17 | Structured Prediction

Learning Objectives:

- Recognize when a problem should be solved using a structured prediction technique.
- Implement the structured perceptron algorithm for sequence labeling.
- Map "argmax" problems to integer linear programs.
- Augment the structured perceptron with losses to derive structured SVMs.

Dependencies:

It is often the case that instead of predicting a *single* output, you need to predict multiple, correlated outputs simultaneously. In natural language processing, you might want to assign a syntactic label (like noun, verb, adjective, etc.) to words in a sentence: there is clear correlation among these labels. In computer vision, you might want to label regions in an image with object categories; again, there is correlation among these regions. The branch of machine learning that studies such questions is **structured prediction**.

In this chapter, we will cover two of the most common algorithms for structured prediction: the structured perceptron and the structured support vector machine. We will consider two types of structure. The first is the "sequence labeling" problem, typified by the natural language processing example above, but also common in computational biology (labeling amino acids in DNA) and robotics (labeling actions in a sequence). For this, we will develop specialized prediction algorithms that take advantage of the sequential nature of the task. We will also consider more general structures beyond sequences, and discuss how to cast them in a generic optimization framework: **integer linear programming** (or **ILP**).

The general framework we will explore is that of *jointly scoring input/output configurations*. We will construct algorithms that learn a function $s(\boldsymbol{x}, \hat{\boldsymbol{y}})$ (s for "score"), where $\boldsymbol{x}$ is an input (like an image) and $\hat{\boldsymbol{y}}$ is some predicted output (like a segmentation of that image). For any given image, there are a *lot* of possible segmentations (i.e., a lot of possible $\hat{\boldsymbol{y}}$s), and the goal of s is to rank them in order of "how good" they are: how compatible they are with the input $\boldsymbol{x}$. The most important thing is that the scoring function s ranks the *true* segmentation $\boldsymbol{y}$ higher than any other imposter segmentation $\hat{\boldsymbol{y}}$. That is, we want to ensure that $s(\boldsymbol{x}, \boldsymbol{y}) > s(\boldsymbol{x}, \hat{\boldsymbol{y}})$ for all $\hat{\boldsymbol{y}} \neq \boldsymbol{y}$. The main challenge we will face is how to do this *efficiently*, given that there are so many imposter $\hat{\boldsymbol{y}}$s.

17.1 Multiclass Perceptron

In order to build up to structured problems, let's begin with a simplified by pedagogically useful stepping stone: multiclass classification with a perceptron. As discussed earlier, in multiclass classification we have inputs $\boldsymbol{x} \in \mathbb{R}^D$ and output labels $y \in \{1, 2, \ldots, K\}$. Our goal is to learn a scoring function s so that $s(x, y) > s(x, \hat{y})$ for all $\hat{y} \neq y$, where y is the true label and $\hat{y}$ is an imposter label. The general form of scoring function we consider is a linear function of a joint feature vector $\boldsymbol{\phi}(\boldsymbol{x}, y)$:

$$s(\boldsymbol{x}, y) = \boldsymbol{w} \cdot \boldsymbol{\phi}(\boldsymbol{x}, y) \tag{17.1}$$

Here, the features $\boldsymbol{\phi}(\boldsymbol{x}, y)$ should denote how "compatible" the input $\boldsymbol{x}$ is with the label y. We keep track of a single weight vector $\boldsymbol{w}$ that learns how to weigh these different "compatibility" features.

A natural way to represent $\boldsymbol{\phi}$, if you know nothing else about the problem, is an *outer product* between $\boldsymbol{x}$ and the label space. This yields the following representation:

$$\boldsymbol{\phi}(\boldsymbol{x}, k) = \Big\langle \underbrace{0, 0, \ldots, 0}_{D(k-1) \text{ zeros}}, \underbrace{\boldsymbol{x}}_{\in \mathbb{R}^D}, \underbrace{0, 0, \ldots, 0}_{D(K-k) \text{ zeros}} \Big\rangle \in \mathbb{R}^{DK} \tag{17.2}$$

In this representation, $\boldsymbol{w}$ effectively encodes a separate weight for every feature/label pair.

How are we going to learn $\boldsymbol{w}$? We will start with $\boldsymbol{w} = \boldsymbol{0}$ and then process each input one at a time. Suppose we get an input $\boldsymbol{x}$ with gold standard label y. We will use the current scoring function to predict a label. In particular, we will predict the label $\hat{y}$ that maximizes the score:

$$\hat{y} = \operatorname*{argmax}_{\hat{y} \in [1,K]} s(\boldsymbol{x}, \hat{y}) \tag{17.3}$$

$$= \operatorname*{argmax}_{\hat{y} \in [1,K]} \boldsymbol{w} \cdot \boldsymbol{\phi}(\boldsymbol{x}, \hat{y}) \tag{17.4}$$

If this predicted output is correct (i.e., $\hat{y} = y$), then, per the normal perceptron, we will do nothing. Suppose that $\hat{y} \neq y$. This means that the score of $\hat{y}$ is greater than the score of y, so we want to update $\boldsymbol{w}$ so that the score of $\hat{y}$ is decreased and the score of y is increased. We do this by:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \boldsymbol{\phi}(\boldsymbol{x}, y) - \boldsymbol{\phi}(\boldsymbol{x}, \hat{y}) \tag{17.5}$$

To make sure this is doing what we expect, let's consider what would happen if we computed scores under the *updated* value of $\boldsymbol{w}$. To make the notation clear, let's say $\boldsymbol{w}^{(\text{old})}$ are the weights before update, and

Algorithm 39 MULTICLASSPERCEPTRONTRAIN(**D**, *MaxIter*)

```
w ← 0                                              // initialize weights
for iter = 1 ... MaxIter do
    for all (x,y) ∈ D do
        ŷ ← argmax_k w · φ(x,k)                    // compute prediction
        if ŷ ≠ y then
            w ← w + φ(x,y) − φ(x,ŷ)                // update weights
        end if
    end for
end for
return w                                           // return learned weights
```

$\boldsymbol{w}^{(\text{new})}$ are the weights after update. Then:

$$\boldsymbol{w}^{(\text{new})} \cdot \boldsymbol{\phi}(\boldsymbol{x}, y) \tag{17.6}$$

$$= \Big(\boldsymbol{w}^{(\text{old})} + \boldsymbol{\phi}(\boldsymbol{x}, y) - \boldsymbol{\phi}(\boldsymbol{x}, \hat{y})\Big) \cdot \boldsymbol{\phi}(\boldsymbol{x}, y) \tag{17.7}$$

$$= \underbrace{\boldsymbol{w}^{(\text{old})} \cdot \boldsymbol{\phi}(\boldsymbol{x}, y)}_{\text{old prediction}} + \underbrace{\boldsymbol{\phi}(\boldsymbol{x}, y) \cdot \boldsymbol{\phi}(\boldsymbol{x}, y)}_{\geq 0} - \underbrace{\boldsymbol{\phi}(\boldsymbol{x}, \hat{y}) \cdot \boldsymbol{\phi}(\boldsymbol{x}, y)}_{=0} \tag{17.8}$$

Here, the first term is the old prediction. The second term is of the form $\boldsymbol{a} \cdot \boldsymbol{a}$ which is non-negative (and, unless $\boldsymbol{\phi}(\boldsymbol{x}, y)$ is the zero vector, positive). The third term is the dot product between $\boldsymbol{\phi}$ for two different labels, which by definition of $\boldsymbol{\phi}$ is zero (see Eq (17.2)).

? Verify the score of $\hat{y}$, $\boldsymbol{w}^{(\text{new})} \cdot \boldsymbol{\phi}(\boldsymbol{x}, \hat{y})$, decreases after an update, as we would want.

This gives rise to the updated multiclass perceptron specified in Algorithm 17.1. As with the normal perceptron, the generalization of the multiclass perceptron increases dramatically if you do weight averaging.

An important note is that MulticlassPerceptronTrain is actually more powerful than suggested so far. For instance, suppose that you have three categories, but believe that two of them are tightly related, while the third is very different. For instance, the categories might be {music, movies, oncology}. You can encode this relatedness by defining a feature expansion $\boldsymbol{\phi}$ that reflects this:

$$\boldsymbol{\phi}(\boldsymbol{x}, \text{music}) = \langle \boldsymbol{x}, \boldsymbol{0}, \boldsymbol{0}, \boldsymbol{x} \rangle \tag{17.9}$$

$$\boldsymbol{\phi}(\boldsymbol{x}, \text{movies}) = \langle \boldsymbol{0}, \boldsymbol{x}, \boldsymbol{0}, \boldsymbol{x} \rangle \tag{17.10}$$

$$\boldsymbol{\phi}(\boldsymbol{x}, \text{oncology}) = \langle \boldsymbol{0}, \boldsymbol{0}, \boldsymbol{x}, \boldsymbol{0} \rangle \tag{17.11}$$

This encoding is identical to the normal encoding in the first three positions, but includes an extra copy of the features at the end, shared between music and movies. By doing so, if the perceptron wants to learn something common to music and movies, it can use this final shared position.

? Suppose you have a *hierarchy* of classes arranged in a tree. How could you use that to construct a feature representation. You can think of the music/movies/oncology example as a binary tree: the left branch of the root splits into music and movies; the right branch of the root is just oncology.

17.2 Structured Perceptron

Let us now consider the sequence labeling task. In sequence labeling, the outputs are themselves variable-length vectors. An input/output pair (which must have the same length) might look like:

$$\boldsymbol{x} = \text{“ monsters eat tasty bunnies “} \tag{17.12}$$

$$\boldsymbol{y} = \quad \text{noun verb adj noun} \tag{17.13}$$

To set terminology, we will refer to the *entire sequence* $\boldsymbol{y}$ as the "output" and a single label within $\boldsymbol{y}$ as a "label". As before, our goal is to learn a scoring function that scores the true output sequence $\boldsymbol{y}$ higher than any imposter output sequence.

As before, despite the fact that $\boldsymbol{y}$ is now a vector, we can *still* define feature functions over the *entire* input/output pair. For instance, we might want to count the number of times "monsters" has been tagged as "noun" in a given output. Or the number of times "verb" is followed by "noun" in an output. Both of these are features that are likely indicative of a *correct* output. We might also count the number of times "tasty" has been tagged as a verb (probably a negative feature) and the number of times two verbs are adjacent (again, probably a negative feature).

More generally, a very standard set of features would be:

- the number of times word w has been labeled with tag l, for all words w and all syntactic tags l
- the number of times tag l is adjacent to tag l' in the output, for all tags l and l'

The first set of features are often called **unary features**, because they talk only about the relationship between the input (sentence) and a *single* (unit) label in the output sequence. The second set of features are often called **Markov features**, because they talk about adjacent labels in the output sequence, which is reminiscent of Markov models which only have short term memory.

Note that for a given input $\boldsymbol{x}$ of length L (in the example, $L = 4$), the number of possible outputs is K^L, where K is the number of syntactic tags. This means that the number of possible outputs *grows exponentially* in the length of the input. In general, we write $\mathcal{Y}(\boldsymbol{x})$ to mean "the set of all possible structured outputs for the input $\boldsymbol{x}$". We have just seen that $|\mathcal{Y}(\boldsymbol{x})| = K^{\text{len}(\boldsymbol{x})}$.

Despite the fact that the inputs and outputs have variable length, the size of the *feature representation* is constant. If there are V words in your vocabulary and K labels for a given word, the the number of unary features is VK and the number of Markov features is K^2, so

Algorithm 40 StructuredPerceptronTrain($\mathbf{D}$, *MaxIter*)

1: $\boldsymbol{w} \leftarrow \mathbf{0}$ // initialize weights
2: **for** *iter* = 1 ... *MaxIter* **do**
3: **for all** $(\boldsymbol{x}, \boldsymbol{y}) \in \mathbf{D}$ **do**
4: $\hat{\boldsymbol{y}} \leftarrow \text{argmax}_{\hat{\boldsymbol{y}} \in \mathcal{Y}(\boldsymbol{x})} \boldsymbol{w} \cdot \phi(\boldsymbol{x}, \hat{\boldsymbol{y}})$ // compute prediction
5: **if** $\hat{\boldsymbol{y}} \neq \boldsymbol{y}$ **then**
6: $\boldsymbol{w} \leftarrow \boldsymbol{w} + \phi(\boldsymbol{x}, \boldsymbol{y}) - \phi(\boldsymbol{x}, \hat{\boldsymbol{y}})$ // update weights
7: **end if**
8: **end for**
9: **end for**
10: **return** $\boldsymbol{w}$ // return learned weights

the total number of features is $K(V + K)$. Of course, more complex feature representations are possible and, in general, are a good idea. For example, it is often useful to have unary features of neighboring words like "the number of times the word immediately preceding a verb was 'monsters'."

Now that we have a fixed size feature representation, we can develop a perceptron-style algorithm for sequence labeling. The core idea is the same as before. We will maintain a *single* weight vector $\boldsymbol{w}$. We will make predictions by choosing the (entire) output sequence $\hat{\boldsymbol{y}}$ that maximizes a score given by $\boldsymbol{w} \cdot \phi(\boldsymbol{x}, \hat{\boldsymbol{y}})$. And if this output sequence is incorrect, we will adjust the weights *word* the correct output sequence $\boldsymbol{y}$ and away from the incorrect output sequence $\hat{\boldsymbol{y}}$. This is summarized in Algorithm 17.2

You may have noticed that Algorithm 17.2 for the structured perceptron is *identical* to Algorithm 17.1, aside from the fact that in the multiclass perceptron the argmax is over the K possible classes, while in the structured perceptron, the argmax is over the K^L possible output sequences!

The only difficulty in this algorithm is in line 4:

$$\hat{\boldsymbol{y}} \leftarrow \underset{\hat{\boldsymbol{y}} \in \mathcal{Y}(\boldsymbol{x})}{\text{argmax}}\, \boldsymbol{w} \cdot \phi(\boldsymbol{x}, \hat{\boldsymbol{y}}) \tag{17.14}$$

In principle, this requires you to search over K^L possible output sequences $\hat{\boldsymbol{y}}$ to find the one that maximizes the dot product. Except for very small K or very small L, this is computationally infeasible. Because of its difficulty, this is often refered to as the **argmax problem** in structured prediction. Below, we consider how to solve the argmax problem for sequences.

17.3 Argmax for Sequences

We now face an *algorithmic* question, not a machine learning question: how to compute the argmax in Eq 17.14 efficiently. In general,

this is not possible. However, under somewhat restrictive assumptions about the form of our features ϕ, we can solve this problem efficiently, by casting it as the problem of computing a maximum weight path through a specifically constructed lattice. This is a variant of the Viterbi algorithm for hidden Markov models, a classic example of dynamic programming. (Later, in Section 17.6, we will consider argmax for more general problems.)

The key observation for sequences is that—so long as we restrict our attention to unary features and Markov features—the feature function ϕ decomposes over the input. This is easiest to see with an example. Consider the input/output sequence from before: $\boldsymbol{x}$ = "monsters eat tasty bunnies" and $\boldsymbol{y}$ = [noun verb adj noun]. If we want to compute the number of times "bunnies" is tagged as "noun" in this pair, we can do this by:

1. count the number of times "bunnies" is tagged as "noun" in the first three words of the sentence
2. add to that the number of times "bunnies" is tagged as "noun" in the final word

We can do a similar exercise for Markov features, like the number of times "adj" is followed by "noun".

However, we don't actually *need* these counts. All we need for computing the argmax sequence is the dot product between the weights $\boldsymbol{w}$ and these counts. In particular, we can compute $\boldsymbol{w} \cdot \boldsymbol{\phi}(\boldsymbol{x}, \boldsymbol{y})$ as the dot product on all-but-the-last word plus the dot product on the last word: $\boldsymbol{w} \cdot \boldsymbol{\phi}_{1:3}(\boldsymbol{x}, \boldsymbol{y}) + \boldsymbol{w} \cdot \boldsymbol{\phi}_4(\boldsymbol{x}, \boldsymbol{y})$. Here, $\boldsymbol{\phi}_{1:3}$ means "features for everything up to and including position 3" and $\boldsymbol{\phi}_4$ means "features for position 4."

More generally, we can write $\boldsymbol{\phi}(\boldsymbol{x}, \boldsymbol{y}) = \sum_{l=1}^{L} \boldsymbol{\phi}_l(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{\phi}_l(\boldsymbol{x}, \boldsymbol{y})$ only includes features about position l.[1] In particular, we're taking advantage of the associative law for addition:

[1] In the case of Markov features, we think of them as pairs that *end* at position l, so "verb adj" would be the active feature for ϕ_3.

$$\boldsymbol{w} \cdot \boldsymbol{\phi}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{w} \cdot \sum_{l=1}^{L} \boldsymbol{\phi}_l(\boldsymbol{x}, \boldsymbol{y}) \qquad \text{decomposition of structure} \qquad (17.15)$$

$$= \sum_{l=1}^{L} \boldsymbol{w} \cdot \boldsymbol{\phi}_l(\boldsymbol{x}, \boldsymbol{y}) \qquad \text{associative law} \qquad (17.16)$$

What this means is that we can build a graph like that in Figure **??**, with one verticle slice per time step (l 1 ... L).[2] Each *edge* in this graph will receive a weight, constructed in such a way that if you take a complete path through the lattice, and add up all the weights, this will correspond exactly to $\boldsymbol{w} \cdot \boldsymbol{\phi}(\boldsymbol{x}, \boldsymbol{y})$.

[2] A graph of this sort is called a **trellis**, and sometimes a **lattice** in the literature.

To complete the construction, let $\boldsymbol{\phi}_l(\boldsymbol{x}, \cdots \circ y \circ y')$ denote the *unary* features at position l together with the *Markov* features that end at

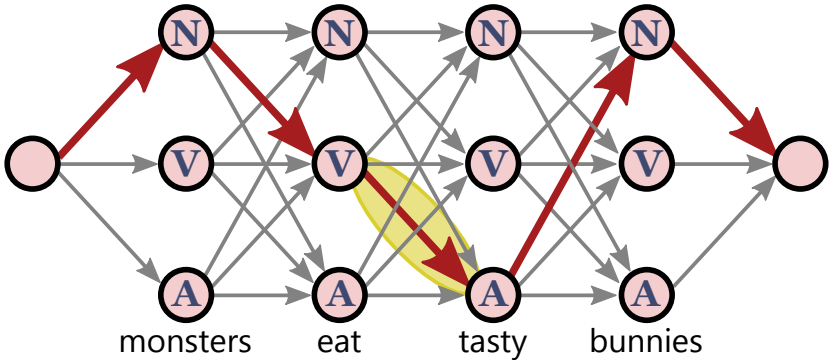

Figure 17.1: A picture of a trellis sequence labeling. At each time step l the corresponding word can have any of the three possible labels. Any path through this trellis corresponds to a unique labeling of this sentence. The gold standard path is drawn with bold red arrows. The highlighted edge corresponds to the edge between $l = 2$ and $l = 3$ for verb/adj as described in the text. That edge has weight $\boldsymbol{w} \cdot \boldsymbol{\phi}_3(\boldsymbol{x}, \cdots \circ \text{verb} \circ \text{adj})$.

position l. These features depend *only* on $\boldsymbol{x}$, y and y', and *not* any of the previous parts of the output.

For example, in the running example "monsters/noun eat/verb tasty/adj bunnies/noun", consider the edge between $l = 2$ and $l = 3$ going from "verb" to "adj". (Note: this is a "correct" edge, in the sense that it belongs to the ground truth output.) The features associated with this edge will be unary features about "tasty/adj" as well as Markov features about "verb/adj". The *weight* of this edge will be exactly the total score (according to $\boldsymbol{w}$) of those features.

Formally, consider an edge in the trellis that goes from time $l - 1$ to l, and transitions from y to y'. Set the weight of this edge to exactly $\boldsymbol{w} \cdot \boldsymbol{\phi}_l(\boldsymbol{x}, \cdots \circ y \circ y')$. By doing so, we guarantee that the sum of weights along any path through this lattice is exactly equal to the score of that path. Once we have constructed the graph as such, we can run any max-weight path algorithm to compute the highest scoring output. For trellises, this can be computed by the Viterbi algorithm, or by applying any of a number of path finding algorithms for more general graphs. A complete derivation of the dynamic program in this case is given in Section 17.7 for those who want to implement it directly.

The main benefit of this construction is that it is guaranteed to exactly compute the argmax output for sequences required in the structured perceptron algorithm, *efficiently*. In particular, it's runtime is $O(LK^2)$, which is an exponential improvement on the naive $O(K^L)$ runtime if one were to enumerate every possible output sequence. The algorithm can be naturally extended to handle "higher order" Markov assumptions, where features depend on triples or quadruples of the output. The trellis becomes larger, but the algorithm remains essentially the same. In order to handle a length M Markov features, the resulting algorithm will take $O(LK^M)$ time. In practice, it's rare that $M > 3$ is necessary or useful.

17.4 Structured Support Vector Machines

In Section 8.7 we saw the support vector machine as a very useful general framework for binary classification. In this section, we will develop a related framework for structured support vector machines. The two main advantages of structured SVMs over the structured perceptron are (1) it is regularized (though averaging in structured perceptron achieves a similar effect) and (2) we can incorporate more complex loss functions.

In particular, one suboptimal thing about the structured perceptron is that all errors are consider equally bad. For structured problems, we often have much more nuanced and elaborate loss functions that we want to optimize. Even for sequence labeling, it is typically far worse to label every word incorrectly than to just label one word incorrectly. It is very common to use **Hamming loss** as a general loss function for structured prediction. Hamming loss simply counts: of all the predictions you made, how many were incorrect? For sequence labeling, it is:

$$\ell^{(\text{Ham})}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \sum_{l=1}^{L} \mathbf{1}[y_l \neq \hat{y}_l] \tag{17.17}$$

In order to build up to structured SVMs, recall that SVMs began with the following optimization problem:

$$\begin{aligned} \min_{\boldsymbol{w},\boldsymbol{\xi}} \quad & \underbrace{\frac{1}{2}||\boldsymbol{w}||^2}_{\text{large margin}} + \underbrace{C\sum_n \xi_n}_{\text{small slack}} && (17.18)\\ \text{subj. to} \quad & y_n\left(\boldsymbol{w}\cdot\boldsymbol{x}_n + b\right) \geq 1 - \xi_n && (\forall n)\\ & \xi_n \geq 0 && (\forall n) \end{aligned}$$

After a bit of work, we were able to reformulate this in terms of a standard loss optimization algorithm with hinge loss:

$$\min_{\boldsymbol{w}} \quad \underbrace{\frac{1}{2}||\boldsymbol{w}||^2}_{\text{large margin}} + \underbrace{C\sum_n \ell^{(\text{hin})}(y_n, \boldsymbol{w}\cdot\boldsymbol{x}_n + b)}_{\text{small slack}} \tag{17.19}$$

We can do a similar derivation in the structured case. The question is: exactly what should the slack be measuring? Our *goal* is for the score of the true output $\boldsymbol{y}$ to beat the score of any imposter output $\hat{\boldsymbol{y}}$. To incorporate loss, we will say that we want the score of the true output to beat the score of any imposter output by *at least* the loss that would be suffered if we were to predict that imposter output. An alternative view is the ranking view: we want the true output to be ranked above any imposter by an amount at least equal to the loss.

To keep notation simple, we will write $s_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{y})$ to denote the score of the pair $\boldsymbol{x}, \boldsymbol{y}$, namely $\boldsymbol{w} \cdot \boldsymbol{\phi}(\boldsymbol{x}, \boldsymbol{y})$. This suggests a set of constraints of the form:

$$s_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{y}) - s_{\boldsymbol{w}}(\boldsymbol{x}, \hat{\boldsymbol{y}}) \geq \ell^{(\text{Ham})}(\boldsymbol{y}, \hat{\boldsymbol{y}}) - \xi_{\hat{\boldsymbol{y}}} \qquad (\forall n, \forall \hat{\boldsymbol{y}} \in \mathcal{Y}(\boldsymbol{x})) \tag{17.20}$$

The rest of the optimization problem remains the same, yielding:

$$\min_{\boldsymbol{w}, \boldsymbol{\xi}} \quad \frac{1}{2} ||\boldsymbol{w}||^2 + C \sum_n \sum_{\hat{\boldsymbol{y}} \in \mathcal{Y}_{\boldsymbol{x}_n}} \xi_{n, \hat{\boldsymbol{y}}} \tag{17.21}$$

$$\begin{aligned} \text{subj. to} \quad & s_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{y}) - s_{\boldsymbol{w}}(\boldsymbol{x}, \hat{\boldsymbol{y}}) \\ & \qquad \geq \ell^{(\text{Ham})}(\boldsymbol{y}_n, \hat{\boldsymbol{y}}) - \xi_{n, \hat{\boldsymbol{y}}} \qquad (\forall n, \forall \hat{\boldsymbol{y}} \in \mathcal{Y}(\boldsymbol{x}_n)) \\ & \xi_{n, \hat{\boldsymbol{y}}} \geq 0 \qquad (\forall n, \forall \hat{\boldsymbol{y}} \in \mathcal{Y}(\boldsymbol{x}_n)) \end{aligned}$$

This optimization problem asks for a large margin and small slack, where there is a slack very for every training example and every possible incorrect output associated with that training example. In general, this is *way too many* slack variables and *way too many* constraints!

There is a very useful, general trick we can apply. If you focus on the first constraint, it roughly says (letting $s()$ denote score): $s(\boldsymbol{y}) \geq \big[s(\hat{\boldsymbol{y}}) + \ell(\boldsymbol{y}, \hat{\boldsymbol{y}})\big]$ for all $\hat{\boldsymbol{y}}$, modulo slack. We'll refer to the thing in brackets as the "loss-augmented score." But if we want to guarantee that the score of the true $\boldsymbol{y}$ beats the loss-augmented score of *all* $\hat{\boldsymbol{y}}$, it's enough to ensure that it beats the loss-augmented score of the most confusing imposter. Namely, it is sufficient to require that $s(\boldsymbol{y}) \geq \max_{\hat{\boldsymbol{y}}} \big[s(\hat{\boldsymbol{y}}) + \ell(\boldsymbol{y}, \hat{\boldsymbol{y}})\big]$, modulo slack. Expanding out the definition of $s()$ and adding slack back in, we can replace the exponentially large number of constraints in Eq (17.21) with the simpler set of constraints:

$$s_{\boldsymbol{w}}(\boldsymbol{x}_n, \boldsymbol{y}_n) \geq \max_{\hat{\boldsymbol{y}} \in \mathcal{Y}(\boldsymbol{x}_n)} \Big[s_{\boldsymbol{w}}(\boldsymbol{x}_n, \hat{\boldsymbol{y}}) + \ell^{(\text{Ham})}(\boldsymbol{y}_n, \hat{\boldsymbol{y}})\Big] - \xi_n \qquad (\forall n)$$

We can now apply the same trick as before to remove ξ_n from the analysis. In particular, because ξ_n is constrained to be ≥ 0 and because we are trying to minimize it's sum, we can figure out that out the optimum, it will be the case that:

$$\xi_n = \max \left\{ 0, \max_{\hat{\boldsymbol{y}} \in \mathcal{Y}(\boldsymbol{x}_n)} \Big[s_{\boldsymbol{w}}(\boldsymbol{x}_n, \hat{\boldsymbol{y}}) + \ell^{(\text{Ham})}(\boldsymbol{y}_n, \hat{\boldsymbol{y}})\Big] - s_{\boldsymbol{w}}(\boldsymbol{x}_n, \boldsymbol{y}_n) \right\} \tag{17.22}$$

$$= \ell^{(\text{s-h})}(\boldsymbol{y}_n, \boldsymbol{x}_n, \boldsymbol{w}) \tag{17.23}$$

This value is referred to as the **structured hinge loss**, which we have denoted as $\ell^{(\text{s-h})}(\boldsymbol{y}_n, \boldsymbol{x}_n, \boldsymbol{w})$. This is because, although it is more complex, it bears a striking resemlance to the **hinge loss** from Chapter 8.

In particular, if the score of the true output beats the score of every the best imposter by at least its loss, then ξ_n will be zero. On the other hand, if some imposter (plus its loss) beats the true output, the loss scales linearly as a function of the difference. At this point, there is nothing special about Hamming loss, so we will replace it with some arbitrary structured loss ℓ.

Plugging this back into the objective function of Eq (17.21), we can write the structured SVM as an *unconstrained* optimization problem, akin to Eq (17.19), as:

$$\min_{\boldsymbol{w}} \quad \frac{1}{2}||\boldsymbol{w}||^2 + C\sum_n \ell^{(\text{s-h})}(\boldsymbol{y}_n, \boldsymbol{x}_n, \boldsymbol{w}) \tag{17.24}$$

This is now in a form that we can optimize using subgradient descent (Chapter 8) or stochastic subgradient descent (Chapter 14).

In order to compute subgradients of Eq (17.24), we need to be able to compute subgradients of the structured hinge loss. Mathematically this is straightforward. If the structured hinge loss on an example $(\boldsymbol{x}, vy)$ is zero, then the gradient with respect to $\boldsymbol{w}$ is also zero. If the structured hinge loss is positive, then the gradient is:

$$\nabla_{\boldsymbol{w}} \ell^{(\text{s-h})}(\boldsymbol{y}, \boldsymbol{x}, \boldsymbol{w}) \quad \textit{if the loss is} > 0 \tag{17.25}$$

expand definition using arbitrary structured loss ℓ

$$= \nabla_{\boldsymbol{w}} \left\{ \max_{\hat{\boldsymbol{y}} \in \mathcal{Y}(\boldsymbol{x}_n)} \Big[\boldsymbol{w} \cdot \boldsymbol{\phi}(\boldsymbol{x}_n, \hat{\boldsymbol{y}}) + \ell(\boldsymbol{y}_n, \hat{\boldsymbol{y}}) \Big] - \boldsymbol{w} \cdot \boldsymbol{\phi}(\boldsymbol{x}_n, \boldsymbol{y}_n) \right\} \tag{17.26}$$

define $\hat{\boldsymbol{y}}_n$ to be the output that attains the maximum above, rearrange

$$= \nabla_{\boldsymbol{w}} \Big\{ \boldsymbol{w} \cdot \boldsymbol{\phi}(\boldsymbol{x}_n, \hat{\boldsymbol{y}}) - \boldsymbol{w} \cdot \boldsymbol{\phi}(\boldsymbol{x}_n, \boldsymbol{y}_n) + \ell(\boldsymbol{y}_n, \hat{\boldsymbol{y}}) \Big\} \tag{17.27}$$

take gradient

$$= \boldsymbol{\phi}(\boldsymbol{x}_n, \hat{\boldsymbol{y}}) - \boldsymbol{\phi}(\boldsymbol{x}_n, \boldsymbol{y}_n) \tag{17.28}$$

Putting this together, we get the full gradient as:

$$\nabla_{\boldsymbol{w}} \ell^{(\text{s-h})}(\boldsymbol{y}_n, \boldsymbol{x}_n, \boldsymbol{w}) = \begin{cases} \boldsymbol{0} & \text{if } \ell^{(\text{s-h})}(\boldsymbol{y}_n, \boldsymbol{x}_n, \boldsymbol{w}) = 0 \\ \boldsymbol{\phi}(\boldsymbol{x}_n, \hat{\boldsymbol{y}}_n) - \boldsymbol{\phi}(\boldsymbol{x}_n, \boldsymbol{y}_n) & \text{otherwise} \end{cases}$$
$$\text{where } \hat{\boldsymbol{y}}_n = \operatorname*{argmax}_{\hat{\boldsymbol{y}}_n \in \mathcal{Y}(\boldsymbol{x}_n)} \Big[\boldsymbol{w} \cdot \boldsymbol{\phi}(\boldsymbol{x}_n, \hat{\boldsymbol{y}}_n) + \ell(\boldsymbol{y}_n, \hat{\boldsymbol{y}}_n) \Big] \tag{17.29}$$

The form of this gradient is very simple: it is equal to the features of the worst imposter minus the features of the truth, unless the truth beats all imposters, in which case it's zero. When plugged into stochastic subgradient descent, you end up with an update that looks very much like the structured perceptron: if the current prediction $(\hat{\boldsymbol{y}}_n)$ is correct, there is no gradient step. But if the current prediction is incorrect, you step $\boldsymbol{w}$ toward the truth and away from the imposter.

Algorithm 41 StochSubGradStructSVM(**D**, *MaxIter*, λ, ℓ)

1: $\mathbf{w} \leftarrow \mathbf{0}$ // initialize weights
2: **for** *iter* = 1 ... *MaxIter* **do**
3: **for all** $(\mathbf{x}, \mathbf{y}) \in \mathbf{D}$ **do**
4: $\hat{\mathbf{y}} \leftarrow \text{argmax}_{\hat{\mathbf{y}} \in \mathcal{Y}(\mathbf{x})} \mathbf{w} \cdot \phi(\mathbf{x}, \hat{\mathbf{y}}) + \ell(\mathbf{y}, \hat{\mathbf{y}})$ // loss-augmented prediction
5: **if** $\hat{\mathbf{y}} \neq \mathbf{y}$ **then**
6: $\mathbf{w} \leftarrow \mathbf{w} + \phi(\mathbf{x}, \mathbf{y}) - \phi(\mathbf{x}, \hat{\mathbf{y}})$ // update weights
7: **end if**
8: $\mathbf{w} \leftarrow \mathbf{w} - \frac{\lambda}{N} \mathbf{w}$ // shrink weights due to regularizer
9: **end for**
10: **end for**
11: **return** $\mathbf{w}$ // return learned weights

We will consider how to compute the loss-augmented argmax in the next section, but before that we summarize an algorithm for optimizing structured SVMs using stochastic subgradient descent: Algorithm 17.4. Of course there are other possible optimization strategies; we are highlighting this one because it is nearly identical to the structured perceptron. The only differences are: (1) on line 4 you use loss-augmented argmax instead of argmax; and (2) on line 8 the weights are shrunk slightly corresponding to the ℓ_2 regularizer on $\mathbf{w}$. (Note: we have used $\lambda = 1/(2C)$ to make the connection to linear models clearer.)

17.5 Loss-Augmented Argmax

The challenge that arises is that we now have a more complicated argmax problem that before. In structured perceptron, we only needed to compute $\hat{\mathbf{y}}_n$ as the output that maximized its score (see Eq 17.14). Here, we need to find the output that maximizes it score *plus* it's loss (Eq (17.29)). This optimization problem is refered to as **loss-augmented search** or **loss-augmented inference**.

Before solving the loss-augmented inference problem, it's worth thinking about why it makes sense. What is $\hat{\mathbf{y}}_n$? It's the output that has the highest score among all outputs, *after* adding the output's corresponding loss to that score. In other words, every incorrect output gets an artificial boost to its score, equal to its loss. The loss is serving to make imposters look *even better* than they really are, so if the truth is to beat an imposter, it has to beat it by a *lot*. In fact, this loss augmentation is essentially playing the role of a margin, where the required margin scales according to the loss.

The algorithmic question, then, is how to compute $\hat{\mathbf{y}}_n$. In the fully general case, this is at least as hard as the normal argmax problem, so we cannot expect a general solution. Moreover, even in cases where the argmax problem is easy (like for sequences), the loss-augmented

argmax problem can still be difficult. In order to make it easier, we need to assume that the loss *decomposes* of the input in a way that's consistent with the features. In particular, if the structured loss function is Hamming loss, this is often straightforward.

As a concrete example, let's consider loss-augmented argmax for sequences under Hamming loss. In comparison to the trellis problem solved in Section 17.7, the only difference is that we want to *reward* paths that go through incorrect nodes in the trellis! In particular, in Figure 17.1, all of the edges that are not part of the gold standard path—those that are thinner and grey—get a free "+1" added to their weights. Since Hamming loss adds one to the score for any word that's predicted incorrectly, this means that every edge in the trellis that leads to an *incorrect* node (i.e., one that does not match the gold truth label) gets a "+1" added to its weight.

Again, consider an edge in the trellis that goes from time $l-1$ to l, and transitions from y to y'. In the non-loss-augmented, the weight of this edge was exactly $\boldsymbol{w} \cdot \boldsymbol{\phi}_l(\boldsymbol{x}, \cdots \circ y \circ y')$. In the loss-augmented cases, the weight of this edge becomes:

$$\underbrace{\boldsymbol{w} \cdot \boldsymbol{\phi}_l(\boldsymbol{x}, \cdots \circ y \circ y')}_{\text{edge score, as before}} + \underbrace{\mathbf{1}[y' \neq \boldsymbol{y}_l]}_{+1 \text{ for mispredictions}} \tag{17.30}$$

Once this loss-augmented graph has been constructed, the same max-weight path algorithm can be run to find the loss-augmented argmax sequence.

17.6 Argmax in General

The general argmax problem for structured perceptron is the algorithmic question of whether the following can be efficiently computed:

$$\hat{\boldsymbol{y}} \leftarrow \operatorname*{argmax}_{\hat{\boldsymbol{y}} \in \mathcal{Y}(\boldsymbol{x})} \boldsymbol{w} \cdot \boldsymbol{\phi}(\boldsymbol{x}, \hat{\boldsymbol{y}}) \tag{17.31}$$

We have seen that *if* the output space $\mathcal{Y}(\boldsymbol{x})$ is sequences *and* the only types of features are unary features and Markov features, then this can be computed efficiently. There are a small number of other structured output spaces and feature restrictions for which efficient problem-specific algorithms exist:

- Binary trees, with context-free features: use the CKY algorithm
- 2d image segmentation, with adjacent-pixel features: use a form of graph cuts
- Spanning trees, with edge-based features: use Kruskal's algorithm (or for directed spanning trees, use Chu-Liu/Edmonds algorithm)

These special cases are often very useful, and many problems can be cast in one of these frameworks. However, it is often the case that you need a more general solution.

One of the most generally useful solutions is to cast the argmax problem as an **integer linear program**, or **ILP**. ILPs are a specific type of mathematical program/optimization problem, in which the objective function being optimized is linear and the constraints are linear. However, unlike "normal" linear programs, in an ILP you are allowed to have integer constraints and disallow fractional values. The general form of an ILP is, for a fixed vector $\boldsymbol{a}$:

$$\max_{\boldsymbol{z}} \quad \boldsymbol{a} \cdot \boldsymbol{z} \quad \text{subj. to} \quad \text{linear constraints on } \boldsymbol{z} \tag{17.32}$$

The main point is that the constraints on $\boldsymbol{z}$ are allowed to include constraints like $z_3 \in \{0, 1\}$, which is considered an integer constraint.

Being able to cast your argmax problem as an ILP has the advantage that there are very good, efficiently, well-engineered ILP solvers out there in the world.[3] ILPs are not a panacea though: in the worst case, the ILP solver will be horribly inefficient. But for prototyping, or if there are no better options, it's a very handy technique.

[3] I like Gurobi best, and it's free for academic use. It also has a really nice Python interface.

Figuring out how exactly to cast your argmax problem as an ILP can be a bit challenging. Let's start with an example of encoding sequence labeling with Markov features as an ILP. We first need to decide what the variables will be. Because we need to encode pairwise features, we will let our variables be of the form:

$$z_{l,k',k} = \mathbf{1}[\text{label } l \text{ is } k \text{ and label } l-1 \text{ is } k'] \tag{17.33}$$

These zs will all be binary indicator variables.

Our next task is to construct the linear objective function. To do so, we need to assign a value to $a_{l,k',k}$ in such a way that $\boldsymbol{a} \cdot \boldsymbol{z}$ will be exactly equal to $\boldsymbol{w} \cdot \boldsymbol{\phi}(\boldsymbol{x}, y(\boldsymbol{z}))$, where $y(\boldsymbol{z})$ denotes the sequence that we can read off of the variables z. With a little thought, we arrive at:

$$a_{l,k',k} = \boldsymbol{w} \cdot \boldsymbol{\phi}_l(\boldsymbol{x}, \langle \ldots, k', k \rangle) \tag{17.34}$$

Finally, we need to construct constaints. There are a few things that these constraints need to enforce:

1. That all the zs are binary. That's easy: just say $z_{l,k',k} \in \{0, 1\}$, for all l, k', k.

2. That for a given position l, there is exactly one active z. We can do this with an equality constraint: $\sum_k \sum_{k'} z_{l,k',k} = 1$ for all l.

3. That the zs are internally consistent: if the label at position 5 is supposed to be "noun" then both $z_{5,\_,\_}$ *and* $z_{6,\_,\_}$ need to agree on

this. We can do this as: $\sum_{k'} z_{l,k',k} = \sum_{k''} z_{l+1,k,k''}$ for all l, k. Effectively what this is saying is that $z_{5,?,\text{verb}} = z_{6,\text{verb},?}$ where the "?" means "sum over all possibilities."

This fully specifies an ILP that you can relatively easily implement (arguably more easily than the dynamic program in Algorithm 17.7) and which will solve the argmax problem for you. Will it be efficient? In this case, probably yes. Will it be as efficient as the dynamic program? Probably not.

It takes a bit of effort and time to get used to casting optimization problems as ILPs, and certainly not all can be, but most can and it's a very nice alternative.

In the case of loss-augmented search for structured SVMs (as opposed to structured perceptron), the objective function of the ILP will need to be modified to include terms corresponding to the loss.

17.7 Dynamic Programming for Sequences

Recall the decomposition we derived earlier:

$$\boldsymbol{w} \cdot \boldsymbol{\phi}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{w} \cdot \sum_{l=1}^{L} \boldsymbol{\phi}_l(\boldsymbol{x}, \boldsymbol{y}) \qquad \text{decomposition of structure} \qquad (17.35)$$

$$= \sum_{l=1}^{L} \boldsymbol{w} \cdot \boldsymbol{\phi}_l(\boldsymbol{x}, \boldsymbol{y}) \qquad \text{associative law} \qquad (17.36)$$

This decomposition allows us to construct the following dynamic program. We will compute $\alpha_{l,k}$ as the score of the *best possible* output prefix up to and including position l that labels the lth word with label k. More formally:

$$\alpha_{l,k} = \max_{\hat{\boldsymbol{y}}_{1:l-1}} \boldsymbol{w} \cdot \boldsymbol{\phi}_{1:l}(\boldsymbol{x}, \hat{\boldsymbol{y}} \circ k) \qquad (17.37)$$

Here, $\hat{\boldsymbol{y}}$ is a sequence of length $l-1$, and $\hat{\boldsymbol{y}} \circ k$ denotes the sequence of length l obtained by adding k onto the end. The max denotes the fact that we are seeking the *best* possible prefix up to position $l-1$, and the forcing the label for position l to be k.

Before working through the details, let's consider an example. Suppose that we've computing the αs up to $l = 2$, and have: $\alpha_{2,\text{noun}} = 2$, $\alpha_{2,\text{verb}} = 9$, $\alpha_{2,\text{adj}} = -1$ (recall: position $l = 2$ is "eat"). We want to extend this to position 3; for example, we want to compute $\alpha_{3,\text{adj}}$. Let's assume there's a single unary feature here, "tasty/adj" and three possible Markov features of the form "?:adj". Assume these weights are as given to the right. [4] Now, the question for $\alpha_{3,\text{adj}}$ is: what's the score of the best prefix that labels "tasty" as "adj"? We can obtain this by taking the best prefix up to "eat" and then appending

[4] $w_{\text{"tasty/adj"}} = 1.2$
$w_{\text{"noun:adj"}} = -5$
$w_{\text{"verb:adj"}} = 2.5$
$w_{\text{"adj:adj"}} = 2.2$

each possible label. Whichever combination is best is the winner. The relevant computation is:

$$\begin{aligned}
\alpha_{3,\text{adj}} &= \max \Big\{ \alpha_{2,\text{noun}} + w_{\text{“tasty/adj”}} + w_{\text{“noun:adj”}} \\
&\qquad \alpha_{2,\text{verb}} + w_{\text{“tasty/adj”}} + w_{\text{“verb:adj”}} \\
&\qquad \alpha_{2,\text{adj}} + w_{\text{“tasty/adj”}} + w_{\text{“adj:adj”}} \Big\} && (17.38) \\
&= \max \Big\{ 2 + 1.2 - 5, \quad 9 + 1.2 + 2.5, \quad -1 + 1.2 + 2.2 \Big\} && (17.39) \\
&= \max \Big\{ -1.8, \quad 12.7, \quad 2.4 \Big\} = 12.7 && (17.40)
\end{aligned}$$

This means that (a) the score for the prefix ending at position 3 labeled as adjective is 12.7, and (b) the “winning” previous label was “verb”. We will need to record these winning previous labels so that we can extract the best path at the end. Let’s denote by $\zeta_{l,k}$ the label at position $l-1$ that achieves the max.

From here, we can formally compute the αs recursively. The main observation that will be necessary is that, because we have limited ourselves to Markov features, $\phi_{l+1}(\boldsymbol{x}, \langle y_1, y_2, \ldots, y_l, y_{l+1} \rangle)$ depends only on the last two terms of y, and does *not* depend on $y_1, y_2, \ldots, y_{l-1}$. The full recursion is derived as:

$$\alpha_{0,k} = 0 \quad \forall k \tag{17.41}$$

$$\zeta_{0,k} = \emptyset \quad \forall k \tag{17.42}$$

the score for any empty sequence is zero

$$\alpha_{l+1,k} = \max_{\hat{\boldsymbol{y}}_{1:l}} \boldsymbol{w} \cdot \boldsymbol{\phi}_{1:l+1}(\boldsymbol{x}, \hat{\boldsymbol{y}} \circ k) \tag{17.43}$$

separate score of prefix from score of position l+1

$$= \max_{\hat{\boldsymbol{y}}_{1:l}} \boldsymbol{w} \cdot \Big(\boldsymbol{\phi}_{1:l}(\boldsymbol{x}, \hat{\boldsymbol{y}}) + \boldsymbol{\phi}_{l+1}(\boldsymbol{x}, \hat{\boldsymbol{y}} \circ k) \Big) \tag{17.44}$$

distributive law over dot products

$$= \max_{\hat{\boldsymbol{y}}_{1:l}} \Big[\boldsymbol{w} \cdot \boldsymbol{\phi}_{1:l}(\boldsymbol{x}, \hat{\boldsymbol{y}}) + \boldsymbol{w} \cdot \boldsymbol{\phi}_{l+1}(\boldsymbol{x}, \hat{\boldsymbol{y}} \circ k) \Big] \tag{17.45}$$

separate out final label from prefix, call it k’

$$= \max_{\hat{\boldsymbol{y}}_{1:l-1}} \max_{k'} \Big[\boldsymbol{w} \cdot \boldsymbol{\phi}_{1:l}(\boldsymbol{x}, \hat{\boldsymbol{y}} \circ k') + \boldsymbol{w} \cdot \boldsymbol{\phi}_{l+1}(\boldsymbol{x}, \hat{\boldsymbol{y}} \circ k' \circ k) \Big] \tag{17.46}$$

swap order of maxes, and last term doesn’t depend on prefix

$$= \max_{k'} \Big[\Big[\max_{\hat{\boldsymbol{y}}_{1:l-1}} \boldsymbol{w} \cdot \boldsymbol{\phi}_{1:l}(\boldsymbol{x}, \hat{\boldsymbol{y}} \circ k') \Big] + \boldsymbol{w} \cdot \boldsymbol{\phi}_{l+1}(\boldsymbol{x}, \langle \ldots, k', k \rangle) \Big] \tag{17.47}$$

apply recursive definition

$$= \max_{k'} \Big[\alpha_{l,k'} + \boldsymbol{w} \cdot \boldsymbol{\phi}_{l+1}(\boldsymbol{x}, \langle \ldots, k', k \rangle) \Big] \tag{17.48}$$

Algorithm 42 ARGMAXFORSEQUENCES($\boldsymbol{x}$, $\boldsymbol{w}$)

$L \leftarrow \text{LEN}(\boldsymbol{x})$
$\alpha_{l,k} \leftarrow 0, \quad \zeta_{k,l} \leftarrow 0, \quad \forall\, k = 1 \dots K, \quad \forall l = 0 \dots L$ // initialize variables
for $l = 0 \dots L\text{-}1$ **do**
  for $k = 1 \dots K$ **do**
    $\alpha_{l+1,k} \leftarrow \max_{k'} \left[\alpha_{l,k'} + \boldsymbol{w} \cdot \boldsymbol{\phi}_{l+1}(\boldsymbol{x}, \langle \dots, k', k \rangle)\right]$ // recursion:
      // here, $\boldsymbol{\phi}_{l+1}(\dots k', k \dots)$ is the set of features associated with
      // output position $l+1$ and two adjacent labels k' and k at that position
    $\zeta_{l+1,k} \leftarrow$ the k' that achieves the maximum above // store backpointer
  end for
end for
$\boldsymbol{y} \leftarrow \langle 0, 0, \dots, 0 \rangle$ // initialize predicted output to L-many zeros
$y_L \leftarrow \text{argmax}_k\, \alpha_{L,k}$ // extract highest scoring final label
for $l = L\text{-}1 \dots 1$ **do**
  $y_l \leftarrow \zeta_{l,y_{l+1}}$ // traceback ζ based on $\boldsymbol{y}_{l+1}$
end for
return $\boldsymbol{y}$ // return predicted output

and record a backpointer to the k' that achieves the max

$$\zeta_{l+1,k} = \underset{k'}{\text{argmax}} \left[\alpha_{l,k'} + \boldsymbol{w} \cdot \boldsymbol{\phi}_{l+1}(\boldsymbol{x}, \langle \dots, k', k \rangle)\right] \tag{17.49}$$

At the end, we can take $\max_k \alpha_{L,k}$ as the score of the best output sequence. To extract the final sequence, we know that the best label for the last word is $\text{argmax}\, \alpha_{L,k}$. Let's call this $\hat{y}_L$ Once we know that, the best *previous* label is $\zeta_{L-1,\hat{y}_L}$. We can then follow a path through ζ back to the beginning. Putting this all together gives Algorithm 17.7.

The main benefit of Algorithm 17.7 is that it is guaranteed to exactly compute the argmax output for sequences required in the structured perceptron algorithm, *efficiently*. In particular, it's runtime is $O(LK^2)$, which is an exponential improvement on the naive $O(K^L)$ runtime if one were to enumerate every possible output sequence. The algorithm can be naturally extended to handle "higher order" Markov assumptions, where features depend on triples or quadruples of the output. The memoization becomes notationally cumbersome, but the algorithm remains essentially the same. In order to handle a length M Markov features, the resulting algorithm will take $O(LK^M)$ time. In practice, it's rare that $M > 3$ is necessary or useful.

In the case of loss-augmented search for structured SVMs (as opposed to structured perceptron), we need to include the scores coming from the loss augmentation in the dynamic program. The only thing that changes between the standard argmax solution (Algorithm 17.7, and derivation in Eq (17.48)) is that the any time an incorrect label is used, the (loss-augmented) score increases by one. Recall that in the non-loss-augmented case, we have the α recursion

as:

$$\alpha_{l+1,k} = \max_{\hat{\boldsymbol{y}}_{1:l}} \boldsymbol{w} \cdot \boldsymbol{\phi}_{1:l+1}(\boldsymbol{x}, \hat{\boldsymbol{y}} \circ k) \tag{17.50}$$

$$= \max_{k'} \left[\alpha_{l,k'} + \boldsymbol{w} \cdot \boldsymbol{\phi}_{l+1}(\boldsymbol{x}, \langle \ldots, k', k \rangle) \right] \tag{17.51}$$

If we define $\tilde{\alpha}$ to be the loss-augmented score, the corresponding recursion is (differences highlighted in blue):

$$\tilde{\alpha}_{l+1,k} = \max_{\hat{\boldsymbol{y}}_{1:l}} \boldsymbol{w} \cdot \boldsymbol{\phi}_{1:l+1}(\boldsymbol{x}, \hat{\boldsymbol{y}} \circ k) + \ell^{(\text{Ham})}_{1:l+1}(\boldsymbol{y}, \hat{\boldsymbol{y}} \circ k) \tag{17.52}$$

$$= \max_{k'} \left[\tilde{\alpha}_{l,k'} + \boldsymbol{w} \cdot \boldsymbol{\phi}_{l+1}(\boldsymbol{x}, \langle \ldots, k', k \rangle) \right] + \mathbf{1}[k \neq \boldsymbol{y}_{l+1}] \tag{17.53}$$

In other words, when computing $\tilde{\alpha}$ in the loss-augmented case, whenever the output prediction is forced to pass through an incorrect label, the score for that cell in the dynamic program gets increased by one. The resulting algorithm is identical to Algorithm 17.7, except that Eq (17.53) is used for computing αs.

17.8 Further Reading

TODO