

601.465/665 — Natural Language Processing

Homework 6: Tagging

Prof. Jason Eisner — Fall 2021
Due date: Tuesday 23 November, 11 pm

In this homework, you will build taggers based on a Hidden Markov Model (HMM) and then on a Conditional Random Field (CRF). Moreover, your CRF tagger will extract useful context features using a bidirectional recurrent neural network (biRNN).

Your code could be used for many kinds of tagging, but you'll evaluate it on the traditional task of tagging a word sequence with a part-of-speech sequence.

This is not the first time you'll be writing code to predict structures using dynamic programming. But last time, the grammar weights were *given* to you. This time you'll have to *train* the weights that you'll use to predict structures. For training, you'll reuse the backprop and SGD techniques that you previously used to train a language model for predicting the next word.

Homework goals: This homework ties together most of the ideas from the course: linguistic modeling, structured prediction, dynamic programming, deep embeddings of structures, and training objectives.

After completing it, you should be comfortable with modern structured prediction methods, including different training objectives (supervised and semi-supervised; generative and discriminative), different modeling techniques including neural nets, and different decoders.

As a result, you should be able to see how to apply the same methods to other kinds of structured prediction (PCFG trees and FST paths).

Collaboration: *You may work in pairs on this homework.* That is, if you choose, you may collaborate with one partner from the class, handing in a single homework with both your names on it. However:

- (a) You should do all the work *together*, for example by pair programming. Don't divide it up into "my part" and "your part."
- (b) Your `README` file should describe at the top what each of you contributed, so that we know you shared the work fairly.
- (c) Your partner for homework 6 can't be the same as your partner from homework 4 (parsing).

In any case, observe **academic integrity** and never claim any work by third parties as your own.

Materials: We provide spreadsheets, some of which were shown in class. You should use Python for this assignment, since we'll be using PyTorch, and we provide Python starter code. The materials can be found in <http://www.cs.jhu.edu/~jason/465/hw-tag/>. Data for the assignment are described in reading section [H](#).

Reading: First read the long handout attached to the end of this homework!

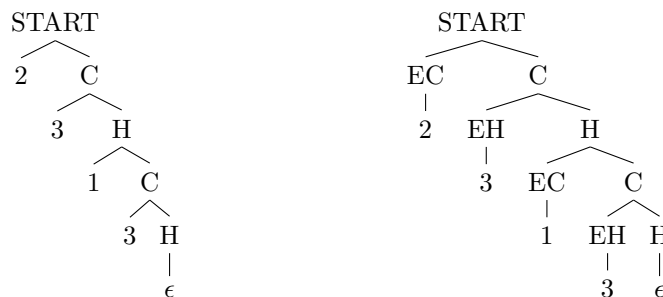
1 Understanding HMMs and the Forward-Backward Algorithm

The forward-backward algorithm is given in Algorithm 4 of the reading handout, and we encourage you to play with the spreadsheet that we used in class.

1

Play as much as you like, but here are some questions to answer in your README:

- (a) Reload the spreadsheet to go back to the default settings. Now, change the first day to have just 1 ice cream.
 - i. What is the new probability (in the initial reconstruction) that day 1 is hot? Explain, by considering the probability of HHH versus CHH as explanations of the first three days of data, 133.
 - ii. How much does this change affect the probability that day 2 is hot? That is, what is that probability before vs. after the change to the day 1 data? What cell in the spreadsheet holds this probability?
 - iii. How does this change affect the final graph after 10 iterations of reestimation? In particular, what is $p(H)$ on days 1 and 2? (*Hint*: Look at where the other 1-ice-cream days fall during the summer.)
- (b) We talked about stereotypes in class. Suppose you bring a very strong bias to interpreting the data: you believe that I *never* eat only 1 ice cream on a hot day. So, again reload the spreadsheet, and set $p(1 | H) = 0$, $p(2 | H) = 0.3$, $p(3 | H) = 0.7$.
 - i. How does this change affect the initial reconstruction of the weather (the leftmost graph)?
 - ii. What does the final graph look like after 10 iterations of reestimation?
 - iii. What is $p(1 | H)$ after 10 iterations? Explain carefully why this is, discussing what happens at each reestimation step, in terms of the 2^{33} paths through the trellis.
- (c) The backward algorithm (which computes all the β probabilities) is exactly analogous to the inside algorithm. Recall that the inside algorithm finds the probability of a sentence by summing over all possible parses. The backward algorithm finds the probability of a sentence by summing over all possible taggings that could have generated that sentence.
 - i. Let's make that precise. Each state (node) in the trellis has an β probability. *Which* state's β probability equals the total probability of the sentence?
 - ii. It is actually possible to regard the backward algorithm as a special case of the inside algorithm! In other words, there is a particular grammar whose parses correspond to taggings of the sentence. One parse of the sequence 2313 would look like the tree on the left:



In the tree on the left, what is the meaning of an H constituent? What is the probability of the rule $H \rightarrow 1 C$? How about the probability of $H \rightarrow \epsilon$? An equivalent approach uses a grammar that instead produces the slightly more complicated parse on the right; why might one prefer that approach?

2 Building an HMM

Write a bigram Viterbi tagger called `tag.py` that can be run on the English data in the `data/` subdirectory. Follow the design described in the reading handout, with the emission probabilities parameterized in terms of pretrained word embeddings that are read from a lexicon file (see reading section C.1). You can use a [lexicon from Homework 3](#).

Starter code and detailed INSTRUCTIONS are given in the `code/` subdirectory. These walk you through a simple, reasonable path for building up your tagger, partly using some pieces we've implemented for you, and checking your work against the ice cream spreadsheet from class.

Hand in answers to the following questions:

- Why does Algorithm 1 initialize $\alpha_{\text{BOS}}(0)$ and $\beta_{\text{EOS}}(n)$ to 1?
- If you train on a **sup** file and then evaluate on a held-out **raw** file, you'll get lower perplexity than if you evaluate on a held-out **dev** file. Why is that? Which perplexity do you think is more important and why?
- V includes the word types from **sup** and **raw** (plus OOV). Why not from **dev** as well?
- The accuracy on known words wasn't stellar. What about the design of the model (and perhaps how it differed from what we discussed in class) limited the performance on known words? Why might we still prefer this modeling decision?
- Did the iterations of semi-supervised training help or hurt overall tagging accuracy? How about tagging accuracy on known, seen, and novel words (respectively)?
- Explain in a few clear sentences why you think the semi-supervised approach helped where it did. How did it get additional value out of the **enraw** file?
- Suggest at least two reasons to explain why the semi-supervised approach didn't always help.
- Experiment with different training strategies for using **enraw**. For example, you could train on **enraw** alone, or a combined corpus of **ensup+enraw**, or **ensup+ensup+ensup+enraw** (so that the supervised data is weighted more heavily). And you could do staged training where you do additional training on **ensup** before or after this. What seems to work? Why?

3 Open-Ended Improvements

Before handing in your tagger, implement an `--awesome` flag that makes it actually do a good job at tagging. With this flag turned on, it should work better than the baseline method described in reading section I.3.

Again, instructions are given in the INSTRUCTIONS file.

You only need one good improvement. Your improvement should demonstrate intellectual engagement with the problem of how to improve your tagging accuracy or speed. It doesn't have to be complicated, but a one-line change such as switching lexicons or optimization algorithms is not enough by itself. More successful or interesting solutions will get more credit.






A good starting point is to study the output on the English data. As a previous question noted, our basic tagging architecture seems to get a lot of known words wrong. Think about why it might incorrectly tag a certain word as a determiner, even though that word only appeared as a noun in training data.

With this in mind, here are some things you might try. Some of them are based on the fact that in natural language, the emission matrix B is sparse. Most words have only a few possible tags, and some tags (the “closed-class” tags) can emit only a small set of words. Thus, many entries in B are zero (or at least very small).

- Use constraints on inference, so that a word that is known from supervised training data can only be tagged with one of its supervised tags.¹
- Use better features that go beyond word embeddings. These might look at the spelling of the word, or its behavior in training data. Maybe the word **actuary** is a noun because it ends in **ry**, or because it has often been tagged as a noun in training data, or because **actuaries** appears in training data as a noun.
- Come up with some way to bias HMM training towards a sparse transition matrix.
- Use a higher-order HMM, e.g., trigram transition probabilities instead of bigrams.

Your tagger might still fall short of the state-of-the-art 97% for English tagging, even though the reduced tagset in Figure 1 of the reading means that you're solving an easier problem than the state-of-the-art taggers. Why? Because you only have 100,000 words of training data (together with embeddings).

What to hand in:

-  3 • Submit the source code for this version of `tag.py` (together with all other source files needed to make it run). It should be able to run both with and without the `--awesome` flag.
-  4 • Upload the trained model you'd like us to use on the **test** data. You need to upload `ic_hmm.pkl`, `en_hmm.pkl`, `en_hmm_raw.pkl`, `en_hmm_awesome.pkl`².
-  5 • Describe what you did.
-  6 • Include your `endev.output` file. Unless you tell us otherwise, we will assume that you trained your system using `tag.py` as shown in the **INSTRUCTIONS**, and tested it as shown on **endev**, but using the `--awesome` flag for both training and testing.
-  7 • Do a simple ablation experiment of your choice and report the result.

¹In other words, set τ_j to a restricted set of tags that depends on w_j . This can be made reasonably fast by precomputing a multi-hot “tag dictionary” vector for each word type. $\alpha(j)$ can be multiplied by w_j 's tag dictionary vector to zero out the tags that are not allowed for w_j .

²For the supervised training on toy example, supervised training on english, semi-supervised training on english, and awesome training on english.

In this homework, the leaderboard will probably show performance on **endev**. This development data is being used to help develop everyone’s systems.

For actual grading, however, we will evaluate your code on test data **entest** that you have never seen (as in Homework 3). The autograder will run your code to both train and test your model. It will compare the actual **output** generated by your tagger to the gold-standard tags on the **entest** data.

As a warning, too much experimentation with the dev data may lead to overfitting. Thus, you might consider splitting the dev data into two parts, holding some of it out to check that your final system actually works on a new dataset.

4 Conditional Random Field Tagger

Finally, add a CRF tagger to your codebase, again following the **INSTRUCTIONS** file. It should begin as a fairly straightforward modification of your HMM tagger, to support discriminative training (maximum *conditional* likelihood). After that, you’ll add more useful, contextual features to create what’s called a biRNN-CRF.

This is a fairly modern architecture. It’s described in reading section **G**—you’ll want to carefully examine reading section **G.3** on how to parameterize the potential functions. Some concrete guidance is given in the **INSTRUCTIONS** file.

8 Turn in your updated **tag.py** program, which now has both HMM tagger and CRF tagger functionalities. (HMM is the default, but `--crf` switches to CRF.)

9 (a) Compare the CRF to the HMM. Does it get better accuracy on **endev** when each is trained on **ensup**? Are there any patterns in the errors that one makes and the other doesn’t?

10 (b) What happens when you include **enraw** in the training data for your CRF?

11 (c) Did your biRNN-CRF do better than the basic CRF? That is, did adding the RNN features improve your tagging accuracy?

12 For extra credit, you are welcome to try to improve performance further, by changing the way the CRF scores the transitions and emissions. Such CRF improvements should be activated by the `--awesome` flag, just like your HMM improvements in question **3**. For example, your scoring function could use the biRNN features differently (see footnote **17** in the reading handout), or it could use hand-crafted features. It’s fine for the `--awesome` flag to activate many of the same changes in the HMM and the CRF. Tell us about what you did, why you expected it to help the CRF, and whether it actually helped.

13 Please upload the trained model you’d like us to use on the **test** data. You need to upload `ic_crf.pkl`, `en_crf.pkl`, `en_crf_raw.pkl`, `en_crf_birnn.pkl`³.

5 Final Question

14 What is the maximum amount of ice cream you have ever eaten in one day? Why? Did you get sick?

³For the supervised training on toy example, supervised training on english, semi-supervised training on english, and birnn model on english with whichever training is better.

601.465/665 — Natural Language Processing

Reading for Homework 6: Tagging

Prof. Jason Eisner — Fall 2021

We don't have a required textbook for this course. Instead, handouts like this one are the main readings. This handout accompanies homework 6, which refers to it.

A HMM Basics

A.1 Notation

In this reading handout, I'll use the following notation for hidden Markov models (HMMs). You might want to use the same notation in your program.

- A sentence is a word sequence w_1, \dots, w_n . We also define $w_0 = \text{BOSW}$ and $w_{n+1} = \text{EOSW}$. The w is for 'word'.
- The corresponding tags are t_1, \dots, t_n . We also define $t_0 = \text{BOS}$ and $t_{n+1} = \text{EOS}$.
- We sometimes write the i^{th} tagged word as w_i/t_i , for example in the data files.
- I'll use "A" to indicate tag-to-tag *transition* probabilities, as in $p_A(t_i | t_{i-1})$.
- I'll use "B" to indicate tag-to-word *emission* probabilities, as in $p_B(w_i | t_i)$.
- It is sometimes convenient to put these probabilities into matrices. Suppose we have integerized the tag types. Then let the **transition matrix** A be a square matrix defined by $A_{st} = p_A(t | s)$. Notice that row s sums to 1: it gives a probability distribution over the tags t that might follow tag s .
- Suppose we have also integerized the word types. Then let the **emission matrix** B be a wide rectangular matrix defined by $B_{tw} = p_A(w | t)$, so row t gives a probability distribution over the words w that might be emitted by tag t . Usually this $k \times V$ matrix is wider than it is tall, since the vocabulary (V words) is much larger than the tagset (k tags).

A.2 Probability Model

Now, given a tagged sentence

$$w_1/t_1 \ w_2/t_2 \ \dots \ w_n/t_n,$$

the probability of generating it (according to the hidden Markov model) is

$$p(\mathbf{t}, \mathbf{w}) = \left(\prod_{i=1}^{n+1} p_A(t_i | t_{i-1}) \right) \cdot \left(\prod_{i=1}^n p_B(w_i | t_i) \right) \quad (1)$$

Algorithm 1 The forward algorithm. \mathbf{w} is the input sentence, and $\boldsymbol{\tau}$ is a sequence of sets of tags. For each $j \in [1, n]$, τ_j denotes the set of possible tags for w_j . The algorithm returns $\sum_{\mathbf{t}: (\forall j) t_j \in \tau_j} p(\mathbf{t}, \mathbf{w})$. In the standard case, each τ_j is the set of all k tags, and then the algorithm returns $p(\mathbf{w})$ (equation (3)). But it can also be made to return $Z = p(\mathbf{t}, \mathbf{w})$ (equation (1)) by taking each $\tau_j = \{t_j\}$. In general, it returns the likelihood when we observe the words \mathbf{w} and we also observe the fact that each t_j falls in τ_j ; this sums over all tagged sentences that are consistent with those observations.

```

1: function FORWARD( $\mathbf{w}, \boldsymbol{\tau}$ )
2:    $n = |\mathbf{w}|$ ;  $w_{n+1} \leftarrow \text{EOSW}$ ;  $\tau_{n+1} \leftarrow \{\text{EOS}\}$   $\triangleright$  add EOSW to input
3:    $\triangleright$  all  $\alpha$  values are initially 0
4:    $\alpha_{\text{BOS}}(0) \leftarrow 1$   $\triangleright$  “start” node of the graph
5:   for  $j \leftarrow 1$  to  $n + 1$  :
6:      $\triangleright$  note: the loops below could be replaced by tensor operations (see equation (12))
7:     for  $t_j \in \tau_j$  :
8:       for  $t_{j-1} \in \tau_{j-1}$  :
9:          $\triangleright$  find total prob of prefix paths ending in  $t_{j-1}t_j$ ; add it to total prob of prefix paths ending in  $t_j$ 
10:         $p \leftarrow p_A(t_j | t_{j-1}) \cdot p_B(w_j | t_j)$   $\triangleright$  probability of the  $t_{j-1} \rightarrow t_j$  arc
11:         $\alpha_{t_j}(j) += \alpha_{t_{j-1}}(j-1) \cdot p$   $\triangleright$  extend prefix paths starting with  $t_{j-1}$ ; keep a running total
12:    $Z \leftarrow \alpha_{\text{EOS}}(n+1)$   $\triangleright$  total prob of all complete paths (from BOS,0 to EOS, $n+1$ )
13:   return  $Z$ 

```

We can also write this as a product of $2n + 1$ matrix entries,

$$p(\mathbf{t}, \mathbf{w}) = \left(\prod_{i=1}^{n+1} A_{t_{i-1}, t_i} \right) \cdot \left(\prod_{i=1}^n B_{t_i, w_i} \right) \quad (2)$$

where the subscripts are the integers that represent these tags and words.

The first product is the prior probability of generating the tag sequence under a bigram model, and the second product is the probability of generating the observed words once we have chosen the tags. We don’t have to generate $t_0 = \text{BOS}$ because it is given, just as ROOT was given in homework 1 and BOSW was given in homework 3. However, just as in homework 3, we do have to generate EOS , in order to know when the sentence ends: $n \geq 0$ is the first natural number such that $t_{n+1} = \text{EOS}$.

In the above equations, we omitted the trivial factors $p_B(w_0 | t_0) = p_B(\text{BOSW} | \text{BOS}) = 1$ and $p_B(w_{n+1} | t_{n+1}) = p_B(\text{EOSW} | \text{EOS}) = 1$. However, our pseudocode will include the latter factor, for convenience.

A.3 Marginal Probabilities

The probability of generating the words \mathbf{w} —marginalizing out the tag sequence that produced those words—is

$$p(\mathbf{w}) = \sum_{\mathbf{t}} p(\mathbf{t}, \mathbf{w}) = \sum_{t_1} \cdots \sum_{t_n} \left(\prod_{i=1}^{n+1} p_A(t_i | t_{i-1}) \right) \cdot \left(\prod_{i=1}^n p_B(w_i | t_i) \right) \quad (3)$$

where we have observed $t_0 = \text{BOS}$ and $t_{n+1} = \text{EOS}$ but have to sum over all possible taggings for the n positions in between. With k possible tags at each position, there are k^n possible taggings

to sum over! Yet as we saw in lecture, this grand summation can be computed in $O(nk^2)$ time by the **forward algorithm** (Algorithm 1). Basically, the trick is that we can express (3) as

$$p(\mathbf{w}) = \alpha_{\text{EOS}}(n + 1) \quad (4)$$

if we first define $\alpha_{t_j}(j)$ for any time step $j \in [1, n + 1]$ and any possible tag t_j at that time step:

$$\alpha_{t_j}(j) \stackrel{\text{def}}{=} \sum_{t_1} \cdots \sum_{t_{j-1}} \left(\prod_{i=1}^j p_A(t_i | t_{i-1}) \right) \cdot \left(\prod_{i=1}^j p_B(w_i | t_i) \right) \quad (5)$$

This represents the total probability of all prefix tag sequences for w_1, \dots, w_j that end with t_j . It can be rearranged and written as an efficient recurrence:

$$\alpha_{t_j}(j) = \sum_{t_{j-1}} \left(\sum_{t_1} \cdots \sum_{t_{j-2}} \left(\prod_{i=1}^{j-1} p_A(t_i | t_{i-1}) \right) \cdot \left(\prod_{i=1}^{j-1} p_B(w_i | t_i) \right) \right) \cdot p_A(t_j | t_{j-1}) \cdot p_B(w_j | t_j) \quad (6)$$

$$= \sum_{t_{j-1}} \alpha_{t_{j-1}}(j-1) \cdot p_A(t_j | t_{j-1}) \cdot p_B(w_j | t_j) \quad (7)$$

where to make the $j = 1$ case work out, we must take as the base case

$$\alpha_{\text{BOS}}(0) = 1 \quad \alpha_{t_0}(0) = 0 \text{ if } t_0 \neq \text{BOS} \quad (8)$$

A.4 Matrix Formulas

For efficiency, let's rewrite equation (7) to use vector and matrix operations. First, we switch to using the parameter matrices A and B . (We also rename t_{j-1} and t_j to s and t for simplicity.)

$$\alpha_t(j) = \sum_s \alpha_s(j-1) \cdot A_{st} \cdot B_{tw_j} \quad (9)$$

Since the B term doesn't depend on s , we can write this as

$$\alpha_t(j) = \left(\sum_s \alpha_s(j-1) \cdot A_{st} \right) B_{tw_j} \quad (10)$$

If we regard $\vec{\alpha}(j)$ and $\vec{\alpha}(j-1)$ as row vectors indexed by tags, this is

$$\vec{\alpha}_t(j) = (\vec{\alpha}(j-1) \cdot A)_t \cdot B_{tw_j} \quad (11)$$

or, if we want to express and compute this for all t at once,

$$\vec{\alpha}(j) = (\vec{\alpha}(j-1) \cdot A) \odot B_{\cdot w_j} \quad (12)$$

where $B_{\cdot w_j}$ denotes column w_j of B , and the \odot operator performs elementwise multiplication of two vectors (that is, if $\vec{c} = \vec{a} \odot \vec{b}$, then $c_t = a_t \cdot b_t$ for each t).¹

Algorithm 2 The Viterbi algorithm, which finds $\operatorname{argmax}_{\mathbf{t}: (\forall j) t_j \in \tau_j} p(\mathbf{t}, \mathbf{w})$. Compare Algorithm 1.

```

1: function VITERBI( $\mathbf{w}$ ,  $\tau$ )
2:    $n = |\mathbf{w}|$ ;  $w_{n+1} \leftarrow \text{EOSW}$ ;  $\tau_{n+1} \leftarrow \{\text{EOS}\}$  ▷ add EOSW to input
3:   ▷ all  $\hat{\alpha}$  values are initially 0 (or  $-\infty$ ), and all backpointers are initially None
4:    $\hat{\alpha}_{\text{BOS}}(0) \leftarrow 1$  ▷ “start” node of the graph
5:   for  $j \leftarrow 1$  to  $n + 1$  :
6:     for  $t_j \in \tau_j$  :
7:       for  $t_{j-1} \in \tau_{j-1}$  :
8:          $p \leftarrow p_A(t_j | t_{j-1}) \cdot p_B(w_j | t_j)$  ▷ probability of the  $t_{j-1} \rightarrow t_j$  arc
9:         if  $\hat{\alpha}_{t_j}(j) < \hat{\alpha}_{t_{j-1}}(j-1) \cdot p$  : ▷ keep a running maximum
10:           $\hat{\alpha}_{t_j}(j) \leftarrow \hat{\alpha}_{t_{j-1}}(j-1) \cdot p$  ▷ increase the maximum
11:           $\text{backpointer}_{t_j}(j) \leftarrow t_{j-1}$  ▷ and remember who provided the new maximum
12:   ▷ follow backpointers to find the best tag sequence that ends at the final state (EOS at time  $n + 1$ )
13:    $t_{n+1} \leftarrow \text{EOS}$ 
14:   for  $j \leftarrow n + 1$  downto 1 :
15:      $t_{j-1} \leftarrow \text{backpointer}_{t_j}(j)$ 
16:   return  $\mathbf{t}$ 

```

A.5 The Viterbi Algorithm

Algorithm 2 famously finds the most probable tagging of \mathbf{w} . Where the forward algorithm uses the semiring $(\oplus, \otimes) = (+, \times)$ to find the *total* probability of all taggings, the Viterbi algorithm uses $(\oplus, \otimes) = (\max, \times)$ to find the *maximum* probability of all taggings. To avoid confusion, we refer to this as $\hat{\alpha}$ rather than α . It then follows backpointers to extract the *argmax*—that is, the tagging that achieves this maximum.

If you want to use tensor operations instead of nested loops to compute the $\hat{\alpha}$ probabilities at position j , try to express equation (12) in PyTorch where you express matrix multiplication without using the @ operator, instead using $*$ and sum . Then for Viterbi, replace sum with max (and figure out how to use argmax for the backpointers). *Hint*: The $*$ operation is **broadcastable**.

¹Can’t we somehow just express equation (9) as a matrix multiplication? Yes, if we first define a new matrix. For any word type w , define the **observable operator** $P(w)$ to be the square matrix whose entries are given by $P(w)_{st} = A_{st}B_{tw}$. This is basically a copy of A , but where each column t has been reweighted by tag t ’s probability of emitting word w . In the “trellis graph” whose paths correspond to possible taggings, the $k \times k$ matrix P gives the weights of the k^2 arcs from time $j - 1$ to time j . Now we can rewrite equation (9) as $\vec{\alpha}(j) = \vec{\alpha}(j - 1)P(w_j)$. In other words, when we observe the word token w_j , we update the $\vec{\alpha}$ vector by multiplying it by w_j ’s observable operator—that is, by a linear transformation.

As the forward algorithm is basically just a sequence of $n + 1$ such updates, we get we get $p(\mathbf{w}) = (\vec{\alpha}(0)P(w_1) \cdot P(w_2) \cdots P(w_n)P(w_{n+1}))_{\text{EOS}}$, or less efficiently, $p(\mathbf{w}) = (P(w_1) \cdot P(w_2) \cdots P(w_n)P(w_{n+1}))_{\text{BOS, EOS}}$, which represents the total weight of all paths of length $n + 1$ from BOS to EOS in the graph.

However, that’s probably not the best way to implement it unless you have space to precompute and store the matrices $P(w)$ for all w . The reason is that each time you construct a copy of $P(w_j)$ from A and B , you have to scale each column of A . Fewer multiplications are used by equation (11), which computes the row vector $\vec{\alpha}(j - 1) \cdot A$ first and then scales each element of that row vector (faster than scaling a whole column).

B Training HMMs²

B.1 Maximum Likelihood Estimation

Let’s model sentences as being generated independently of one another—all by the same HMM with parameters A, B . Then the probability of generating an observed sequence of sentences $\mathbf{w}_1, \mathbf{w}_2, \dots$ is given by $\prod_m p(\mathbf{w}_m)$. This is known as the **likelihood** of the parameters A, B (given this dataset).

To estimate the parameters of a model,³ a pretty good estimation procedure (with a lot of theoretical justification) is to maximize their likelihood, or equivalently their log-likelihood:

$$\mathcal{L}(A, B) \stackrel{\text{def}}{=} \sum_m \log p(\mathbf{w}_m) \quad (13)$$

But it’s wise to add a regularization term to this objective, to resist overfitting to the training data. The scale of this regularization term is, as usual, a hyperparameter C that you can tune on dev data.

$$\operatorname{argmax}_{A, B} \mathcal{L}(A, B) - C \cdot R(A, B) \quad (14)$$

In statistics, the likelihood of the parameters is just the probability of generating *all the observed data* from those parameters.⁴ So when you observe more (or less) data, it affects the definition of $\mathcal{L}(A, B)$. Examples:

- If you have more sentences in your training corpus, then equation (13) will have more summands.
- If for a particular observed sentence \mathbf{w}_m you also observe the tags \mathbf{t}_m , then you should replace $p(\mathbf{w}_m)$ with $p(\mathbf{t}_m, \mathbf{w}_m)$ in equation (13). That is, use equation (1) instead of equation (3). The probability $p(\mathbf{t}_m, \mathbf{w}_m)$ can be regarded as still summing over all the tag sequences that are *consistent with the observations*: but now we have enough observations that there is only one such tag sequence, so $p(\mathbf{t}_m, \mathbf{w}_m)$ is the only summand.
- As an advanced case, if you observe \mathbf{w}_m and *part* of \mathbf{t}_m , then you should modify equation (3) to marginalize over only the unobserved tags. For example, if you observe that $t_3 = \text{NOUN}$, then just take $t_3 = \text{NOUN}$ in the formula and remove the summation \sum_{t_3} over all possible values of t_3 . Algorithm 1 actually handles this case.

B.2 Discussion: Supervised, Unsupervised, and Semi-Supervised Learning

For the **tagging task**, we are interested in learning the mapping $\mathbf{w} \mapsto \mathbf{t}$ from a sentence to its tagging. (This is just an example of the common machine learning problem of learning an $x \mapsto y$ mapping.) But what kind of data do we learn from?

²The training methods discussed in this section are much more general than just HMMs. Very similar methods apply to PCFGs, for example. They also apply to lots of other models. One nice thing about HMMs (and PCFGs) is that the probability Z of generating the observed data can be computed efficiently, by the forward (or inside) algorithm. But the methods can be used even for models where we only have slow or approximate methods for computing Z .

³An alternative is to “go full Bayesian” and get a posterior distribution over the parameters, rather than just picking one value for the parameters.

⁴Here’s a [quick explanation](#) of what “likelihood” means in general.

Supervised learning. If we observe the tags for all sentences, then we have a *supervised* learning problem. In that case, you already know how to estimate the probabilities $p_A(t | t_{\text{prev}})$ and $p_B(w | t)$.

Specifically, if you have a small number of probabilities to estimate, then you can just use the relative frequencies of the different transitions and emissions. This turns out to be equivalent to maximizing the log-likelihood.

If you have a larger vocabulary, then you should probably take steps to avoid overfitting. You could smooth the counts, or maximize the *regularized* log-likelihood, or use a model of $p_B(w | t)$ that has fewer parameters (see reading section C below).

Unsupervised learning. The name “hidden Markov model” is properly used for the case where we don’t observe the tags. Although the tags are governed by a Markov model (bigram model), they are *hidden* (or *unobserved* or *missing*). The log-likelihood in equation (13) is sometimes called the *incomplete-data log-likelihood* for this reason.

There are different reasons we might want to estimate an HMM. If you’re using the HMM for language modeling, then the tags t are *latent* random variables—unseen forces that we posit to help explain the observed data w . In that case, we don’t actually care about which tags are used, only that they are helpful in modeling $p(w)$.

But if we are actually going to evaluate on the tagging task, then this is an *unsupervised* learning problem. And it’s hard! We want to predict the correct tags despite never having seen any tags in training data.

Semi-supervised learning. Seeing even *some* of the tags makes the unsupervised problem easier. It is then called semi-supervised. For one thing, we might observe in the tagged sentences that **the** usually gets tagged as **Det** and **caviar** usually gets tagged as **Noun**, rather than vice-versa. (The pure unsupervised learning problem has no information to break this tie—swapping the names **Det** and **Noun** through the model would not change its likelihood function.)

Even a semi-supervised learner never observes **caviar/Noun** in the tagged sentences, it might be able to guess this tag based on observing **...the caviar with ...** in an untagged sentence. It knows from the tagged sentences that **the/Det** and **with/Prep** are good guesses for those frequent words—and then it will want to guess **caviar/Noun** in between, because it knows from other tagged sentences that **Det Noun** and **Noun Prep** are common bigrams.

B.3 SGD

When some or all tags are unobserved, the likelihood function can have multiple local optima. The global optimum is hard to find. But we can at least try to improve the likelihood. That’s what you’ll do in the homework, using PyTorch’s **backward** method to compute the gradient of log-likelihood.⁵

The most obvious approach to (locally) maximizing $\mathcal{L}(A, B)$ is simply to use stochastic gradient ascent (SGD). In other words, sample a sentence from the training corpus, and adjust A and B to

⁵Make sure your log-likelihood function is computed using PyTorch operations. For example, to construct a PyTorch tensor that represents $\log p$, be careful to write `torch.log(p)`, or the more concise version `p.log()`, where `p` is already a PyTorch tensor. Don’t write `math.log(p)` or `numpy.log(p)`, because those aren’t PyTorch operations. They will just return an ordinary scalar that doesn’t remember its dependence on `p`, so the **backward** method will not see that it needs to propagate gradient information back to `Z`. That will screw up the training of your model.

improve its log-probability. If the sentence is tagged, its probability is given by equation (1); if it's untagged, by equation (3).

The only difficulty is that we need to ensure that the rows of A and B are probability distributions! The entries of each row have to be non-negative and sum to 1. An SGD update to A or B might violate these constraints.

Do we have to abandon SGD and switch to a **constrained optimization** algorithm? No: we can still use SGD if we change our setup slightly. Instead of taking the transition and emission probabilities to be parameters of the model, we can define them in terms of parameter matrices W^A and W^B , which have the same shape as A and B respectively. Each row of A (or B) is defined to be the result of passing the corresponding row of W^A (or W^B) through a softmax function. This is called a **softmax parameterization**. In other words,

$$p_A(t | s) = \frac{\exp W_{st}^A}{\sum_{t'} \exp W_{st'}^A} \quad p_B(w | t) = \frac{\exp W_{tw}^B}{\sum_{w'} \exp W_{tw'}^B} \quad (15)$$

Each of these formulas is just a very simple conditional log-linear model, with one feature for each tag-tag bigram or tag-word pair. The weight of this feature is a single matrix element.

The point is that the elements of W^A and W^B can be arbitrary real numbers, and we'll still get well-defined conditional probability distributions. So we don't have to worry that an SGD update to W^A and W^B will give us invalid parameters.

In reading section C below, we'll look at some fancier options for taking this a little further, by defining W^A and W^B in terms of further real-valued parameters.

Unfortunately, you have to recompute the probability matrices A and B each time you update the parameters. Even with tensor operations, this is a bit expensive: in particular, B is a large matrix if the vocabulary is large. So you probably don't want to update the parameters after every sentence. Instead, it's probably wise for each SGD update to be the total gradient from a "minibatch" of several sentences. Then you can recompute A and B less often.

Maximizing likelihood by SGD will be the primary method for this homework. However, the following subsections draw a connection to the alternative method that you saw in class, namely maximizing likelihood by EM.

B.4 Monte Carlo Expectation Maximization (MCEM)

The incomplete-data likelihood *sums* over all possible taggings for each sentence. A variant approach is to *impute* the sentence's tagging, i.e., guess what it might be. This gives us a "supervised" version of the sentence, and we can train on that by increasing the *complete*-data likelihood. In other words, we use imputation to convert the unsupervised learning problem into a supervised learning problem!

How does this work? We run SGD as before. But once we've chosen a sentence \mathbf{w}_m from the training corpus, we also randomly sample a *guess* $\hat{\mathbf{t}}_m$ of its tag sequence. The guess $\hat{\mathbf{t}}_m$ should be sampled from the posterior distribution $p(\mathbf{t} | \mathbf{w}_m)$ defined by the *current* model parameters. In other words, we will probably sample a tagging that the model thinks has a high chance of being correct.

If we pretend that $\hat{\mathbf{t}}_m$ really is correct, then $(\hat{\mathbf{t}}_m, \mathbf{w}_m)$ is a *supervised* version of the sentence. We can then take a gradient step to increase the log of its probability $p(\hat{\mathbf{t}}_m, \mathbf{w}_m)$ as defined by equation (1). This is a simple formula that does not have to sum over all possible tag sequences.

Algorithm 3 Forward algorithm with backward sampling. This is a variant of the forward algorithm (Algorithm 1) that keeps *randomized* backpointers, and then follows them just as in the Viterbi algorithm (Algorithm 2).

```

1: function RANDOM-TAGGING( $\mathbf{w}$ ,  $\boldsymbol{\tau}$ )
2:    $n = |\mathbf{w}|$ ;  $w_{n+1} \leftarrow \text{EOSW}$ ;  $\tau_{n+1} \leftarrow \{\text{EOS}\}$  ▷ add EOSW to input
3:   ▷ all  $\alpha$  values are initially 0
4:    $\alpha_{\text{BOS}}(0) \leftarrow 1$  ▷ “start” node of the graph
5:   for  $j \leftarrow 1$  to  $n + 1$  :
6:     for  $t_j \in \tau_j$  :
7:       for  $t_{j-1} \in \tau_{j-1}$  :
8:          $p \leftarrow p_A(t_j | t_{j-1}) \cdot p_B(w_j | t_j)$  ▷ probability of the  $t_{j-1} \rightarrow t_j$  arc
9:          $\alpha_{t_j}(j) += \alpha_{t_{j-1}}(j-1) \cdot p$ 
10:        with probability of  $(\alpha_{t_{j-1}}(j-1) \cdot p) / \alpha_{t_j}(j)$  ▷ keep a running sample
11:          backpointer $_{t_j}(j) \leftarrow t_{j-1}$ 
12:   ▷ follow backpointers to extract a random tag sequence that ends at the final state (EOS at time  $n + 1$ )
13:    $t_{n+1} \leftarrow \text{EOS}$ 
14:   for  $j \leftarrow n + 1$  downto 1 :
15:      $t_{j-1} \leftarrow \text{backpointer}_{t_j}(j)$ 
16:   return  $\mathbf{t}$ 

```

This property is useful in advanced machine learning models where summation is computationally expensive, but which do have a more efficient procedure for sampling the latent variable \mathbf{t} from the posterior $p(\mathbf{t} | \mathbf{w}_m)$, either exactly or approximately.

Even though we are taking a gradient step and it is stochastic, this is not actually an SGD procedure for maximizing equation (13), because it is *not* true that $\nabla \log p(\hat{\mathbf{t}}_m, \mathbf{w}_m)$ is $\nabla \log p(\mathbf{w}_m)$ on average. Even so, it does turn out to be a *correct* procedure for (locally) maximizing equation (13). It is known as Monte Carlo EM.⁶ Intuitively, we are using the current parameters to make the guesses, which help us improve the parameters, which mean that future steps will have better guesses, and so on.

But what is the sampling procedure for HMMs? That is, how do we draw a random tagging from the posterior distribution $p(\mathbf{t} | \mathbf{w}_m)$? A “backward sampling” algorithm is shown in Algorithm 3. It is sort of a randomized version of the Viterbi algorithm—instead of following the *best* backpointers to find the *most probable* tag sequence, it follows *random* backpointers to find a *random* tag sequence. Of course, the tag sequences aren’t all equally probable! If most of the paths to **Noun** at token 8 come through **Det** at token 7, then most of $\alpha_{\text{Noun}}(8)$ will be contributed by $\alpha_{\text{Det}}(7)$, and backpointer $_{\text{Noun}}(8)$ will probably be **Det**. However, to figure out which paths contribute most, we first have to do all the work of the forward algorithm. So in the case of HMMs, sampling is not in fact more efficient than summing. Therefore, we won’t ask you to use Monte Carlo EM. It’s smarter to compute the entire sum and improve it by following its gradient (reading section B.3).

⁶“Monte Carlo” algorithms are algorithms that use randomness. The name was originally a military code name, referring to a casino in Monaco—casinos also use randomness.

B.5 Expectation Maximization (EM)

The procedure shown on the spreadsheet in class was the classical EM procedure, which is also based on summation. The EM procedure can be understood as “MCEM without randomness.” We’ll average over all ways of guessing the tag sequence, instead of randomly picking one as in the previous section.

So instead of increasing $\log p(\hat{\mathbf{t}}_m, \mathbf{w}_m)$ for a *particular* (random) tag sequence $\hat{\mathbf{t}}_m$, EM increases the expected value of that quantity, namely $\sum_{\mathbf{t}} p(\mathbf{t} | \mathbf{w}_m) \cdot \log p(\mathbf{t}, \mathbf{w}_m)$.

Traditional EM doesn’t even have SGD’s randomness of choosing a different example \mathbf{w}_m at each step. Rather, the **M step** is a “batch” procedure that tries to maximize the expected log-likelihood of the whole training corpus, i.e.,

$$\sum_m \sum_{\mathbf{t}} p(\mathbf{t} | \mathbf{w}_m) \cdot \log p(\mathbf{t}, \mathbf{w}_m) \quad (16)$$

This looks for parameters A, B that have high log-probabilities $\log p(\mathbf{t}, \mathbf{w}_m)$ for the “currently” probable taggings \mathbf{t} . The current coefficients $p(\mathbf{t} | \mathbf{w}_m)$ are held fixed during the M step even though we are adjusting our parameters. Those coefficients were used in the **E step** and correspond to our current guesses.

Once we’ve maximized equation (16), we use these new parameters to make new guesses—another E step—and then do another M step, and so on. Again, this turns out to be a valid procedure for (locally) maximizing equation (13).⁷

The key reason that EM is efficient is that for an HMM, we can compute equation (16) efficiently.⁸ We don’t have to iterate over all k^n possible taggings for a sentence of length n . It turns out that to compute equation (16), or its gradient, *we only need the total fractional counts of the transitions and emissions*.⁹ So that is all that the E step computes. For a given sentence, the forward-backward algorithm of Algorithm 4 computes the fractional counts in $O(nk^2)$ time via dynamic programming. We compute the total counts by running Algorithm 4 on all the sentences.¹⁰

One way of thinking about the M step is that it is just like training on a supervised corpus, where each untagged training sentence \mathbf{w}_m has given rise to lots of tagged training sentences $(\mathbf{t}, \mathbf{w}_m)$, which are all “fractionally” in the corpus. Our old supervised training methods work. Specifically, you can maximize equation (16) by setting the transition and emission probabilities as ratios of

⁷Why? It can be shown (via something called Jensen’s inequality) that equation (16) is always \leq equation (13). So by finding parameters that make the lower bound (16) high, we can ensure that the log-likelihood (13) is also high. Furthermore, it can be shown that when EM converges—meaning that two successive E steps return the same distribution $p(\mathbf{t} | \mathbf{w}_m)$ —the two quantities are equal and both are at a local maximum. Thus, EM locally maximizes equation (13). Monte Carlo EM (reading section B.4) is essentially the stochastic version of this procedure.

⁸For fancier models where this is not true, a useful generalization is **variational EM**. Namely, when there is no efficient way to work with $p(\mathbf{t} | \mathbf{w}_m)$ in equation (16), we can use a more tractable distribution $q_m(\mathbf{t})$ over the taggings. No matter what distribution q_m we pick, we’ll still get a lower bound on the log-likelihood (still by Jensen’s inequality). So the variational E step tries to increase the lower bound $\sum_m \sum_{\mathbf{t}} q_m(\mathbf{t}) \cdot \log p(\mathbf{t}, \mathbf{w}_m)$ by reestimating the distributions q_m (while keeping them tractable), and the M step tries to increase it by reestimating the model p .

⁹Can you prove this? Hint: start by using equation (1) to expand the $\log p$ term in equation (16).

¹⁰How about a matrix version? Footnote 1 noted that the forward algorithm amounted to computing a sequence of row vectors via $\vec{\alpha}(j) = \vec{\alpha}(j-1)P(w_j)$, where the base case $\vec{\alpha}(0)$ is the row vector that is one-hot at BOS. In the same way, the backward algorithm amounts to computing a sequence of column vectors via $\vec{\beta}(j-1) = P(w_j)\vec{\beta}(j)$, where the base case $\vec{\beta}(n+1)$ is the column vector that is one-hot at EOS. We obtain Z as $\alpha_{\text{EOS}}(n+1)$ or alternatively as $Z = \beta_{\text{BOS}}(0)$. The posterior distribution over the tag T_j is given by the vector $\vec{\alpha}(j) \odot \vec{\beta}(j) / Z$.

Algorithm 4 Forward-backward algorithm for computing emission counts $c(t, w)$ and expected transition counts $c(s, t)$. This algorithm increases the existing counts $c(\dots)$ with the counts from this sentence.

```

1: function FORWARD-BACKWARD( $\mathbf{w}, \boldsymbol{\tau}$ )
2:   run Algorithm 1 to compute the  $\alpha$  values as well as  $n, w_{n+1}, \tau_{n+1}$ , and  $Z$ 
3:    $\triangleright$  now for the backward pass; all  $\beta$  values are initially 0
4:    $\beta_{\text{EOS}}(n+1) \leftarrow 1$ 
5:   for  $j \leftarrow n+1$  downto 1 :
6:     for  $t_j \in \tau_j$  :
7:        $c(t_j, w_j) += \alpha_{t_j}(j) \cdot \beta_{t_j}(j) / Z$   $\triangleright$  increment count by  $p(T_j = t_j | \mathbf{w}, \boldsymbol{\tau})$ 
8:       for  $t_{j-1} \in \tau_{j-1}$  :
9:          $\triangleright$  find total prob of all suffix paths starting with  $t_{j-1}t_j$ ,
           and add it to total prob of suffix paths starting with  $t_{j-1}$ 
10:         $p \leftarrow p_A(t_j | t_{j-1}) \cdot p_B(w_j | t_j)$   $\triangleright$  probability of the  $t_{j-1} \rightarrow t_j$  arc (same as in forward pass)
11:         $c(t_{j-1}, t_j) += \alpha_{t_{j-1}}(j-1) \cdot p \cdot \beta_{t_j}(j) / Z$   $\triangleright$  increment by  $p(T_{j-1} = t_{j-1}, T_j = t_j | \mathbf{w}, \boldsymbol{\tau})$ 
12:         $\beta_{t_{j-1}}(j-1) += p \cdot \beta_{t_j}(j)$   $\triangleright$  extend suffix paths starting with  $t_j$ ; keep a running total

```

fractional counts. Or you can maximize it by gradient ascent.¹¹ If the probabilities are obtained from a log-linear model (such as equation (15)), the gradient is the usual gradient—the observed counts minus the expected counts, where we take the “observed” counts to be the fractional counts we imputed (since we didn’t actually observe anything).

C Parameterization

C.1 Word Embeddings

So far, we have had a separate parameter for every emission probability $p(w | t)$: namely B_{tw} or the softmax parameter W_{tw}^B (equation (15)). However, with a large vocabulary, this is a lot of parameters. For words that only appear once or twice in the training data, these parameters may take many epochs to converge to their final values.

One way to address this would be to initialize the parameters to something reasonable – for example, get an initial estimate of $p(w | t)$ from supervised data by counting and smoothing, and take the log of this estimate to get W_{tw}^B .

But perhaps a better way to reduce the number of parameters is to share parameters among similar words. How do we know which words are similar? We can use pretrained word embeddings! Let $\vec{w} \in \mathbb{R}^D$ be the embedding of word type w . This contains lots of information about w from some large unsupervised corpus. Hopefully, it contains enough information to guess which tags will tend to emit w .

For each tag t , we can learn a weight vector $\vec{\theta}_t^B$. Now we can define

$$W_{tw}^B = \vec{\theta}_t^B \cdot \vec{w} \tag{17}$$

¹¹If you’re following a gradient at the M step, you don’t need to go all the way up to a maximum. It’s enough to just improve equation (16)—that procedure is known as **generalized EM**.

so the matrix W^B can be computed “all at once” by

$$W^B = \Theta^B E^\top \tag{18}$$

where Θ^B is a matrix whose rows are the d -dimensional weight vectors for the different tags, and E is a matrix whose rows are the d -dimensional embeddings for the different words. (W^B can then be converted to B by softmaxing each row.)

Notice that equation (15) still gives a log-linear model of $p(w | t)$, with kd features in total. Each tag t has a separate set of d features, which fire on (t, w) with values corresponding to the dimensions of \vec{w} .

When the vocabulary is small, as in the ice cream example, you could just take E to be the identity matrix. In this case, the word-embeddings are one-hot vectors. Now we have $W^B = \Theta^B$, so row t of W^B can be adjusted freely: it just represents an unconstrained softmax parameterization of $p(w | t)$.

Fine-tuning the word embeddings. Once the model is trained and the Θ (and W^A) parameters are working well, you can optionally adjust the E parameters as well during subsequent SGD steps. This is the “fine-tuning” stage. It does risk overfitting to the training data and forgetting the pretrained information. The simplest remedy is to monitor your evaluation metric on development data, and stop fine-tuning when that metric starts going down.

C.2 Word Frequencies

There is one potential problem here, namely that the embeddings may not contain information about word frequency. $p(w | t)$ might be just as high for a rare word as for a frequent word, if those words have similar embeddings! Fortunately, this shouldn’t be a problem, for us, because if $p(w | t)$ is 50 times too high for *all* tags, then this just increases the log-likelihood by a constant, $\log 50$, but doesn’t change the relative probabilities of the taggings. Still, you can fix the problem by defining

$$W_{tw}^B = \vec{\theta}_t \cdot \vec{w} + \log \hat{p}(w) \tag{19}$$

where $\hat{p}(w)$ is estimated by counting words in the training data. This means that $p(w | t)$ will be defined as proportional to $\hat{p}(w) \cdot \exp(\vec{\theta}_t \cdot \vec{w})$. Since Bayes’ Theorem says that $p(w | t) \propto p(w) \cdot p(t | w)$, this means that really we’re using $\exp(\vec{\theta}_t \cdot \vec{w})$ to model $p(t | w)$, up to a constant. That seems like a reasonable thing to do.

Even better, you can concatenate $\log \hat{p}(w)$ onto the vector \vec{w} as an additional dimension. If training $\vec{\theta}_t$ learns a weight of γ for this dimension, then this means it defines $p(w | t)$ as proportional to $\hat{p}(w)^\gamma \cdot \exp(\vec{\theta}_t \cdot \vec{w})$. We might expect that it would learn $\gamma \approx 1$ (and this might be a good choice for initialization), but since the other dimensions of \vec{w} already do reflect the frequency of w to some extent, then perhaps it will learn $\gamma < 1$.

In the homework, you’ll have the option to take this even farther, and augment \vec{w} with information about the frequency with which w had various tags in supervised training data.

C.3 Tag Embeddings

Our tag set is small enough that it’s reasonable to directly learn the $k \times k$ matrix W^A . However, if k were large, then it might also be worth reducing the number of transition parameters by learning

low-dimensional embeddings of the tags. For example, you could define $W_{st}^A = \theta_s^{\text{left}} \cdot \theta_t^{\text{right}}$. [Chiu & Rush \(2020\)](#) show that HMMs are actually surprisingly good language models—competitive with neural models—if you use this trick to give them a huge number of states!

C.4 Don't Guess When You Know

“Don't guess when you know” is a good general principle. If there are things that *can't* or *must* happen, then you may as well ensure that their probabilities in your model are 0 or 1, respectively. You shouldn't use data to estimate smoothed values of these probabilities in $(0, 1)$ when you *know* that the true values are 0 or 1.

It's not the end of the world if you estimate 0.00001 for something that should be 0, but it's good software design to nail down that 0, unless this special case makes the code slow or ugly.

In particular, we know about p_B :

- $p_B(\text{BOSW} \mid \text{BOS}) = 1$
- $p_B(\text{BOSW} \mid t) = 0$ for $t \neq \text{BOS}$
- $p_B(\text{EOSW} \mid \text{EOS}) = 1$
- $p_B(\text{EOSW} \mid t) = 0$ for $t \neq \text{EOS}$

To avoid all of these special cases, let's just say that B only includes the emission probabilities $p_B(w_j \mid t_j)$ for $j \in [1, n]$. In fact, equation (1) only considers those emission probabilities. (It's true that Algorithm 1 line 10 does look up $p(\text{EOSW} \mid \text{EOS})$, but that factor can be skipped or forced to 1 by a simple “if” test. There's no need to actually look it up in B .)

Then we don't need rows in the B matrix for BOS and EOS. And we don't need columns for BOSW and EOSW—we can actually leave them out of the vocabulary. Then each row of B is a probability distribution over the vocabulary and cannot give any probability to BOSW or EOSW. Omitting the BOSW and EOSW columns makes it easy to produce each row as a softmax (equation (15)). It also means we don't have to worry about the fact that BOSW and EOSW don't have embeddings in the lexicon.

You can physically omit the rows in B for BOS and EOS if you add them last to the tag integerizer; then a matrix with $k - 2$ rows will have rows indexed by the other tags. Alternatively, it's okay if those tags have rows in the matrix, but they should never be accessed. Set them to 0, or to NaN so that you get an error if they are accidentally used.

We also know about p_A :

- $p_A(\text{BOS} \mid s) = 0$ for any preceding tag s
- $p_A(t \mid \text{EOS})$ doesn't matter (will not be used)

So it would be nice to omit the BOS column and the EOS row from the A matrix.

The problem is that $\vec{\alpha}(j)$ should have k elements—one for each tag including BOS (nonzero for $j = 0$) and EOS (nonzero for $j = n + 1$). So if we're going to multiply by A to transform that vector (equation (12)), we need A to be $k \times k$.

The best option is probably to force the BOS column to be 0. For example, if you set the BOS column of W^A to $-\infty$, then exponentiating it will give a column of zeroes in A . Then you can set the EOS row to be NaN.

(In Python, use `math.nan` and `-math.inf`, or `float("nan")` and `float("-inf")`.)

D Numerical Issues

For long sentences or large vocabularies, the α and Z probabilities for a sentence will be quite small. You'll need a method for preventing underflow. Here are a few options.¹²

D.1 The Scaling Trick

A simple, efficient, and effective option is to just scale the probabilities up when they get small. Recall our update rule (equation (12)):

$$\vec{\alpha}(j) \leftarrow (\vec{\alpha}(j-1) \cdot A) \odot B_{\cdot w_j} \quad (20)$$

Suppose after we compute this, we divide it by a small constant κ_j :

$$\vec{\alpha}(j) \leftarrow \vec{\alpha}(j) / \kappa_j \quad (21)$$

Now Z in Algorithm 1 will be off by a factor of $\prod_j \kappa_j$. But you can easily multiply that back in at the end of the algorithm (line 12).

$$Z \leftarrow \alpha_{\text{EOS}}(n+1) \cdot \prod_j \kappa_j \quad (22)$$

except that to avoid underflow, you should switch to the log domain:

$$\log Z \leftarrow \log \alpha_{\text{EOS}}(n+1) + \sum_j \log \kappa_j \quad (23)$$

So you just have to keep a running total $\sum_j \log \kappa_j$ as you go. Since this is just adding a constant to $\log Z$, it doesn't even affect the gradients in reading section B.3 or equation (27). The only reason to keep this running total is to report $\log Z$ accurately so that you can watch this log-likelihood improve over time.

You could choose κ_j to be any positive number. An elegant choice is $\kappa_j = \sum_t \alpha_t(j)$ because then the rescaled $\vec{\alpha}(j)$ then gives the relative probabilities of the tags (i.e., they sum to 1). Another reasonable choice would be $\kappa_j = \max_t \alpha_t(j)$, in which case the rescaled $\vec{\alpha}(j)$ will have a max value of 1.

A couple of fancy improvements you could consider:

- At steps j where this sum or max is not very small, you could optionally just skip the rescaling step (equivalent to taking $\kappa_j = 1$). This means you only have to rescale every so often, when the probabilities are in danger of underflowing.
- You could ensure that κ_j is a power of 2, which means that there are especially fast ways to multiply $\vec{\alpha}(j)$ by κ_j . (This is probably not worth it, unless you're writing low-level code, but it connects nicely to the discussion in reading section D.3 below.)

¹²These methods can also help prevent *overflow*. Overflow could arise during exponentiation in equation (15) or equations (40)–(45) if you have allowed the weights to get too large. However, if the weights are really large enough to cause overflow, then it probably means you've taken too large a gradient step or you haven't regularized enough.

Whatever value of κ_j you choose, it’s just a bookkeeping convenience to prevent underflow. For our purposes, it’s just some constant that you picked. You should make it a scalar rather than a tensor, so that PyTorch will not waste time back-propagating through it.

Note that this trick works for tagging specifically. How would you make it work for parsing? (The most obvious approach is wrong. I can see a solution, but it took a little thought!) The next solution is more general, but also more annoying.

D.2 Log Probabilities

A common general trick for avoiding underflow is to store your probabilities “in the log domain.” That is, when you see something like $p = q \cdot r$, implement it as something like $lp = lq + lr$, where lp, lq, lr are variables storing the logs of the probabilities. Note that $\log 0 = -\infty$ (which is a valid floating-point number).

Addition in the log domain. The forward algorithm requires you to add probabilities, as in $p \leftarrow p + q$. But you are now storing these probabilities p and q as their logs, lp and lq .

You might try to write $lp \leftarrow \log(\exp lp + \exp lq)$, but the \exp operation will probably underflow and return 0—that is why you are using logs in the first place!

Instead, the function `logaddexp(lp, lq)` is available in numpy and PyTorch. It safely does the computation above. That is, it adds two values that are represented in the log domain. (If you want to sum a whole sequence of values, use `logsumexp`.) How does it work?

$$\text{logaddexp}(x, y) \stackrel{\text{def}}{=} \begin{cases} x + \log(1 + \exp(y - x)) & \text{if } y \leq x \\ y + \log(1 + \exp(x - y)) & \text{otherwise} \end{cases} \quad (24)$$

You can check for yourself that this equals $\log(\exp x + \exp y)$; that the \exp can’t overflow (because its argument is always ≤ 0); and that you get an appropriate answer even if the \exp underflows.

The sub-expression $\log(1 + z)$ can be computed more quickly and accurately by the specialized function `log1p(z) = z - z2/2 + z3/3 - ...` (Taylor series), which is usually available in the math library of your programming language (or see http://www.johndcook.com/cpp_log_one_plus_x.html). This avoids ever computing $1 + z$, which would lose most of z ’s significant digits for small z .

Make sure to handle the special case where $p = 0$ or $q = 0$ (see above).

Matrix operations. Recall that equation (12) involves a product of the row vector $\vec{\alpha}(j - 1)$ with the square matrix A . But if the inputs are represented in the log domain, you can’t use the ordinary matrix product operator `@`. You need a version that uses `+` and `logsumexp` (in place of the usual `*` and `sum`). You’ll have to figure out how to do this efficiently with PyTorch tensor operations. Just as in reading section A.5, a good first step is to try to implement `@` using only `*` and `sum`.

Since A and B are now represented as matrices lA and lB whose entries are in the log domain, the softmax formulas (15) become

$$\log p_A(t | s) = W_{st}^A - \text{logsumexp}_{t'} W_{st'}^A \quad \log p_B(w | t) = W_{tw}^B - \text{logsumexp}_{w'} W_{tw'}^B \quad (25)$$

In other words, each row of lA is obtained by shifting the corresponding row of W^A by its `logsumexp`, and similarly for lB .

What kind of logs? To simplify your code and avoid bugs, this section has recommended that you use log-probabilities rather than negative log-probabilities. Then you won't have to remember to negate the output to log or the input to exp. (The convention of negating log-probabilities helps to keep minus signs out of the *printed numbers*; but when you're coding, it's safer to keep minus signs out of the *formulas and code* instead.)

Similarly, I recommend that you use natural logarithms (\log_e) because they are simpler than \log_2 , slightly faster, less prone to programming mistakes, and supported by library functions like `logsumexp`.

Yes, it's conventional to *report* $-\log_2$ probabilities, (the unit here is "bits"). But you can store $\log_e x$ internally, and convert to bits only when and if you print it out: $-\log_2 x = -(\log_e x) / \log_e 2$. (As it happens, you won't be required to print any log-probabilities for this homework, only perplexities: see equation (49).)

Problem with log 0. You may find that your gradient contains partial derivatives of `nan`. NaN or `nan` stands for "not a number"; this special floating-point value is used to represent an indeterminate quantity (see <https://en.wikipedia.org/wiki/NaN>).

The issue arises from the behavior of the `logsumexp` or `logaddexp` operator when all its arguments are $-\infty$. (That case arises when you are summing up the paths to a tag that is impossible—such as BOS at a position $j > 0$. Each of these paths has a log-alpha value of $\log 0 = -\infty$.)

I've reported this problem at <https://github.com/pytorch/pytorch/issues/49724>, with a detailed explanation of why it happens. Hopefully it will get fixed. If you are still running into it, just use either of the following workarounds:

- I built an improved version of `logsumexp` that treats $-\infty$ properly during backprop. To get it, do `import logsumexp_safe`. This will redefine the `logsumexp` and `logaddexp` operations so that they accept a new keyword argument `safe_inf=True`, which makes the `nan` values turn into 0 when appropriate.

The same problem will arise if you ever backpropagate through `log(softmax(...))`. In particular, be careful not to take the log of the *A* and *B* matrices. You can revise `updateAB` in `hmm.py` so that instead of computing attributes *A* and *B* using `softmax` (corresponding to equation (15)), it computes attributes *lA* and *lB* using `log_softmax` (corresponding to equation (25)).

- Alternatively, represent a 0 probability by `log 1e-45` instead of `log 0`, where `1e-45` is a number very close to 0. This is a hack, but it's a simple way to avoid the problem.

D.3 High Precision Arithmetic

A final possibility is to use a type of number that won't underflow. 32-bit floating point numbers can get as small as 10^{-38} , and 64-bit floating point numbers can get as small as 10^{-308} . But there are Python libraries for high-precision arithmetic, such as `mpmath`, `bigfloat`, and `mxNumber`. These use more than 64 bits to represent a number—indeed, they can expand the size of the representation as necessary.

Recall that a floating point number stores a fixed-point mantissa together with an integer exponent which is basically the \log_2 of the number (rounded towards zero). So to avoid underflow, really what we want is to have a lot of bits in the exponent.

The previous alternatives can be actually thought of specialized floating-point representations¹³ that focus on the exponent:

- In the scaling trick of reading section D.1, the running total $\sum_{i=1}^j \log \kappa_i$ can be regarded as a possibly large, possibly non-integer exponent that is shared by all of the values in $\vec{\alpha}(j)$.
- The log-domain trick of reading section D.2 can be thought of as a numeric representation where the mantissa is 1 and the exponent is itself a floating point number (allowing it to be very large). Indeed, `logaddexp` is actually rather similar to floating-point addition.¹⁴

Using the general high-precision arithmetic libraries is probably too slow and unnecessarily precise, and won't work with PyTorch.

E Posterior Decoding

Decoding refers to extracting a prediction from the model. Recall that a Viterbi decoder (reading section A.5) finds the single most likely overall sequence. By contrast, a posterior decoder will separately choose the best tag at each position—the tag with highest posterior marginal probability—even if this gives an unlikely overall sequence.

In Algorithm 4, the probability that tag T_j has value t_j is computed at line 7 (which adds it to $c(t_j, w_j)$). The posterior marginal distribution over all possible tags at position j can be found efficiently as $\vec{\alpha}(j) \odot \vec{\beta}(j) / Z$. See also reading section F for some ways to compute this without implementing Algorithm 4 (the backward pass).

Here's an example of how posterior decoding works (repeated from the HMM slides in class). Suppose you have a 2-word string, and the HMM assigns positive probability to three different tag sequences, as shown at the left of this table:

prob	actual sequence		score if predicted sequence is ...				
			N V	Det Adj	Det N	Det V	...
0.45	N	V	2	0	0	1	...
0.35	Det	Adj	0	2	1	1	...
0.2	Det	N	0	1	2	1	...
expected score			0.9	0.9	0.75	1.0	...

The Viterbi decoder will return N V because that's the most probable tag sequence. However, the HMM itself says that this has only a 45% chance of being correct. There are two other possible answers, as shown by the rows of the table, so N V might be totally wrong.

So is N V a good output for our system? Suppose we will be evaluated by the number of correct tags in the output. The N V column shows how many tags we might get right if we output N V: we

¹³Indeed, the previous alternatives could be packaged up as numeric classes. For example, you could make a `PosNum` class that stores any positive number p as $\log p$ (and perhaps stores 0 as $-\infty$), which enables it to represent very small probabilities. You'd then implement arithmetic operations such as `*`, `+`, and `max`. You'd also need a constructor that turns a nonnegative real into a `PosNum` (using `log`), and a method for extracting the real value of a `PosNum` (using `exp`).

¹⁴Before adding two numbers, floating-point addition has to multiply the mantissa of the smaller number by 2^{y-x} , where $y-x \leq 0$ is the difference between the integer exponents. This is similar to equation (24). But in the floating-point case, multiplying the mantissa by 2^{y-x} —which is an integer power of 2—is easily accomplished by right-shifting the bits of its binary representation.

have a 45% chance of getting 2 tags right, but a 55% chance of getting 0 tags right, so *on average* we expect to get only 0.9 tags right. The **Det Adj** or **Det N** columns show how many tags we’d expect to get right if we predicted those sequences instead.

It’s not hard to see that with this evaluation setup, the best way to maximize our score is to separately predict the most likely tag at every position. We predict $t_1 = \mathbf{Det}$ because that has a 0.55 chance of being right, so it adds 0.55 to the expected score. And we predict $t_2 = \mathbf{V}$ because that has an 0.45 chance of being right, so it adds 0.45—more than if we had chosen **Adj** or **N**.

Thus, our best output is **Det V**, where *on average* we expect to get 1.0 tags right. This is not the highest-probability output—in fact it has probability 0 of being correct, according to the HMM! (That’s why there’s no **Det V** row in the table.) It’s just a *good compromise* that is likely to get a pretty good score. It can never achieve the maximum score of 2 (only the three rows in the table can do that), but it also is never completely wrong with a score of 0.

F Back-Propagation As An Alternative to the Backward Pass

If you want fractional counts, either to use in EM or for some other purpose such as posterior decoding, it’s not actually necessary to implement the backward pass from Algorithm 4. **It turns out** that the back-propagation algorithm (available as `backward()` in PyTorch) automatically computes the same quantities—in fact, it computes them in the same way!

First run the forward algorithm to compute Z for a given sentence. The β probabilities are actually just the partial derivatives of Z :

$$\beta_t(j) = \frac{\partial Z}{\partial \alpha_t(j)} \tag{26}$$

This falls out from the fact that $Z = \sum_{t \in \tau_j} \alpha_t(j) \cdot \beta_t(j)$, i.e., the sum of all paths through tag t at time j . Moreover, it turns out that

$$c(s, t) = \frac{\partial \log Z}{\partial \log A_{st}} \qquad c(t, w) = \frac{\partial \log Z}{\partial \log B_{tw}} \tag{27}$$

where $c(\dots)$ represent the fractional counts for the given sentence. (If you want the total fractional counts for a batch or minibatch of sentences, replace $\partial \log Z$ with $\partial \sum_m \log Z_m$.)

Fundamentally, equations (27) work because the HMM probability $p(\mathbf{t}, \mathbf{w})$ (equation (1)) is a log-linear function of the counts of the different types of transitions and emissions in the tagged sentence (\mathbf{t}, \mathbf{w}) , where the weights of these transition and emission types are log-probabilities. Remember that in a log-linear model, the partial derivatives of $\log Z$ with respect to the feature weights are just expected feature counts. That’s where (27) comes from.

F.1 Computational Details

If you computed $\log Z$ from $\log A_{st}$ and $\log B_{tw}$, then calling back-propagation on $\log Z$ will directly compute (27) for you. (First compute $\log Z$, making sure that gradients are being tracked.¹⁵)

¹⁵Gradient tracking will have been turned off for efficiency if your fractional-count method is called in the context “`with torch.no_grad():`” ... for example, if it’s called while `train()` is evaluating the `loss()` in the `hmm.py` starter code. So a fractional-count method that uses back-prop will need to override this, by indenting the computation of $\log Z$ under a line “`with torch.enable_grad():`” (such a line in Python is a **context manager** that does some setup before running a block of code and some cleanup afterwards).

Then call the `retain_grad()` method on the tensors holding the log-probabilities, before calling the `.backward()` method on $\log Z$. Finally you can look at the `.grad` attributes of the tensors holding the log-probabilities. The `.grad` attributes usually only stay available at leaf nodes of the computation graph, but `retain_grad` lets you keep them at intermediate nodes as well.)

Alternatively, if you only have easy access to the probabilities A_{st} and B_{tw} and not their logs, then you can use those instead, through the following change of variables:

$$\begin{aligned} c(s, t) &= \frac{\partial \log Z}{\partial \log A_{st}} & c(t, w) &= \frac{\partial \log Z}{\partial \log B_{tw}} \\ &= \frac{\partial \log Z}{\partial A_{st}} \cdot \frac{\partial A_{st}}{\partial \log A_{st}} = \frac{\partial \log Z}{\partial A_{st}} \cdot A_{st} & &= \frac{\partial \log Z}{\partial B_{tw}} \cdot \frac{\partial B_{tw}}{\partial \log B_{tw}} = \frac{\partial \log Z}{\partial B_{tw}} \cdot B_{tw} \end{aligned} \quad (28)$$

F.2 Position-Specific Counts

There’s one more trick you need to know about. Equation (27) considers the effect on $\log Z$ of changing a weight. That’s the total effect through every position in the sentence—so it gives a fractional count that’s summed over all positions. But what if you just want to know whether a certain transition or emission happened at a particular token position j ? For example, you’d want that for posterior decoding (reading section E).

Well, you could get this by equation (27) if there were a particular log-probability or probability that was *only* used at position j . And you can arrange that! Just use a temporary copy of that log-probability or probability in your computation.

For example, when you look up $p(w_j | t_j)$ in matrix B at Algorithm 1 line 10, you can assign it to a temporary variable like this:

$$\text{twprob}[j][t_j] \leftarrow B_{t_j w_j} + 0 \quad \triangleright \text{adding } 0 \text{ or calling } .\text{clone}() \text{ makes a new Tensor with same value} \quad (29)$$

$$\text{twprob}[j][t_j].\text{retain_grad}() \quad (30)$$

Then after you call `Z.log().backward()`, you will find the partial derivatives with respect to `twprob[j][t_j]` and $B_{t_j w_j}$ in their respective `.grad` fields. The former partial derivative is position-specific, and the latter collects the sum of all the position-specific partial derivatives. You can use the trick of equation (28) to convert these partial derivatives into fractional counts.

If you were storing B in the log domain as a matrix lB (reading section D.2), then your code can look more like

$$\text{twlogprob}[j][t_j] \leftarrow lB_{t_j w_j} + 0 \quad (31)$$

$$\text{twlogprob}[j][t_j].\text{retain_grad}() \quad (32)$$

and you won’t need the trick of equation (28).

F.3 Generalizations

This connection between gradients and expected counts also holds for the CRF models that we’ll see next in reading section G. And it’s also true for fancier probabilistic models based on CFGs and FSTs and more. If you’re curious to understand the details, I published [a tutorial paper](#) that explains all of this in detail—looking at the inside-outside and forward-backward algorithms.

G Linear-Chain Conditional Random Fields (CRFs)

G.1 HMMs versus CRFs

An HMM defines $p(\mathbf{t}, \mathbf{w})$. It also defines the conditional distribution

$$p(\mathbf{t} \mid \mathbf{w}) = p(\mathbf{t}, \mathbf{w}) / p(\mathbf{w}) \quad (33)$$

This is the ratio of equations (1) and (3); the denominator $p(\mathbf{w})$ can be computed by the forward algorithm. Posterior decoding and Viterbi decoding depend only on $p(\mathbf{t} \mid \mathbf{w})$ —they are trying to find the individual tags or tag sequences that are most probable given \mathbf{w} .

A linear-chain CRF is a variant that directly defines $p(\mathbf{t} \mid \mathbf{w})$ and has no opinion about $p(\mathbf{w})$. (That is, it is a *discriminative* model, whereas the HMM is *generative*.) Thus, we can't train it to maximize the incomplete-data log-likelihood $\sum_m p(\mathbf{w}_m)$, nor the complete-data log-likelihood $\sum_m p(\mathbf{t}_m, \mathbf{w}_m)$. However, as long as we have supervised data, we can train it to maximize the *conditional* log-likelihood:

$$\mathcal{L}(A, B) \stackrel{\text{def}}{=} \sum_m \log p(\mathbf{t}_m \mid \mathbf{w}_m) \quad (34)$$

Of course, one could also train an HMM discriminatively, to maximize only equation (34).

G.2 Defining a CRF

The CRF formula and algorithms are very similar. Instead of equations (1) and (3), we have

$$\tilde{p}(\mathbf{t}, \mathbf{w}) = \left(\prod_{i=1}^{n+1} \phi_A(t_{i-1}, t_i) \right) \cdot \left(\prod_{i=1}^n \phi_B(t_i, w_i) \right) \quad (35)$$

$$Z(\mathbf{w}) = \sum_{\mathbf{t}} p(\mathbf{t}, \mathbf{w}) \quad (36)$$

$$p(\mathbf{t} \mid \mathbf{w}) = \tilde{p}(\mathbf{t}, \mathbf{w}) / Z(\mathbf{w}) \quad (37)$$

In this handout, we'll systematically use \tilde{p} as in equation (35) to denote an unnormalized probability distribution—just like a probability distribution, but it might not sum to 1. (We referred to this as u in the log-linear handout.) We'll use p to denote the normalized version, which in this case is the conditional distribution (37).

The factors in equation (35), which must be ≥ 0 , are returned by the **potential functions** ϕ_A and ϕ_B . Equation (1) is just a special case where these factors are conditional probabilities (with the result that \tilde{p} is actually a probability distribution over tagged sentences: $\sum_{\mathbf{t}, \mathbf{w}} \tilde{p}(\mathbf{t}, \mathbf{w}) = 1$).

A more compact way of writing equations (35)–(37) is to use the “proportional to” symbol \propto . The constant of proportionality is understood to be “whatever ensures that $\sum_{\mathbf{t}} p(\mathbf{t} \mid \mathbf{w}) = 1$.”

$$p(\mathbf{t} \mid \mathbf{w}) \propto \left(\prod_{i=1}^{n+1} \phi_A(t_{i-1}, t_i) \right) \cdot \left(\prod_{i=1}^n \phi_B(t_i, w_i) \right) \quad (38)$$

So far, we haven't gained anything. It turns out that any CRF distribution with potentials defined as in (38) could also have been obtained as the conditional distribution (33) of some HMM

(Smith and Johnson (2007)). Thus, maximizing the conditional log-likelihood of a CRF is no better than maximizing the conditional log-likelihood of an HMM.

However, let’s change equation (38) to allow the potential functions to also depend on the sentence and the position in it:

$$\tilde{p}(\mathbf{t} \mid \mathbf{w}) \propto \left(\prod_{i=1}^{n+1} \phi_A(t_{i-1}, t_i, \mathbf{w}, i) \right) \cdot \left(\prod_{i=1}^n \phi_B(t_i, w_i, \mathbf{w}, i) \right) \quad (39)$$

Now we have a much more powerful model, because now when a potential function evaluates a transition (t_{i-1}, t_i) or an emission (t_i, w_i) , it can look at the context in \mathbf{w} of that transition or emission. Information from the whole sentence can help inform our local tagging judgment.

Equation (39) is called a linear-chain CRF, because the random variables that we are trying to predict— T_1, T_2, \dots, T_n —are linked together in a chain by the ϕ_A potential functions, making each tag interdependent with the next one. The ϕ_B potential functions link the tags to the words.¹⁶

G.3 Parameterization

What kind of potential functions should we learn? We have lots of choices, but here is a straightforward approach.

We don’t need to use a softmax parameterization for ϕ , because potential functions don’t need to sum to 1. They only need to be ≥ 0 . We can just replace equation (15) with its numerator

$$\phi_A(s, t) = \exp W_{st}^A \qquad \phi_B(t, w) = \exp W_{tw}^B \quad (40)$$

where W^A is defined as before (reading section C.3) and so is W^B (equation (18)). But that’s just for the simple case (38); it doesn’t give us any dependence on context as allowed by (39).

So let’s introduce such dependence. The obvious way to generalize equation (40) along the lines of equation (18) is to build unnormalized log-linear functions:

$$\phi_A(s, t, \mathbf{w}, j) = \exp \left(\vec{\theta}^A \cdot \vec{f}^A(s, t, \mathbf{w}, j) \right) \qquad \phi_B(t, w, \mathbf{w}, j) = \exp \left(\vec{\theta}^B \cdot \vec{f}^B(t, w, \mathbf{w}, j) \right) \quad (41)$$

Here each \vec{f} function returns a feature vector. We could hand-craft those features, but it’s easier to let a neural network discover the relevant features. That is, \vec{f} should construct a vector embedding of the argument tuple—the transition or emission in context.

Here’s one way to do that. First, let \vec{h}_j be a (row) vector embedding of the sentence prefix $w_1 \cdots w_j$, and let \vec{h}'_j be a (column) vector embedding of the sentence suffix $w_{j+1} \cdots w_n$. We can get these using left-to-right and right-to-left RNNs. Those familiar computations are actually very similar to the computations of $\vec{\alpha}$ and $\vec{\beta}$, particularly the versions in footnotes 1 and 10:

$$\vec{h}_j = \sigma \left(M \begin{bmatrix} 1; \vec{h}_{j-1}; \vec{w}_j \end{bmatrix} \right) \qquad \vec{h}'_{j-1} = \sigma \left(M' \begin{bmatrix} 1; \vec{w}_j; \vec{h}'_j \end{bmatrix} \right) \quad (42)$$

The most important difference from $\vec{\alpha}, \vec{\beta}$ is that the definitions of \vec{h}, \vec{h}' include a nonlinearity, the σ (“sigmoid”) function. This is defined by $\sigma(x) = 1/(1 + \exp(-x))$, and is applied separately

¹⁶Actually the ϕ_B factors don’t actually add any power, because the ϕ_A functions can look at the words just as well. But it may still be convenient to keep the ϕ_B factors, because they provide a kind of backoff (from tag bigrams to tag unigrams), and because they may be parameterized differently from the ϕ_A factors.

(“elementwise”) to each element of its argument. The square-bracket-semicolon notation denotes concatenation of multiple column vectors into a column vector. The subscripts on \vec{h}_{j-1} and \vec{h}'_j denote the inter-word positions immediately before and after w_j (similar to our convention in parsing). The matrices M and M' are parameters, as are the vectors used for the base cases \vec{h}_{-1} (just before $w_0 = \text{BOSW}$) and \vec{h}'_{n+1} (just after $w_{n+1} = \text{EOSW}$).

Now we’ve got prefix and suffix embeddings. Let’s suppose we also have a vector embedding \vec{t} for each tag t , and a vector embedding \vec{w} for each word w . We can then combine these to construct our embeddings of the argument tuples used in (45):

$$\vec{f}^A(s, t, \mathbf{w}, j) = \sigma(U^A [1; \vec{h}_{j-2}; \vec{s}; \vec{t}; \vec{h}'_j]) \quad (43)$$

$$\vec{f}^B(t, w, \mathbf{w}, j) = \sigma(U^B [1; \vec{h}_{j-1}; \vec{t}; \vec{w}; \vec{h}'_j]) \quad (44)$$

For example, equation (43) is encoding a tuple of the form (prefix, s , t , suffix). That is, \vec{h}_{j-2} encodes the prefix before the tag bigram st at position j , and \vec{h}'_j encodes the suffix after that tag bigram token. So equation (43) encodes this tag bigram token *in context*.¹⁷

In each of equations (42)–(44), we are essentially following the standard recipe from class for encoding tuples: concatenate the embeddings of the tuple’s elements, then apply a sigmoided affine transformation. The tag embeddings \vec{s} , \vec{t} and the matrices U^A , U^B are additional parameters. The word embeddings \vec{w} may be pretrained vectors that are held constant, although they too could be treated as parameters and fine-tuned.

Summary: We switched to discriminative training of our HMM, and then switched from an HMM to a CRF, which can examine the entirety of \mathbf{w} when figuring out how to tag w_j . The old way to build a CRF was to design feature templates so that each feature would fire on certain tags and tag bigrams in the context of certain specified patterns in \mathbf{w} . But in practice, such features rarely looked at much of \mathbf{w} (usually just the local context around the tag or tag bigram). Our more modern approach just uses a biRNN (42) to scan \mathbf{w} for useful patterns. These patterns combine with the tags or tag bigrams through a further neural network (43)–(44) to define the feature vectors \vec{f} . This whole system is trained via equation (34) so that the neural machinery is encouraged to extract whatever features turn out to be useful for the tagging task. If you have hand-designed features that you expect to be useful as well (why guess when you know?), you can simply append them to the feature vectors in (43)–(44), and equation (45) will learn $\vec{\theta}$ -weights for them too.

¹⁷Of course, you don’t *have* to do it this way. There are other reasonable ways to embed a tag bigram in context: for example, instead of \vec{h}_{j-2} and \vec{h}'_j , you could use \vec{h}_j and \vec{h}'_{j-2} , so that the prefix and suffix embeddings also include the two words that are being tagged by st . Or you could simply drop the prefix and suffix embeddings altogether from equation (43), in which case the resulting ϕ_B in equation (45) does not consider context any more than it did in equation (40). The point is, you have a lot of options, just as when you define features for a log-linear model.

Also, a common choice is to replace equation (45) with something like

$$\phi_A(s, t, \mathbf{w}, j) = \exp(\vec{\theta}^{A,s,t} \cdot \vec{f}^A(\mathbf{w}, j)) \quad \phi_B(t, w, \mathbf{w}, j) = \exp(\vec{\theta}^{B,t,w} \cdot \vec{f}^B(\mathbf{w}, j))$$

where the vectors $\vec{\theta}^{A,s,t}$ and $\vec{\theta}^{B,t,w}$ are learned embeddings of the possible transitions and emissions, akin to the learned embeddings of possible vocabulary words that we’d use if we were predicting a word. The advantage of this architecture is that the context encodings $\vec{f}^A(\mathbf{w}, j)$, $\vec{f}^B(\mathbf{w}, j)$ of position j can be reused for all the possible transitions and emissions at position j . We no longer need to embed each transition or emission separately in context. So this architecture is a little more efficient than equation (45), though also a little less flexible.

G.4 Algorithms

To compute the CRF’s conditional log-likelihood (34), the expensive part is computing the normalizing constant $Z(\mathbf{w})$ (36). Since this sums over all taggings, we would like to still do this by running some small variant of the forward algorithm (Algorithm 1).

Remember that the core of the forward algorithm is the successive update of $\vec{\alpha}$ vectors by equation (12). We’ll just replace that equation with a slight variant:

$$\vec{\alpha}(j) = \left(\vec{\alpha}(j-1) \cdot A^{(j)} \right) \odot \vec{b}^{(j)} \quad (45)$$

where $A^{(j)}$ is a matrix of potentials for bigrams *at position j in the given input sentence \mathbf{w}* , and $\vec{b}^{(j)}$ is a vector of potentials for unigrams *at position j in the given input sentence \mathbf{w}* .

All we have to do is to package up the appropriate potentials that were defined in equation (45):

$$A_{st}^{(j)} = \phi_A(s, t, \mathbf{w}, j) \quad b_t^{(j)} = \phi_B(t, w_j, \mathbf{w}, j) \quad (46)$$

In short, the new idea is that A and B are no longer fixed throughout a sentence or a minibatch of sentences. At each position j in a sentence, we construct contextual versions $A^{(j)}$ and $\vec{b}^{(j)}$, which depend on the sentence \mathbf{w} and the learned parameters. This is the main trick of CRFs.

Crucially, the transition and emission potentials at position j *don’t* depend on the tags at positions other than $j-1$ and j . (It’s still considering only tag bigrams.) They can therefore be reused across exponentially many taggings. We can still use dynamic programming algorithms to efficiently work with the CRF distribution (37), taking advantage of the linear chain structure of equation (39).

In particular, for training, we can use the forward algorithm to compute the conditional log-likelihood (and then compute its gradient by back-propagation and follow it by SGD, as usual). The forward algorithm should use $A^{(j)}$ and $\vec{b}^{(j)}$ at step j . For extracting actual taggings, the Viterbi algorithm, the backward sampling algorithm, and posterior decoding can be similarly extended to use the same $A^{(j)}$ and $\vec{b}^{(j)}$. You can lazily compute each matrix “on demand” only when you first need to use it. Or alternatively, you can eagerly compute all the matrices needed for the minibatch as soon as you receive the minibatch, and then look each matrix up when you need to use it.

H Data Resources for the Homework

There are three datasets, available at <http://cs.jhu.edu/~jason/465/hw-tag/data>.

The datasets are

- **ic**: Ice cream cone sequences with 1-character tags (C, H). Start with this easy dataset.
- **en**: English word sequences with 1-character tags (documented in Figure 1).
- **cz**: Czech word sequences with 2-character tags.

This homework will focus on the **en** dataset. The **ic** dataset is to help you test your code.

The **cz** dataset is provided just so that you can see what a harder tagging task looks like. The tagset is larger due to the morphological complexity of Czech (in fact, the original tags had 6 characters), and spelling features are more important in Czech than in English. You are welcome to try your code on it!

Each dataset consists of three files:

C	Coordinating conjunction or Cardinal number
D	Determiner
E	Existential <i>there</i>
F	Foreign word
I	Preposition or subordinating conjunction
J	Adjective
L	List item marker (<i>a., b., c., ...</i>) (rare)
M	Modal (<i>could, would, must, can, might ...</i>)
N	Noun
P	Pronoun or Possessive ending (<i>'s</i>) or Predeterminer
R	Adverb or Particle
S	Symbol, mathematical (rare)
T	The word <i>to</i>
U	Interjection (rare)
V	Verb
W	<i>wh</i> -word (question word)
BOS	Context for the start of a sentence
EOS	Marks the end of a sentence
,	Comma
.	Period
:	Colon, semicolon, or dash
-	Parenthesis
'	Open quotation mark
'	Close quotation mark
\$	Currency symbol

Figure 1: Tags in the **en** dataset. These are the preterminals from **wallstreet.gr** in homework 3, but stripped down to their first letters. For example, all kinds of nouns (formerly **NN**, **NNS**, **NNP**, **NNPS**) are simply tagged as **N** in this homework. Using only the first letters reduces the number of tags, speeding things up and increasing accuracy. (However, it results in a couple of unnatural categories, **C** and **P**.)

- **sup**: tagged data for supervised training (**ensup** provides 100,000 words)
- **dev**: tagged data for testing (25,000 words for **endev**); your tagger should ignore the tags in this file except when measuring the accuracy of its tagging
- **raw**: untagged data for reestimating parameters (100,000 words for **enraw**)

The file format is quite simple. Each line has a single word/tag pair separated by the **/** character. (In the **raw** file, only the word appears.) Punctuation marks count as words.

H.1 Tag and Word Vocabularies

Take the tag vocabulary to be all the tag types that appeared at least once in **sup**. For simplicity, we will not include an OOV tag. (OOT?) Thus, any novel tag will simply have probability 0 (even

with smoothing).¹⁸

Take the word vocabulary to be all the word types that appeared at least once in **sup** \cup **raw** (or just in **sup** if no **raw** file is provided, as in the case of **vtag**), plus an OOV type in case any out-of-vocabulary word types show up in **dev**.¹⁹

As in homework 3, you should use the same vocabulary size V throughout a run of your program, that your perplexity results will be comparable to one another. So you need to construct the vocabulary before you Viterbi-tag **dev** the first time (even though you have not used **raw** yet in any other way).

You are strongly encouraged to test your code using the artificial **ic** dataset. This dataset is small and should run fast. More important, it is designed so you can check your work: if you run the forward-backward algorithm, the initial parameters, intermediate results, and perplexities should all agree *exactly* with the results on the spreadsheet we used in class. If you are training with SGD, then you should still converge to the same place as EM converges to.

- **icsup** has been designed so that your initial unsmoothed supervised training on it will yield the initial parameters from the spreadsheet (transition and emission probabilities).
- **icdev** has exactly the data from the spreadsheet. Running your Viterbi tagger with the above parameters on **icdev** should produce the same values as the spreadsheet’s iteration 0:²⁰
 - $\hat{\alpha}$ probabilities for each day
 - weather tag for each day (shown on the graph)²¹

I Measuring Tagging Performance

There are various metrics that you could report to measure the quality of a part-of-speech tagger.

¹⁸Is this simplification okay? How bad is it to assign probability 0 to novel tags?

- Effect on perplexity (reading section I.2): You might occasionally assign probability 0 to the *correct* tagging of a **test** sentence, because it includes novel tag types of probability 0. This yields perplexity of ∞ .
- Effect on accuracy (reading section I.1): The effect on accuracy will be minimal, however. The decoder will simply never guess any novel tags. But few **test** tokens require a novel tag, anyway.

¹⁹It would not be safe to assign 0 probability to novel *words*, because words are actually observed in **dev**. If any novel words showed up in **dev**, we’d end up computing $p(\vec{t}, \vec{w}) = 0$ for *every* tagging \vec{t} of the dev corpus \vec{w} , and we couldn’t identify the *best* tagging. So we need to hold out some smoothed probability for the OOV word.

²⁰To check your work, you only have to look at iteration 0, at the left of the spreadsheet. But for your interest, the spreadsheet does do reestimation. It is just like the forward-backward spreadsheet, but uses the Viterbi approximation. Interestingly, this approximation *prevents* it from really learning the pattern in the ice cream data, especially when you start it off with bad parameters. Instead of making gradual adjustments that converge to a good model, it jumps right to a model based on the Viterbi tag sequence. This sequence tends never to change again, so we have convergence to a mediocre model after one iteration. This is not surprising. The forward-backward algorithm interprets the interpreting the world in terms of its stereotypes but then uses those interpretations to update its stereotypes. The Viterbi approximation turns it into a blinkered fanatic that is absolutely positive that its interpretation of each sequence is correct, and therefore it learns less, particularly when it has only one sequence to learn from.

²¹You won’t be able to check your backpointers directly.

I.1 Accuracy

In these task-specific metrics, you look at some subset of the tokens in your evaluation sentences (**dev** or **test**) and ask what percentage of them received the correct tag.

accuracy looks at all test tokens, except for the sentence boundary markers BOSW and EOSW. (No one in NLP tries to take credit for tagging EOSW correctly with EOS!)

known-word accuracy considers only tokens of words (other than BOSW and EOSW) that also appeared in **sup**. So we have observed some possible parts of speech.

seen-word accuracy considers tokens of words that did not appear in **sup**, but did appear in **raw** untagged data. Thus, we have observed the words in context and have used EM to try to infer their parts of speech.

novel-word accuracy considers only tokens of words that did *not* appear in **sup** or **raw**. These are very hard to tag, since context at test time is the only clue to the correct tag. But they constitute about 9% of all tokens in **endev**, so it is important to tag them as accurately as possible.

vtagem's output must also include the perplexity per *untagged* raw word. This is defined on **raw** data \vec{w} as

$$\exp\left(-\frac{\log p(w_1, \dots, w_n | w_0)}{n}\right)$$

Note that this does not mention the tags for raw data, which we don't even know. It is easy to compute, since you found $Z = p(w_1, \dots, w_n | w_0)$ while running the forward-backward algorithm (Algorithm 4). It is the total probability of *all* paths (tag sequences compatible with the optional dictionary) that generate the raw word sequence.

I.2 Perplexity

As usual, perplexity is a useful task-independent metric that may correlate with accuracy.

Given a tagged corpus, the model's perplexity per tagged word is given by²²

$$\text{perplexity per tagged word} = 2^{\text{cross-entropy per tagged word}} \tag{47}$$

where

$$\text{cross-entropy per tagged word} = \frac{-\log_2 p(w_1, t_1, \dots, w_n, t_n | w_0, t_0)}{n}$$

Since the power of 2 and the log base 2 cancel each other out, you can equivalently write this using a power of e and log base e :

$$\text{perplexity per tagged word} = \exp(-\text{log-likelihood per tagged word}) \tag{48}$$

$$= \exp\left(-\frac{\log p(w_1, t_1, \dots, w_n, t_n | w_0, t_0)}{n}\right) \tag{49}$$

²²Using the notation from reading section A.1.

This is equivalent because $e^{-(\log x)/n} = (e^{\log x})^{-1/n} = x^{-1/n} = (2^{\log_2 x})^{-1/n} = 2^{-(\log_2 x)/n}$.

Why is the corpus probability in the formula conditioned on w_0/t_0 ? Because the model only generates $w_1/t_1, \dots, w_n/t_n$. You knew in advance that $w_0/t_0 = \text{BOSW}/\text{BOS}$ would be the left context for generating those tagged words. The model has no distribution $p(w_0, t_0)$. Instead, Algorithm 2 explicitly hard-codes your prior knowledge that $t_0 = \text{BOS}$.

When you have untagged data, you can also compute the model’s perplexity on that:

$$\begin{aligned} \text{perplexity per untagged word} &= \exp(-\log\text{-likelihood per untagged word}) \\ &= \exp\left(-\frac{\log p(w_1, \dots, w_n, | w_0 t_0)}{n}\right) \end{aligned} \quad (50)$$

where the forward or backward algorithm can compute

$$p(w_1, \dots, w_n, | w_0, t_0) = \sum_{t_1, \dots, t_n} p(w_1, t_1, \dots, w_n, t_n | w_0, t_0) \quad (51)$$

Notice that

$$p(w_1, t_1, \dots, w_n, t_n | w_0, t_0) = p(w_1, \dots, w_n | w_0, t_0) \cdot p(t_1, \dots, t_n | \vec{w}, t_0) \quad (52)$$

so the tagged perplexity (48) can be regarded as the product of two perplexities—namely, how perplexed is the model by the words (in (50)), and how perplexed is it by the tags given the words?

To *evaluate* a trained model, you should ordinarily consider its perplexity on *held-out* data. Lower perplexity is better. Of course, likelihood-based training is equivalent to seeking low perplexity on *training* data.

I.3 Baselines and Ablations

When you build a system, you should compare it with simpler methods, to make sure that all of the work you put into it is worthwhile.

In the case of an HMM tagger, a very simple **baseline method** is to tag each word with its most frequent tag from supervised training data. If the word didn’t appear in supervised training, just back off and tag it with the most frequent tag overall.

If you can’t improve over this baseline, where the improvement is evaluated on held-out data, then there was no point to all of the HMM stuff!

In addition, if you build a fancy system, you should try turning off parts of it to see whether they helped. This is called an **ablation experiment**. For example, you could see whether pretrained word embeddings really improved your accuracy, compared to simple one-hot embeddings.

One good ablation for an HMM is to replace the *bigram* transition probabilities $p_A(t_i | t_{i-1})$ with *unigram* transition probabilities $p_{At}(t_i)$. The bigram case is the plain-vanilla definition of an HMM. It is sometimes called a 1st-order HMM, meaning that each tag depends on 1 previous tag—just enough to make things interesting. A fancier trigram version using $p_A(t_i | t_{i-2}, t_{i-1})$ would be called a 2nd-order HMM. So by analogy, the unigram case can be called a 0th-order HMM. The order refers to how far we look back when determining the current tag.

Since the unigram HMM throws away the tag-to-tag dependencies, it is really just tagging each word in isolation. To be precise, the best path maximizes $p_A(t_i) \cdot p_B(w_i | t_i)$ at each position i

separately. But that is simply the Bayes' Theorem method for maximizing $p(t_i | w_i)$. So this is really just the baseline method again—picking the most probable tag for each word! The only difference is that it uses smoothed probabilities (which might be better, for example if they use pretrained word embeddings).

Note that a unigram HMM can actually be implemented to be very fast. The most probable tag for each word type can be precomputed. then the tagger can just look up each word token's most probable part of speech in a hash table—with overall runtime $O(n)$. No dynamic programming is needed.