Structure Prediction

Athens NLP Summer School (AthNLP 2024)

Vlad Niculae · Language Technology Lab, University of Amsterdam
https://vene.ro/mlsd · v.niculae@uva.nl

Αρις (Aris)

Ωρίων (Orion)

Structure Prediction

1. Overview
2. **Structured inputs**
   - Recap: Encoding sequences. RNN, CNN, transformer
   - Encoding graphs
3. **Structured outputs**
   - Probabilistic models of structures
   - Directed acyclic graphs
   - Algorithms for paths in DAGs: Maximization, probabilities, sampling
   - Application: Sequence tagging
   - Application: Sequence segmentation
   - Evaluating structured outputs

Machine Learning

Understanding, choosing, designing:

- models
- learning algorithms
- evaluation metrics
- experiment methodology

to learn and evaluate mappings
from inputs x to outputs y.

Machine Learning

Understanding, choosing, designing:

- models
- learning algorithms
- evaluation metrics
- experiment methodology

to learn and evaluate mappings from inputs x to outputs y.

... for Structures

structure, *noun: the way in which a complex object's parts are organized in relationship to one another.*

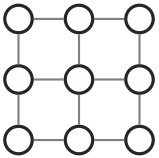
Many objects we want to do ML on have interesting structure:

language, images, shapes, networks…

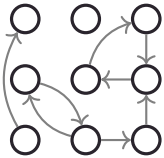
This lecture: how to work with structure in the input and the output.

A few examples of structure

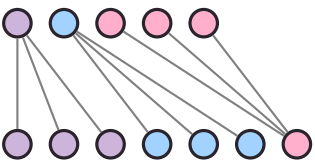
Sequence



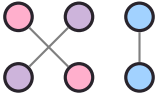
Grid



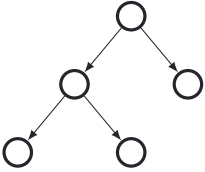
Graph



Alignments



Permutations



Hierarchy

Structures in NLP

- **Sequence** of (sub)words/characters: the usual way we encode linguistic data.
- **Segmentation** into entities / events / sections / speakers / ...
- **Inter-word dependencies**: syntactic or semantic analysis (graphs, trees)
- **Alignment**: between multi-lingual documents / speech to phonemes / ...

Structure is at the heart of
all models and algorithms designed for NLP.

Context and acknowledgements

These slides are a condensed version of my UvA course “Machine Learning for Structures,” with materials publicly available at `https://vene.ro/mlds`.

The original course covers more applications beyond NLP.

Slide help acknowledgements:

- Caio Corro
- Stela Topalova
- Mara Pîslar
- all the students taking my class

Funding acknowledgements:

- my institute IvI at UvA
- NWO VI.Veni.212.228
- Horizon Europe UTTER 101070631

Recap: ML classifiers

Learn to map from inputs $x \in \mathcal{X}$ to corresponding outputs $y \in \mathcal{Y}$ given a set of training pairs (x, y).

Classification: $\mathcal{Y} = \{1, 2, \ldots, K\}$.

Feature encoder $\boldsymbol{\phi} : \mathcal{X} \to \mathbb{R}^d$.

(could be hand-crafted or a neural net)

To make predictions:

$$\hat{y}(x) = \operatorname*{argmax}_{y \in \mathcal{Y}} \boldsymbol{w}_y \cdot \boldsymbol{\phi}(x)$$

Another way to think of this:

weight matrix $\boldsymbol{W}$ with rows $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_k$:

$\boldsymbol{a}(x) = \boldsymbol{W}\boldsymbol{\phi}(x) \in \mathbb{R}^K$ is a vector of scores for each of the k classes

$\text{score}(y; x) = [\boldsymbol{a}(x)]_y$

The highest-scoring class wins:

$$\hat{y}(x) = \operatorname*{argmax}_{y \in \mathcal{Y}} \text{score}(y; x)$$

Recap: Probabilistic classifiers, logistic regression

We can give a probabilistic interpretation to the ML classifier by interpreting scores as probabilities by applying softmax:

$$\Pr(y \mid x) = \frac{\exp(\text{score}(y; x))}{Z}, \qquad \text{where} \qquad Z = \sum_{y \in \mathcal{Y}} \exp(\text{score}(y; x)).$$

y	1	2	3	4
$\text{score}(y; x)$	-1.5	0.2	0.9	-1.1
$\Pr(y \mid x)$	0.05	0.29	0.58	0.08

This motivates logistic regression as a training objective (loss):
train params to maximize $\sum_{(x,y) \in \mathcal{D}} \log \Pr(y \mid x)$.

Why is softmax the way it is:

exp ensures all probabilities are non-negative.
Z is the normalizing constant to ensure probabilities sum to 1.

Handling structures

We made no assumptions about the form of $x \in \mathcal{X}$:

this is abstracted into the feature encoder $\boldsymbol{\phi}(x)$.

In the next part (30min), we recap feature encoders for structured objects.

maybe with a few extensions you haven't seen.

Afterward, we will look at structured outputs $\mathcal{Y}$.

Structure Prediction

1. Overview
2. Structured inputs
   - Recap: Encoding sequences. RNN, CNN, transformer
   - Encoding graphs
3. Structured outputs
   - Probabilistic models of structures
   - Directed acyclic graphs
   - Algorithms for paths in DAGs: Maximization, probabilities, sampling
   - Application: Sequence tagging
   - Application: Sequence segmentation
   - Evaluating structured outputs

Structure Prediction

Sequence input: Bag-of-words

(Jurafsky and Martin, 2024, Ch. 4.1)

Simple but powerful idea: for each vocabulary item, a feature that counts it:

$$\phi_i(x) = \text{number of occurrences of word } v_i \text{ in } x.$$

This leads to:

		!	.	book	fairly	good	is	long	nt	the	this
	text	ϕ_1	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_5	ϕ_6	ϕ_7	ϕ_8	ϕ_9
x_1	“this book is good!”	1	0	1	0	1	1	0	0	0	1
x_2	“fairly long book”	0	0	1	1	0	0	1	0	0	0
x_3	“the book isn’t good.”	0	1	1	0	1	1	0	1	1	0
					…						

Variants: zero-one, normalized frequencies.

Sequence input: Bag-of-words

(Jurafsky and Martin, 2024, Ch. 4.1)

Simple but powerful idea: for each vocabulary item, a feature that counts it:

$$\phi_i(x) = \text{number of occurrences of word } v_i \text{ in } x.$$

This leads to:

		!	.	book	fairly	good	is	long	nt	the	this
	text	ϕ_1	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_5	ϕ_6	ϕ_7	ϕ_8	ϕ_9
x_1	“this book is good!”	1	0	1	0	1	1	0	0	0	1
x_2	“fairly long book”	0	0	1	1	0	0	1	0	0	0
x_3	“the book isn’t good.”	0	1	1	0	1	1	0	1	1	0
					…						

Variants: zero-one, normalized frequencies.

Order is lost: $\boldsymbol{\phi}$(“doesn’t word order matter”) = $\boldsymbol{\phi}$(“word order doesn’t matter”)

Sequence inputs: Getting some structure back

Sequential order = a fundamental *structure* of language.

n-grams: treat *n consecutive tokens* as a single one.

Bigram tokenization:
“the book isn’t good.“ → [`the_book`, `book_is`, `is_n’t`, `n’t_good`, `good_.`]

This captures some local order.

Can even combine: 1-gram ∪ 2-gram ∪ . . . ∪ *n*-gram: [1]

But, it comes at a cost: how many features are needed?

[1] Ensure combination is reversible or else we won’t be able to distinguish features.
For instance, here, _ must not appear in any unigram.

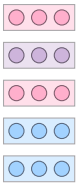
Embeddings of discrete tokens

(Jurafsky and Martin, 2024, Ch. 5)
(Murphy, 2022, sec. 1.5.4.3)

Neural networks perform continuous operations.

For sequential **discrete** data, (language, DNA, etc), we must first represent each token as a continuous "embedding" vector.

$$\underbrace{\begin{bmatrix} 3 \\ 2 \\ 3 \\ 1 \\ 1 \end{bmatrix}}_{\in \mathbb{R}^L} \rightarrow \underbrace{\begin{bmatrix} \boldsymbol{e}(3) \\ \boldsymbol{e}(2) \\ \boldsymbol{e}(3) \\ \boldsymbol{e}(1) \\ \boldsymbol{e}(1) \end{bmatrix}}_{\in \mathbb{R}^{L \times d}}$$



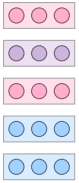
Embeddings of discrete tokens

(Jurafsky and Martin, 2024, Ch. 5)
(Murphy, 2022, sec. 1.5.4.3)

Neural networks perform continuous operations.

For sequential **discrete** data, (language, DNA, etc), we must first represent each token as a continuous "embedding" vector.

$$\underbrace{\begin{bmatrix} 3 \\ 2 \\ 3 \\ 1 \\ 1 \end{bmatrix}}_{\in \mathbb{R}^L} \rightarrow \underbrace{\begin{bmatrix} \boldsymbol{e}(3) \\ \boldsymbol{e}(2) \\ \boldsymbol{e}(3) \\ \boldsymbol{e}(1) \\ \boldsymbol{e}(1) \end{bmatrix}}_{\in \mathbb{R}^{L \times d}}$$



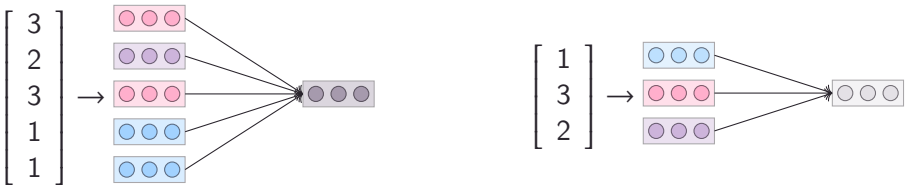
The function $\boldsymbol{e}(i)$ retrieves the ith row from an *embedding matrix* $\mathbf{E} \in \mathbb{R}^{|V| \times d}$.

The embeddings could be fixed or learned as model parameters.

Continuous bag-of-words

(Goldberg, 2017, Ch. 13)

Different-length sequences can be encoded by pooling their embeddings.



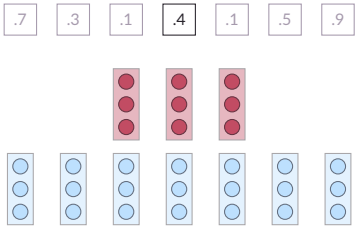
- average pooling: $\boldsymbol{z} = \frac{1}{L}(\boldsymbol{z}_1 + \ldots + \boldsymbol{z}_L)$
- max pooling: $[\boldsymbol{z}]_j = \max([\boldsymbol{z}_1]_j, \ldots, [\boldsymbol{z}_L]_j$ (coordinate-wise)

Just like in the standard bag of words, word order doesn't matter.

Sequence convolutions

aka 1-d convolution with d channels

(Goldberg, 2017, Ch. 13)



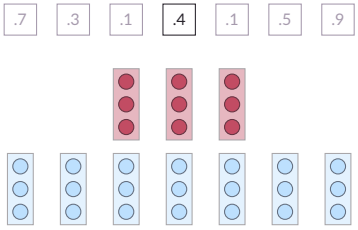
- Denote L=sequence length, d=embedding size, k=window size.

To reduce visual noise on slides, we now use the same color for all words, even if they're different words in general.

Sequence convolutions

aka 1-d convolution with d channels

(Goldberg, 2017, Ch. 13)



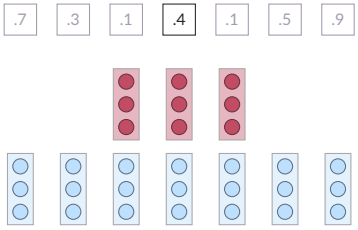
- Denote L=sequence length, d=embedding size, k=window size.
- In the single-channel case, a filter was a dim-k vector. Now, a filter is a $d \times k$ matrix.

To reduce visual noise on slides, we now use the same color for all words, even if they're different words in general.

Sequence convolutions

aka 1-d convolution with d channels

(Goldberg, 2017, Ch. 13)



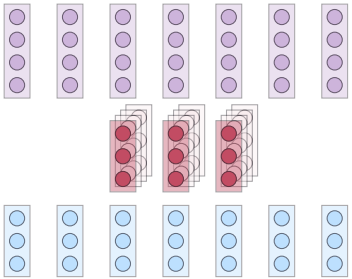
- Denote L=sequence length, d=embedding size, k=window size.
- In the single-channel case, a filter was a dim-k vector. Now, a filter is a $d \times k$ matrix.
- Output is still a single number per window.

To reduce visual noise on slides, we now use the same color for all words, even if they're different words in general.

Sequence convolutions

(Goldberg, 2017, Ch. 13)

aka 1-d convolution with d channels



- Denote L=sequence length, d=embedding size, k=window size.
- In the single-channel case, a filter was a dim-k vector. Now, a filter is a $d \times k$ matrix.
- Output is still a single number per window.
- Apply m filters in parallel: output is a dim-m vector per window:

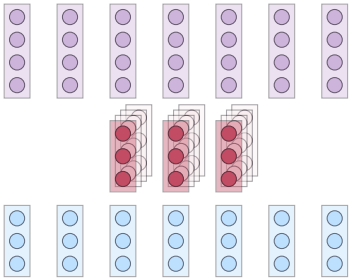
  a "layer" maps $(L, d) \rightarrow (L, m)$, for any L.

To reduce visual noise on slides, we now use the same color for all words, even if they're different words in general.

Sequence convolutions

(Goldberg, 2017, Ch. 13)

aka 1-d convolution with d channels



- Denote L=sequence length, d=embedding size, k=window size.
- In the single-channel case, a filter was a dim-k vector. Now, a filter is a $d \times k$ matrix.
- Output is still a single number per window.
- Apply m filters in parallel: output is a dim-m vector per window:

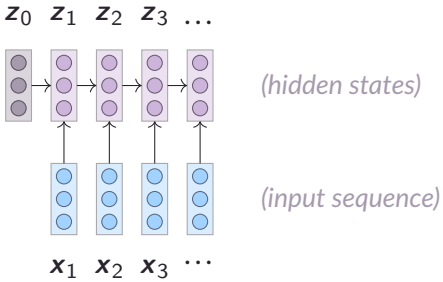
  a "layer" maps $(L, d) \rightarrow (L, m)$, for any L.
- Kind of like "continuous" n-grams!

To reduce visual noise on slides, we now use the same color for all words, even if they're different words in general.

Recurrent neural networks (RNN)

Recurrently encoding a sequence of input vectors $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \to (\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n)$:

$$\boldsymbol{z}_t = \boldsymbol{\phi}(\boldsymbol{x}_t, \boldsymbol{z}_{t-1})$$



The simplest RNN is the Elman RNN:

$$\boldsymbol{z}_t = \tanh\left(\underbrace{\boldsymbol{W}\boldsymbol{x}_t}_{\text{linear of inputs}} + \underbrace{\boldsymbol{U}\boldsymbol{z}_{t-1}}_{\text{linear of prev. state}} + \boldsymbol{b}\right)$$

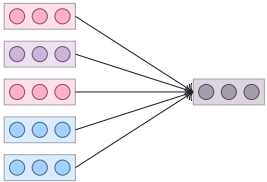
Each hidden state depends on the previous ones. Therefore, cannot parallelize, must compute in order $\boldsymbol{z}_1, \boldsymbol{z}_2, \ldots$

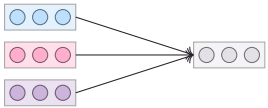
The initial state $\boldsymbol{z}_0$ is a fixed parameter.

The final state $\boldsymbol{z}_n$ has seen the entire sequence.

Pooling

$$\boldsymbol{z} = \text{AveragePool}(\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n) := \frac{1}{n} \sum_{j=1}^{n} \boldsymbol{z}_i$$



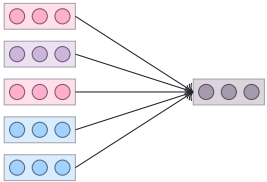


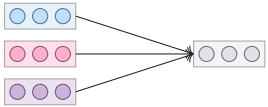
Used to get one representation of a variable-size set or sequence.

Combine n input vectors into one single output vector,
with equal contribution.

Pooling

$$\boldsymbol{z} = \text{AveragePool}(\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n) := \frac{1}{n} \sum_{j=1}^{n} \boldsymbol{z}_i$$





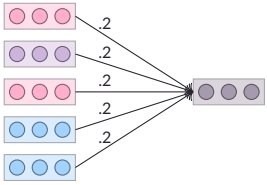
Used to get one representation of a variable-size set or sequence.

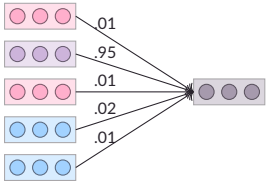
Combine n input vectors into one single output vector,
with equal contribution.

But what if some of the inputs should contribute more than others?

Weighted average pooling

$$\boldsymbol{z} = \sum_i \alpha_i \boldsymbol{z}_i, \quad \text{where} \quad \alpha_i \geq 0, \sum_i \alpha_i = 1$$

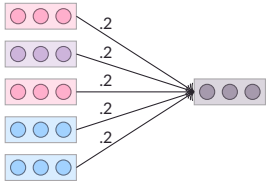


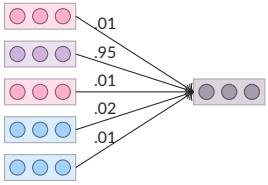


The weights $\boldsymbol{\alpha}$ control the relative importance of the inputs.

Weighted average pooling

$$\boldsymbol{z} = \sum_i \alpha_i \boldsymbol{z}_i, \quad \text{where} \quad \alpha_i \geq 0, \sum_i \alpha_i = 1$$





The weights $\boldsymbol{\alpha}$ control the relative importance of the inputs.

But how to come up with these weights?
How to decide what's important in a given context?

Attention

Key idea: have a representation of the "context" as a vector $\boldsymbol{q} \in \mathbb{R}^d$.

Then, say the importance of $\boldsymbol{z}_i$ is proportional to its alignment ($\sim$ angle) to $\boldsymbol{q}$:

$$\alpha_i = \underbrace{\frac{\exp(\boldsymbol{q} \cdot \boldsymbol{z}_i)}{\sum_j \exp(\boldsymbol{q} \cdot \boldsymbol{z}_j)}}_{[\text{softmax}([\boldsymbol{q}\cdot\boldsymbol{z_1},\ldots,\boldsymbol{q}\cdot\boldsymbol{z}_n])]_i} \quad ; \qquad \text{Attn}(\boldsymbol{q}; \boldsymbol{z}_1, \ldots, \boldsymbol{z}_n) := \sum_i \alpha_i \boldsymbol{z}_i.$$

Attention

Key idea: have a representation of the "context" as a vector $\boldsymbol{q} \in \mathbb{R}^d$.

Then, say the importance of $\boldsymbol{z}_i$ is proportional to its alignment ($\sim$ angle) to $\boldsymbol{q}$:

$$\alpha_i = \underbrace{\frac{\exp(\boldsymbol{q} \cdot \boldsymbol{z}_i)}{\sum_j \exp(\boldsymbol{q} \cdot \boldsymbol{z}_j)}}_{[\text{softmax}([\boldsymbol{q}\cdot\boldsymbol{z_1},\ldots,\boldsymbol{q}\cdot\boldsymbol{z}_n])]_i} \quad ; \qquad \text{Attn}(\boldsymbol{q}; \boldsymbol{z}_1, \ldots, \boldsymbol{z}_n) := \sum_i \alpha_i \boldsymbol{z}_i.$$

This is the basic **attention mechanism**:

Pool a bunch of vectors, with varying weights,
depending on how aligned they are with a context.

Attention

Key idea: have a representation of the "context" as a vector $\boldsymbol{q} \in \mathbb{R}^d$.

Then, say the importance of $\boldsymbol{z}_i$ is proportional to its alignment ($\sim$ angle) to $\boldsymbol{q}$:

$$\alpha_i = \underbrace{\frac{\exp(\boldsymbol{q} \cdot \boldsymbol{z}_i)}{\sum_j \exp(\boldsymbol{q} \cdot \boldsymbol{z}_j)}}_{[\text{softmax}([\boldsymbol{q}\cdot\boldsymbol{z_1},\ldots,\boldsymbol{q}\cdot\boldsymbol{z}_n])]_i} ; \qquad \text{Attn}(\boldsymbol{q}; \boldsymbol{z}_1, \ldots, \boldsymbol{z}_n) := \sum_i \alpha_i \boldsymbol{z}_i.$$

This is the basic **attention mechanism**:

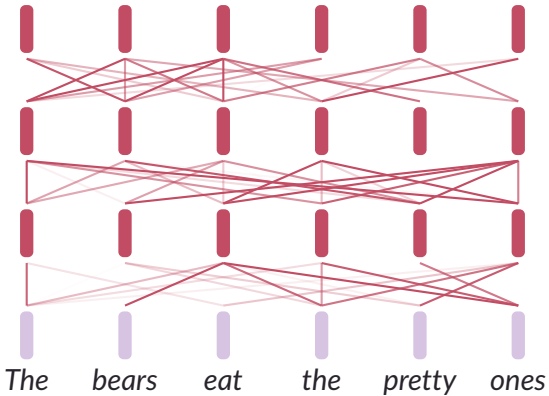
Pool a bunch of vectors, with varying weights,
depending on how aligned they are with a context.

What could be the context?

- Could be just a static learned parameter.
- If training on multiple tasks or domains, $\boldsymbol{q}$ can be an embedding of the domain.
- In machine translation (say EN→NL), $\boldsymbol{z}_i$ are the EN words,

Transformer

Stacked multi-head attention (+ some annoying details like LayerNorm)



- Combines some of the strengths of CNN and RNN:
- Global even without much depth: every output depends on every input.
- Parallelizable: each position and each head can be computed separately. (still one layer at a time)
- Sequence-aware thanks to positional embeddings.

Structure Prediction

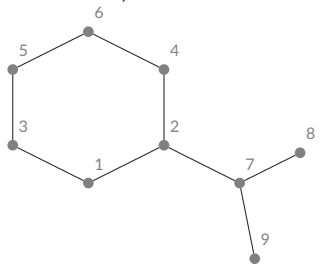
1. **Overview**
2. **Structured inputs**
   - Recap: Encoding sequences. RNN, CNN, transformer
   - Encoding graphs
3. **Structured outputs**
   - Probabilistic models of structures
   - Directed acyclic graphs
   - Algorithms for paths in DAGs: Maximization, probabilities, sampling
   - Application: Sequence tagging
   - Application: Sequence segmentation
   - Evaluating structured outputs

Encoding general graphs

Graph-structured data: proteins, molecules, social networks, etc.

A graph $\mathcal{G} = (V, E)$:

- $V = \{1, \ldots, n\}$ is the set of nodes.
- $E \subseteq V \times V$ are the edges, e.g., $(u, v) \in E$ means an edge from u to v
- Directed vs undirected graphs: in a nutshell, undirected means $(u, v) \in E \iff (v, u) \in E$.
- the adjacenty matrix $\boldsymbol{A} \in \{0, 1\}^{n \times n}$ encodes the set of edges E: $a_{uv} = 1 \iff (u, v) \in E$.



Each node can have a *type* (e.g., carbon, hydrogen, ...).

For simplicity, we assume all edges are of the same type.

Graph datasets

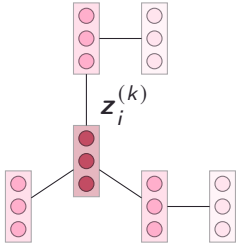
(Hamilton, 2020)

Two main scenarios, but the tools we use are the same

1. Each data point $\boldsymbol{x}^{(i)}$ is a graph.
   - e.g., molecule solubility, malicious software detection, protein classification, …
   - in NLP: syntactic/semantic–annotated texts, natural language generation (from AMR, from knowledge graphs).
   - can be given as a sequence of node labels $(x_1^{(i)}, \ldots, x_{n_i}^{(i)})$ and an adjacency matrix $\boldsymbol{A}^{(i)}$
2. Data points are parts of one big graph.
   - e.g., node classification (classifying bots on twitter), link prediction (instagram follow suggestions), community detection, …
   - in NLP: linking, knowledge base completion
   - harder to set up experiments, dev set/test set, etc.

Node representations with graph neural nets

Encoding a **graph** of input vectors $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \rightarrow (\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n)$:



- We apply an iterative process.
- At iteration 0, $\boldsymbol{z}_i^{(0)} = \boldsymbol{x}_i$ (the input embedding)
- At each iteration, a node's embedding is updated as a function of the embeddings of its neighbors, i.e., message passing along the edges:

$$\boldsymbol{m}_i^{(k)} = \sum_{j \in N(i)} \boldsymbol{z}_j^{(k)}$$

$$\boldsymbol{z}_i^{(k+1)} = \tanh\left(\boldsymbol{W}_{\text{self}} \boldsymbol{z}_i^{(k)} + \boldsymbol{W}_{\text{neigh}} \boldsymbol{m}_i^{(k)} + \boldsymbol{b}\right)$$

- Apply this update in parallel for every node, then repeat.

Pooling

As defined, a GNN gives us rich embeddings of every node.

To get a single embedding of the entire graph, we turn again to pooling.

Unlike for RNNs, there is no single node that could be taken as representative of the entire graph (especially if k is small and the graph is wide).

We turn to the kind of pooling used for CNNs:

1. average pooling: $\boldsymbol{z} = \frac{1}{n}(\boldsymbol{z}_1 + \ldots + \boldsymbol{z}_n)$
2. max pooling: $[\boldsymbol{z}]_j = \max([\boldsymbol{z}_1]_j, \ldots, [\boldsymbol{z}_n]_j)$

Permutation equivariance

The structure of a graph doesn't change if we number the nodes in another order.

The output of a GNN should not change either.

Mathematically, given a graph represented as $(\boldsymbol{X}, \boldsymbol{A})$, for any permutation matrix $\boldsymbol{P}$, a GNN satisfies

$$\mathrm{GNN}(\boldsymbol{P}\boldsymbol{X}, \boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^\top) = \boldsymbol{P}\,\mathrm{GNN}(\boldsymbol{X}, \boldsymbol{A}).$$

GNN variants

Many variations can be built on top of this idea.

- The update $\boldsymbol{z}_i^{(k+1)} = \tanh(\boldsymbol{W}_{\text{self}}\boldsymbol{z}_i^{(k)} + \boldsymbol{W}_{\text{neigh}}\boldsymbol{m}_i^{(k)} + \boldsymbol{b})$ resembles an RNN.
  $\rightarrow$ gated variants (GGNN)!
- Separate weight matrices per iteration $(\boldsymbol{W}_{\{\text{self,neigh}\}}^{(k)}, \boldsymbol{b}^{(k)})$
- Supporting different edge types:
  - first, notice that $\boldsymbol{W}_{\text{neigh}} \sum_j \boldsymbol{z}_j = \sum_j \boldsymbol{W}_{\text{neigh}}\boldsymbol{z}_j$.
  - then, if $e(i,j)$ is the type of the edge from i to j, we could compute $\sum_j \boldsymbol{W}_{e(i,j)}\boldsymbol{z}_j$.
- Different normalization over neighbors (more next time).

Self-attention for graphs

Self-attention (and thus Transformers) are permutation equivariant.

Remember in GNN we computed the message from neighbors as a sum:

$$\boldsymbol{m}_i = \sum_{j \in N(i)} \boldsymbol{z}_i$$

Self-attention for graphs

Self-attention (and thus Transformers) are permutation equivariant.

Remember in GNN we computed the message from neighbors as a sum:

$$\boldsymbol{m}_i = \sum_{j \in N(i)} \boldsymbol{z}_i$$

Instead, self-attention over neighbors:

$$\alpha_{ij} = \frac{\exp(\boldsymbol{q}_i \cdot \boldsymbol{k}_j)}{\sum_{j' \in N(i)} \exp(\boldsymbol{q}_i \cdot \boldsymbol{k}_{j'})}$$

$$\boldsymbol{m}_i = \sum_{j \in N(i)} \alpha_{ij} \boldsymbol{v}_j$$

Self-attention for graphs

Self-attention (and thus Transformers) are permutation equivariant.

Remember in GNN we computed the message from neighbors as a sum:

$$\boldsymbol{m}_i = \sum_{j \in N(i)} \boldsymbol{z}_i$$

Instead, self-attention over neighbors:

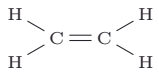
$$\alpha_{ij} = \frac{\exp(\boldsymbol{q}_i \cdot \boldsymbol{k}_j)}{\sum_{j' \in N(i)} \exp(\boldsymbol{q}_i \cdot \boldsymbol{k}_{j'})}$$

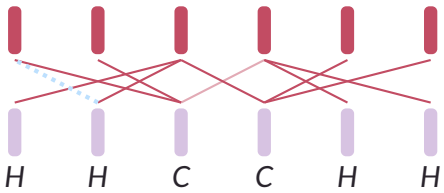
$$\boldsymbol{m}_i = \sum_{j \in N(i)} \alpha_{ij} \boldsymbol{v}_j$$

In other words: self-attention constrained by the adjacency structure

(no attention allowed where there is no edge)

Ethylene (C_2H_4):





Structure Prediction

1. **Overview**
2. **Structured inputs**
   - Recap: Encoding sequences. RNN, CNN, transformer
   - Encoding graphs
3. **Structured outputs**
   - Probabilistic models of structures
   - Directed acyclic graphs
   - Algorithms for paths in DAGs: Maximization, probabilities, sampling
   - Application: Sequence tagging
   - Application: Sequence segmentation
   - Evaluating structured outputs

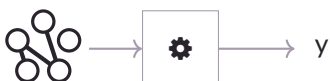
Structure Prediction

1. **Overview**
2. **Structured inputs**
   - Recap: Encoding sequences. RNN, CNN, transformer
   - Encoding graphs
3. **Structured outputs**
   - Probabilistic models of structures
   - Directed acyclic graphs
   - Algorithms for paths in DAGs: Maximization, probabilities, sampling
   - Application: Sequence tagging
   - Application: Sequence segmentation
   - Evaluating structured outputs

Smith, 2011

So far, we've studied this scenario:

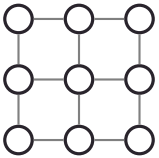
- Structured inputs
- Familiar unstructured outputs: classification / regression.



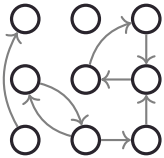
In the next part of class, we study **structured outputs**.

Reminder: Kinds of structure

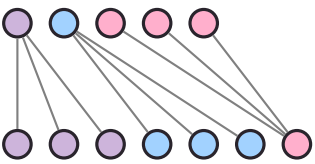
Sequence



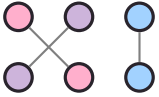
Grid



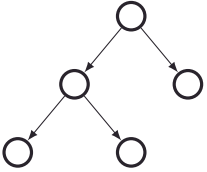
Graph



Alignments



Permutations



Hierarchy

Structured outputs are:

- discrete objects
- made of smaller parts
- which interact with each other and/or constrain each other.

Structured outputs are:

- discrete objects
- made of smaller parts
- which interact with each other and/or constrain each other.

Example: What are the possible ways to assign 4 jockeys to 4 horses?

$$\begin{aligned}\mathcal{Y} = \{&(1,2,3,4),\\ &(1,2,4,3),\\ &(1,3,2,4),\\ &\dots,\\ &(4,3,2,1)\}\end{aligned}$$

Structured outputs are:

- discrete objects
- made of smaller parts
- which interact with each other and/or constrain each other.

Example: What are the possible ways to assign 4 jockeys to 4 horses?

$$\begin{aligned}\mathcal{Y} = \{&(1,2,3,4),\\ &(1,2,4,3),\\ &(1,3,2,4),\\ &\dots,\\ &(4,3,2,1)\}\end{aligned}$$

We can't just predict the best jockey for each horse, or the best horse for each jockey, since we might end up with double assignments.

Structured outputs are:

- discrete objects
- made of smaller parts
- which interact with each other and/or constrain each other.

Example: What are the possible ways to assign 4 jockeys to 4 horses?

$$\begin{aligned}\mathcal{Y} = \{&(1,2,3,4),\\ &(1,2,4,3),\\ &(1,3,2,4),\\ &\ldots,\\ &(4,3,2,1)\}\end{aligned}$$

We can't just predict the best jockey for each horse, or the best horse for each jockey, since we might end up with double assignments.

What is $|\mathcal{Y}|$?

Recap: Logistic regression and perceptron losses

The two losses we've seen for multi-class classification:
(changing notation slightly)

$$L_{\text{LR}}(y) = -\log \Pr(Y = y|x) \qquad = -\text{score}(y) + \log \sum_{y' \in \mathcal{Y}} \exp\left(\text{score}(y')\right)$$

$$L_{\text{Perc}}(y) \qquad = -\text{score}(y) + \max_{y' \in \mathcal{Y}} \text{score}(y')$$

For classification:

- we had $\mathcal{Y} = \{1, 2, \ldots, K\}$
- the model (linear or NN) outputs a vector $\boldsymbol{a}$ of scores for each class, so $\text{score}(y) = a_y$.

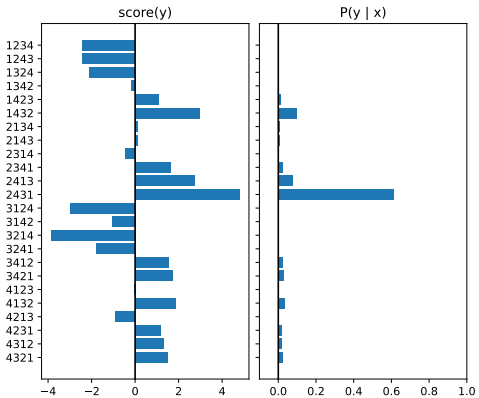
Can we generalize this to structured $\mathcal{Y}$?

Probabilistic models of structures

Our model must be able to assign a score to every possible structure, $\text{score}(y; x, \theta)$. For brevity we just write $\text{score}(y)$, but remember it depends on input and params.

From this, we can get a probability distribution over possible structures:

$$\Pr(y \mid x) = \frac{\exp\left(\text{score}(y)\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(\text{score}(y')\right)}$$



Modelling challenges

Essential computational prerequisites:

- $\text{score}(y)$
- for prediction: $\arg\max_{y \in \mathcal{Y}} \text{score}(y)$
- for learning: $\log \sum_{y \in \mathcal{Y}} \exp\left(\text{score}(y)\right)$

The challenges: unlike multi-class classification,

- $\mathcal{Y}$ can vary for each data point (e.g., with n. horses)
- $|\mathcal{Y}|$ can get very large: we can't just for-loop over it.

Generally intractable!
But, for certain structures and scoring functions, efficient algorithms exist.

Structure Prediction

1. **Overview**
2. **Structured inputs**
   - Recap: Encoding sequences. RNN, CNN, transformer
   - Encoding graphs
3. **Structured outputs**
   - Probabilistic models of structures
   - Directed acyclic graphs
   - Algorithms for paths in DAGs: Maximization, probabilities, sampling
   - Application: Sequence tagging
   - Application: Sequence segmentation
   - Evaluating structured outputs

Computations for structures

Recall: Structured outputs are:

- discrete objects
- made of smaller parts
- which interact with each other and/or constrain each other,

and we must know how to compute:

- $\text{score}(y)$
- for prediction: $\arg\max_{y \in \mathcal{Y}} \text{score}(y)$
- for learning: $\log \sum_{y \in \mathcal{Y}} \exp\left(\text{score}(y)\right)$

For large problems, we can't enumerate $\mathcal{Y}$ (could be exponentially large).

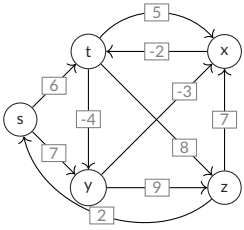
So, we must actually make use of its structure.

Recap: Graphs

Definition 1: Weighted directed graph

A weighted directed graph is $G = (V, E, w)$ where:

- V is the set of vertices (nodes) of G.
- $E \subset V \times V$ is the set of arcs of G:
  $uv \in E$ means there is an arc from node $u \in V$ to node $v \in V$ ($u \neq v$).
  Arcs are ordered pairs, so $uv \neq vu$.
- $w : E \to \mathbb{R}$ is a weight function assigning a weight to each edge.

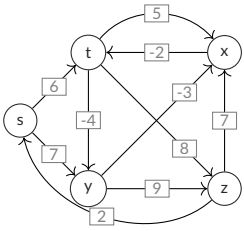


Recap: Graphs

Definition 1: Weighted directed graph

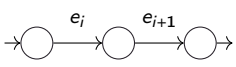
A weighted directed graph is $G = (V, E, w)$ where:

- V is the set of vertices (nodes) of G.
- $E \subset V \times V$ is the set of arcs of G:
  $uv \in E$ means there is an arc from node $u \in V$ to node $v \in V$ $(u \neq v)$.
  Arcs are ordered pairs, so $uv \neq vu$.
- $w : E \to \mathbb{R}$ is a weight function assigning a weight to each edge.



Definition 2: Paths

A path A in G is a sequence of edges: $A = e_1 e_2 \ldots e_k$, with each $e_i \in E$, two-by-two "linked", i.e., if $e_i = u_i v_i$ and $e_{i+1} = u_{i+1} v_{i+1}$ then we must have $v_i = u_{i+1}$.

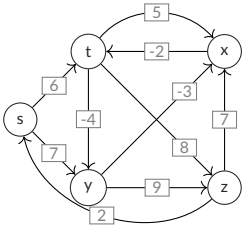


Recap: Graphs

Definition 1: Weighted directed graph

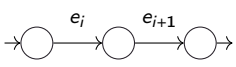
A weighted directed graph is $G = (V, E, w)$ where:

- V is the set of vertices (nodes) of G.
- $E \subset V \times V$ is the set of arcs of G:
  $uv \in E$ means there is an arc from node $u \in V$ to node $v \in V$ ($u \neq v$).
  Arcs are ordered pairs, so $uv \neq vu$.
- $w : E \to \mathbb{R}$ is a weight function assigning a weight to each edge.



Definition 2: Paths

A path A in G is a sequence of edges: $A = e_1 e_2 \dots e_k$, with each $e_i \in E$, two-by-two "linked", i.e., if $e_i = u_i v_i$ and $e_{i+1} = u_{i+1} v_{i+1}$ then we must have $v_i = u_{i+1}$.



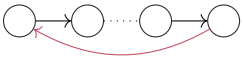
The weight of a path is the sum of arc weights: $w(A) = \sum_{e \in P} w(e)$.

We denote path concatenation by $A_1 \frown A_2$ (when legal).

Directed acyclic graphs

Definition 3: Cycle

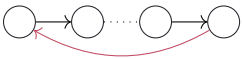
A cycle is a path $e_1 e_2 \dots e_k$ wherein the last edge e_k points to the node from which the first edge e_1 departs.



Directed acyclic graphs

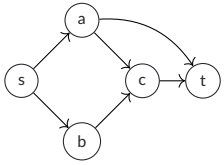
Definition 3: Cycle

A cycle is a path $e_1 e_2 \ldots e_k$ wherein the last edge e_k points to the node from which the first edge e_1 departs.



Definition 4. Directed acyclic graph (DAG)

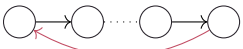
A DAG is a directed graph that contains no cycles.



Directed acyclic graphs

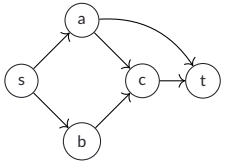
Definition 3: Cycle

A cycle is a path $e_1 e_2 \dots e_k$ wherein the last edge e_k points to the node from which the first edge e_1 departs.



Definition 4. Directed acyclic graph (DAG)

A DAG is a directed graph that contains no cycles.



Definition 4. Topological ordering

A topological ordering of a directed graph $G = (V, E)$ is an ordering of its nodes $v_1, v_2, \dots, v_n$ such that if $v_i v_j \in E$ then $i < j$.

TOs:
s, *a*, *b*, *c*, *t*
s, *b*, *a*, *c*, *t*

G is a DAG if and only if G admits a topological ordering.
Rough intuition: "backward" edges against the ordering $\iff$ cycles.

Structure Prediction

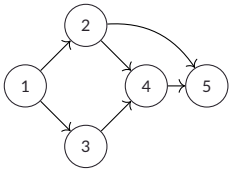
1. **Overview**
2. **Structured inputs**
   - Recap: Encoding sequences. RNN, CNN, transformer
   - Encoding graphs
3. **Structured outputs**
   - Probabilistic models of structures
   - Directed acyclic graphs
   - Algorithms for paths in DAGs: Maximization, probabilities, sampling
   - Application: Sequence tagging
   - Application: Sequence segmentation
   - Evaluating structured outputs

Paths in DAGs

(Smith, 2011, Sec. 2.2.4)

Label nodes in topological order $V = \{1, \ldots, n\}$.

Let $\mathcal{Y}_i$ be the set of paths starting at 1 and ending at i.



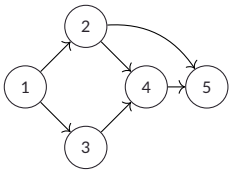
Paths in DAGs

(Smith, 2011, Sec. 2.2.4)

Label nodes in topological order $V = \{1, \ldots, n\}$.

Let $\mathcal{Y}_i$ be the set of paths starting at 1 and ending at i.

Let's assume our space of structures is $\mathcal{Y} = \mathcal{Y}_n$.



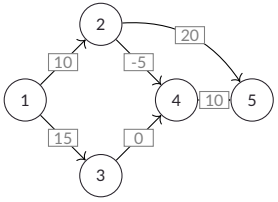
Important things to compute:

- $\text{score}(y) = w(y)$
- $\text{argmax}_{y \in \mathcal{Y}_n} w(y)$
- $\log \sum_{y \in \mathcal{Y}_n} \exp w(y)$

Max-scoring path

(Smith, 2011, Sec. 2.2.4)

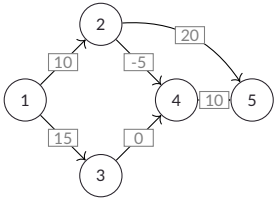
- The greedy path from 1 to 5 might not be best.
- From *Data Structures and Algorithms* you might recall Dijkstra's algorithm.
  - Requires no "negative cycles" — always true for DAGs.
  - Complexity: $\Theta(|V| \log |V| + |E|)$ with "Fibonacci heaps"; $\Theta(|V|^2)$ with a straightforward implementation. .



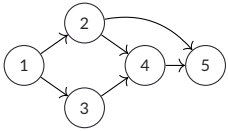
Max-scoring path

(Smith, 2011, Sec. 2.2.4)

- The greedy path from 1 to 5 might not be best.
- From *Data Structures and Algorithms* you might recall Dijkstra's algorithm.
  - Requires no "negative cycles" — always true for DAGs.
  - Complexity: $\Theta(|V| \log |V| + |E|)$ with "Fibonacci heaps"; $\Theta(|V|^2)$ with a straightforward implementation. .
- In the case of DAGs, we can do better.



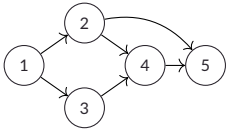
Dynamic programming recurrence



Goal: the max weight of a path from 1 to i:

$$m_i = \max_{y \in \mathcal{Y}_i} w(y).$$

Dynamic programming recurrence



Goal: the max weight of a path from 1 to i:

$$m_i = \max_{y \in \mathcal{Y}_i} w(y).$$

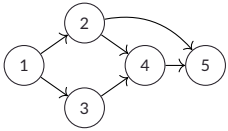
Define predecessors of i as $P_i := \{j \in V : ji \in E\}$.

Insight 1.

Any path from to i is an extension of some path to predecessor $j \in P_i$ by arc ji.

In other words: if $y \in \mathcal{Y}_i$ then $y = y' \frown ji$ for some $j \in P_i$ and some $y' \in \mathcal{Y}_j$.

Dynamic programming recurrence



Goal: the max weight of a path from 1 to i:

$$m_i = \max_{y \in \mathcal{Y}_i} w(y).$$

Define predecessors of i as $P_i := \{j \in V : ji \in E\}$.

Insight 1.

Any path from to i is an extension of some path to predecessor $j \in P_i$ by arc ji.

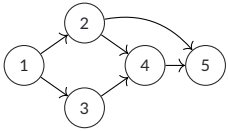
In other words: if $y \in \mathcal{Y}_i$ then $y = y' \frown ji$ for some $j \in P_i$ and some $y' \in \mathcal{Y}_j$.

Proposition: DP recurrence for max

For any $i > 1$, the best path from 1 to i is the best among the extensions of the best path to the predecessors of i:

$$m_i = \max_{j \in P_i} \big(m_j + w(ji)\big)$$

Dynamic programming recurrence



Goal: the max weight of a path from 1 to i:

$$m_i = \max_{y \in \mathcal{Y}_i} w(y).$$

Define predecessors of i as $P_i := \{j \in V : ji \in E\}$.

Insight 1.

Any path from to i is an extension of some path to predecessor $j \in P_i$ by arc ji.

In other words: if $y \in \mathcal{Y}_i$ then $y = y' \frown ji$ for some $j \in P_i$ and some $y' \in \mathcal{Y}_j$.

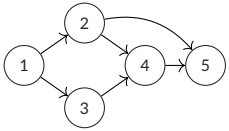
Proposition: DP recurrence for max

For any $i > 1$, the best path from 1 to i is the best among the extensions of the best path to the predecessors of i:

$$m_i = \max_{j \in P_i} \big(m_j + w(ji)\big)$$

Proof: $m_i := \max_{y \in \mathcal{Y}_i} w(y)$

Dynamic programming recurrence



Goal: the max weight of a path from 1 to i:

$$m_i = \max_{y \in \mathcal{Y}_i} w(y).$$

Define predecessors of i as $P_i := \{j \in V : ji \in E\}$.

Insight 1.

Any path from to i is an extension of some path to predecessor $j \in P_i$ by arc ji.

In other words: if $y \in \mathcal{Y}_i$ then $y = y'^\frown ji$ for some $j \in P_i$ and some $y' \in \mathcal{Y}_j$.

Proposition: DP recurrence for max

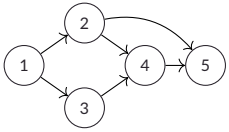
For any $i > 1$, the best path from 1 to i is the best among the extensions of the best path to the predecessors of i:

$$m_i = \max_{j \in P_i} \big(m_j + w(ji)\big)$$

Proof: $m_i := \max_{y \in \mathcal{Y}_i} w(y)$

$$= \max_{j \in P_i} \max_{y' \in \mathcal{Y}_j} (w(y') + w(ji))$$

Dynamic programming recurrence



Goal: the max weight of a path from 1 to i:

$$m_i = \max_{y \in \mathcal{Y}_i} w(y).$$

Define predecessors of i as $P_i := \{j \in V : ji \in E\}$.

Insight 1.

Any path from to i is an extension of some path to predecessor $j \in P_i$ by arc ji.

In other words: if $y \in \mathcal{Y}_i$ then $y = y' \frown ji$ for some $j \in P_i$ and some $y' \in \mathcal{Y}_j$.

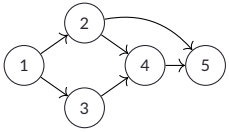
Proposition: DP recurrence for max

For any $i > 1$, the best path from 1 to i is the best among the extensions of the best path to the predecessors of i:

$$m_i = \max_{j \in P_i} \big(m_j + w(ji)\big)$$

Proof:
$$\begin{aligned} m_i &:= \max_{y \in \mathcal{Y}_i} w(y) \\ &= \max_{j \in P_i} \max_{y' \in \mathcal{Y}_j} \big(w(y') + w(ji)\big) \\ &= \max_{j \in P_i} \left(\max_{y' \in \mathcal{Y}_j} (w(y')) + w(ji) \right) \end{aligned}$$

Dynamic programming recurrence



Goal: the max weight of a path from 1 to i:

$$m_i = \max_{y \in \mathcal{Y}_i} w(y).$$

Define predecessors of i as $P_i := \{j \in V : ji \in E\}$.

Insight 1.

Any path from to i is an extension of some path to predecessor $j \in P_i$ by arc ji.

In other words: if $y \in \mathcal{Y}_i$ then $y = y'^\frown ji$ for some $j \in P_i$ and some $y' \in \mathcal{Y}_j$.

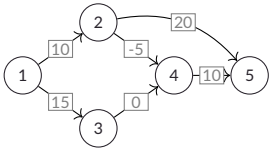
Proposition: DP recurrence for max

For any $i > 1$, the best path from 1 to i is the best among the extensions of the best path to the predecessors of i:

$$m_i = \max_{j \in P_i} \big(m_j + w(ji)\big)$$

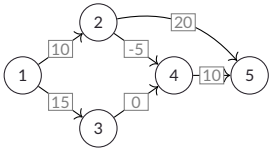
Proof:
$$\begin{aligned} m_i &:= \max_{y \in \mathcal{Y}_i} w(y) \\ &= \max_{j \in P_i} \max_{y' \in \mathcal{Y}_j} \big(w(y') + w(ji)\big) \\ &= \max_{j \in P_i} \left(\max_{y' \in \mathcal{Y}_j} (w(y')) + w(ji) \right) \\ &= \max_{j \in P_i} \big(m_j + w(ji)\big) . \end{aligned}$$

The Viterbi algorithm



$m_i = \max_{j \in P_i} \left(m_j + w(ji) \right)$ holds for any graph;
but we would chase our own tail forever.

The Viterbi algorithm



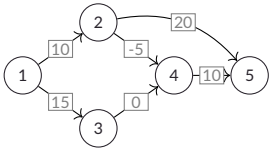
$m_i = \max_{j \in P_i} \left(m_j + w(ji) \right)$ holds for any graph; but we would chase our own tail forever.

Insight 2.

In a topologically-ordered DAG, any path from 1 to i must only contain nodes $j < i$.

(So, we may compute $m_1, \ldots, m_n$ in order.)

The Viterbi algorithm



$m_i = \max_{j \in P_i} \left(m_j + w(ji)\right)$ holds for any graph; but we would chase our own tail forever.

Insight 2.

In a topologically-ordered DAG, any path from 1 to i must only contain nodes $j < i$.

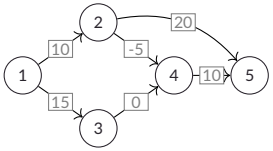
(So, we may compute $m_1, \ldots, m_n$ in order.)

General Viterbi algorithm for DAGs

input: Topologically-ordered DAG $G = (V, E, w)$, $V = \{1, \ldots, n\}$
output: maximum path weights $m_1, \ldots, m_n$.

initialize $m_1 \leftarrow 0$
for $i = 2, \ldots, n$ **do**
$m_i \leftarrow \max_{j \in P_i} \left(m_j + w(ji)\right)$

The Viterbi algorithm



$m_i = \max_{j \in P_i} \left(m_j + w(ji)\right)$ holds for any graph; but we would chase our own tail forever.

Insight 2.

In a topologically-ordered DAG, any path from 1 to i must only contain nodes $j < i$.

(So, we may compute $m_1, \ldots, m_n$ in order.)

Insight 3.

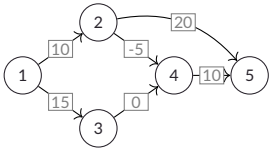
A path acheiving maximal weight is made up of the edges $j^\star i$, where $j^\star$ is the node selected by the max at each iteration.

General Viterbi algorithm for DAGs

input: Topologically-ordered DAG
$G = (V, E, w)$, $V = \{1, \ldots, n\}$
output: maximum path weights $m_1, \ldots, m_n$.

initialize $m_1 \leftarrow 0$
for $i = 2, \ldots, n$ **do**
$m_i \leftarrow \max_{j \in P_i} \left(m_j + w(ji)\right)$

The Viterbi algorithm



$m_i = \max_{j \in P_i} \left(m_j + w(ji)\right)$ holds for any graph; but we would chase our own tail forever.

Insight 2.

In a topologically-ordered DAG, any path from 1 to i must only contain nodes $j < i$.

(So, we may compute $m_1, \ldots, m_n$ in order.)

Insight 3.

A path acheiving maximal weight is made up of the edges $j^\star i$, where $j^\star$ is the node selected by the max at each iteration.

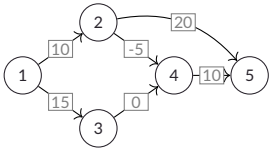
General Viterbi algorithm for DAGs

input: Topologically-ordered DAG
$G = (V, E, w)$, $V = \{1, \ldots, n\}$
output: maximum path weights $m_1, \ldots, m_n$.

initialize $m_1 \leftarrow 0$
for $i = 2, \ldots, n$ **do**
$m_i \leftarrow \max_{j \in P_i} \left(m_j + w(ji)\right)$
$\pi_i \leftarrow \arg\max_{j \in P_i} \left(m_j + w(ji)\right)$

Reconstruct path: follow backpointers
output: optimal path y from 1 to n (optional)
$y = []$; $i \leftarrow n$
while $i > 1$ **do**
$y \leftarrow \pi_i i^\frown y$
$i \leftarrow \pi_i$

The Viterbi algorithm



$m_i = \max_{j \in P_i} \left(m_j + w(ji)\right)$ holds for any graph; but we would chase our own tail forever.

Insight 2.

In a topologically-ordered DAG, any path from 1 to i must only contain nodes $j < i$.

(So, we may compute $m_1, \ldots, m_n$ in order.)

Insight 3.

A path acheiving maximal weight is made up of the edges $j^\star i$, where $j^\star$ is the node selected by the max at each iteration.

General Viterbi algorithm for DAGs

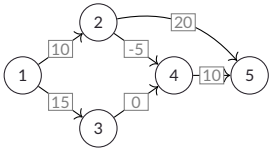
input: Topologically-ordered DAG
$G = (V, E, w)$, $V = \{1, \ldots, n\}$
output: maximum path weights $m_1, \ldots, m_n$.

initialize $m_1 \leftarrow 0$
for $i = 2, \ldots, n$ **do**
$m_i \leftarrow \max_{j \in P_i} \left(m_j + w(ji)\right)$
$\pi_i \leftarrow \arg\max_{j \in P_i} \left(m_j + w(ji)\right)$

Reconstruct path: follow backpointers
output: optimal path y from 1 to n (optional)
$y = []$; $i \leftarrow n$
while $i > 1$ **do**
$y \leftarrow \pi_i i \frown y$
$i \leftarrow \pi_i$

Complexity: $\Theta(|V| + |E|)$.

The Viterbi algorithm



General Viterbi algorithm for DAGs

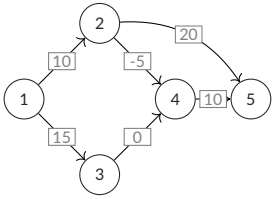
input: Topologically-ordered DAG
$G = (V, E, w)$, $V = \{1, \ldots, n\}$
output: maximum path weights $m_1, \ldots, m_n$.

initialize $m_1 \leftarrow 0$
for $i = 2, \ldots, n$ **do**
$m_i \leftarrow \max_{j \in P_i} \left(m_j + w(ji)\right)$
$\pi_i \leftarrow \arg\max_{j \in P_i} \left(m_j + w(ji)\right)$

Reconstruct path: follow backpointers
output: optimal path y from 1 to n (optional)
$y = []$; $i \leftarrow n$
while $i > 1$ **do**
$y \leftarrow \pi_i i \frown y$
$i \leftarrow \pi_i$

Complexity: $\Theta(|V| + |E|)$.

Probability distributions



A weighted DAG induces a probability distributions over all paths from 1 to n:

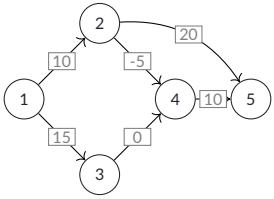
$$\Pr(y) = \frac{\exp(w(y))}{\sum_{y' \in \mathcal{Y}_n} \exp(w(y'))}$$

y	$w(y)$	$\exp(w(y))$	$\Pr(y)$
$1 \to 2 \to 5$			
$1 \to 2 \to 4 \to 5$			
$1 \to 3 \to 4 \to 5$			

To assess $\Pr(y)$ even for a single path, the denominator sums over all paths.

Next goal: calculate this denominator efficiently.

Probability distributions



A weighted DAG induces a probability distributions over all paths from 1 to n:

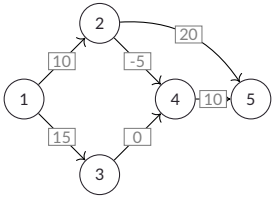
$$\Pr(y) = \frac{\exp(w(y))}{\sum_{y' \in \mathcal{Y}_n} \exp(w(y'))}$$

y	$w(y)$	$\exp(w(y))$	$\Pr(y)$
$1 \to 2 \to 5$	$10 + 20 = 30$		
$1 \to 2 \to 4 \to 5$	$10 - 5 + 10 = 15$		
$1 \to 3 \to 4 \to 5$	$15 + 0 + 10 = 25$		

To assess $\Pr(y)$ even for a single path, the denominator sums over all paths.

Next goal: calculate this denominator efficiently.

Probability distributions



A weighted DAG induces a probability distributions over all paths from 1 to n:

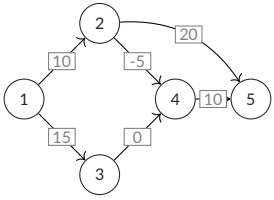
$$\Pr(y) = \frac{\exp(w(y))}{\sum_{y' \in \mathcal{Y}_n} \exp(w(y'))}$$

y	$w(y)$	$\exp(w(y))$	$\Pr(y)$
$1 \to 2 \to 5$	$10 + 20 = 30$	$1.1 \cdot 10^{13}$	
$1 \to 2 \to 4 \to 5$	$10 - 5 + 10 = 15$	$3.3 \cdot 10^{6}$	
$1 \to 3 \to 4 \to 5$	$15 + 0 + 10 = 25$	$7.2 \cdot 10^{10}$	

To assess $\Pr(y)$ even for a single path, the denominator sums over all paths.

Next goal: calculate this denominator efficiently.

Probability distributions



A weighted DAG induces a probability distributions over all paths from 1 to n:

$$\Pr(y) = \frac{\exp(w(y))}{\sum_{y' \in \mathcal{Y}_n} \exp(w(y'))}$$

y	$w(y)$	$\exp(w(y))$	$\Pr(y)$
$1 \to 2 \to 5$	$10 + 20 = 30$	$1.1 \cdot 10^{13}$	.9930
$1 \to 2 \to 4 \to 5$	$10 - 5 + 10 = 15$	$3.3 \cdot 10^{6}$	.0001
$1 \to 3 \to 4 \to 5$	$15 + 0 + 10 = 25$	$7.2 \cdot 10^{10}$	.0069

To assess $\Pr(y)$ even for a single path, the denominator sums over all paths.

Next goal: calculate this denominator efficiently.

Log-probability DP recurrence

Since $\exp w(y)$ can be huge, it's better to work with log-probabilities:

$$\log \Pr(y) = w(y) - \log \sum_{y' \in \mathcal{Y}_n} \exp w(y')$$

so we aim to compute this log-sum-exp directly.

Log-probability DP recurrence

Since $\exp w(y)$ can be huge, it's better to work with log-probabilities:

$$\log \Pr(y) = w(y) - \log \sum_{y' \in \mathcal{Y}_n} \exp w(y')$$

so we aim to compute this log-sum-exp directly.

Insight 1 (from before).

If $y \in \mathcal{Y}_i$ then $y = y' \frown ji$ for some $j \in P_i$ and some $y' \in \mathcal{Y}_j$.

Log-probability DP recurrence

Since $\exp w(y)$ can be huge, it's better to work with log-probabilities:

$$\log \Pr(y) = w(y) - \log \sum_{y' \in \mathcal{Y}_n} \exp w(y')$$

so we aim to compute this log-sum-exp directly.

Insight 1 (from before).

If $y \in \mathcal{Y}_i$ then $y = y' \frown ji$ for some $j \in P_i$ and some $y' \in \mathcal{Y}_j$.

Insight 4: addition distributes over log-sum-exp.

$$c + \log \sum_i \exp(z_i) = \log \sum_i \exp(c + z_i)$$

Log-probability DP recurrence

Since $\exp w(y)$ can be huge, it's better to work with log-probabilities:

$$\log \Pr(y) = w(y) - \log \sum_{y' \in \mathcal{Y}_n} \exp w(y')$$

so we aim to compute this log-sum-exp directly.

Insight 1 (from before).

If $y \in \mathcal{Y}_i$ then $y = y' \frown ji$ for some $j \in P_i$ and some $y' \in \mathcal{Y}_j$.

Insight 4: addition distributes over log-sum-exp.

$$c + \log \sum_i \exp(z_i) = \log \sum_i \exp(c + z_i)$$

Denote $q_i := \log \sum_{y \in \mathcal{Y}_i} \exp(w(y))$.

Proposition: DP recurrence for log-sum-exp.

$$q_i = \log \sum_{j \in P_i} \exp\left(q_j + w(ji)\right)$$

Compare with the DP recurrence for max:

$$m_i = \max_{j \in P_i} (m_j + w(ji)).$$

Log-probability DP recurrence

Since $\exp w(y)$ can be huge, it's better to work with log-probabilities:

$$\log \Pr(y) = w(y) - \log \sum_{y' \in \mathcal{Y}_n} \exp w(y')$$

so we aim to compute this log-sum-exp directly.

Insight 1 (from before).

If $y \in \mathcal{Y}_i$ then $y = y' \frown ji$ for some $j \in P_i$ and some $y' \in \mathcal{Y}_j$.

Insight 4: addition distributes over log-sum-exp.

$$c + \log \sum_i \exp(z_i) = \log \sum_i \exp(c + z_i)$$

Denote $q_i := \log \sum_{y \in \mathcal{Y}_i} \exp(w(y))$.

Proposition: DP recurrence for log-sum-exp.

$$q_i = \log \sum_{j \in P_i} \exp\left(q_j + w(ji)\right)$$

Compare with the DP recurrence for max:

$$m_i = \max_{j \in P_i} (m_j + w(ji)).$$

Proof: $q_i = \log \sum_{j \in P_i} \sum_{y' \in \mathcal{Y}_j} \exp\left(w(y') + w(ji)\right)$

Log-probability DP recurrence

Since $\exp w(y)$ can be huge, it's better to work with log-probabilities:

$$\log \Pr(y) = w(y) - \log \sum_{y' \in \mathcal{Y}_n} \exp w(y')$$

so we aim to compute this log-sum-exp directly.

Insight 1 (from before).

If $y \in \mathcal{Y}_i$ then $y = y' \frown ji$ for some $j \in P_i$ and some $y' \in \mathcal{Y}_j$.

Insight 4: addition distributes over log-sum-exp.

$$c + \log \sum_i \exp(z_i) = \log \sum_i \exp(c + z_i)$$

Denote $q_i := \log \sum_{y \in \mathcal{Y}_i} \exp(w(y))$.

Proposition: DP recurrence for log-sum-exp.

$$q_i = \log \sum_{j \in P_i} \exp\left(q_j + w(ji)\right)$$

Compare with the DP recurrence for max:

$$m_i = \max_{j \in P_i} (m_j + w(ji)).$$

Proof:

$$q_i = \log \sum_{j \in P_i} \sum_{y' \in \mathcal{Y}_j} \exp\left(w(y') + w(ji)\right)$$

$$= \log \sum_{j \in P_i} \exp\left(\log \sum_{y' \in \mathcal{Y}_j} \exp(w(y')) + w(ji) \right)$$

Log-probability DP recurrence

Since $\exp w(y)$ can be huge, it's better to work with log-probabilities:

$$\log \Pr(y) = w(y) - \log \sum_{y' \in \mathcal{Y}_n} \exp w(y')$$

so we aim to compute this log-sum-exp directly.

Insight 1 (from before).

If $y \in \mathcal{Y}_i$ then $y = y' \frown ji$ for some $j \in P_i$ and some $y' \in \mathcal{Y}_j$.

Insight 4: addition distributes over log-sum-exp.

$$c + \log \sum_i \exp(z_i) = \log \sum_i \exp(c + z_i)$$

Denote $q_i := \log \sum_{y \in \mathcal{Y}_i} \exp(w(y))$.

Proposition: DP recurrence for log-sum-exp.

$$q_i = \log \sum_{j \in P_i} \exp\left(q_j + w(ji)\right)$$

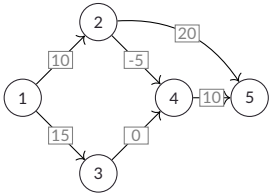
Compare with the DP recurrence for max:

$$m_i = \max_{j \in P_i} (m_j + w(ji)).$$

Proof:

$$\begin{aligned} q_i &= \log \sum_{j \in P_i} \sum_{y' \in \mathcal{Y}_j} \exp\left(w(y') + w(ji)\right) \\ &= \log \sum_{j \in P_i} \exp\left(\log \sum_{y' \in \mathcal{Y}_j} \exp(w(y')) + w(ji)\right) \\ &= \log \sum_{j \in P_i} \exp\left(q_j + w(ji)\right). \end{aligned}$$

The Forward algorithm



General forward algorithm for DAGs

input: Topologically-ordered DAG
$G = (V, E, w)$, $V = \{1, \ldots, n\}$
output: $q_n := \log \sum_{y \in \mathcal{Y}_n} \exp w(y)$.

initialize $q_1 \leftarrow 0$
for $i = 2, \ldots, n$ **do**
$q_i \leftarrow \log \sum_{j \in P_i} \exp\left(q_j + w(ji)\right)$

Complexity: $\Theta(|V| + |E|)$.

Lets us calculate the log-probability of any given sequence $\log \Pr(y)$.

Can use autodiff to get $\nabla_w \log \Pr(y)$.

Spot a pattern?

(Mohri, 2002)

Why are these two algorithms so similar?

Deriving the DP recurrences was almost identical.

Spot a pattern?

(Mohri, 2002)

Why are these two algorithms so similar?

Deriving the DP recurrences was almost identical.

The pattern:

- $x \oplus y = \max(x, y); \quad x \otimes y = x + y$ form a semiring over $\mathbb{R} \cup \{-\infty\}$.
- $x \oplus y = \log(e^x + e^y); x \otimes y = x + y$ form a semiring over $\mathbb{R} \cup \{-\infty\}$.

Spot a pattern?

(Mohri, 2002)

Why are these two algorithms so similar?

Deriving the DP recurrences was almost identical.

The pattern:

- $x \oplus y = \max(x, y)$; $\quad x \otimes y = x + y$ form a semiring over $\mathbb{R} \cup \{-\infty\}$.
- $x \oplus y = \log(e^x + e^y)$; $x \otimes y = x + y$ form a semiring over $\mathbb{R} \cup \{-\infty\}$.

This is a very productive generalization that leads to other algorithms too:

- the boolean semiring $x \oplus y = x \vee y$, $x \otimes y = x \wedge y$ over $\{0, 1\}$ yields an algorithm for path existence;
- there is a semiring that leads to top-k paths.

Sampling paths

Goal: draw samples from the distribution over paths: $y_1, \ldots, y_k \sim \Pr(Y = y)$.

Motivation:

- analyze not just the most likely path, but a set of "typical" paths
- perform inferences

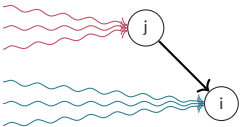
  $$\mathbb{E}_{\Pr(Y)}[F(Y)]$$

  for arbitrary functions F,
- train structured latent variable models

Sampling: One arc at a time

Probability that the last arc of a path ending in i is ji:

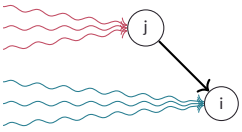
$\Pr(ji|y \text{ ends in } i) =$



Sampling: One arc at a time

Probability that the last arc of a path ending in i is ji:

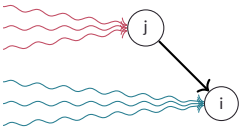
$$\Pr(ji|y \text{ ends in } i) = \frac{\sum_{[y';ji]\in \mathcal{Y}_i} \exp(w(y') + w(ji))}{\sum_{y\in\mathcal{Y}_i} \exp(w(y))}$$



Sampling: One arc at a time

Probability that the last arc of a path ending in i is ji:

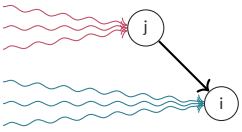
$$\Pr(ji|y \text{ ends in } i) = \frac{\sum_{[y';ji]\in\mathcal{Y}_i} \exp(w(y') + w(ji))}{\sum_{y\in\mathcal{Y}_i} \exp(w(y))}$$
$$= \frac{\exp(w(ji))\sum_{y'\in\mathcal{Y}_j} \exp(w(y'))}{\sum_{y\in\mathcal{Y}_i} \exp(w(y))}$$



Sampling: One arc at a time

Probability that the last arc of a path ending in i is ji:

$$\begin{aligned}\Pr(ji|y \text{ ends in } i) &= \frac{\sum_{[y';ji]\in\mathcal{Y}_i} \exp(w(y') + w(ji))}{\sum_{y\in\mathcal{Y}_i} \exp(w(y))} \\ &= \frac{\exp(w(ji))\sum_{y'\in\mathcal{Y}_j} \exp(w(y'))}{\sum_{y\in\mathcal{Y}_i} \exp(w(y))} \\ &= \exp(w(ji) + q_j - q_i)\end{aligned}$$

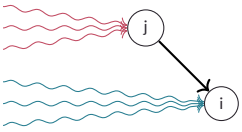


Sampling: One arc at a time

Probability that the last arc of a path ending in i is ji:

$$\begin{aligned}\Pr(ji|y \text{ ends in } i) &= \frac{\sum_{[y';ji]\in\mathcal{Y}_i} \exp(w(y') + w(ji))}{\sum_{y\in\mathcal{Y}_i} \exp(w(y))} \\ &= \frac{\exp(w(ji))\sum_{y'\in\mathcal{Y}_j} \exp(w(y'))}{\sum_{y\in\mathcal{Y}_i} \exp(w(y))} \\ &= \exp(w(ji) + q_j - q_i)\end{aligned}$$

All paths end in n, so draw the final arc jn first.



Sampling: One arc at a time

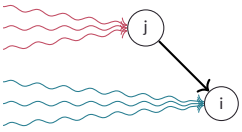
Probability that the last arc of a path ending in i is ji:

$$\Pr(ji|y \text{ ends in } i) = \frac{\sum_{[y';ji]\in\mathcal{Y}_i} \exp(w(y') + w(ji))}{\sum_{y\in\mathcal{Y}_i} \exp(w(y))}$$
$$= \frac{\exp(w(ji))\sum_{y'\in\mathcal{Y}_j} \exp(w(y'))}{\sum_{y\in\mathcal{Y}_i} \exp(w(y))}$$
$$= \exp(w(ji) + q_j - q_i)$$

All paths end in n, so draw the final arc jn first.

Repeat same reasoning on the subgraph with nodes $1, \ldots, j$, i.e., replace n with j and repeat until we hit 1.

Resembles the backpointers from Viterbi: think "stochastic backpointers".



Sampling: One arc at a time

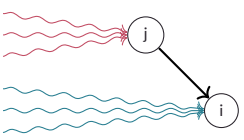
Probability that the last arc of a path ending in i is ji:

$$\Pr(ji|y \text{ ends in } i) = \frac{\sum_{[y';ji]\in\mathcal{Y}_i} \exp(w(y') + w(ji))}{\sum_{y\in\mathcal{Y}_i} \exp(w(y))}$$
$$= \frac{\exp(w(ji))\sum_{y'\in\mathcal{Y}_j} \exp(w(y'))}{\sum_{y\in\mathcal{Y}_i} \exp(w(y))}$$
$$= \exp(w(ji) + q_j - q_i)$$

All paths end in n, so draw the final arc jn first.

Repeat same reasoning on the subgraph with nodes $1, \ldots, j$, i.e., replace n with j and repeat until we hit 1.

Resembles the backpointers from Viterbi: think "stochastic backpointers".



Forward filtering, backward sampling for DAGs

input: Topologically-ordered DAG;
output: y: a sample from $\Pr(y)$.

initialize $q_1 \leftarrow 0$
for $i = 2, \ldots, n$ **do**
  $q_i \leftarrow \log \sum_{j\in P_i} \exp\left(q_j + w(ji)\right)$

$y = [\,]$; $i \leftarrow n$
while $i > 1$ **do**
  sample $j \in P_i$ w.p. $p_j = \exp(w(ji) + q_j - q_i)$
  $y \leftarrow ji^\frown y$
  $i \leftarrow j$

Dynamic programming in DAG conclusion

If we can cast our problem as finding paths in a DAG, then dynamic programming (DP) lets us calculate:

- $\operatorname{argmax}_{y \in \mathcal{Y}} \operatorname{score}(y)$
- $\log \sum_{y \in \mathcal{Y}} \exp \operatorname{score}(y)$ and therefore probabilities
- samples from the distribution over structures

in linear time $\Theta(|V| + |E|)$.

Next we see a bunch of structures that fit this pattern, and some that do not.

Some structures solvable by DP cannot be represented via DAGs.

Dynamic programming in DAG: references and historical notes

The best modern reference for DP as taught in this course is Huang (2008).

Historically, DP is credited to Bellman (1954) in optimal policies and control.

Popularity of DP in NLP came via hidden markov models (HMM) in the 70s and 80s in speech, especially at IBM Research and Bell Labs through a limited-circulation text (Ferguson, 1980): Rabiner gives a first-hand history (Rabiner, n.d.).

Viterbi (1967) was working on information theory / codes. Forward comes from Markov process and is due to Baum (1972). FFBS (Frühwirth-Schnatter, 1994) originates from state space models. There is a lot of reinvention and misattribution around DP, and confusing naming. I tried to name things simply and logically but it can be ambiguous.

Structure Prediction

1. **Overview**
2. **Structured inputs**
   - Recap: Encoding sequences. RNN, CNN, transformer
   - Encoding graphs
3. **Structured outputs**
   - Probabilistic models of structures
   - Directed acyclic graphs
   - Algorithms for paths in DAGs: Maximization, probabilities, sampling
   - Application: Sequence tagging
   - Application: Sequence segmentation
   - Evaluating structured outputs

Sequence tagging

Given a sequence of n items $\boldsymbol{x} = (x_1, \ldots, x_n)$, assign to each of them one of K tags:

$$\boldsymbol{y} = (y_1, \ldots, y_n) \quad \text{where each} \quad y_i \in \{1, \ldots, K\}.$$

Sequence tagging

Given a sequence of n items $\boldsymbol{x} = (x_1, \ldots, x_n)$, assign to each of them one of K tags:

$$\boldsymbol{y} = (y_1, \ldots, y_n) \quad \text{where each} \quad y_i \in \{1, \ldots, K\}.$$

Example 1: Part-of-speech (POS) tagging in NLP

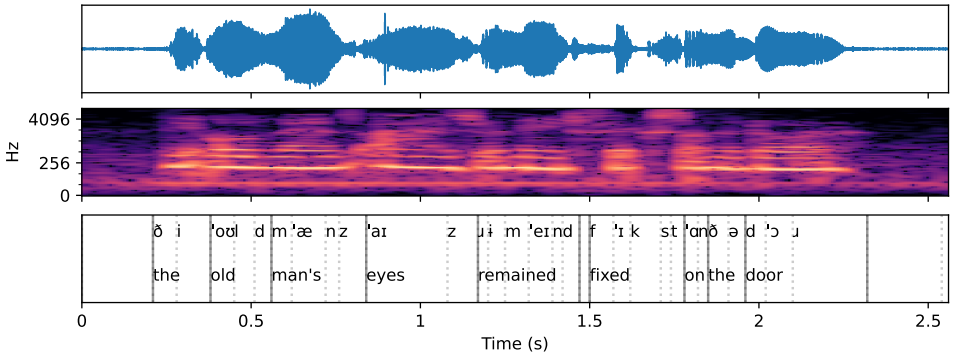
	the	old	man	the	boat
$\boldsymbol{y}_a$	det	adj	noun	det	noun
$\boldsymbol{y}_b$	det	noun	verb	det	noun

Sequence tagging

Given a sequence of n items $\boldsymbol{x} = (x_1, \ldots, x_n)$, assign to each of them one of K tags:

$$\boldsymbol{y} = (y_1, \ldots, y_n) \quad \text{where each} \quad y_i \in \{1, \ldots, K\}.$$

Example 2: Frame-level phoneme classification (may be part of speech recognition)

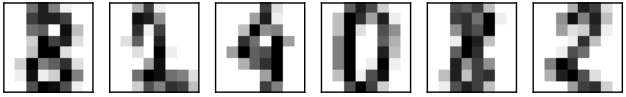


Sequence tagging

Given a sequence of n items $\boldsymbol{x} = (x_1, \ldots, x_n)$, assign to each of them one of K tags:

$$\boldsymbol{y} = (y_1, \ldots, y_n) \quad \text{where each} \quad y_i \in \{1, \ldots, K\}.$$

Example 3: Optical character recognition



Characterizing the output space

Given a sequence of n items $\boldsymbol{x} = (x_1, \ldots, x_n)$, assign to each of them one of K tags:

$$\boldsymbol{y} = (y_1, \ldots, y_n) \quad \text{where each} \quad y_i \in \{1, \ldots, K\}.$$

Input $\boldsymbol{x} = (x_1, \ldots, x_n)$, e.g., a sequence of words.

Output $\boldsymbol{y} = (y_1, \ldots, y_n)$, e.g., a sequence of part-of-speech tags.

For each data point (sentence), $|\boldsymbol{y}| = |\boldsymbol{x}|$; different data points have different lengths.

Characterizing the output space

Given a sequence of n items $\boldsymbol{x} = (x_1, \ldots, x_n)$, assign to each of them one of K tags:

$$\boldsymbol{y} = (y_1, \ldots, y_n) \quad \text{where each} \quad y_i \in \{1, \ldots, K\}.$$

Input $\boldsymbol{x} = (x_1, \ldots, x_n)$, e.g., a sequence of words.

Output $\boldsymbol{y} = (y_1, \ldots, y_n)$, e.g., a sequence of part-of-speech tags.

For each data point (sentence), $|\boldsymbol{y}| = |\boldsymbol{x}|$; different data points have different lengths.

For fixed length n, some possible outputs:

- $(1, 1, \ldots, 1, 1) \in \mathcal{Y}$
- $(1, 1, \ldots, 1, 2) \in \mathcal{Y}$
- $(K, K, \ldots, K, K) \in \mathcal{Y}$

How many in terms of n?

Part-of-speech tags

	Tag	Description	Example
Open Class	**ADJ**	Adjective: noun modifiers describing properties	*red*, *young*, *awesome*
	ADV	Adverb: verb modifiers of time, place, manner	*very*, *slowly*, *home*, *yesterday*
	NOUN	words for persons, places, things, etc.	*algorithm*, *cat*, *mango*, *beauty*
	VERB	words for actions and processes	*draw*, *provide*, *go*
	PROPN	Proper noun: name of a person, organization, place, etc..	*Regina*, *IBM*, *Colorado*
	INTJ	Interjection: exclamation, greeting, yes/no response, etc.	*oh*, *um*, *yes*, *hello*
Closed Class Words	**ADP**	Adposition (Preposition/Postposition): marks a noun's spacial, temporal, or other relation	*in, on, by, under*
	AUX	Auxiliary: helping verb marking tense, aspect, mood, etc.,	*can, may, should, are*
	CCONJ	Coordinating Conjunction: joins two phrases/clauses	*and*, *or*, *but*
	DET	Determiner: marks noun phrase properties	*a, an, the, this*
	NUM	Numeral	*one, two, first, second*
	PART	Particle: a preposition-like form used together with a verb	*up, down, on, off, in, out, at, by*
	PRON	Pronoun: a shorthand for referring to an entity or event	*she, who, I, others*
	SCONJ	Subordinating Conjunction: joins a main clause with a subordinate clause such as a sentential complement	*that*, *which*
Other	**PUNCT**	Punctuation	; , ()
	SYM	Symbols like $ or emoji	$, %
	X	Other	asdf, qwfg

Figure 8.1 The 17 parts of speech in the Universal Dependencies tagset (Nivre et al., 2016a). Features can be added to make finer-grained distinctions (with properties like number, case, definiteness, and so on).

Source: (Jurafsky and Martin, 2024) `https://web.stanford.edu/~jurafsky/slp3/8.pdf` ©Jurafsky and Martin

Designing a simple scorer

Writing $\boldsymbol{y} = (y_1, \ldots, y_n)$, take $\text{score}(\boldsymbol{y}) = \sum_j a_{j,y_j}$.

$\boldsymbol{A}$ is a matrix of scores,
e.g., computed by a NN encoder.

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

Designing a simple scorer

Writing $\boldsymbol{y} = (y_1, \ldots, y_n)$, take
$\text{score}(\boldsymbol{y}) = \sum_j a_{j,y_j}$.

$\boldsymbol{A}$ is a matrix of scores,
e.g., computed by a NN encoder.

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

	the	old	man	the	boat
$\boldsymbol{y}_a$	det	adj	noun	det	noun
$\boldsymbol{y}_b$	det	noun	verb	det	noun

Designing a simple scorer

Writing $\boldsymbol{y} = (y_1, \ldots, y_n)$, take
$\text{score}(\boldsymbol{y}) = \sum_j a_{j,y_j}$.

$\boldsymbol{A}$ is a matrix of scores,
e.g., computed by a NN encoder.

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

	the	old	man	the	boat
$\boldsymbol{y}_a$	det	adj	noun	det	noun
$\boldsymbol{y}_b$	det	noun	verb	det	noun

$\text{score}(\boldsymbol{y}_a) =$

Designing a simple scorer

Writing $\boldsymbol{y} = (y_1, \ldots, y_n)$, take
$\text{score}(\boldsymbol{y}) = \sum_j a_{j,y_j}$.

$\boldsymbol{A}$ is a matrix of scores,
e.g., computed by a NN encoder.

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

	the	old	man	the	boat
$\boldsymbol{y}_a$	det	adj	noun	det	noun
$\boldsymbol{y}_b$	det	noun	verb	det	noun

$\text{score}(\boldsymbol{y}_a) = 21$

Designing a simple scorer

Writing $\boldsymbol{y} = (y_1, \ldots, y_n)$, take
$\text{score}(\boldsymbol{y}) = \sum_j a_{j,y_j}$.

$\boldsymbol{A}$ is a matrix of scores,
e.g., computed by a NN encoder.

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

	the	old	man	the	boat
$\boldsymbol{y}_a$	det	adj	noun	det	noun
$\boldsymbol{y}_b$	det	noun	verb	det	noun

$\text{score}(\boldsymbol{y}_a) = 21$

$\text{score}(\boldsymbol{y}_b) =$

Designing a simple scorer

Writing $\boldsymbol{y} = (y_1, \ldots, y_n)$, take
$\text{score}(\boldsymbol{y}) = \sum_j a_{j,y_j}$.

$\boldsymbol{A}$ is a matrix of scores,
e.g., computed by a NN encoder.

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

	the	old	man	the	boat
$\boldsymbol{y}_a$	det	adj	noun	det	noun
$\boldsymbol{y}_b$	det	noun	verb	det	noun

$\text{score}(\boldsymbol{y}_a) = 21$

$\text{score}(\boldsymbol{y}_b) = 17$

Designing a simple scorer

A first attempt:
separate classifier for each position.

1. embed and encode $\boldsymbol{x}$, eg, with a CNN.

$$(x_1, \ldots, x_n) \to (\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n)$$

2. For each position j, apply a classification head with K outputs. E.g.,

$$\boldsymbol{a}_j = \boldsymbol{W}^\top \boldsymbol{z}_j + \boldsymbol{b}$$

   Think of $\boldsymbol{A}$ as a matrix with n rows and K columns, where $a_{j,c}$ is the score of assigning tag c at position j.

3. Writing $\boldsymbol{y} = (y_1, \ldots, y_n)$, take $\text{score}(\boldsymbol{y}) = \sum_j a_{j,y_j}$.

```
words = [21, 79, 14]  # indices
emb = Embedding(vocab_sz, dim)
clf = Linear(dim, n_tags)

# optionally add RNN, CNN, whatever

Z = emb(words)  # (3 × dim)
A = clf(Z)      # (3 × n_tags)

# computing the score of a given tag sequence:
y = [2, 0, 2]

y_score = sum(A[i, yi]
              for y, yi in enumerate(y))

# or, if you want to be fancy/fast:
y_score = A[torch.arange(len(y)), y].sum()
```

Finding the best sequence

With our $\text{score}(\boldsymbol{y}) = \sum_j a_{j,y_j}$, can we compute:

$$\max_{\boldsymbol{y} \in \mathcal{Y}} \text{score}(\boldsymbol{y})$$

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

Finding the best sequence

With our $\text{score}(\boldsymbol{y}) = \sum_j a_{j,y_j}$, can we compute:

$$\max_{\boldsymbol{y} \in \mathcal{Y}} \text{score}(\boldsymbol{y})$$

$$= \max_{y_1 \in [K], \ldots, y_n \in [K]} \text{score}([y_1, \ldots, y_n])$$

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

Finding the best sequence

With our $\text{score}(\boldsymbol{y}) = \sum_j a_{j,y_j}$, can we compute:

$$\begin{aligned}
&\max_{\boldsymbol{y} \in \mathcal{Y}} \text{score}(\boldsymbol{y}) \\
&= \max_{y_1 \in [K], \ldots, y_n \in [K]} \text{score}([y_1, \ldots, y_n]) \\
&= \max_{y_1 \in [K], \ldots, y_n \in [K]} \sum_j a_{j,y_j}
\end{aligned}$$

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

Finding the best sequence

With our $\text{score}(\boldsymbol{y}) = \sum_j a_{j,y_j}$, can we compute:

$$
\begin{aligned}
&\max_{\boldsymbol{y} \in \mathcal{Y}} \text{score}(\boldsymbol{y}) \\
&= \max_{y_1 \in [K], \ldots, y_n \in [K]} \text{score}([y_1, \ldots, y_n]) \\
&= \max_{y_1 \in [K], \ldots, y_n \in [K]} \sum_j a_{j,y_j} \\
&= \sum_j \max_{y_j \in [K]} a_{j,y_j}
\end{aligned}
$$

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

Finding the best sequence

With our $\text{score}(\boldsymbol{y}) = \sum_j a_{j,y_j}$, can we compute:

$$
\begin{aligned}
&\max_{\boldsymbol{y} \in \mathcal{Y}} \text{score}(\boldsymbol{y}) \\
&= \max_{y_1 \in [K], \ldots, y_n \in [K]} \text{score}([y_1, \ldots, y_n]) \\
&= \max_{y_1 \in [K], \ldots, y_n \in [K]} \sum_j a_{j,y_j} \\
&= \sum_j \max_{y_j \in [K]} a_{j,y_j}
\end{aligned}
$$

So, $\arg\max_{\boldsymbol{y}} \text{score}(\boldsymbol{y})$ is made up of the tags selected independently at each position.

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

Normalizing constant (log-sum-exp)

With our $\text{score}(\boldsymbol{y}) = \sum_j a_{j,y_j}$, can we compute:

$$\log \sum_{\boldsymbol{y} \in \mathcal{Y}} \exp\left(\text{score}(\boldsymbol{y})\right)$$

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

Normalizing constant (log-sum-exp)

With our $\text{score}(\boldsymbol{y}) = \sum_j a_{j,y_j}$, can we compute:

$$\log \sum_{\boldsymbol{y} \in \mathcal{Y}} \exp\left(\text{score}(\boldsymbol{y})\right)$$

$$= \log \sum_{y_1=1}^{K} \cdots \sum_{y_n=1}^{K} \exp \sum_{j=1}^{n} a_{j,y_j}$$

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

Normalizing constant (log-sum-exp)

With our $\text{score}(\boldsymbol{y}) = \sum_j a_{j,y_j}$, can we compute:

$$\log \sum_{\boldsymbol{y} \in \mathcal{Y}} \exp\left(\text{score}(\boldsymbol{y})\right)$$

$$= \log \sum_{y_1=1}^{K} \cdots \sum_{y_n=1}^{K} \exp \sum_{j=1}^{n} a_{j,y_j}$$

$$= \log \sum_{y_1=1}^{K} \cdots \sum_{y_n=1}^{K} \prod_{j=1}^{n} \exp a_{j,y_j}$$

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

Normalizing constant (log-sum-exp)

With our $\text{score}(\boldsymbol{y}) = \sum_j a_{j,y_j}$, can we compute:

$$\log \sum_{\boldsymbol{y} \in \mathcal{Y}} \exp\left(\text{score}(\boldsymbol{y})\right)$$

$$= \log \sum_{y_1=1}^{K} \cdots \sum_{y_n=1}^{K} \exp \sum_{j=1}^{n} a_{j,y_j}$$

$$= \log \sum_{y_1=1}^{K} \cdots \sum_{y_n=1}^{K} \prod_{j=1}^{n} \exp a_{j,y_j}$$

$$= \log \prod_{j=1}^{n} \sum_{y_j=1}^{K} \exp a_{j,y_j}$$

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

Normalizing constant (log-sum-exp)

With our $\text{score}(\boldsymbol{y}) = \sum_j a_{j,y_j}$, can we compute:

$$
\begin{aligned}
&\log \sum_{\boldsymbol{y} \in \mathcal{Y}} \exp\left(\text{score}(\boldsymbol{y})\right) \\
&= \log \sum_{y_1=1}^{K} \cdots \sum_{y_n=1}^{K} \exp \sum_{j=1}^{n} a_{j,y_j} \\
&= \log \sum_{y_1=1}^{K} \cdots \sum_{y_n=1}^{K} \prod_{j=1}^{n} \exp a_{j,y_j} \\
&= \log \prod_{j=1}^{n} \sum_{y_j=1}^{K} \exp a_{j,y_j} \\
&= \sum_{j=1}^{n} \log \sum_{y_j=1}^{K} \exp a_{j,y_j}
\end{aligned}
$$

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

Normalizing constant (log-sum-exp)

With our $\text{score}(\boldsymbol{y}) = \sum_j a_{j,y_j}$, can we compute:

$$
\begin{aligned}
&\log \sum_{\boldsymbol{y} \in \mathcal{Y}} \exp\left(\text{score}(\boldsymbol{y})\right) \\
&= \log \sum_{y_1=1}^{K} \cdots \sum_{y_n=1}^{K} \exp \sum_{j=1}^{n} a_{j,y_j} \\
&= \log \sum_{y_1=1}^{K} \cdots \sum_{y_n=1}^{K} \prod_{j=1}^{n} \exp a_{j,y_j} \\
&= \log \prod_{j=1}^{n} \sum_{y_j=1}^{K} \exp a_{j,y_j} \\
&= \sum_{j=1}^{n} \log \sum_{y_j=1}^{K} \exp a_{j,y_j}
\end{aligned}
$$

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

Probabilistic interpretation: independence

$$
\begin{aligned}
\log \Pr(\boldsymbol{y}) &= \text{score}(\boldsymbol{y}) - \log \sum_{\boldsymbol{y}' \in \mathcal{Y}} \exp \text{score}(\boldsymbol{y}') \\
&= \sum_j \underbrace{\left(a_{j,y_j} - \log \sum_{k \in [K]} \exp a_{j,k} \right)}_{\log \Pr(y_j)}
\end{aligned}
$$

Fully-local vs. fully-global

For sequence tagging, the separable (fully-local) score

$$\text{score}(\boldsymbol{y}) = \sum_j a_{j,y_j}$$

amounts to applying a probabilistic classifier to each of the n positions separately! (any "magic" comes from the feature represtntation / neural net encoder.)

Can we design a richer $\text{score}(\boldsymbol{y})$ taking into account the sequential structure of $\boldsymbol{y}$?

Fully-local vs. fully-global

Entirely global model: like classification, where *each possible sequence* is a class.

$\boldsymbol{y}$	score($\boldsymbol{y}$)
det det det det det	−1000
det det det det noun	−940
det det det det verb	−800
...	
det noun verb det noun	400
...	
verb verb verb verb verb	−1100

As expressive as possible: score is any function of the sequence.

Fully-local vs. fully-global

Entirely global model: like classification, where *each possible sequence* is a class.

$\boldsymbol{y}$	score($\boldsymbol{y}$)
det det det det det	−1000
det det det det noun	−940
det det det det verb	−800
...	
det noun verb det noun	400
...	
verb verb verb verb verb	−1100

As expressive as possible: score is any function of the sequence.

But completely intractable: $O(K^n)$ time and space.

Fully-local vs. fully-global

Entirely global model: like classification, where *each possible sequence* is a class.

$\boldsymbol{y}$	score($\boldsymbol{y}$)
det det det det det	−1000
det det det det noun	−940
det det det det verb	−800
...	
det noun verb det noun	400
...	
verb verb verb verb verb	−1100

As expressive as possible: score is any function of the sequence.

But completely intractable: $O(K^n)$ time and space.

Structure output prediction is about the space in between these two extremes.

Scoring with transitions

Idea: scoring transitions between adjacent tags

$$\text{score}(\boldsymbol{y}) = \sum_{j=1}^{n} a_{j,y_j} + \sum_{j=2}^{n} t_{y_{j-1},y_j}$$

For example, $\text{score}(\texttt{[NOUN, DET, VERB]}) = +a_{2,\text{DET}}\, a_{1,\text{NOUN}} + a_{3,\text{VERB}} + t_{\text{NOUN,DET}} + t_{\text{DET,VERB}}$

Scoring with transitions

A rich scorer that takes into account the sequential nature of $\boldsymbol{y}$ while still allowing efficient computation:

scoring transitions between adjacent tags

$$\text{score}(\boldsymbol{y}) = \sum_{j=1}^{n} a_{j,y_j} + \sum_{j=2}^{n} t_{y_{j-1},y_j}$$

For example, $\text{score}(\texttt{[NOUN, DET, VERB]}) = a_{1,\text{NOUN}} + a_{2,\text{DET}} + a_{3,\text{VERB}} + t_{\text{NOUN,DET}} + t_{\text{DET,VERB}}$

Sequence modeling with transition scores

$$\text{score}(\boldsymbol{y}) = \sum_{j=1}^{n} a_{j,y_j} + \sum_{j=2}^{n} t_{y_{j-1},y_j}$$

The tag scores $\boldsymbol{A} \in \mathbb{R}^{n \times K}$ can be computed as before (e.g., with a convnet.)

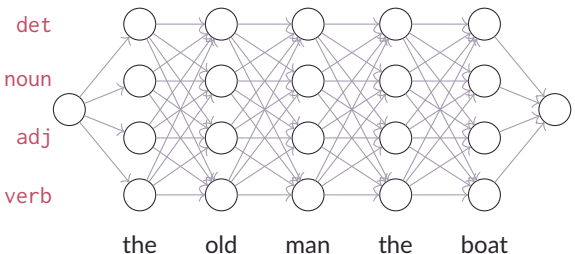
The transition scores $\boldsymbol{T} \in \mathbb{R}^{K \times K}$:

- could be a learned parameter. (size does not depend on n)
- could be predicted by the neural net as a function of $\boldsymbol{x}$.

Unlike in the separable case, with transition scores, we no longer get n parallel classifiers: the different tags impact one another. (This makes the model more expressive and more interesting.)

Sequence tagging as a DAG

$$\text{score}(\boldsymbol{y}) = \sum_{j=1}^{n} a_{j,y_j} + \sum_{j=2}^{n} t_{y_{j-1},y_j}$$



$G = (V, E, w)$ where:

$$V = \{(j, c) : j \in [n], c \in [K]\} \cup \{s, t\}$$

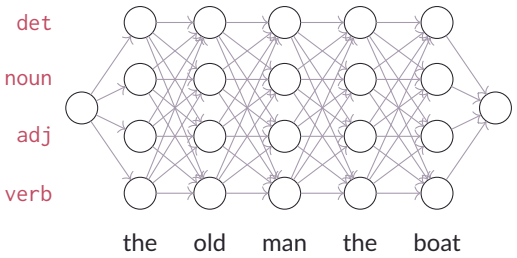
$$E = \{(j-1, c') \to (j, c) : j \in [2, n], c, c' \in [K]\} \cup \{s \to (1, c) : c \in [K]\} \cup \{(n, c) \to t : c \in [K]\}$$

$$w\big((j-1, c') \to (j, c)\big) = a_{j,c} + t_{c',c}$$
$$w(s \to (1, c)) = a_{1,c}$$
$$w((n, c) \to t) = 0$$

$|V| \in \Theta(nK); \quad |E| \in \Theta(nK^2)$

Topological ordering?

Viterbi for sequence tagging



General Viterbi (reminder sketch)

initialize $m_1 \leftarrow 0$
for $i = 2, \ldots, n$ **do**
$m_i \leftarrow \max_{j \in P_i} \left(m_j + w(ji) \right)$
$\pi_i \leftarrow \arg\max_{j \in P_i} \left(m_j + w(ji) \right)$
follow backpointers to get best path

Viterbi for sequence tagging

input: Unary scores $\boldsymbol{A}$ ($n \times K$ array)
Transition scores $\boldsymbol{T}$ ($K \times K$ array)

Forward: compute scores recursively
$m_{1c} = a_{1c}$ for all $c \in [K]$
for $j = 2$ to n **do**
for $c = 1$ to K **do**
$m_{j,c} \leftarrow \max_{c' \in [K]} \left(m_{j-1,c'} + a_{j,c} + t_{c',c} \right)$
$\pi_{j,c} \leftarrow \arg\max_{c' \in [K]} \left(m_{j-1,c'} + a_{j,c} + t_{c',c} \right)$
$f^\star = \max_{c' \in [K]} m_{n,c'}$

Backward: follow backpointers
$y_n = \arg\max_{c'} m_n(c')$
for $j = n - 1$ **down to** 1 **do**
$y_j = \pi_{j+1, y_{j+1}}$

output: $f^\star$ and $\boldsymbol{y}^\star = [y_1, \ldots, y_n]$

Viterbi for sequence tagging: Example

$m_{j,c}$ is stored as a matrix $\boldsymbol{M}$, same shape as $\boldsymbol{A}$.

Apply $m_{1,c} = a_{1,c}$ to get the first row: (copied from $\boldsymbol{A}$)

Then iteratively: $m_{j,c} = \max_{c' \in [K]} m_{j-1,c'} + a_{j,c} + t_{c',c}$

At the end, take the maximum over the last row.

$\boldsymbol{M} =$

	det	noun	adj	verb
the				
old				
man				
the				
boat				

unary and transition scores:

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

$\boldsymbol{T} =$

	det	noun	adj	verb
det	−4	3	2	−1
noun	−3	−2	−1	2
adj	−2	2	1	1
verb	1	−1	0	0

Viterbi for sequence tagging: Example

$m_{j,c}$ is stored as a matrix $\boldsymbol{M}$, same shape as $\boldsymbol{A}$.

Apply $m_{1,c} = a_{1,c}$ to get the first row: (copied from $\boldsymbol{A}$)

Then iteratively: $m_{j,c} = \max_{c' \in [K]} m_{j-1,c'} + a_{j,c} + t_{c',c}$

At the end, take the maximum over the last row.

$\boldsymbol{M} =$

	det	noun	adj	verb
the	5	0	0	0
old				
man				
the				
boat				

unary and transition scores:

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

$\boldsymbol{T} =$

	det	noun	adj	verb
det	−4	3	2	−1
noun	−3	−2	−1	2
adj	−2	2	1	1
verb	1	−1	0	0

Viterbi for sequence tagging: Example

$m_{j,c}$ is stored as a matrix $\boldsymbol{M}$, same shape as $\boldsymbol{A}$.

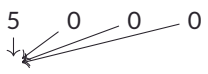
Apply $m_{1,c} = a_{1,c}$ to get the first row: (copied from $\boldsymbol{A}$)

Then iteratively: $m_{j,c} = \max_{c' \in [K]} m_{j-1,c'} + a_{j,c} + t_{c',c}$

At the end, take the maximum over the last row.

$\boldsymbol{M} =$

	det	noun	adj	verb
the	5	0	0	0
old				
man				
the				
boat				



unary and transition scores:

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

$\boldsymbol{T} =$

	det	noun	adj	verb
det	−4	3	2	−1
noun	−3	−2	−1	2
adj	−2	2	1	1
verb	1	−1	0	0

Viterbi for sequence tagging: Example

$m_{j,c}$ is stored as a matrix $\boldsymbol{M}$, same shape as $\boldsymbol{A}$.

Apply $m_{1,c} = a_{1,c}$ to get the first row: (copied from $\boldsymbol{A}$)

Then iteratively: $m_{j,c} = \max_{c' \in [K]} m_{j-1,c'} + a_{j,c} + t_{c',c}$

At the end, take the maximum over the last row.

$\boldsymbol{M} =$

	det	noun	adj	verb
the	5	0	0	0
old	1			
man				
the				
boat				

unary and transition scores:

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

$\boldsymbol{T} =$

	det	noun	adj	verb
det	−4	3	2	−1
noun	−3	−2	−1	2
adj	−2	2	1	1
verb	1	−1	0	0

Viterbi for sequence tagging: Example

$m_{j,c}$ is stored as a matrix $\boldsymbol{M}$, same shape as $\boldsymbol{A}$.

Apply $m_{1,c} = a_{1,c}$ to get the first row: (copied from $\boldsymbol{A}$)

Then iteratively: $m_{j,c} = \max_{c' \in [K]} m_{j-1,c'} + a_{j,c} + t_{c',c}$

At the end, take the maximum over the last row.

$\boldsymbol{M} =$

	det	noun	adj	verb
the	5	0	0	0
old	1			
man				
the				
boat				

unary and transition scores:

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

$\boldsymbol{T} =$

	det	noun	adj	verb
det	−4	3	2	−1
noun	−3	−2	−1	2
adj	−2	2	1	1
verb	1	−1	0	0

Viterbi for sequence tagging: Example

$m_{j,c}$ is stored as a matrix $\boldsymbol{M}$, same shape as $\boldsymbol{A}$.

Apply $m_{1,c} = a_{1,c}$ to get the first row: (copied from $\boldsymbol{A}$)

Then iteratively: $m_{j,c} = \max_{c' \in [K]} m_{j-1,c'} + a_{j,c} + t_{c',c}$

At the end, take the maximum over the last row.

$\boldsymbol{M} =$

	det	noun	adj	verb
the	5	0	0	0
old	1	9		
man				
the				
boat				

unary and transition scores:

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

$\boldsymbol{T} =$

	det	noun	adj	verb
det	−4	3	2	−1
noun	−3	−2	−1	2
adj	−2	2	1	1
verb	1	−1	0	0

Viterbi for sequence tagging: Example

$m_{j,c}$ is stored as a matrix $\boldsymbol{M}$, same shape as $\boldsymbol{A}$.

Apply $m_{1,c} = a_{1,c}$ to get the first row: (copied from $\boldsymbol{A}$)

Then iteratively: $m_{j,c} = \max_{c' \in [K]} m_{j-1,c'} + a_{j,c} + t_{c',c}$

At the end, take the maximum over the last row.

$\boldsymbol{M} =$

	det	noun	adj	verb
the	5	0	0	0
old	1	9	10	4
man				
the				
boat				

unary and transition scores:

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

$\boldsymbol{T} =$

	det	noun	adj	verb
det	−4	3	2	−1
noun	−3	−2	−1	2
adj	−2	2	1	1
verb	1	−1	0	0

Viterbi for sequence tagging: Example

$m_{j,c}$ is stored as a matrix $\boldsymbol{M}$, same shape as $\boldsymbol{A}$.

Apply $m_{1,c} = a_{1,c}$ to get the first row: (copied from $\boldsymbol{A}$)

Then iteratively: $m_{j,c} = \max_{c' \in [K]} m_{j-1,c'} + a_{j,c} + t_{c',c}$

At the end, take the maximum over the last row.

$\boldsymbol{M} =$

	det	noun	adj	verb
the	5	0	0	0
old	1	9	10	4
man	8	15	11	12
the	18	13	14	17
boat	18	26	20	17

unary and transition scores:

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

$\boldsymbol{T} =$

	det	noun	adj	verb
det	−4	3	2	−1
noun	−3	−2	−1	2
adj	−2	2	1	1
verb	1	−1	0	0

Viterbi for sequence tagging: Example

$m_{j,c}$ is stored as a matrix $\boldsymbol{M}$, same shape as $\boldsymbol{A}$.

Apply $m_{1,c} = a_{1,c}$ to get the first row: (copied from $\boldsymbol{A}$)

Then iteratively: $m_{j,c} = \max_{c' \in [K]} m_{j-1,c'} + a_{j,c} + t_{c',c}$

At the end, take the maximum over the last row.

$\boldsymbol{M} =$

	det	noun	adj	verb
the	5	0	0	0
old	1	9	10	4
man	8	15	11	12
the	18	13	14	17
boat	18	26	20	17

To find the best tag sequence $\boldsymbol{y}^\star$, keep track of the path.

unary and transition scores:

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

$\boldsymbol{T} =$

	det	noun	adj	verb
det	−4	3	2	−1
noun	−3	−2	−1	2
adj	−2	2	1	1
verb	1	−1	0	0

Viterbi for sequence tagging: Example

$m_{j,c}$ is stored as a matrix $\boldsymbol{M}$, same shape as $\boldsymbol{A}$.

Apply $m_{1,c} = a_{1,c}$ to get the first row: (copied from $\boldsymbol{A}$)

Then iteratively: $m_{j,c} = \max_{c' \in [K]} m_{j-1,c'} + a_{j,c} + t_{c',c}$

At the end, take the maximum over the last row.

$\boldsymbol{M} =$

	det	noun	adj	verb
the	**5**	0	0	0
old	1	**9**	10	4
man	8	15	11	**12**
the	**18**	13	14	17
boat	18	**26**	20	17

To find the best tag sequence $\boldsymbol{y}^\star$, keep track of the path.

unary and transition scores:

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

$\boldsymbol{T} =$

	det	noun	adj	verb
det	−4	3	2	−1
noun	−3	−2	−1	2
adj	−2	2	1	1
verb	1	−1	0	0

The two main recurrences of sequence tagging:

(Dynamic programming applied to the sequence tagging DAG)

$$m_{j,c} = \max_{c' \in [K]} \left(m_{j-1,c'} + a_{jc} + t_{c'c} \right),$$
$$q_{j,c} = \log \sum_{c' \in [K]} \exp \left(q_{j-1,c'} + a_{jc} + t_{c'c} \right).$$

The Forward algorithm

Forward algorithm for sequence tagging

input: Unary scores $\boldsymbol{A}$ ($n \times K$ array)
Transition scores $\boldsymbol{T}$ ($K \times K$ array)

Forward: compute scores recursively

$q_{1,c} = a_{1,c}$ for all $c \in [K]$
for $j = 2$ **to** n **do**
  for $c = 1$ **to** K **do**
    $q_{j,c} = \log \sum_{c' \in [K]} \exp\left(q_{j-1,c'} + a_{j,c} + t_{c',c}\right)$
return $\log Z = \log \sum_{c' \in [K]} \exp\left(q_{n,c'}\right)$

	the	old	man	the	boat	
y_a	det	adj	noun	det	noun	score$(y_a) = 25$
y_b	det	noun	verb	det	noun	score$(y_b) = 26$
y_c	noun	noun	noun	noun	noun	score$(y_c) = 1$

Applying the Forward algorithm yields

$\boldsymbol{Q} =$

	det	noun	adj	verb
the	5.00	0.00	0.00	0.00
old	1.73	9.00	10.00	4.19
man	8.18	15.01	11.05	12.70
the	18.88	13.92	14.37	17.03
boat	18.08	26.88	20.90	18.38

unary and transition scores:

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

$\boldsymbol{T} =$

	det	noun	adj	verb
det	−4	3	2	−1
noun	−3	−2	−1	2
adj	−2	2	1	1
verb	1	−1	0	0

	the	old	man	the	boat	
y_a	det	adj	noun	det	noun	score$(y_a) = 25$
y_b	det	noun	verb	det	noun	score$(y_b) = 26$
y_c	noun	noun	noun	noun	noun	score$(y_c) = 1$

Applying the Forward algorithm yields

$\boldsymbol{Q} =$

	det	noun	adj	verb
the	5.00	0.00	0.00	0.00
old	1.73	9.00	10.00	4.19
man	8.18	15.01	11.05	12.70
the	18.88	13.92	14.37	17.03
boat	18.08	26.88	20.90	18.38

$\log Z \approx 26.885$

unary and transition scores:

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

$\boldsymbol{T} =$

	det	noun	adj	verb
det	−4	3	2	−1
noun	−3	−2	−1	2
adj	−2	2	1	1
verb	1	−1	0	0

	the	old	man	the	boat	
$\boldsymbol{y}_a$	det	adj	noun	det	noun	$\text{score}(\boldsymbol{y}_a) = 25$
$\boldsymbol{y}_b$	det	noun	verb	det	noun	$\text{score}(\boldsymbol{y}_b) = 26$
$\boldsymbol{y}_c$	noun	noun	noun	noun	noun	$\text{score}(\boldsymbol{y}_c) = 1$

Applying the Forward algorithm yields

$\boldsymbol{Q} =$

	det	noun	adj	verb
the	5.00	0.00	0.00	0.00
old	1.73	9.00	10.00	4.19
man	8.18	15.01	11.05	12.70
the	18.88	13.92	14.37	17.03
boat	18.08	26.88	20.90	18.38

$$\log Z \approx 26.885$$

$$\log P(\boldsymbol{y}_a) = \text{score}(\boldsymbol{y}_a) - \log Z = 25 - 26.885 = -1.885$$

unary and transition scores:

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

$\boldsymbol{T} =$

	det	noun	adj	verb
det	−4	3	2	−1
noun	−3	−2	−1	2
adj	−2	2	1	1
verb	1	−1	0	0

	the	old	man	the	boat	
$\boldsymbol{y}_a$	det	adj	noun	det	noun	$\text{score}(\boldsymbol{y}_a) = 25$
$\boldsymbol{y}_b$	det	noun	verb	det	noun	$\text{score}(\boldsymbol{y}_b) = 26$
$\boldsymbol{y}_c$	noun	noun	noun	noun	noun	$\text{score}(\boldsymbol{y}_c) = 1$

Applying the Forward algorithm yields

$\boldsymbol{Q} =$

	det	noun	adj	verb
the	5.00	0.00	0.00	0.00
old	1.73	9.00	10.00	4.19
man	8.18	15.01	11.05	12.70
the	18.88	13.92	14.37	17.03
boat	18.08	26.88	20.90	18.38

$$\log Z \approx 26.885$$

$$\log P(\boldsymbol{y}_a) = \text{score}(\boldsymbol{y}_a) - \log Z = 25 - 26.885 = -1.885$$

$$\log P(\boldsymbol{y}_b) = \text{score}(\boldsymbol{y}_b) - \log Z = 26 - 26.885 = -0.885$$

unary and transition scores:

$\boldsymbol{A} =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

$\boldsymbol{T} =$

	det	noun	adj	verb
det	−4	3	2	−1
noun	−3	−2	−1	2
adj	−2	2	1	1
verb	1	−1	0	0

	the	old	man	the	boat	
y_a	det	adj	noun	det	noun	$\text{score}(y_a) = 25$
y_b	det	noun	verb	det	noun	$\text{score}(y_b) = 26$
y_c	noun	noun	noun	noun	noun	$\text{score}(y_c) = 1$

Applying the Forward algorithm yields

$Q =$

	det	noun	adj	verb
the	5.00	0.00	0.00	0.00
old	1.73	9.00	10.00	4.19
man	8.18	15.01	11.05	12.70
the	18.88	13.92	14.37	17.03
boat	18.08	26.88	20.90	18.38

$$\log Z \approx 26.885$$

$$\log P(y_a) = \text{score}(y_a) - \log Z = 25 - 26.885 = -1.885$$

$$\log P(y_b) = \text{score}(y_b) - \log Z = 26 - 26.885 = -0.885$$

$$\log P(y_c) = \text{score}(y_c) - \log Z = 1 - 26.885 = -25.885$$

unary and transition scores:

$A =$

	det	noun	adj	verb
the	5	0	0	0
old	0	1	3	0
man	0	3	0	1
the	5	0	0	0
boat	0	5	0	0

$T =$

	det	noun	adj	verb
det	−4	3	2	−1
noun	−3	−2	−1	2
adj	−2	2	1	1
verb	1	−1	0	0

Putting it all together

At this point, we have all the ingredients needed to train a probabilistic sequence tagger with transition scores!

1. Receiving an input sequence $\boldsymbol{x}$, the model returns unary and transition scores $\boldsymbol{A}$ and $\boldsymbol{T}$.
2. If we're at test time:
   run Viterbi to get predicted sequence; compute accuracies etc.
3. If training time:
   run Forward algorithm to compute the training objective
   $-\log P(\boldsymbol{y} \mid \boldsymbol{x}) = -\operatorname{score}(\boldsymbol{y}) + \log \sum_{\boldsymbol{y}' \in \mathcal{Y}} \exp \operatorname{score}(\boldsymbol{y}')$.

Structure Prediction

Sequence segmentation

The rod cutting problem: We have a rod of length n units, and we can cut it at every marker. What cuts to make to maximize the total value of the resulting pieces?

Sequence segmentation

The rod cutting problem: We have a rod of length n units, and we can cut it at every marker. What cuts to make to maximize the total value of the resulting pieces?

Sequence segmentation

The rod cutting problem: We have a rod of length n units, and we can cut it at every marker. What cuts to make to maximize the total value of the resulting pieces?

DNA/RNA: A C A G A T T A C C

Word segmentation: 私 は 日 本 語 を 学 習

Sequence segmentation

The rod cutting problem: We have a rod of length n units, and we can cut it at every marker. What cuts to make to maximize the total value of the resulting pieces?

DNA/RNA: A C A G A T T A C C

Word segmentation: 私 は 日 本 語 を 学 習

Entity Extraction: Mayor Halsema to visit the University of Amsterdam next Friday

Sequence segmentation

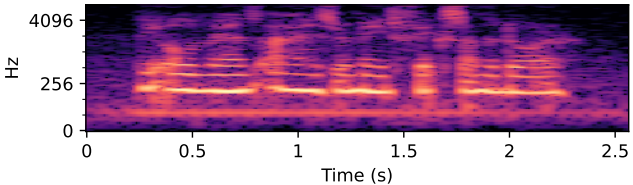
The rod cutting problem: We have a rod of length n units, and we can cut it at every marker. What cuts to make to maximize the total value of the resulting pieces?

DNA/RNA: A C A G A T T A C C

Word segmentation: 私 は 日 本 語 を 学 習

Entity Extraction: Mayor Halsema to visit the University of Amsterdam next Friday

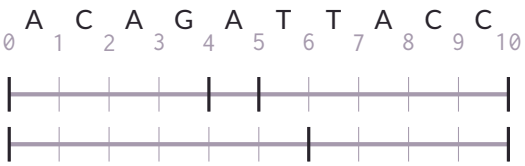
Speech:



Representing and scoring segmentations

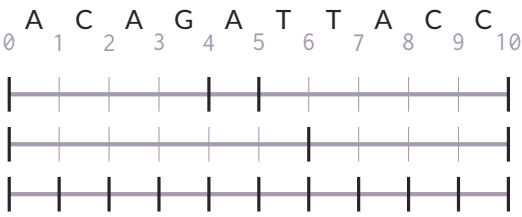
cuts	segments
[4,5]	[(0,4), (4,5), (5,10)]

Representing and scoring segmentations



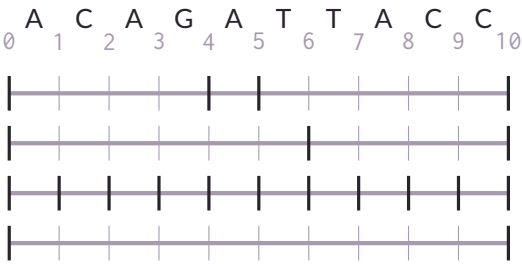
cuts	segments
[4,5]	[(0,4), (4,5), (5,10)]
[6]	[(0,6), (6,10)]

Representing and scoring segmentations



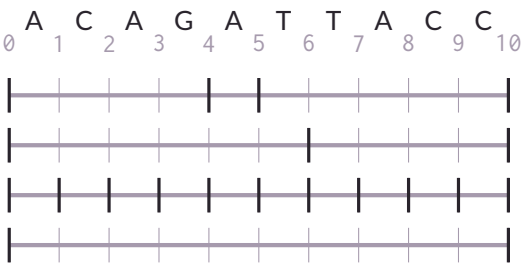
cuts	segments
[4,5]	[(0,4), (4,5), (5,10)]
[6]	[(0,6), (6,10)]
[1,2,...,9]	[(0,1), …, (9,10)]

Representing and scoring segmentations



cuts	segments
[4,5]	[(0,4), (4,5), (5,10)]
[6]	[(0,6), (6,10)]
[1,2,...,9]	[(0,1), … , (9,10)]
[]	[(0, 10)]

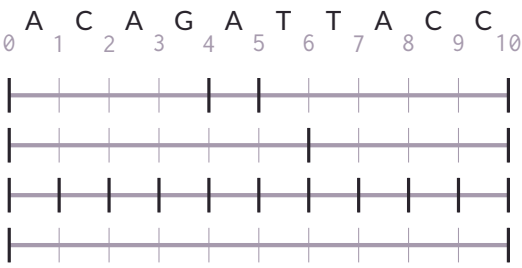
Representing and scoring segmentations



cuts	segments
[4,5]	[(0,4), (4,5), (5,10)]
[6]	[(0,6), (6,10)]
[1,2,...,9]	[(0,1), …, (9,10)]
[]	[(0, 10)]

- How many possible segments?

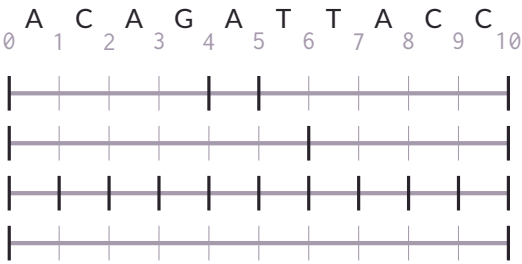
Representing and scoring segmentations



cuts	segments
[4,5]	[(0,4), (4,5), (5,10)]
[6]	[(0,6), (6,10)]
[1,2,...,9]	[(0,1), …, (9,10)]
[]	[(0, 10)]

- How many possible segments?
- How many possible *segmentations*?

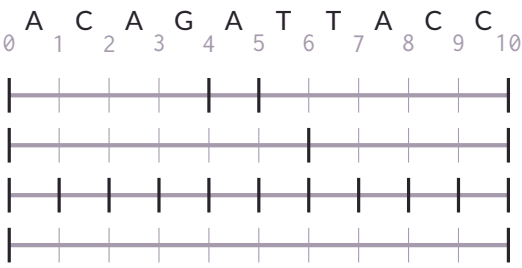
Representing and scoring segmentations



cuts	segments	score
[4,5]	[(0,4), (4,5), (5,10)]	$a_{0,4} + a_{4,5} + a_{5,10}$
[6]	[(0,6), (6,10)]	$a_{0,6} + a_{6,10}$
[1,2,...,9]	[(0,1), …, (9,10)]	$a_{0,1} + a_{1,2} + \ldots + a_{9,10}$
[]	[(0, 10)]	$a_{0,10}$

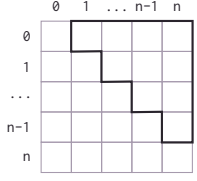
- How many possible segments?
- How many possible *segmentations*?
- Scoring: assign a score to every possible segment (i, j).

Representing and scoring segmentations

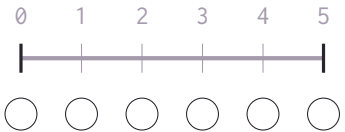


cuts	segments	score
[4,5]	[(0,4), (4,5), (5,10)]	$a_{0,4} + a_{4,5} + a_{5,10}$
[6]	[(0,6), (6,10)]	$a_{0,6} + a_{6,10}$
[1,2,...,9]	[(0,1), ..., (9,10)]	$a_{0,1} + a_{1,2} + \ldots + a_{9,10}$
[]	[(0, 10)]	$a_{0,10}$

- How many possible segments?
- How many possible *segmentations*?
- Scoring: assign a score to every possible segment (i, j).
- You can visualize this as the "upper triangle" of a $(n+1) \times (n+1)$ matrix:

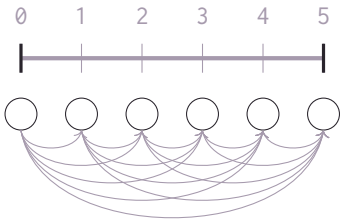


Dynamic programming: DAG formulation



Nodes: one per fencepost. $V = \{0, 1, \ldots, n\}$.

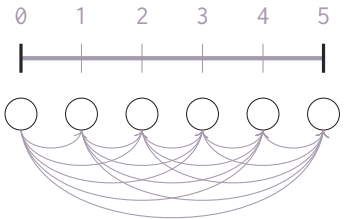
Dynamic programming: DAG formulation



Nodes: one per fencepost. $V = \{0, 1, \ldots, n\}$.

Edges: one per segment.
$E = \{(i, j) : 0 \leq i < j \leq n\}$.

Dynamic programming: DAG formulation

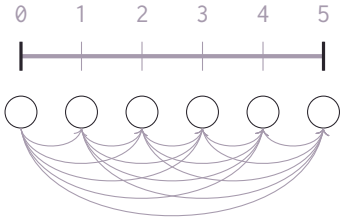


Nodes: one per fencepost. $V = \{0, 1, \ldots, n\}$.

Edges: one per segment.
$E = \{(i, j) : 0 \leq i < j \leq n\}$.

Topologic order?

Dynamic programming: DAG formulation



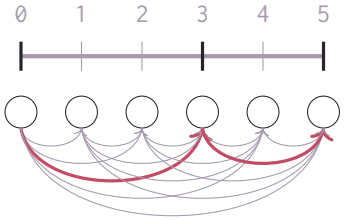
Nodes: one per fencepost. $V = \{0, 1, \ldots, n\}$.

Edges: one per segment.
$E = \{(i, j) : 0 \leq i < j \leq n\}$.

Topologic order?

Any path from 0 to n corresponds
to a segmentation of the sequence.

Dynamic programming: DAG formulation



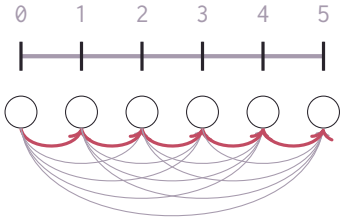
Nodes: one per fencepost. $V = \{0, 1, \ldots, n\}$.

Edges: one per segment.
$E = \{(i, j) : 0 \leq i < j \leq n\}$.

Topologic order?

Any path from 0 to n corresponds
to a segmentation of the sequence.

Dynamic programming: DAG formulation



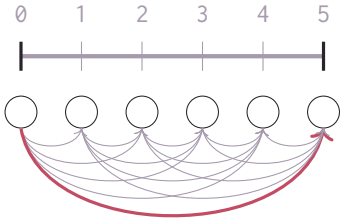
Nodes: one per fencepost. $V = \{0, 1, \ldots, n\}$.

Edges: one per segment.
$E = \{(i, j) : 0 \leq i < j \leq n\}$.

Topologic order?

Any path from 0 to n corresponds
to a segmentation of the sequence.

Dynamic programming: DAG formulation



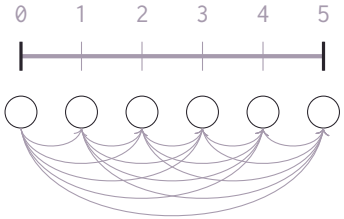
Nodes: one per fencepost. $V = \{0, 1, \ldots, n\}$.

Edges: one per segment.
$E = \{(i, j) : 0 \leq i < j \leq n\}$.

Topologic order?

Any path from 0 to n corresponds
to a segmentation of the sequence.

Dynamic programming: DAG formulation



Nodes: one per fencepost. $V = \{0, 1, \ldots, n\}$.

Edges: one per segment.
$E = \{(i, j) : 0 \leq i < j \leq n\}$.

Topologic order?

Any path from 0 to n corresponds to a segmentation of the sequence.

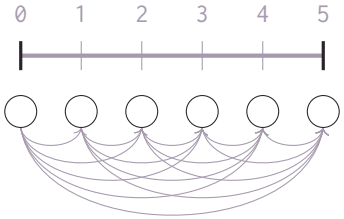
Viterbi for segmentation

input: segment scores $\boldsymbol{A} \in \mathbb{R}^{n \times n}$

Forward: compute recursively
$m_1 = a_{01}; \pi_1 = 0$
for $j = 2$ to n **do**
  $m_j \leftarrow \max_{0 \leq i < j} m_i + a_{ij}$
  $\pi_j \leftarrow \arg\max_{0 \leq i < j} m_i + a_{ij}$
$f^\star = m_n$
Backward: follow backpointers
$\boldsymbol{y}^\star = []; j \leftarrow n$
while $j > 0$ **do**
  $\boldsymbol{y}^\star = [(\pi_j, j)] + \boldsymbol{y}^\star$
  $j = \pi_j$

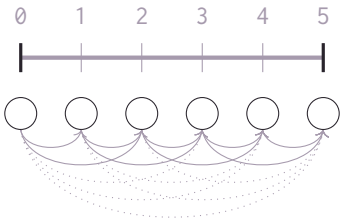
Analogously, we can obtain a *Forward* algorithm for $\log Z$: exercise for you.

Extension 1: Bounded segment length



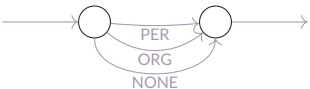
- can be much faster if we limit segment lengts to $L \ll n$.
- in terms of the DAG: discard edges ij where $j - i > L$
- exercise: how does this impact the complexity of Viterbi?

Extension 1: Bounded segment length



- can be much faster if we limit segment lengts to $L \ll n$.
- in terms of the DAG: discard edges ij where $j - i > L$
- exercise: how does this impact the complexity of Viterbi?

Extension 2: Labeled segments



- each segment also receives a label (e.g., `PERSON`, `ORGANIZATION`, `NONE`...)
- the labels are independent given the cuts: for any two nodes in the DAG, we only need to pick the best edge between them.

Extension 3: Labeled + transitions

- drawing inspiration from sequence tagging: what if we want a reward/penalty for consecutive `PERSON`→`ORGANIZATION` segments?
- labels no longer independent given cuts.
- still solvable via DP, but must keep track of transitions.
- essentially a combination of the sequence tagging DAG and the segmentation DAG.

Segmentation structure: Summary

- Segmentations of a length-n sequence: $O(2^n)$ possible segmentations, $O(n^2)$ possible segments.
- Dynamic programming gives polynomial-time probabilistic segmentation models.
- Extensions can accommodate maximum lengths, labels, transitions.

Tagging and segmentation: Historical notes

The model we derived for sequence tagging was first proposed (with non-neural features) under the name “linear chain conditional random field” (Lafferty et al., 2001) and is often informally just called “CRF”; this is confusing.

The segmentation model is technically also a CRF, often called semi-Markov CRF or semi-CRF attributed to Sarawagi and Cohen (2004), to the best of my knowledge the first attestation of the Viterbi algorithm in this model is due to Bridle and Sedgwick (1977). However this conference paper is garbled in the IEEE online archive and can only be found uncorrupted in libraries. It is also (unreferenced) one of the teaching examples of DP in Cormen et al. (2009).

Structure Prediction

1. **Overview**
2. **Structured inputs**
   - Recap: Encoding sequences. RNN, CNN, transformer
   - Encoding graphs
3. **Structured outputs**
   - Probabilistic models of structures
   - Directed acyclic graphs
   - Algorithms for paths in DAGs: Maximization, probabilities, sampling
   - Application: Sequence tagging
   - Application: Sequence segmentation
   - Evaluating structured outputs

Evaluation

(Jurafsky and Martin, 2024, Sec. 4.7)

Well, what would we do in the unstructured case?

Notation: Iverson Bracket

$$[\![p]\!] = \begin{cases} 1, & p \text{ is true,} \\ 0, & \text{otherwise.} \end{cases}$$

- Accuracy:
  What fraction of test cases are correctly classified?

$$\text{Acc} = \frac{1}{N} \sum_{i=1}^{N} [\![y^{(i)} = \widehat{y}^{(i)}]\!]$$

Evaluation

Well, what would we do in the unstructured case?

- Per-class Precision:
  What fraction of the test cases predicted as class c are correctly predicted?

$$P_{(c)} = \frac{\sum_{i=1}^{N} [\![y^{(i)} = c \& y^{(i)} = \widehat{y}^{(i)}]\!]}{\sum_{i=1}^{N} [\![\widehat{y}^{(i)} = c]\!]}$$

- Per-class Recall:
  What fraction of the test cases from class c are correctly predicted?

$$R_{(c)} = \frac{\sum_{i=1}^{N} [\![y^{(i)} = c \& y^{(i)} = \widehat{y}^{(i)}]\!]}{\sum_{i=1}^{N} [\![y^{(i)} = c]\!]}$$

- Per-class F_1 score: $F_{1,(c)} = 2(P_c^{-1} + R_c^{-1})^{-1}$
  Balances precision and recall (harmonic mean).

Binary clf.: usual (and intuitive) to only compute P/R/F for the "positive" class.

Evaluation

Another way to think about P/R/F:

For class c,

TP	TN
FP	FN

- $TP_{(c)}$: true positives: $y^{(i)} = c$ and $\widehat{y}^{(i)} = c$.
- $FP_{(c)}$: false positives: $y^{(i)} \neq c$ and $\widehat{y}^{(i)} = c$.
- $FN_{(c)}$: false negatives: $y^{(i)} = c$ and $\widehat{y}^{(i)} \neq c$.
- $TN_{(c)}$: true negatives: $y^{(i)} \neq c$ and $\widehat{y}^{(i)} \neq c$.

Then,

$$P_{(c)} = \frac{TP_{(c)}}{TP_{(c)} + FP_{(c)}} \quad R_{(c)} = \frac{TP_{(c)}}{TP_{(c)} + FN_{(c)}} \quad \text{Acc}_{(c)} = \frac{1}{N} \sum_{c} TP_{(c)} + TN_{(c)}$$

Evaluation

Macro-average P (or R,F) score over classes

- weighted (by class frequency): denoting $N_c = \sum_{i=1}^{N} [\![y^{(i)} = c]\!]$,

$$\sum_{c=1}^{K} \frac{N_c}{N} P_{(c)}$$

- unweighted:

$$\frac{1}{K} \sum_{c=1}^{K} P_{(c)}$$

Micro-average:

First add up TP, FP, FN, TN over classes.
Then compute P/R/F for this “total” class.

Be explicit and thoughtful!

For instance:

many rare classes that are very easy to recognize -> unweighted F_1 would give an overly optimistic summary close to 1.

class proportions will change at test time or performance should be equally good on all classes, unweighted can make more sense!

Structured evaluation: POS tagging

For sequential data, accuracy already becomes more complicated:

sequence-level?

$$\text{Acc}_{\text{seq}} = \frac{\sum_{i=1}^{N} [\![\boldsymbol{y}^{(i)} = \hat{\boldsymbol{y}}^{(i)}]\!]}{N}$$

or (micro-averaged) tag accuracy? (writing $n^{(i)} = |\boldsymbol{y}^{(i)}|$):

$$\text{Acc}_{\text{tag}} = \frac{\sum_{i=1}^{N} \sum_{j=1}^{n^{(i)}} [\![y_j^{(i)} = \hat{y}_j^{(i)}]\!]}{\sum_{i=1}^{N} n^{(i)}}$$

(could also imagine a macro-averaged version, but it's not meaningful here)

Example:

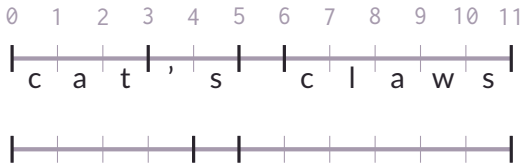
true:	PRO	VERB	NUM	NOUN	ADV
pred:	PRO	VERB	NUM	NOUN	PRO
words:	there	are	70	children	there

true:	INTJ
pred:	X
words:	eeeeek

$$\text{Acc}_{\text{seq}} = \frac{0}{2} = 0$$

$$\text{Acc}_{\text{tag}} = \frac{4}{6} = 0.667$$

Structured evaluation: Segmentations



Gold segments: $\boldsymbol{y} = [(0, 3), (3, 5), (5, 6), (6, 11)]$
Predicted: $\widehat{\boldsymbol{y}} = [(0, 4), (4, 5), (5, 11)]$

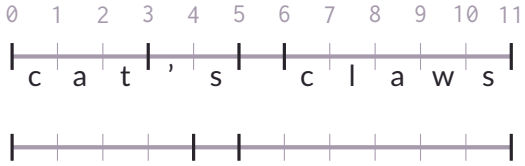
The number of pred. and gold segments differ.

We could interpret this as binary clf of cuts, and evaluate cut accuracy or P/R/F.

Not a great idea:
above, we correctly got the positive cut at 5.
(and correctly said no cut at 1,2,…)

But **no correct segments were returned!**

Structured evaluation: Segmentations



Gold segments: $\boldsymbol{y} = [(0, 3), (3, 5), (5, 6), (6, 11)]$
Predicted: $\widehat{\boldsymbol{y}} = [(0, 4), (4, 5), (5, 11)]$

The number of pred. and gold segments differ.

We could interpret this as binary clf of cuts, and evaluate cut accuracy or P/R/F.

Not a great idea:
above, we correctly got the positive cut at 5.
(and correctly said no cut at 1,2,…)

But **no correct segments were returned!**

Segment-level P/R/F (Sproat and Emerson, 2003):

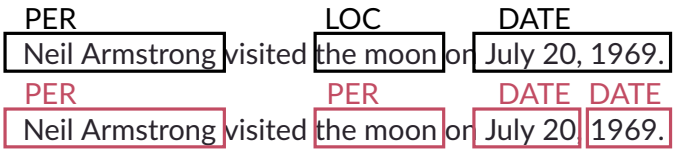
True positive segments (appearing both in $\boldsymbol{y}$ and $\widehat{\boldsymbol{y}}$).
False positive segments (in $\widehat{\boldsymbol{y}}$ but not in $\boldsymbol{y}$)
False negative segments (in $\boldsymbol{y}$ but not in $\widehat{\boldsymbol{y}}$)

$$P = \frac{TP}{TP+FP} = \frac{\text{n. correctly predicted segments}}{\text{n. predicted segments}}$$
$$R = \frac{TP}{TP+FN} = \frac{\text{n. correctly predicted segments}}{\text{n. gold segments}}$$

For this prediction, both are zero.

More advanced metrics: overlap-aware, or "out-of-vocabulary" rates on held-out data.

Structured evaluation: Labeled segmentations



Gold segments:
{ (PER, 0, 2), (LOC, 3, 5), (DATE, 6, 11)}

Pred segments:
{ (PER, 0, 2), (PER, 3, 5), (DATE, 6, 9), (DATE, 9, 11)}

TP = {(PER, 0, 2)}

FP = {(PER, 3, 5), (DATE, 6, 9), (DATE, 9, 11)}

FN = {(LOC, 3, 5), (DATE, 6, 11)}

$P = 1/1 + 3 = .25$ $R = 1/1 + 2 = .33$ $F_1 = .2845$

This is the standard way to evaluate chunking/NER (Tjong Kim Sang and Buchholz, 2000; Tjong Kim Sang, 2002)

Per-class P/R/F, and adding "unlabeled P/R/F" possible, but not standard.

Note: segment accuracy is not useful: the set TN would contain almost all possible segments.

Summary

- Structured objects are made of smaller parts that can interact in a large number of combinations.
- This combinatorial nature adds complexity to evaluation and learning, but also gives us rich, powerful representations.
- We've seen how to encode some structures into feature vectors, taking these relationships and interactions into account.
- We've seen how to predict structures, by representing them as paths in DAGs and using dynamic programming algorithms.
- We built such models for sequence tagging and segmentations. Other structures can be modeled in this way too, and some cannot. My full course on this at `https://vene.ro/mlsd` goes deeper.

References I

- Baum, Leonard E. (1972). “An inequality and associated maximization technique in statistical estimation of probabilistic functions of a Markov process”. In.
- Bellman, Richard (1954). “The theory of dynamic programming”. In: *Bulletin of the American Mathematical Society* 60.6, pp. 503–515.
- Bridle, J. and N. Sedgwick (1977). “A method for segmenting acoustic patterns, with applications to automatic speech recognition”. In: *ICASSP ’77. IEEE International Conference on Acoustics, Speech, and Signal Processing*. Vol. 2, pp. 656–659.
- Cormen, Thomas H et al. (2009). *Introduction to algorithms (third edition)*. MIT press.
- Ferguson, JD (1980). “Application of hidden Markov models to text and speech”. In: *Princeton, NJ), IDA-CRD*.

References II

- Frühwirth-Schnatter, Sylvia (1994). “Data augmentation and dynamic linear models”. In: *Journal of Time Series Analysis* 15.2, pp. 183–202. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-9892.1994.tb00184.x.
- Goldberg, Yoav (2017). *Neural Network Methods in Natural Language Processing*. Morgan & Claypool.
- Hamilton, William L. (2020). *Graph Representation Learning*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool.
- Huang, Liang (Aug. 2008). “Advanced Dynamic Programming in Semiring and Hypergraph Frameworks”. In: *Coling 2008: Advanced Dynamic Programming in Computational Linguistics: Theory, Algorithms and Applications - Tutorial notes*. Ed. by Liang Huang. Manchester, UK: Coling 2008 Organizing Committee, pp. 1–18.

References III

- Jurafsky, Daniel and James H. Martin (2024). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. 3rd. Online manuscript released August 20, 2024.
- Lafferty, John, Andrew McCallum, Fernando Pereira, et al. (2001). “Conditional random fields: Probabilistic models for segmenting and labeling sequence data”. In: *Icml*. Vol. 1. 2. Williamstown, MA, p. 3.
- Mohri, Mehryar (2002). “Semiring frameworks and algorithms for shortest-distance problems”. In: *J. Autom. Lang. Comb.* 7.3, pp. 321–350.
- Murphy, Kevin P. (2022). *Probabilistic Machine Learning: An introduction*. MIT Press.
- Rabiner, Lawrence R (n.d.). *First-hand: The Hidden Markov Model*. `https://ethw.org/First-Hand:The_Hidden_Markov_Model`.

References IV

- Sarawagi, Sunita and William W Cohen (2004). “Semi-Markov Conditional Random Fields for Information Extraction”. In: *Advances in Neural Information Processing Systems*. Ed. by L. Saul, Y. Weiss, and L. Bottou. Vol. 17. MIT Press.
- Smith, Noah A. (2011). *Linguistic Structure Prediction*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool.
- Sproat, Richard and Thomas Emerson (July 2003). “The First International Chinese Word Segmentation Bakeoff”. In: *Proceedings of the Second SIGHAN Workshop on Chinese Language Processing*. Sapporo, Japan: Association for Computational Linguistics, pp. 133–143.
- Tjong Kim Sang, Erik F. (2002). “Introduction to the CoNLL-2002 Shared Task: Language-Independent Named Entity Recognition”. In: *COLING-02: The 6th Conference on Natural Language Learning 2002 (CoNLL-2002)*.
- Tjong Kim Sang, Erik F. and Sabine Buchholz (2000). “Introduction to the CoNLL-2000 Shared Task Chunking”. In: *Fourth Conference on Computational Natural Language Learning and the Second Learning Language in Logic Workshop*.

References V

Viterbi, A. (1967). “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm”. In: *IEEE Transactions on Information Theory* 13.2, pp. 260–269.