(y_n, y_{n-1})$. In the hidden Markov model, the Viterbi recurrence had an alternative interpretation as the max-product algorithm (see Equation 7.53); analogously, the forward recurrence is known as the **sum-product algorithm**, because of the form of [7.78]. The forward variable can also be computed through a recurrence:

$$\alpha_m(y_m) = \sum_{\boldsymbol{y}_{1:m-1}} \prod_{n=1}^{m} \exp s_n(y_n, y_{n-1}) \tag{7.79}$$

$$= \sum_{y_{m-1}} (\exp s_m(y_m, y_{m-1})) \sum_{\boldsymbol{y}_{1:m-2}} \prod_{n=1}^{m-1} \exp s_n(y_n, y_{n-1}) \tag{7.80}$$

$$= \sum_{y_{m-1}} (\exp s_m(y_m, y_{m-1})) \times \alpha_{m-1}(y_{m-1}). \tag{7.81}$$

Using the forward recurrence, it is possible to compute the denominator of the conditional probability,

$$\sum_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})} \Psi(\boldsymbol{w}, \boldsymbol{y}) = \sum_{\boldsymbol{y}_{1:M}} (\exp s_{M+1}(\blacklozenge, y_M)) \prod_{m=1}^{M} \exp s_m(y_m, y_{m-1}) \tag{7.82}$$

$$= \alpha_{M+1}(\blacklozenge). \tag{7.83}$$

[9] The terminology of "potentials" and "partition functions" comes from statistical mechanics (Bishop, 2006).

The conditional log-likelihood can be rewritten,

$$\ell = \frac{\lambda}{2}||\boldsymbol{\theta}||^2 - \sum_{i=1}^{N} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) + \log \alpha_{M+1}(\blacklozenge). \quad [7.84]$$

Probabilistic programming environments, such as TORCH (Collobert et al., 2011) and DYNET (Neubig et al., 2017), can compute the gradient of this objective using automatic differentiation. The programmer need only implement the forward algorithm as a computation graph.

As in logistic regression, the gradient of the likelihood with respect to the parameters is a difference between observed and expected feature counts:

$$\frac{d\ell}{d\theta_j} = \lambda\theta_j + \sum_{i=1}^{N} E[f_j(\boldsymbol{w}^{(i)}, \boldsymbol{y})] - f_j(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}), \quad [7.85]$$

where $f_j(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)})$ refers to the count of feature j for token sequence $\boldsymbol{w}^{(i)}$ and tag sequence $\boldsymbol{y}^{(i)}$. The expected feature counts are computed "under the hood" when automatic differentiation is applied to Equation 7.84 (Eisner, 2016).

Before the widespread use of automatic differentiation, it was common to compute the feature expectations from marginal tag probabilities $\mathrm{p}(y_m \mid \boldsymbol{w})$. These marginal probabilities are sometimes useful on their own, and can be computed using the **forward-backward algorithm**. This algorithm combines the forward recurrence with an equivalent **backward recurrence**, which traverses the input from w_M back to w_1.

*Forward-backward algorithm

Marginal probabilities over tag bigrams can be written as,[10]

$$\Pr(Y_{m-1} = k', Y_m = k \mid \boldsymbol{w}) = \frac{\sum_{\boldsymbol{y}: Y_m = k, Y_{m-1} = k'} \prod_{n=1}^{M} \exp s_n(y_n, y_{n-1})}{\sum_{\boldsymbol{y}'} \prod_{n=1}^{M} \exp s_n(y'_n, y'_{n-1})}. \quad [7.86]$$

The numerator sums over all tag sequences that include the transition $(Y_{m-1} = k') \rightarrow (Y_m = k)$. Because we are only interested in sequences that include the tag bigram, this sum can be decomposed into three parts: the *prefixes* $\boldsymbol{y}_{1:m-1}$, terminating in $Y_{m-1} = k'$; the

[10] Recall the notational convention of upper-case letters for random variables, e.g. Y_m, and lower case letters for specific values, e.g., y_m, so that $Y_m = k$ is interpreted as the event of random variable Y_m taking the value k.

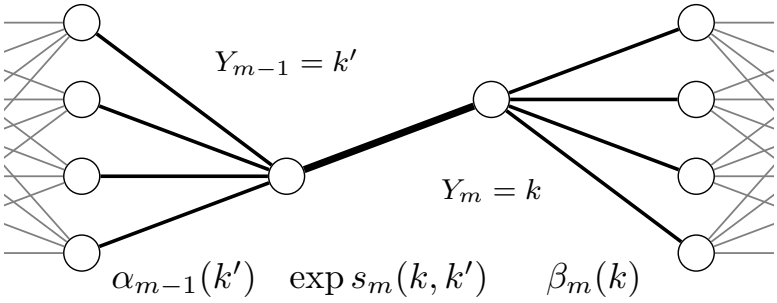

Figure 7.3: A schematic illustration of the computation of the marginal probability $\Pr(Y_{m-1} = k', Y_m = k)$, using the forward score $\alpha_{m-1}(k')$ and the backward score $\beta_m(k)$.

transition $(Y_{m-1} = k') \rightarrow (Y_m = k)$; and the *suffixes* $\boldsymbol{y}_{m:M}$, beginning with the tag $Y_m = k$:

$$\sum_{\boldsymbol{y}: Y_m = k, Y_{m-1} = k'} \prod_{n=1}^{M} \exp s_n(y_n, y_{n-1}) = \sum_{\boldsymbol{y}_{1:m-1}: Y_{m-1} = k'} \prod_{n=1}^{m-1} \exp s_n(y_n, y_{n-1}) \times \exp s_m(k, k') \times \sum_{\boldsymbol{y}_{m:M}: Y_m = k} \prod_{n=m+1}^{M+1} \exp s_n(y_n, y_{n-1}). \quad [7.87]$$

The result is product of three terms: a score that sums over all the ways to get to the position $(Y_{m-1} = k')$, a score for the transition from k' to k, and a score that sums over all the ways of finishing the sequence from $(Y_m = k)$. The first term of Equation 7.87 is equal to the **forward variable**, $\alpha_{m-1}(k')$. The third term — the sum over ways to finish the sequence — can also be defined recursively, this time moving over the trellis from right to left, which is known as the **backward recurrence**:

$$\beta_m(k) \triangleq \sum_{\boldsymbol{y}_{m:M}: Y_m = k} \prod_{n=m}^{M+1} \exp s_n(y_n, y_{n-1}) \quad [7.88]$$

$$= \sum_{k' \in \mathcal{Y}} \exp s_{m+1}(k', k) \sum_{\boldsymbol{y}_{m+1:M}: Y_m = k'} \prod_{n=m+1}^{M+1} \exp s_n(y_n, y_{n-1}) \quad [7.89]$$

$$= \sum_{k' \in \mathcal{Y}} \exp s_{m+1}(k', k) \times \beta_{m+1}(k'). \quad [7.90]$$

To understand this computation, compare with the forward recurrence in Equation 7.81.

In practice, numerical stability demands that we work in the log domain,

$$\log \alpha_m(k) = \log \sum_{k' \in \mathcal{Y}} \exp\left(\log s_m(k, k') + \log \alpha_{m-1}(k')\right) \quad [7.91]$$

$$\log \beta_{m-1}(k) = \log \sum_{k' \in \mathcal{Y}} \exp\left(\log s_m(k', k) + \log \beta_m(k')\right). \quad [7.92]$$

The application of the forward and backward probabilities is shown in Figure 7.3. Both the forward and backward recurrences operate on the trellis, which implies a space complexity $\mathcal{O}(MK)$. Because both recurrences require computing a sum over K terms at each node in the trellis, their time complexity is $\mathcal{O}(MK^2)$.

7.6 Neural sequence labeling

In neural network approaches to sequence labeling, we construct a vector representation for each tagging decision, based on the word and its context. Neural networks can perform tagging as a per-token classification decision, or they can be combined with the Viterbi algorithm to tag the entire sequence globally.

7.6.1 Recurrent neural networks

Recurrent neural networks (RNNs) were introduced in chapter 6 as a language modeling technique, in which the context at token m is summarized by a recurrently-updated vector,

$$\boldsymbol{h}_m = g(\boldsymbol{x}_m, \boldsymbol{h}_{m-1}), \quad m = 1, 2, \ldots M,$$

where $\boldsymbol{x}_m$ is the vector **embedding** of the token w_m and the function g defines the recurrence. The starting condition $\boldsymbol{h}_0$ is an additional parameter of the model. The long short-term memory (LSTM) is a more complex recurrence, in which a memory cell is through a series of gates, avoiding repeated application of the non-linearity. Despite these bells and whistles, both models share the basic architecture of recurrent updates across a sequence, and both will be referred to as RNNs here.

A straightforward application of RNNs to sequence labeling is to score each tag y_m as a linear function of $\boldsymbol{h}_m$:

$$\psi_m(y) = \boldsymbol{\beta}_y \cdot \boldsymbol{h}_m \quad [7.93]$$

$$\hat{y}_m = \operatorname*{argmax}_y \psi_m(y). \quad [7.94]$$

The score $\psi_m(y)$ can also be converted into a probability distribution using the usual softmax operation,

$$\mathrm{p}(y \mid \boldsymbol{w}_{1:m}) = \frac{\exp \psi_m(y)}{\sum_{y' \in \mathcal{Y}} \exp \psi_m(y')}. \quad [7.95]$$

Using this transformation, it is possible to train the tagger from the negative log-likelihood of the tags, as in a conditional random field. Alternatively, a hinge loss or margin loss objective can be constructed from the raw scores $\psi_m(y)$.

The hidden state $\boldsymbol{h}_m$ accounts for information in the input leading up to position m, but it ignores the subsequent tokens, which may also be relevant to the tag y_m. This can be addressed by adding a second RNN, in which the input is reversed, running the recurrence from w_M to w_1. This is known as a **bidirectional recurrent neural network** (Graves and Schmidhuber, 2005), and is specified as:

$$\overleftarrow{\boldsymbol{h}}_m = g(\boldsymbol{x}_m, \overleftarrow{\boldsymbol{h}}_{m+1}), \quad m = 1, 2, \ldots, M. \tag{7.96}$$

The hidden states of the left-to-right RNN are denoted $\overrightarrow{\boldsymbol{h}}_m$. The left-to-right and right-to-left vectors are concatenated, $\boldsymbol{h}_m = [\overleftarrow{\boldsymbol{h}}_m; \overrightarrow{\boldsymbol{h}}_m]$. The scoring function in Equation 7.93 is applied to this concatenated vector.

Bidirectional RNN tagging has several attractive properties. Ideally, the representation $\boldsymbol{h}_m$ summarizes the useful information from the surrounding context, so that it is not necessary to design explicit features to capture this information. If the vector $\boldsymbol{h}_m$ is an adequate summary of this context, then it may not even be necessary to perform the tagging jointly: in general, the gains offered by joint tagging of the entire sequence are diminished as the individual tagging model becomes more powerful. Using backpropagation, the word vectors $\boldsymbol{x}$ can be trained "end-to-end", so that they capture word properties that are useful for the tagging task. Alternatively, if limited labeled data is available, we can use word embeddings that are "pre-trained" from unlabeled data, using a language modeling objective (as in § 6.3) or a related word embedding technique (see chapter 14). It is even possible to combine both fine-tuned and pre-trained embeddings in a single model.

Neural structure prediction The bidirectional recurrent neural network incorporates information from throughout the input, but each tagging decision is made independently. In some sequence labeling applications, there are very strong dependencies between tags: it may even be impossible for one tag to follow another. In such scenarios, the tagging decision must be made jointly across the entire sequence.

Neural sequence labeling can be combined with the Viterbi algorithm by defining the local scores as:

$$s_m(y_m, y_{m-1}) = \boldsymbol{\beta}_{y_m} \cdot \boldsymbol{h}_m + \eta_{y_{m-1}, y_m}, \tag{7.97}$$

where $\boldsymbol{h}_m$ is the RNN hidden state, $\boldsymbol{\beta}_{y_m}$ is a vector associated with tag y_m, and η_{y_{m-1}, y_m} is a scalar parameter for the tag transition (y_{m-1}, y_m). These local scores can then be incorporated into the Viterbi algorithm for inference, and into the forward algorithm for training. This model is shown in Figure 7.4. It can be trained from the conditional log-likelihood objective defined in Equation 7.76, backpropagating to the tagging parameters

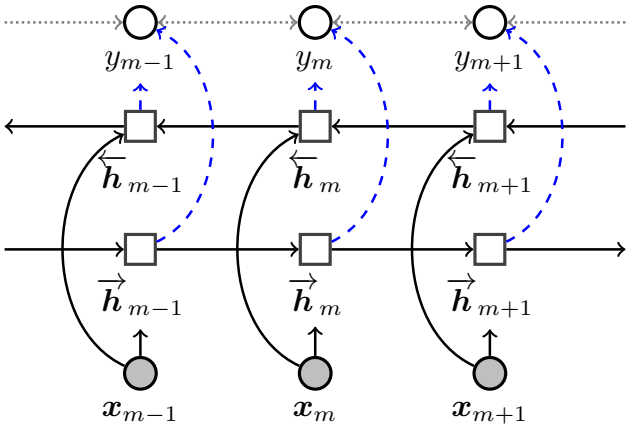

Figure 7.4: **Bidirectional LSTM** for sequence labeling. The solid lines indicate computation, the dashed lines indicate probabilistic dependency, and the dotted lines indicate the optional additional probabilistic dependencies between labels in the biLSTM-CRF.

$\boldsymbol{\beta}$ and $\boldsymbol{\eta}$, as well as the parameters of the RNN. This model is called the **LSTM-CRF**, due to its combination of aspects of the long short-term memory and conditional random field models (Huang et al., 2015).

The LSTM-CRF is especially effective on the task of **named entity recognition** (Lample et al., 2016), a sequence labeling task that is described in detail in § 8.3. This task has strong dependencies between adjacent tags, so structure prediction is especially important.

7.6.2 Character-level models

As in language modeling, rare and unseen words are a challenge: if we encounter a word that was not in the training data, then there is no obvious choice for the word embedding $\boldsymbol{x}_m$. One solution is to use a generic **unseen word** embedding for all such words. However, in many cases, properties of unseen words can be guessed from their spellings. For example, *whimsical* does not appear in the Universal Dependencies (UD) English Treebank, yet the suffix *-al* makes it likely to be adjective; by the same logic, *unflinchingly* is likely to be an adverb, and *barnacle* is likely to be a noun.

In feature-based models, these morphological properties were handled by suffix features; in a neural network, they can be incorporated by constructing the embeddings of unseen words from their spellings or morphology. One way to do this is to incorporate an additional layer of bidirectional RNNs, one for each word in the vocabulary (Ling et al., 2015). For each such character-RNN, the inputs are the characters, and the output is the concatenation of the final states of the left-facing and right-facing passes, $\boldsymbol{\phi}_w =$

$[\overrightarrow{\boldsymbol{h}}^{(w)}_{N_w}; \overleftarrow{\boldsymbol{h}}^{(w)}_{0}]$, where $\overrightarrow{\boldsymbol{h}}^{(w)}_{N_w}$ is the final state of the right-facing pass for word w, and N_w is the number of characters in the word. The character RNN model is trained by backpropagation from the tagging objective. On the test data, the trained RNN is applied to out-of-vocabulary words (or all words), yielding inputs to the word-level tagging RNN. Other approaches to compositional word embeddings are described in § 14.7.1.

7.6.3 Convolutional Neural Networks for Sequence Labeling

One disadvantage of recurrent neural networks is that the architecture requires iterating through the sequence of inputs and predictions: each hidden vector $\boldsymbol{h}_m$ must be computed from the previous hidden vector $\boldsymbol{h}_{m-1}$, before predicting the tag y_m. These iterative computations are difficult to parallelize, and fail to exploit the speedups offered by **graphics processing units (GPUs)** on operations such as matrix multiplication. **Convolutional neural networks** achieve better computational performance by predicting each label y_m from a set of matrix operations on the neighboring word embeddings, $\boldsymbol{x}_{m-k:m+k}$ (Collobert et al., 2011). Because there is no hidden state to update, the predictions for each y_m can be computed in parallel. For more on convolutional neural networks, see § 3.4. Character-based word embeddings can also be computed using convolutional neural networks (Santos and Zadrozny, 2014).

7.7 *Unsupervised sequence labeling

In unsupervised sequence labeling, the goal is to induce a hidden Markov model from a corpus of *unannotated* text $(\boldsymbol{w}^{(1)}, \boldsymbol{w}^{(2)}, \ldots, \boldsymbol{w}^{(N)})$, where each $\boldsymbol{w}^{(i)}$ is a sequence of length $M^{(i)}$. This is an example of the general problem of **structure induction**, which is the unsupervised version of structure prediction. The tags that result from unsupervised sequence labeling might be useful for some downstream task, or they might help us to better understand the language's inherent structure. For part-of-speech tagging, it is common to use a tag dictionary that lists the allowed tags for each word, simplifying the problem (Christodoulopoulos et al., 2010).

Unsupervised learning in hidden Markov models can be performed using the **Baum-Welch algorithm**, which combines the forward-backward algorithm (§ 7.5.3) with expectation-maximization (EM; § 5.1.2). In the M-step, the HMM parameters from expected counts:

$$\Pr(W = i \mid Y = k) = \phi_{k,i} = \frac{E[\text{count}(W = i, Y = k)]}{E[\text{count}(Y = k)]}$$
$$\Pr(Y_m = k \mid Y_{m-1} = k') = \lambda_{k',k} = \frac{E[\text{count}(Y_m = k, Y_{m-1} = k')]}{E[\text{count}(Y_{m-1} = k')]}$$

The expected counts are computed in the E-step, using the forward and backward recurrences. The local scores follow the usual definition for hidden Markov models,

$$s_m(k, k') = \log \mathrm{p}_E(w_m \mid Y_m = k; \boldsymbol{\phi}) + \log \mathrm{p}_T(Y_m = k \mid Y_{m-1} = k'; \lambda). \quad [7.98]$$

The expected transition counts for a single instance are,

$$E[\text{count}(Y_m = k, Y_{m-1} = k') \mid \boldsymbol{w}] = \sum_{m=1}^{M} \Pr(Y_{m-1} = k', Y_m = k \mid \boldsymbol{w}) \quad [7.99]$$

$$= \frac{\sum_{\boldsymbol{y}: Y_m = k, Y_{m-1} = k'} \prod_{n=1}^{M} \exp s_n(y_n, y_{n-1})}{\sum_{\boldsymbol{y}'} \prod_{n=1}^{M} \exp s_n(y'_n, y'_{n-1})}. \quad [7.100]$$

As described in § 7.5.3, these marginal probabilities can be computed from the forward-backward recurrence,

$$\Pr(Y_{m-1} = k', Y_m = k \mid \boldsymbol{w}) = \frac{\alpha_{m-1}(k') \times \exp s_m(k, k') \times \beta_m(k)}{\alpha_{M+1}(\blacklozenge)}. \quad [7.101]$$

In a hidden Markov model, each element of the forward-backward computation has a special interpretation:

$$\alpha_{m-1}(k') = \mathrm{p}(Y_{m-1} = k', \boldsymbol{w}_{1:m-1}) \quad [7.102]$$

$$\exp s_m(k, k') = \mathrm{p}(Y_m = k, w_m \mid Y_{m-1} = k') \quad [7.103]$$

$$\beta_m(k) = \mathrm{p}(\boldsymbol{w}_{m+1:M} \mid Y_m = k). \quad [7.104]$$

Applying the conditional independence assumptions of the hidden Markov model (defined in Algorithm 12), the product is equal to the joint probability of the tag bigram and the entire input,

$$\begin{aligned}\alpha_{m-1}(k') \times \exp s_m(k, k') \times \beta_m(k) =& \mathrm{p}(Y_{m-1} = k', \boldsymbol{w}_{1:m-1}) \\ & \times \mathrm{p}(Y_m = k, w_m \mid Y_{m-1} = k') \\ & \times \mathrm{p}(\boldsymbol{w}_{m+1:M} \mid Y_m = k) \\ =& \mathrm{p}(Y_{m-1} = k', Y_m = k, \boldsymbol{w}_{1:M}). \end{aligned} \quad [7.105]$$

Dividing by $\alpha_{M+1}(\blacklozenge) = \mathrm{p}(\boldsymbol{w}_{1:M})$ gives the desired probability,

$$\frac{\alpha_{m-1}(k') \times s_m(k, k') \times \beta_m(k)}{\alpha_{M+1}(\blacklozenge)} = \frac{\mathrm{p}(Y_{m-1} = k', Y_m = k, \boldsymbol{w}_{1:M})}{\mathrm{p}(\boldsymbol{w}_{1:M})} \quad [7.106]$$

$$= \Pr(Y_{m-1} = k', Y_m = k \mid \boldsymbol{w}_{1:M}). \quad [7.107]$$

The expected emission counts can be computed in a similar manner, using the product $\alpha_m(k) \times \beta_m(k)$.

7.7.1 Linear dynamical systems

The forward-backward algorithm can be viewed as Bayesian state estimation in a discrete state space. In a continuous state space, $\boldsymbol{y}_m \in \mathbb{R}^K$, the equivalent algorithm is the **Kalman smoother**. It also computes marginals $\text{p}(\boldsymbol{y}_m \mid \boldsymbol{x}_{1:M})$, using a similar two-step algorithm of forward and backward passes. Instead of computing a trellis of values at each step, the Kalman smoother computes a probability density function $q_{\boldsymbol{y}_m}(\boldsymbol{y}_m; \boldsymbol{\mu}_m, \Sigma_m)$, characterized by a mean $\boldsymbol{\mu}_m$ and a covariance Σ_m around the latent state. Connections between the Kalman smoother and the forward-backward algorithm are elucidated by Minka (1999) and Murphy (2012).

7.7.2 Alternative unsupervised learning methods

As noted in § 5.5, expectation-maximization is just one of many techniques for structure induction. One alternative is to use **Markov Chain Monte Carlo (MCMC)** sampling algorithms, which are briefly described in § 5.5.1. For the specific case of sequence labeling, Gibbs sampling can be applied by iteratively sampling each tag y_m conditioned on all the others (Finkel et al., 2005):

$$\text{p}(y_m \mid \boldsymbol{y}_{-m}, \boldsymbol{w}_{1:M}) \propto \text{p}(w_m \mid y_m)\text{p}(y_m \mid \boldsymbol{y}_{-m}). \quad [7.108]$$

Gibbs Sampling has been applied to unsupervised part-of-speech tagging by Goldwater and Griffiths (2007). **Beam sampling** is a more sophisticated sampling algorithm, which randomly draws entire sequences $\boldsymbol{y}_{1:M}$, rather than individual tags y_m; this algorithm was applied to unsupervised part-of-speech tagging by Van Gael et al. (2009). Spectral learning (see § 5.5.2) can also be applied to sequence labeling. By factoring matrices of co-occurrence counts of word bigrams and trigrams (Song et al., 2010; Hsu et al., 2012), it is possible to obtain globally optimal estimates of the transition and emission parameters, under mild assumptions.

7.7.3 Semiring notation and the generalized viterbi algorithm

The Viterbi and Forward recurrences can each be performed over probabilities or log probabilities, yielding a total of four closely related recurrences. These four recurrence scan in fact be expressed as a single recurrence in a more general notation, known as **semiring algebra**. Let the symbols $\oplus$ and $\otimes$ represent generalized addition and multiplication respectively.[11] Given these operators, a generalized Viterbi recurrence is denoted,

$$v_m(k) = \bigoplus_{k' \in \mathcal{Y}} s_m(k, k') \otimes v_{m-1}(k'). \quad [7.109]$$

[11] In a semiring, the addition and multiplication operators must both obey associativity, and multiplication must distribute across addition; the addition operator must be commutative; there must be additive and multiplicative identities $\overline{0}$ and $\overline{1}$, such that $a \oplus \overline{0} = a$ and $a \otimes \overline{1} = a$; and there must be a multiplicative annihilator $\overline{0}$, such that $a \otimes \overline{0} = \overline{0}$.

Each recurrence that we have seen so far is a special case of this generalized Viterbi recurrence:

- In the max-product Viterbi recurrence over probabilities, the $\oplus$ operation corresponds to maximization, and the $\otimes$ operation corresponds to multiplication.
- In the forward recurrence over probabilities, the $\oplus$ operation corresponds to addition, and the $\otimes$ operation corresponds to multiplication.
- In the max-product Viterbi recurrence over log-probabilities, the $\oplus$ operation corresponds to maximization, and the $\otimes$ operation corresponds to addition.[12]
- In the forward recurrence over log-probabilities, the $\oplus$ operation corresponds to log-addition, $a \oplus b = \log(e^a + e^b)$. The $\otimes$ operation corresponds to addition.

The mathematical abstraction offered by semiring notation can be applied to the software implementations of these algorithms, yielding concise and modular implementations. For example, in the OpenFST library, generic operations are parametrized by the choice of semiring (Allauzen et al., 2007).

Exercises

1. Extend the example in § 7.3.1 to the sentence *they can can fish*, meaning that "they can put fish into cans." Build the trellis for this example using the weights in Table 7.1, and identify the best-scoring tag sequence. If the scores for noun and verb are tied, then you may assume that the backpointer always goes to noun.

2. Using the tagset $\mathcal{Y} = \{N, V\}$, and the feature set $\boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) = \{(w_m, y_m), (y_m, y_{m-1})\}$, show that there is no set of weights that give the correct tagging for both *they can fish* (N V V) and *they can can fish* (N V V N).

3. Work out what happens if you train a structured perceptron on the two examples mentioned in the previous problem, using the transition and emission features (y_m, y_{m-1}) and (y_m, w_m). Initialize all weights at 0, and assume that the Viterbi algorithm always chooses N when the scores for the two tags are tied, so that the initial prediction for *they can fish* is N N N.

4. Consider the garden path sentence, *The old man the boat.* Given word-tag and tag-tag features, what inequality in the weights must hold for the correct tag sequence to outscore the garden path tag sequence for this example?

[12] This is sometimes called the **tropical semiring**, in honor of the Brazilian mathematician Imre Simon.

5. Using the weights in Table 7.1, explicitly compute the log-probabilities for all possible taggings of the input *fish can*. Verify that the forward algorithm recovers the aggregate log probability.

6. Sketch out an algorithm for a variant of Viterbi that returns the top-n label sequences. What is the time and space complexity of this algorithm?

7. Show how to compute the marginal probability $\Pr(y_{m-2} = k, y_m = k' \mid \boldsymbol{w}_{1:M})$, in terms of the forward and backward variables, and the potentials $s_n(y_n, y_{n-1})$.

8. Suppose you receive a stream of text, where some of tokens have been replaced at random with *NOISE*. For example:

   - Source: *I try all things, I achieve what I can*
   - Message received: *I try NOISE NOISE, I NOISE what I NOISE*

   Assume you have access to a pre-trained bigram language model, which gives probabilities $\text{p}(w_m \mid w_{m-1})$. These probabilities can be assumed to be non-zero for all bigrams.

   Show how to use the Viterbi algorithm to recover the source by maximizing the bigram language model log-probability. Specifically, set the scores $s_m(y_m, y_{m-1})$ so that the Viterbi algorithm selects a sequence of words that maximizes the bigram language model log-probability, while leaving the non-noise tokens intact. Your solution should not modify the logic of the Viterbi algorithm, it should only set the scores $s_m(y_m, y_{m-1})$.

9. Let $\alpha(\cdot)$ and $\beta(\cdot)$ indicate the forward and backward variables as defined in § 7.5.3. Prove that $\alpha_{M+1}(\blacklozenge) = \beta_0(\lozenge) = \sum_y \alpha_m(y)\beta_m(y), \forall m \in \{1, 2, \ldots, M\}$.

10. Consider an RNN tagging model with a $\tanh$ activation function on the hidden layer, and a hinge loss on the output. (The problem also works for the margin loss and negative log-likelihood.) Suppose you initialize all parameters to zero: this includes the word embeddings that make up $\boldsymbol{x}$, the transition matrix Θ, the output weights $\boldsymbol{\beta}$, and the initial hidden state $\boldsymbol{h}_0$.

    a) Prove that for any data and for any gradient-based learning algorithm, all parameters will be stuck at zero.

    b) Would a sigmoid activation function avoid this problem?