# Dynamic Programming for NLP

April 8, 2016

## 0.1 Dynamic Programming for NLP

The strategy of dynamic programming reduces the complexity of a search problem which decomposes into frequently-reused instances of the same problem.

You encountered dynamic programming for n-gram segmentation in HW4. We have also discussed two more dynamic programming algorithms in lecture: Viterbi and forward-backward. For HW5, you will need to implement the Viterbi algorithm. The forward-backward algorithm is an optional extension.

In this notebook, we sketch an efficient implementation of the Viterbi algorithm. We will rereshape the forward-backward algorithm in section.

### 0.1.1 Viterbi Algorithm

Recall our definition of a first-order Markov model (a kind of history-based model) in lecture:

$$f(\mathbf{x}, c_{1:n}) = \sum_{i=1}^{n} \hat{\mathbf{y}}(c_{i-1})_{c_i}.$$

We fix some observation sequence $\mathbf{x}$ of length $n$. We refer to classes and states interchangeably.

We refer to $\hat{\mathbf{y}}$ as the edge scores. This is what varies across applications of the Viterbi algorithm. For example, in a Hidden Markov Model (HMM), the edge scores are the joint probability of transition and emission, given the observation. In a Maximum-Entropy Markov Model (MEMM), the edge scores are a softmax over a linear model on the features.

The Viterbi algorithm relies on the fact that a maximal path consists of a maximal path to the penultimate state followed by a maximizing transition and emission to the final state. The next timestep reuses each maximal path in the previous timestep. This combination of optimal substructure and overlapping subproblems is why dynamic programming works here.

Next, we consider some practical aspects of implementing the Viterbi algorithm.

**Log Scores** We often use log scores for two reasons: 1. avoid numerical underflow as scores shrink 2. turn multiplications for joint probabilities into cheaper additions

**Boundary States** You will need to think about how to denote "start" and "end" states to demarcate sequences of interest. The sequence model has no natural notion of a "start" or "end" state. (Note that the "start" state is distinct from the earliest timestep in the Viterbi trellis, which is sometimes also called a "start" state.)

We can introduce these boundary states while preprocessing or training. (In the dataset for HW5, blank rows delimit sentences.)

For example, as described in lecture, we often denote the start state for sentences as <s> or `BOS` (beginning-of-sentence), and the end state as </s> or `EOS` (end-of-sentence). These boundary states let a language model recognize patterns at the start and end of a sentence, which can sometimes be an informative feature in NER. Without these boundary states, count-based initial state probabilities would contain no information about being at the start or end of a sentence.

**Precomputing Edge Scores and Efficient Matrix Operations** The edge scores depend on the observed sequence, and so we must compute them anew every time we run the algorithm.

We described the "simple" algorithm in lecture using elementwise operations and computing all the edge scores at once. We then mentioned in lecture the idea of precomputing the edge scores as needed. We can reformulate this precomputation with matrix operations with the analogy of applying `:forward` to a batch.

Required matrices: - **Maximal Path Scores.** We denoted the log-score of the most likely path as $\pi \in \mathbb{R}^{n \times |\mathcal{C}|}$. Then $\pi[i, c_i]$ is the log-score of the highest-scoring path which ends in state $c_i$ in timestep $i$. - **Backpointers.** We denoted the backpointers as $bp \in \mathcal{C}^{n \times |\mathcal{C}|}$. These allow us to recover the maximal paths by following pointers from the last timestep to the first. - **Forwarding.** We can "forward" the max scores through each timestep in $O(|\mathcal{C}|^2)$ space, compared to the naive approach of computing all of the edge scores across all timesteps in advance. (Here, "forward" is in the sense of forwarding a batch, and not the forward algorithm which we discuss later.)

The key idea per timestep is to precompute the edge scores, then compute the maxes and backpointers simultaneously by adding the `:expand`ed maxes from the previous timestep and using `:max`.

**Example** Below is an example of the Viterbi algorithm in Torch: a simple HMM POS tagger for a toy sentence, "deal talks fail." Compare to the neural network POS tagger you developed in HW2 based on [Collobert et al. 2011].

```
In [6]: -- Simple HMM Part-of-Speech (POS) Tagger to demonstrate Viterbi Algorithm
        --
        -- "deal talks fail" phrase from John Longley's lecture notes at University of
        -- Edinburgh (Informatics 2A: Lecture 16)

        -- Note that we have a START boundary state but not an END one in this example.
        -- Column is the "from" state, row is the "to" state
        initial = torch.Tensor({
            -- START, N, V
            {1.0},
            {0.0},
            {0.0},
        }):log() -- precompute the log-probabilities once

        transition = torch.Tensor({
            {0.0, 0.0, 0.0},
            {0.8, 0.4, 0.6},
            {0.2, 0.6, 0.4}
        }):log()

        emission = torch.Tensor({
            -- START, deal, fail, talks
            {1.0, 0.0, 0.0},
            {0.0, 0.45, 0.4},
            {0.0, 0.1, 0.4},
            {0.0, 0.45, 0.1}
        }):log()

        -- number of classes
        C = initial:size(1)

        -- log-scores of transition and emission
        -- corresponds to the vector y in the lecture notes
        -- i: timestep for the computed score
        function score_hmm(observations, i)
```

```
    local observation_emission = emission[observations[i]]:view(C, 1):expand(C, C)
    -- NOTE: allocates a new Tensor
    return observation_emission + transition
end

-- Viterbi algorithm.
-- observations: a sequence of observations, represented as integers
-- logscore: the edge scoring function over classes and observations in a history-based model
function viterbi(observations, logscore)
    local n = observations:size(1)
    local max_table = torch.Tensor(n, C)
    local backpointer_table = torch.Tensor(n, C)

    -- first timestep
    -- the initial most likely paths are the initial state distribution
    -- NOTE: another unnecessary Tensor allocation here
    local maxes, backpointers = (initial + emission[observations[1]]):max(2)
    max_table[1] = maxes

    -- remaining timesteps ("forwarding" the maxes)
    for i=2,n do
        -- precompute edge scores
        y = logscore(observations, i)
        scores = y + maxes:view(1, C):expand(C, C)

        -- compute new maxes (NOTE: another unnecessary Tensor allocation here)
        maxes, backpointers = scores:max(2)

        -- record
        max_table[i] = maxes
        backpointer_table[i] = backpointers
    end

    -- follow backpointers to recover max path
    local classes = torch.Tensor(n)
    maxes, classes[n] = maxes:max(1)
    for i=n,2,-1 do
        classes[i-1] = backpointer_table[{i, classes[i]}]
    end

    return classes
end

-- "START deal talks fail" = 1 2 4 3
-- => START N N V = 1 2 2 3
-- successfully looks ahead to infer the POS tags
-- because N is unlikely to both follow N and emit "fail"
print(viterbi(torch.Tensor({1, 2, 4, 3}), score_hmm))
```

```
Out[6]:  1
         2
         2
         3
        [torch.DoubleTensor of size 4]
```