s 601.465/665 — Natural Language Processing
Homework 7: Neuralization

Prof. Jason Eisner — Fall 2025
Due date: Friday 21 November, 11 pm

This homework is a continuation of the previous one. You will "neuralize" your CRF tagger by using neural networks to predict its transition and emission potentials. In particular,

- You will allow different potential matrices in different contexts.

- You will extract context features that are useful for tagging using a bidirectional recurrent neural network (biRNN). These will let you score tag, perhaps tagging words that it has not seen in unigrams and bigrams in context.

- Your neural network will use pretrained word embeddings to encode the context, which helps it smooth across similar word types. For example, might correctly tag words that it has never seen in training data.

To maximize the training objective (conditional log-likelihood), you will use PyTorch's back-propagation to compute its gradient, and take gradient steps using SGD as in the language modeling homework.

Homework goals: See the previous homework handout.

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 PDF submission to Gradescope should describe at the top what each of you contributed, so that we know you shared the work fairly.
  (c) It is okay to use the same partner for homeworks 6 and 7, though it is not required.
In any case, observe academic integrity and never claim any work by third parties as your own.

Materials: You should build on your code from the previous homework. You will need the forward algorithm and Viterbi algorithm from the previous homework to work correctly—but you won't need the full forward-backward algorithm (E step) or the M step.

You will also continue to work with the data from the previous homework (which were described in its reading section F). You can reuse the lexicons of word embeddings from Homework 3.

We provide additional starter code and INSTRUCTIONS for working with it at https://www.cs.jhu.edu/~jason/465/hw-rnn/.

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

✏️₁₅ **Autograding:** We will post instructions about what files to submit beyond your writeup. We'll probably ask you to upload your best trained models.

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.

1 Implement a Neural CRF

The biRNN-CRF is a fairly modern neural architecture. There are many variants, but this style of model is commonly used to do structured prediction.

A specific version is spelled out in reading section H. You'll want to carefully examine reading section H.4 on how to parameterize the potential functions.

Your easiest route is to follow the steps given in the `INSTRUCTIONS` file.

✏️₁₆   (a) Describe the deterministic patterns in the `next` and `pos` datasets.

✏️₁₇   (b) Why can't your models from the previous homework fit these patterns? Why can a biRNN-CRF do it?

✏️₁₈   (c) The dimensionality $d \geq 0$ is up to you to pick. Does it matter (can it be too small or large)?

Hint: Remember to avoid **for** loops. That is, don't compute the potentials one at a time—for every transition and emission that might occur on the untagged sentence—even if the formulas happen to be presented that way. Often with neural nets, you're doing the same operation many times over on different inputs. And if you're lucky, you don't need to do them in any particular order: the output of one computation isn't needed as the input of the next. In that case, "tensorize" and work on all the inputs in parallel! Combine your work into a few big tensor operations instead of many little ones. It will be So Much Faster.

2 Experiment

Compare your biRNN-CRF to the simple stationary CRF, primarily using `ensup` for training and `endev` for evaluation. Again, `INSTRUCTIONS` gives some tips.

Your performance will depend on some hyperparameters that control the model architecture (RNN dimensionality, choice of lexicon) and also on hyperparameters that control training (learning rate, regularization, minibatch size). Experiment a bit with these until you've learned something. :-)

What did you find out?

✏️₁₉   (a) Were you able to get better cross-entropy or accuracy with the biRNN-CRF than with the simple stationary CRF?

☞20   (b) How did hyperparameters affect those metrics? Discuss.

☞21   (c) How did hyperparameters affect the progress and speed of training? Discuss.

☞22   (d) What happens if you evaluate your trained model on the training data (`ensup`)? Discuss.

Note: The simple CRF (`ConditionalRandomFieldBackprop`) uses an SGD optimizer by default, whereas we snuck in a smarter AdamW optimizer in the `ConditionalRandomFieldNeural` subclass. To make your comparison fair, you might want to use AdamW for both. This is a small change to the code.

3   Informed Embeddings

Your original CRF had one parameter for every (t, w) pair (the log-potential). Your BiRNN-CRF has fewer parameters: even if there is a large vocabulary of words, they all have to use the same embedding dimensions. This means you can't easily learn special properties of particular words: words with similar embeddings will behave similarly.

If the pretrained embeddings have dimensions that implicitly encode part of speech, then this might work. However, the pretrained embeddings we borrowed from HW3 were more semantic than syntactic. They were trained using CBOW, which doesn't really consider the adjacency or order of word tokens. Therefore, their dimensions may not be useful features for our task.

You could try using embeddings from some other kind of pretrained model, such as an RNN language model. But here's another idea. For the training words, at least, you know how often they showed up in training data, and also how often they showed up with particular tags. Maybe you can derive additional features from those frequencies. These are certainly related to the POS tagging task, and they may distinguish words whose CBOW embeddings would otherwise be close.

Augment your word embeddings with these features, as sketched in `INSTRUCTIONS`.

☞23   (a) Which works best—the CBOW features, the frequency-based features, or both together? How about simple one-hot embeddings of the training words?

☞24   (b) What if you isolate the effect of the embeddings by turning off the RNN context features? (Just set the RNN dimensionality d of $\vec{h}, \vec{h}'$ to be 0, which gives you back a stationary model that sees words only via their embeddings.)

☞25   (c) Do any of these options beat your original stationary model from the previous homework?

☞26   (d) You might expect the CBOW embeddings to help most for "seen" words, the frequency-based features to help most for "known" words, and the RNN context features to help most for "novel" words. Why? Do you in fact see evidence of that pattern?

"Seen" here refers to words that appeared only in the *untagged lexicon* (not in supervised data). This is a bit different from the previous homework (EM), where "seen" referred to words that appeared only in *untagged text* (not in supervised data). "Novel" continues to refer to words that are OOV for the model: all OOV words are treated alike, so only context tells us how to tag them. (See reading section G.1 in the previous homework.)

4 Extensions [extra credit]

As in the previous homework, you can try to improve performance for extra credit. You've already tried experimenting with the hyperparameters, but what else might help? Tell us what you tried, why you thought it was a good idea, and how much it helped (or didn't).

Just a good design discussion—working out a plan that *might* improve performance—will get you a few points of extra credit. But for full extra credit, you should actually try it.[1]

Here are some ideas to get you started:

- What if you treat the word embeddings as parameters and tune them along with everything else? (This can only improve the known-word embeddings. However, that may help unknown words if the known-words appear in their context where the RNNs can see them.)

- What if you enrich the embeddings with features for affixes or word shapes? (See `lexicon.py`.)

- Can you find a more appropriate embedding lexicon online and try to use it?

- What if you change the neural architecture? Formulas (45), (47) and (48) are not the only way you could define potentials in terms of parameters. Start by writing out your alternative formulas.

  - In fact, those formulas were a little awkward to compute in PyTorch, because you had to run the feed-forward network separately[2] for each tag unigram and tag bigram at position j. What if you only ran the feed-forward network once at position j (on some appropriate input vector), and it gave you a whole vector of contextually influenced log-potentials, all computed from the same hidden layer? Perhaps that'll work better, and it may also be faster.
  - You could try shallower or deeper architectures.
  - You could try dropping the RNN and just using a fixed-size window of words.

- Perhaps different groups of parameters need to train for different amounts of time, or in a particular order, or with different regularization. Can you come up with a training recipe that works better?

  - Look into "parameter groups" in PyTorch to figure out how to set per-parameter options.
  - Freeze some parameters at the start of training and don't try to learn them until other parameters have already learned to do a pretty good job (after which . you might freeze the other parameters so that they don't lose their mojo).
  - Pretrain the two RNNs on a task like language modeling or cloze language modeling (cf. ELMo and BERT). This allows you to use much more data.
  - A quick trick: if some parameter tensor `X` needs a larger learning rate, you can replace `X` in your formulas by (say) `X*10`. (This means the optimal parameter `X` will tend to be $10\times$ smaller.) But then an ϵ change to `X` will have $10\times$ the effect, which means that the gradient will be $10\times$ bigger. So `X` will move $10\times$ faster—and the quantity `X*10` that you're actually using will move $100\times$ faster! Also, the L_2 regularizer on `X` will be $100\times$ weaker, unless you change it.

---

[1] It may be convenient to use the `--awesome` flag to invoke your new behavior (which might involve a new subclass).
[2] Though hopefully not one at a time, as noted in the earlier hint!

- Neural nets are data-hungry, especially if they're complex with a lot of parameters. What if you trained on a lot more data? Can you find some?

- Maybe part-of-speech tagging doesn't really need that much context, and a simple stationary CRF (or a unigram model) does pretty well. Can you come up with an artificial tagging dataset where your biRNN-CRF architecture will greatly outperform the simpler models? In other words, can you show that the biRNN-CRF tagger is able to learn something important when it's really there to be learned? (To make this interesting, the patterns in your data should be harder to learn than in the simple `next` and `pos` datasets. They should be complicated and/or noisy, rather than simple and deterministic.)

601.465/665 — Natural Language Processing
Reading for Homework 7: Neuralization

Prof. Jason Eisner — Fall 2025

We don't have a required textbook for this course. Instead, handouts like this one are the main readings. This handout accompanies homework 7. It is a continuation of the homework 6 handout and refers back to that handout.

H   Fancier CRF Featurization

Last homework (reading section E.3) introduced a simple restricted family of CRFs:

$$p(\boldsymbol{t} \mid \boldsymbol{w}) \propto \left(\prod_{i=1}^{n+1} \phi_{\mathrm{A}}(t_{i-1}, t_i) \right) \cdot \left(\prod_{i=1}^{n} \phi_{\mathrm{B}}(t_i, w_i) \right) \tag{30}$$

where the potential functions are defined in terms of weights

$$\phi_{\mathrm{A}}(s,t) = \exp W^A_{st} > 0 \qquad\qquad \phi_{\mathrm{B}}(t,w) = \exp W^B_{tw} > 0 \tag{31}$$

and are stored in potential matrices as $A = \exp W^A$, $B = \exp W^B$. We will now upgrade this to a more modern architecture using word embeddings and neural networks.

H.1   Features from Word Embeddings

Our CRF used a separate parameter W^B_{tw} for each possible emission, corresponding to the HMM parameter $p(w \mid t)$. 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 the slow convergence would be to initialize the parameters to something reasonable. For example, you could initialize the potential matrices A and B to HMM transition and emission probabilities estimated via equation (17) from the supervised data (more precisely, you would initialize W^A and W^B to the logs of those probabilities).

But perhaps a better approach is to reduce the number of parameters, by sharing parameters among similar words. This has a smoothing effect. How do we know which words are similar? We can use pretrained word embeddings!

Let $\vec{w} \in \mathbb{R}^e$ 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 $\vec{\theta}^B_t$. Now we can define

$$W^B_{tw} = \vec{\theta}^B_t \cdot \vec{w} \tag{42}$$

Equivalently, the matrix W^B can be computed "all at once" by

$$W^B = \Theta^B E^\top \tag{43}$$

where Θ^B is a matrix whose rows are the d-dimensional weight vectors $\vec{\theta}_t^B$ for the different tags, and E is a matrix whose rows are the d-dimensional embeddings $\vec{w}$ for the different words.

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.

Still linear? Notice that even when we define W^B by equation (43), equation (30) is still a conditional log-linear model over taggings. Its parameters (weights) are now stored in the W^A and Θ^B matrices. In particular, each tag t has a separate set of d features, whose weights $\vec{t}$ are stored in a row of Θ^B. Whenever that tag is applied to a word w, all d of those features fire with strengths corresponding to the dimensions of $\vec{w}$. So you can think of the feature functions as detecting properties of $\vec{w}$, and their weights $\vec{t}$ as evaluating how compatible those properties are with t. Fundamentally, the reason that we still have a log-linear model is that the unnormalized log-probabilities are linear in W^B, which in turn is linear in the new parameters Θ^B.

This model is log-linear only because the embeddings E—which serve as linear coefficients—are fixed. If we decide to fine-tune the embeddings E along with the weights Θ^B (see below), then our unnormalized log-probabilities are no longer linear functions of the parameters, because we are multiplying pairs of parameters together! In this case, the CRF training objective (35) is no longer necessarily convex.

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. This is known as "early stopping."

H.2 Features from Tag Embeddings

A natural question arises: Should we also use tag embeddings? Not necessarily. Our tag set on this homework is small enough that it's reasonable to directly learn the $k \times k$ matrix W^A, which stores a separate feature weight for each tag bigram st. This is still a log-linear model over taggings $\boldsymbol{t}$, where the st feature counts the number of times st appears in $\boldsymbol{t}$.

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 $W_{st}^A = \vec{\theta}_s^A \cdot \vec{\theta}_t^A$, or equivalently $W^A = \Theta^A (\Theta^A)^\top$. Again, this is no longer a log-linear model because it multiplies pairs of weights together

H.3 Context-Dependent Features

The CRF family (30) is not very expressive. It turns out that any distribution in that family could also have been obtained as the conditional distribution (24) of some HMM (Smith and Johnson (2007)). Thus, maximizing the conditional log-likelihood of this CRF is no better than discriminatively training an HMM (maximizing its conditional log-likelihood (25)). However, the last homework said:

> This is warmup for the next homework, where we'll unlock the full power of CRFs by incorporating many more features to help discriminate among taggings $\boldsymbol{t}$. The resulting

CRFs will be richer than HMMs, but will remain efficient thanks to their linear-chain structure.

Now we unlock the power. Let's change equation (30) to allow the potential functions to also depend on the sentence $\boldsymbol{w}$ and the position in it:

$$\tilde{p}(\boldsymbol{t} \mid \boldsymbol{w}) \propto \left(\prod_{i=1}^{n+1} \phi_{\mathrm{A}}(t_{i-1}, t_i, \boldsymbol{w}, i)\right) \cdot \left(\prod_{i=1}^{n} \phi_{\mathrm{B}}(t_i, w_i, \boldsymbol{w}, i)\right) \tag{44}$$

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) at position i in a tagging, it can look at the context in $\boldsymbol{w}$ of that transition or emission. Information from the whole sentence $\boldsymbol{w}$ can help inform our local tagging judgment. Crucially, because the CRF is *not* a generative model—see reading section E.2—the sentence $\boldsymbol{w}$ is fixed, and so the feature functions can look freely at it.

Equation (44) is called a linear-chain CRF, because the random variables that we are trying to predict—$T_1, T_2, \ldots, T_n$—are linked together in a chain by the ϕ_{A} potential functions, making each tag interdependent with the next one.[29] The ϕ_{B} potential functions link the tags to the words.[30]

H.4 Parameterization

What kind of context-dependent potential functions should we learn? We have lots of choices, but here is a straightforward approach using bidirectional RNNs.

The obvious way to generalize equation (31) along the lines of equation (42) is to build unnormalized log-linear functions for use in (44):

$$\phi_{\mathrm{A}}(s, t, \boldsymbol{w}, i) = \exp\left(\vec{\theta}^A \cdot \vec{f}^A(s, t, \boldsymbol{w}, i)\right) \qquad \phi_{\mathrm{B}}(t, w, \boldsymbol{w}, i) = \exp\left(\vec{\theta}^B \cdot \vec{f}^B(t, w, \boldsymbol{w}, i)\right) \tag{45}$$

Here each $\vec{f}$ function returns a feature vector. We could hand-craft those features using our linguistic intuition, but it's easier to let a neural network discover the relevant features. That is, $\vec{f}$ should construct a vector embedding of its 4 arguments—the transition or emission *in context*.

Here's one way to do that, using the entire context. First, let $\vec{h}_j$ be a vector embedding of the sentence prefix $w_1 \cdots w_j$, and let $\vec{h}'_j$ be a vector embedding of the sentence suffix $w_{j+1} \cdots w_n$. Crucially, these depend only on $\boldsymbol{w}$ and not on $\boldsymbol{t}$. The prefix and suffix can contain any number of tokens, so a good way to embed them is to use left-to-right and right-to-left RNNs.

$$\vec{h}_j = \sigma(M \left[1; \vec{h}_{j-1}; \vec{w}_j\right]) \in \mathbb{R}^d \qquad\qquad \vec{h}'_{j-1} = \sigma(M' \left[1; \vec{w}_j; \vec{h}'_j\right]) \in \mathbb{R}^d \tag{46}$$

---

[29]I've written the CRF in the form (44) for similarity to the HMM (1). But it's actually reasonable to restrict the first product in (44) to range over just the $n-1$ bigrams of the sentence proper (that is, $\prod_{i=2}^{n}$ rather than $\prod_{i=1}^{n+1}$). This would drop the bigram factors that involve BOS and EOS—we don't really need those symbols at all in a CRF! In contrast to the generative HMM, we no longer need to generate EOS to determine the number of tags, because that is determined by the input as $n = |\boldsymbol{w}|$. And because the potential functions in equation (44) can look at context, they can see that t_1 is the first tag—and evaluate whether it is a good first tag—when they score (t_1, t_2) and (t_1, w_1). Similarly, they can see that t_n is the last tag when they score (t_{n-1}, t_n) and (t_n, w_n). They don't need BOS and EOS for this. (Note: If you would like to drop those two bigram factors in your CRF, but still share your dynamic programming code with the HMM, you could keep them but treat them as having value 1 in the CRF version.)

[30]Actually 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.

where you pick the RNN dimensionality d. For the base cases, a simple choice is to define $\vec{h}_{-1} = \vec{0}$ (just before $w_0 = \text{BOSW}$) and $\vec{h}'_{n+1} = \vec{0}$ (just after $w_{n+1} = \text{EOSW}$) rather than learning these vectors as parameters. 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). Notice that it is $\vec{h}_0$ and $\vec{h}'_n$ that embed the empty prefix and empty suffix (no words).[31]

The recurrent equations (46) are efficient because they embed all $O(n)$ prefixes and suffixes in $O(n)$ total time. They're rather similar to the computations of $\vec{\alpha}$ and $\vec{\beta}$, particularly the versions in footnotes 1 and 17. The most important difference 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 ("elementwise") to each element of its argument. The square-bracket-semicolon notation denotes concatenation of multiple column vectors into a column vector. The matrices M and M' are parameters of the model.

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) (for all i values needed by equation (44)):

$$\vec{f}^A(s, t, \boldsymbol{w}, i) = \sigma(U^A \left[1; \vec{h}_{i-2}; \vec{s}; \vec{t}; \vec{h}'_i \right]) \tag{47}$$

$$\vec{f}^B(t, w, \boldsymbol{w}, i) = \sigma(U^B \left[1; \vec{h}_{i-1}; \vec{t}; \vec{w}; \vec{h}'_i \right]) \tag{48}$$

For example, equation (47) is encoding a tuple of the form $(\text{prefix}, s, t, \text{suffix})$. That is, $\vec{h}_{i-2}$ encodes the prefix before the tag bigram st ending at position i, and $\vec{h}'_i$ encodes the suffix after that tag bigram token. So equation (47) encodes this tag bigram token *in context*.[32]

In each of equations (46)–(48), 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.

H.5 Summary of Neural Featurization

The last homework discussed training an HMM discriminatively, and then switched from an HMM to a CRF, which is always trained discriminatively. One advantage of discriminative training is

---

[31]If you use the simplification in footnote 29, you will only need the embeddings $\vec{h}_0, \ldots, \vec{h}_n$ and $\vec{h}'_n, \ldots, \vec{h}'_0$ in equations (47) and (48) below. Without that simplification, you will unfortunately also need $\vec{h}_{-1}$ and $\vec{h}'_{n+1}$.

[32]Of 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 $\vec{h}_{i-2}$ and $\vec{h}'_i$, you could use $\vec{h}_i$ and $\vec{h}'_{i-2}$, 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 (47), in which case the resulting ϕ_B in equation (45) does not consider context any more than it did in equation (31). 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

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

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 encoding $\vec{f}(\boldsymbol{w}, i)$ of position i can be reused for all the possible transitions and emissions at position i. We no longer need to embed each transition and each emission separately in context. So this architecture is a little more efficient than equation (45), though also a little less flexible.

that it is more focused on the actual prediction task. But in this homework, we turn to another advantage of discriminative training of $p(\boldsymbol{t} \mid \boldsymbol{w})$: it allows more complex models like equation (44), which can freely examine the entirety of $\boldsymbol{w}$ when figuring out how to tag w_i. This is because it doesn't have the responsibility of generating $\boldsymbol{w}$ in any particular order.

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 $\boldsymbol{w}$. But in practice, such features rarely looked at much of $\boldsymbol{w}$ (usually just the local context around the tag or tag bigram).

Our more modern approach just uses a biRNN (46) to scan $\boldsymbol{w}$ for useful patterns. These patterns combine with the tags or tag bigrams through a further neural network (47)–(48) to define the feature vectors $\vec{f}$. This whole system is trained via equation (25) so that the neural machinery is encouraged to extract whatever features turn out to be useful for the tagging task.

We also made use of pretrained word embeddings, and noted that they could optionally be fine-tuned as part of the system too.

Along the way, we lost the log-linear property, since the CRF's log-potential functions are no longer linear functions of the parameters. We are now doing log-*non*linear modeling (also known as "deep learning").

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 (47)–(48) (outside the σ), and then training the potentials (45) will learn $\vec{\theta}$-weights for them too.

H.6 Bonus Reading: Why Not HMMs?

What would happen if you tried to use the CRF's complicated features in an HMM? That is, could you make the HMM's transition and emission probabilities depend on both left and right context in w?

No, because then your formula for $p(\boldsymbol{t}, \boldsymbol{w})$ would no longer be a valid use of chain rule with backoff. It would have w_i and w_j depending circularly on each other. That's not allowed: in general you wouldn't have $\sum_{\boldsymbol{t},\boldsymbol{w}} p(\boldsymbol{t}, \boldsymbol{w}) = 1$ anymore!

However, there's a cool trick that most people don't know. It *is* possible to generalize an HMM so that its transition and emission probabilities depend freely on *left* context in w: transition $p(t_i \mid t_{i-1}, w_1, \ldots, w_{i-1})$, emission $p(w_i \mid t_i, w_1, \ldots, w_{i-1})$. Since the probabilities don't depend on *right* context, this avoids the circularity. It's still the chain rule with backoff: we just no longer back off from the words to the left.

To extend the HMM in this way, you do need some reasonable model of these fancier probabilities, e.g., a neural net that uses an RNN to encode $w_1, \ldots, w_{i-1}$. You shouldn't simply use count ratios, as they would often be 0/0 in practice.

But aside from that, this extension to the HMM has no apparent disadvantages! The Viterbi algorithm and the forward-backward algorithm remain efficient as long as the transition and emission probabilities at position i don't depend on any tags before t_{i-1}. So you can still find the Viterbi tagging and compute posterior marginals, just as for a standard HMM or a CRF.

Because it's still a generative model (like an standard HMM but not a CRF), you can still compute $p(w)$ when t is unobserved. That allows you to use your model as a language model and to train it on untagged sentences w.

You can even apply the same trick to a PCFG, replacing $p(\text{RHS} \mid \text{LHS})$ with $p(\text{RHS} \mid \text{LHS}, \text{words to left of LHS})$.

I   CRF Algorithms

Now that our CRF uses context-dependent and non-linear features, does anything change?

I.1   Dynamic Programming with Context-Dependent Features

To compute the CRF's conditional log-likelihood (25), the expensive part is computing the normalizing constant $Z(\boldsymbol{w})$ (28). Since this sums over exponentially many taggings, we would like to still do this by running some small variant of the forward algorithm (Algorithm 1 in the previous handout).

Remember that the core of the forward algorithm is the successive update of $\vec{\alpha}$ vectors by equation (12). To deal with context-dependent features, we'll just replace that equation with a slight variant:

$$\vec{\alpha}(j) = \left(\vec{\alpha}(j-1) \cdot A^{(j)}\right) \odot \vec{b}^{(j)} \tag{49}$$

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

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

$$A_{st}^{(j)} = \phi_{\mathrm{A}}(s, t, \boldsymbol{w}, j) \qquad\qquad b_t^{(j)} = \phi_{\mathrm{B}}(t, w_j, \boldsymbol{w}, j) \tag{50}$$

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 $\boldsymbol{w}$ 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. (The model still considers 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 (29), taking advantage of the linear chain structure of equation (44).

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 modified 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 a sentence or minibatch as soon as you receive it, and then look each matrix up when you need to use it.

I.2   Using Back-Propagation

At the end of the previous homework, you switched from EM training to SGD training in order to train a CRF. In this homework, you will continue with SGD training, but switch to using back-propagation to compute the gradient.

In the previous homework (reading section E.4), you "manually" computed the gradient of the regularized CRF objective (35), following the usual recipe for log-linear models. In particular, the conditional log-likelihood (32) had a gradient that was a difference between observed and expected counts. You found the expected counts by using the forward-backward algorithm (Algorithm 4).

However, back-propagation is a more general and automatic way of computing gradients! In the new log-*non*linear models on this homework, the gradient is no longer a simple difference between observed and expected counts. But back-prop will still be able to compute it efficiently, sending a gradient signal back through the recurrent computations of the RNNs.

Due to the nonlinearities, the objective function may have multiple local optima even for fully supervised learning. (Unsupervised learning introduced its own nonlinearities through the summation over different taggings—this uses the nonlinear operator `logaddexp` on the log-probabilities—which is why the incomplete-data log-likelihood had multiple local optima for EM to get stuck in.)

Computing the regularized CRF objective (35) isn't too hard—as noted above, the conditional log-likelihood (32) can be computed with just the forward algorithm. If you compute it in PyTorch, you can then also use PyTorch's back-propagation method to find its gradient.[33]

On this homework, you will compute gradients of your objective function using PyTorch's `backward` method (which is general back-propagation, not the HMM backward algorithm). Thus, make sure your log-likelihood function is computed using PyTorch operations.[34]

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.

- 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.

J Vectorization and GPUs for speed

Remember that you'll be much faster if you can avoid Python loops in favor of PyTorch's vectorized computations (as discussed on HW2 and HW3).

---

[33]So internally, when back-propagation is applied to the forward algorithm, it must do something like Algorithm 4, using the backward algorithm to secretly compute expected counts. Indeed, it's possible to extract those expected counts: check out the bonus reading section K.

[34]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.

This also allows you to speed up your experiments by running on a GPU. You are again free to try this, as you were in Homework 3 (please see the Kaggle section of its reading handout). We have again provided a Kaggle dataset for you.

For example, don't compute the elements of (50) one at a time. The tensor A has 3 dimensions, indexed by s, t, j; can you compute all its entries in parallel? Note that those entries are defined by (45) and depend on feature vectors (47) and (48). So you'll want to compute all of those in parallel too. Start with matrices H and H' that collect up all the of biRNN hidden state vectors (46) in their rows or columns.

Better yet, if you are running on a minibatch, can you work in parallel on all M sentences in the minibatch? This expands most of the tensors with an extra dimension, indexed by the sentence number $0 \leq m < M$. For example, you'd have a 4-dimensional tensor that stacks up the A tensors for the M different sentences. You'd also run your RNNs on all sentences in parallel. Note that you'd have to modify methods like `log_forward` and `viterbi_tagging` to take a `List[Sentence]` instead of just a single `Sentence`. (One tricky issue is that sentences in the same minibatch may have different lengths, so you need to "pad" shorter sentences with extra EOS symbols. The details of how to do or skip computations on those extra symbols is left as an exercise.)

More vectorization may speed you up even on a CPU, and it also allows you to make better use of GPU parallelism. Ideally, you won't leave any of the GPU processors unused. Thus, you may want to start by trying a very large minibatch size that gives you an out-of-memory runtime error, and reduce it only as much as needed to make that error go away.

K   Back-Propagation As An Alternative to the Backward Pass in HMMs and CRFs

As long as we're now using backprop (reading section I.2), this section suggests an interesting alternative way to implement something in the homeworks.

Suppose you want fractional counts from an HMM, either to use in EM or for some other purpose such as posterior decoding. It's not actually necessary to implement the forward-backward algorithm. It turns out that the back-propagation algorithm automatically computes the same quantities. In fact, as footnote 33 mentioned, 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:

$$\beta_t(j) = \frac{\partial Z}{\partial \alpha_t(j)} \tag{51}$$

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

$$c(s, t) = \frac{\partial \log Z}{\partial \log A_{st}} \qquad\qquad c(t, w) = \frac{\partial \log Z}{\partial \log B_{tw}} \tag{52}$$

where $c(\ldots)$ 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 (52) work because the HMM probability $p(\boldsymbol{t}, \boldsymbol{w})$ (equation (1)) is a log-linear function of the counts of the different types of transitions and emissions in the tagged

sentence $(\boldsymbol{t}, \boldsymbol{w})$, where the weights of these transition and emission types are log-probabilities given in the A and B matrices. 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 (52) comes from.

The same trick can be used when the A and B hold the potentials of a CRF, as introduced in reading section E.3 in the last assignment.

And when the potential functions vary by position j (reading section H.3), you can just take the partial derivatives with respect to $\log A^{(j)}$ and $\log b^{(j)}$ from reading section I.1. These will give you the fractional probabilities of the different tags at position j, rather than summing over all positions as equation (52) did.[35]

This connection between gradients and expected counts also holds 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—focusing on the inside-outside and forward-backward algorithms.

K.1 Computational Details

If you computed $\log Z$ from versions of A, B that were represented in logspace (such as lA, lB in reading section C.2, or W^A, W^B in equation (31)), then calling back-propagation on $\log Z$ will directly compute (52) for you.

First compute $\log Z$, making sure that gradients are being tracked.[36] 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; these will be tensors of gradients. The `.grad` attributes usually only stay available at leaf nodes of the computation graph (that is, parameters), but `retain_grad` lets you them at intermediate nodes as well.

Alternatively, if you only have easy access to the probabilities or potentials A and B, and not their logs, then you can use those instead, by making the following change of variables in equation (52):

$$
\begin{aligned}
c(s,t) &= \frac{\partial \log Z}{\partial \log A_{st}} & c(t,w) &= \frac{\partial \log Z}{\partial \log B_{tw}} \\
&= \frac{\partial \log Z}{\partial A_{st}} \cdot \frac{\partial A_{st}}{\partial \log A_{st}} = \frac{\partial \log Z}{\partial A_{st}} \cdot A_{st} & &= \frac{\partial \log Z}{\partial B_{tw}} \cdot \frac{\partial B_{tw}}{\partial \log B_{tw}} = \frac{\partial \log Z}{\partial B_{tw}} \cdot B_{tw} \quad (53)
\end{aligned}
$$

L Neuralization and SGD for HMMs

While this homework has focused on CRFs, it is also possible to use neural nets to define the transition and emission probabilities of an HMM. In this case, we train the parameters of those neural nets.

---

[35]If you want the fractional probabilities at j but A and B do not vary by position, you can make position-specific versions by defining $A^{(j)} = A + 0$ and $B^{(j)} = B + 0$ for each j. Then use these position-specific versions in the computation and compute gradients with respect to them.

[36]Warning: Gradient 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).

There is unfortunately not a role for the biRNN in an HMM. Because the HMM is generative, there seems to be no way to introduce RNNs or other context features without breaking the dynamic programming structure that makes HMMs efficient.[37]

However, the HMM transition and emission probabilities $p_A(t \mid s)$ and $p_B(w \mid t)$ can be defined using embeddings or other features of the tags and words, just as we did earlier for CRFs (reading sections H.1 and H.2). Chiu & Rush (2020) show that HMMs are actually surprisingly good language models—competitive with RNNs—if you use tag embeddings to give them a huge number of states!

L.1 Training a Neuralized HMM

You can train your neuralized HMM by either SGD or by EM. Each time you update the underlying parameters that control the transition and emission probabilities, you need to recompute the probabilities (that is, the matrices A, B).

To train by SGD, you simply try to maximize the HMM's objective function, namely the incomplete-data log-likelihood (14). This adjusts the underlying parameters. You might want to use an L_2 regularizer on these parameters.

To train by EM, you do each E step as usual, and then use SGD for the M step. A full M step would maximize the model's expected log-likelihood (18). However, it is actually enough to do a **generalized M step** that merely increases (18)—typically by taking just one or a few gradient steps. In this case you can even use mini-batches.

In both cases, you need to use back-propagation to find the gradient. You also need to choose some hyperparameters: a mini-batch size and a stepsize schedule for SGD. You were able to avoid the hyperparameters in the last assignment because for the simple non-neuralized HMM, you could use EM with a closed-form M step (17)—no SGD was required.

L.2 HMM Parameterization for SGD

For an HMM, you 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. If you treated A or B as the parameters of the model, then a direct gradient step to adjust them might violate these constraints.

Even without neuralization, there is an obvious solution. Take the underlying model parameters to be 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. (All rows can be computed at once with a single call to PyTorch's `softmax`.) This is called a **softmax parameterization** of A, B—it's like the logarithmic parameterization of A, B in the CRF case (equation (31)), but with normalization. In other words,

$$p_{\mathrm{A}}(t \mid s) = \frac{\exp W_{st}^A}{\sum_{t'} \exp W_{st'}^A} \qquad\qquad p_{\mathrm{B}}(w \mid t) = \frac{\exp W_{tw}^B}{\sum_{w'} \exp W_{tw'}^B} \qquad (54)$$

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 given by the corresponding matrix element.

---

[37]Again, that is a reason to prefer CRFs for tagging—they allow a more expressive model of $p(\boldsymbol{t} \mid \boldsymbol{w})$. Because they are only a conditional model that conditions on a fixed $\boldsymbol{w}$, they can extract rich features such as biRNN features from that $\boldsymbol{w}$. They do not have to produce a distribution over all values of $\boldsymbol{w}$.

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

The neuralization idea is that W^A and W^B can in turn be parameterized in terms of word and/or tag embeddings, *exactly* as in the CRF case (reading sections H.1–H.2). In that case, SGD will update the underlying parameters by following their gradient. Then you will recompute W^A, W^B, which leads to recomputing A, B via equation (54).

Boundary symbols. Omitting the BOSW and EOSW columns makes it easy to produce each row as a softmax. It also means you don't have to worry about the fact that BOSW and EOSW don't have embeddings in the lexicon.

Scaling trick. If you are avoiding underflow through the scaling trick (reading section C.1), then κ_j is just some *constant* that you picked. The choice of κ_j does not actually affect the results—you are just dividing by it in one place and multiplying by it in another.

So you don't have to compute the gradient of κ_j (which would be 0). You can make it be an ordinary float rather than a tensor. Then even if you happened to choose it by some calculation that depends on your forward computation, PyTorch doesn't know that, and will not waste time back-propagating through that computation to determine how to improve κ_j.

Logspace computation. If you are avoiding underflow by running the forward algorithm in logspace (reading section C.2), you may prefer to skip equation (54) and directly construct logspace versions lA, lB of A, B. This can be done simply by using `logsoftmax` in PyTorch, instead of `softmax` as equation (54) did.[38]

Mathematically, the entries of lA, lB are logarithms of (54), which can be rewritten as

$$\log p_{\mathrm{A}}(t \mid s) = W^A_{st} - \mathrm{logsumexp}_{t'} W^A_{st'} \qquad \log p_{\mathrm{B}}(w \mid t) = W^B_{tw} - \mathrm{logsumexp}_{w'} W^B_{tw'} \tag{55}$$

In other words, each row of lA is obtained by shifting the corresponding row of W^A by its logsumexp, and similarly for lB. ($\mathrm{logsumexp}_{t'}$ is a summation operator just like $\sum_{t'}$, but it adds up quantities represented in log-space and gives a result represented in log-space. Thus, $\mathrm{logsumexp}_{t'} \cdots$ returns $\log \sum_{t'} \exp \cdots$, just as $\mathrm{logaddexp}(x, y)$ returned $\log(\exp x + \exp y)$ in reading section C.2. Underflow-resistant `logsumexp` and `logaddexp` operators are available in PyTorch.)

L.3   Word and Tag Frequencies

There is one potential problem with the softmax parameterization of B, namely that the word embeddings may not contain information about word frequency. The HMM is a generative model that needs to generate the words. But $p_B(w \mid t)$ might be just as high for a rare word as for a frequent word, if those words have similar embeddings!

Fortunately, this shouldn't actually be a problem for the tagging task, because if $p_B(w \mid t)$ 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.

---

[38]It's better not to use `log(softmax(...))`, since back-propagating through that can encounter the $\log 0$ bug mentioned in reading section I.2.

But if you want your HMM to be able to generate sentences as well as tag them, then you can fix the problem by defining

$$W_{tw}^B = \vec{\theta}_t \cdot \vec{w} + \log \hat{p}(w) \tag{56}$$

where $\hat{p}(w)$ is estimated by counting words in the training data (with some simple smoothing). This means that $p(w \mid t)$ will be defined as proportional to $\hat{p}(w) \cdot \exp(\vec{\theta}_t \cdot \vec{w})$. Since Bayes' Theorem says that $p(w \mid t) \propto p(w) \cdot p(t \mid w)$, this means that really we're using $\exp \vec{\theta}_t \cdot \vec{w}$ to model $p(t \mid w)$, 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 $p(w \mid t)$ 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 $\vec{w}$ to some extent, then perhaps it will learn $\gamma < 1$.

You can take this even farther, and augment $\vec{w}$ with information about the frequency with which w had various tags in supervised training data, such $\log \hat{p}(t \mid w)$. This vector of logprobs is itself a kind of simple syntactic embedding of the word. It should be useful for CRFs as well as for HMMs.