# Structure Prediction

## Part 0:

Machine Learning for Structured Data
Vlad Niculae · LTL, UvA · https://vene.ro/mlsd

# Structure Prediction

# 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.*

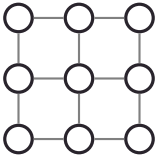
Many objects we want to do ML on have interesting structure:

language, images, shapes, networks…

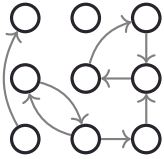
This lecture: how to work with structure in the input and the output.

# A few examples of structure

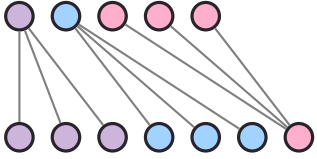
Sequence



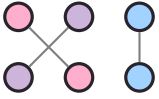
Grid



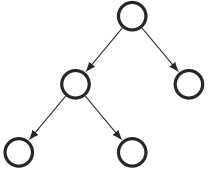
Graph



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.

# 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 cover feature encoders for structured objects.

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

# Structure Prediction

# Structure Prediction

# Encoding Sequences

You've already seen several methods for encoding text as a sequence of discrete tokens. We recap.

# Sequence input: Bag-of-words

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 | $\phi_1$ | $\phi_1$ | $\phi_2$ | $\phi_3$ | $\phi_4$ | $\phi_5$ | $\phi_6$ | $\phi_7$ | $\phi_8$ | $\phi_9$ |
| $x_1$ | “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

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 | $\phi_1$ | $\phi_1$ | $\phi_2$ | $\phi_3$ | $\phi_4$ | $\phi_5$ | $\phi_6$ | $\phi_7$ | $\phi_8$ | $\phi_9$ |
| $x_1$ | “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?

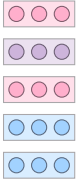
[1] Ensure combination is reversible or else we won't be able to distinguish features.
For instance, here, _ must not appear in any unigram.

# Embeddings of Discrete Tokens

Neural networks perform continuous operations.

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

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

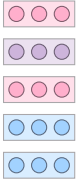


# Embeddings of Discrete Tokens

Neural networks perform continuous operations.

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

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

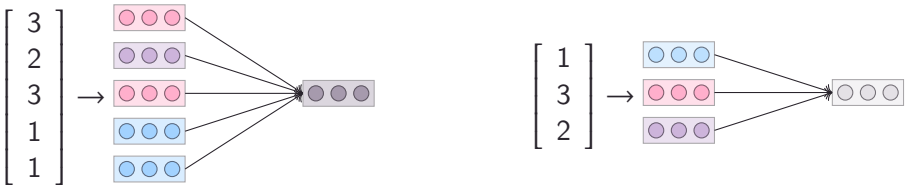


The function $\boldsymbol{e}(i)$ retrieves the $i$th row from an *embedding matrix* $\mathbf{E} \in \mathbb{R}^{|V| \times d}$.

The embeddings could be fixed or learned as model parameters.

# Continuous Bag Of Words

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

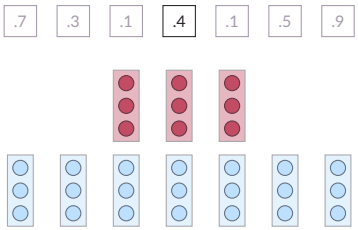



- average pooling: $\boldsymbol{z} = \frac{1}{L}(\boldsymbol{z}_1 + \ldots + \boldsymbol{z}_L)$
- max pooling: $[\boldsymbol{z}]_j = \max([\boldsymbol{z}_1]_j, \ldots, [\boldsymbol{z}_L]_j$ (coordinate-wise)

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

# Sequence convolutions

**aka 1-d convolution with $d$ channels**

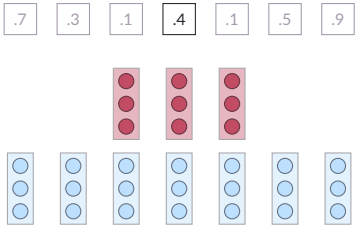



- Denote $L$=sequence length, $d$=embedding size, $k$=window size.

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**

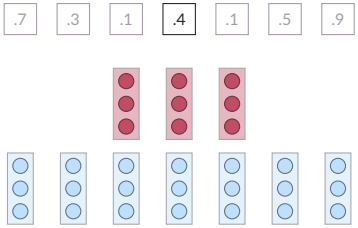



- Denote $L$=sequence length, $d$=embedding size, $k$=window size.
- In the single-channel case, a filter was a dim-$k$ vector. Now, a filter is a $d \times k$ matrix.

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**

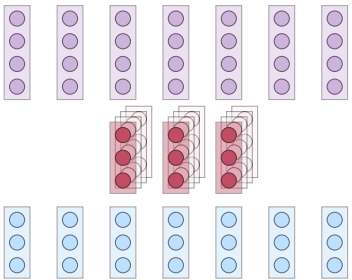



- Denote $L$=sequence length, $d$=embedding size, $k$=window size.
- In the single-channel case, a filter was a dim-$k$ vector. Now, a filter is a $d \times k$ matrix.
- Output is still a single number per window.

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

**aka 1-d convolution with $d$ channels**



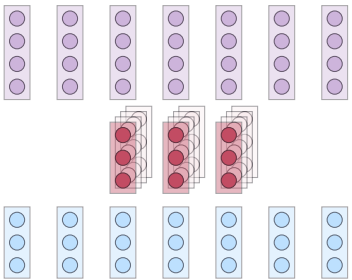
- Denote $L$=sequence length, $d$=embedding size, $k$=window size.
- In the single-channel case, a filter was a dim-$k$ vector. Now, a filter is a $d \times k$ matrix.
- Output is still a single number per window.
- Apply $m$ filters in parallel: output is a dim-$m$ vector per window:

  a "layer" maps $(L, d) \to (L, m)$, 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

**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:

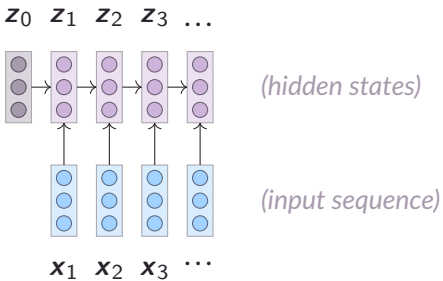
  a "layer" maps $(L, d) \to (L, m)$, for any $L$.
- Kind of like "continuous" n-grams!

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:

$$\boldsymbol{z}_t = \sigma\Bigg(\underbrace{\boldsymbol{W}\boldsymbol{x}_t}_{\text{lin. func. of inputs}} + \underbrace{\boldsymbol{U}\boldsymbol{z}_{t-1}}_{\text{lin. func. of prev. state}} + \boldsymbol{b}\Bigg)$$

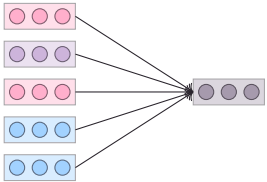
Each hidden state depends on the previous ones. Therefore, cannot parallelize, must compute in order $\boldsymbol{z}_1, \boldsymbol{z}_2, \ldots$

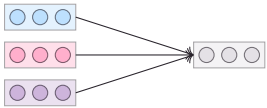
The initial state $\boldsymbol{z}_0$ is a fixed parameter.

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

# Let's talk about pooling.

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



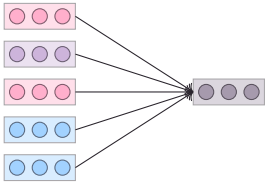


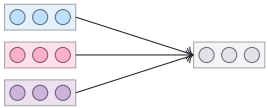
Used to get one representation of a variable-size set or sequence.

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

# Let's talk about pooling.

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





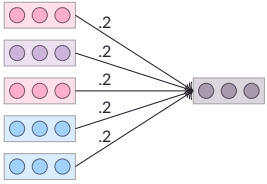
Used to get one representation of a variable-size set or sequence.

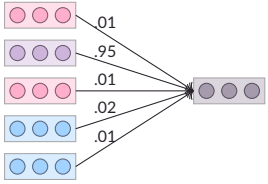
Combine $n$ input vectors into one single output vector,
with equal contribution.

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

# Weighted Average Pooling

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

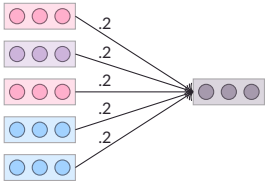


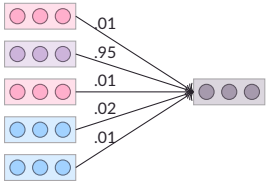




The weights $\boldsymbol{\alpha}$ control the relative importance of the inputs.

# 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 $\boldsymbol{q}$:

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

# 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 $\boldsymbol{q}$:

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

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 $\boldsymbol{q}$:

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

This is the basic **attention mechanism**:

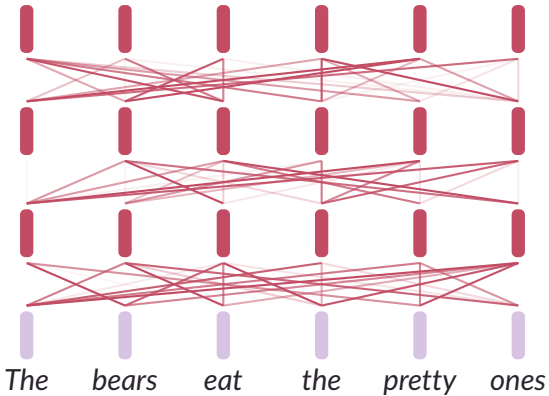
Pool a bunch of vectors, with varying weights,
depending on how aligned they are with a context.

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,
  $\boldsymbol{q}$ can be an embedding of the last NL word predicted (one by one).

# 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.

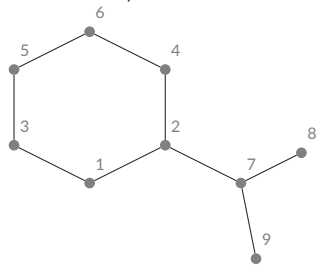
# Structure Prediction

# 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.

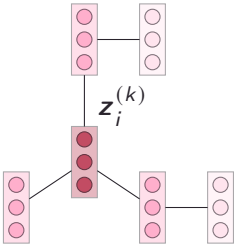
# Graph datasets

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, …
   - can be given as a sequence of node labels $(x_1^{(i)}, \ldots, x_{n_i}^{(i)})$ and an adjacency matrix $\boldsymbol{A}^{(i)}$
   - this is what you have in assignment 1
2. Data points are parts of one big graph.
   - e.g., node classification (classifying bots on twitter), link prediction (instagram follow suggestions), community detection, …
   - much 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:

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

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

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

# Efficiently computing the messages

The message received by each node is a sum of its neighbors' embeddings:

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

# Efficiently computing the messages

The message received by each node is a sum of its neighbors' embeddings:

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

Denote by $\boldsymbol{Z} \in \boldsymbol{R}^{n \times d}$ the matrix of stacked node embeddings,
($n$ =num. nodes, $d$ =embedding dimension).

The $i$th column of the adjacency matrix $\boldsymbol{a}_i$ encodes the (in-)neighbors of node $i$.

$$\boldsymbol{a}_i^\top \boldsymbol{Z} =$$

# Efficiently computing the messages

The message received by each node is a sum of its neighbors' embeddings:

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

Denote by $\boldsymbol{Z} \in \boldsymbol{R}^{n \times d}$ the matrix of stacked node embeddings,
($n$ =num. nodes, $d$ =embedding dimension).

The $i$th column of the adjacency matrix $\boldsymbol{a}_i$ encodes the (in-)neighbors of node $i$.

$$\boldsymbol{a}_i^\top \boldsymbol{Z} = \sum_j a_{ij} \boldsymbol{z}_j =$$

# Efficiently computing the messages

The message received by each node is a sum of its neighbors' embeddings:

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

Denote by $\boldsymbol{Z} \in \boldsymbol{R}^{n \times d}$ the matrix of stacked node embeddings,
($n$ =num. nodes, $d$ =embedding dimension).

The $i$th column of the adjacency matrix $\boldsymbol{a}_i$ encodes the (in-)neighbors of node $i$.

$$\boldsymbol{a}_i^\top \boldsymbol{Z} = \sum_j a_{ij} \boldsymbol{z}_j = \boldsymbol{m}_i$$

# Efficiently computing the messages

The message received by each node is a sum of its neighbors' embeddings:

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

Denote by $\boldsymbol{Z} \in \boldsymbol{R}^{n \times d}$ the matrix of stacked node embeddings,
($n$ =num. nodes, $d$ =embedding dimension).

The $i$th column of the adjacency matrix $\boldsymbol{a}_i$ encodes the (in-)neighbors of node $i$.

$$\boldsymbol{a}_i^\top \boldsymbol{Z} = \sum_j a_{ij} \boldsymbol{z}_j = \boldsymbol{m}_i$$

Compute all messages at once:

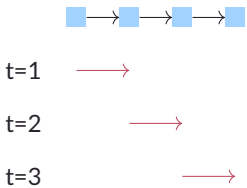
$$\boldsymbol{M} = \boldsymbol{A}^\top \boldsymbol{Z}$$

# Structure Prediction

# RNN vs GNN

The sequence (chain) graph is also a graph, we could use a GNN.

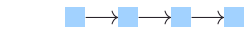
**RNN:** sequential updates



# RNN vs GNN

The sequence (chain) graph is also a graph, we could use a GNN.

**RNN:** sequential updates

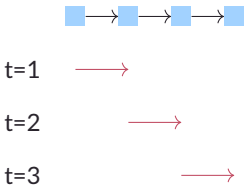



**GNN:** parallel local updates

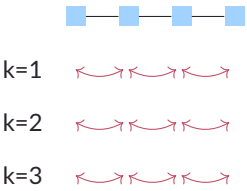
# RNN vs GNN

The sequence (chain) graph is also a graph, we could use a GNN.

**RNN:** sequential updates




**GNN:** parallel local updates

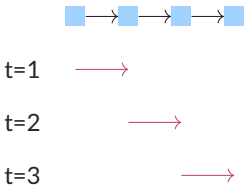



- Propagates through entire sequence with $L$ “messages”.
- Embeddings only aware of nodes to the left (without bidirectional “hack”)
- Defined for sequences only (some extensions possible).

# RNN vs GNN

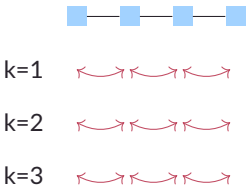
The sequence (chain) graph is also a graph, we could use a GNN.

**RNN:** sequential updates




- Propagates through entire sequence with $L$ “messages”.
- Embeddings only aware of nodes to the left (without bidirectional “hack”)
- Defined for sequences only (some extensions possible).

**GNN:** parallel local updates




- After $k$ iterations, every node got updates from its neighborhood up to $k$ steps away.
- Can be used for any graph.

# 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: $[\boldsymbol{z}]_j = \max([\boldsymbol{z}_1]_j, \ldots, [\boldsymbol{z}_n]_j)$

# Structure Prediction

# 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

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

# Structure Prediction

# GNN variants

Many variations can be built on top of this idea.

- The update $\boldsymbol{z}_i^{(k+1)} = \phi(\boldsymbol{W}_{\text{self}}\boldsymbol{z}_i^{(k)} + \boldsymbol{W}_{\text{neigh}}\boldsymbol{m}_i^{(k)} + \boldsymbol{b})$ 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:

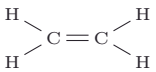
$$\alpha_{ij} = \frac{\exp(\boldsymbol{q}_i \cdot \boldsymbol{k}_j)}{\sum_{j' \in N(i)} \exp(\boldsymbol{q}_i \cdot \boldsymbol{k}_{j'})}$$

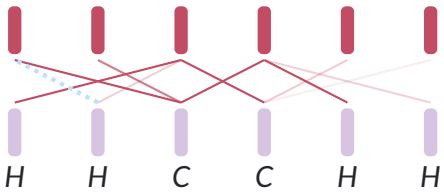
$$\boldsymbol{m}_i = \sum_{j \in N(i)} \alpha_{ij} \boldsymbol{v}_j$$

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

(no attention allowed where there is no edge)

Ethylene ($C_2H_4$):



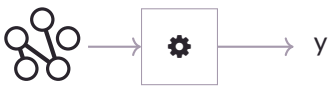



# Structure Prediction

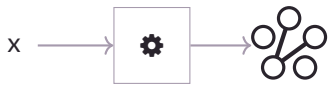
# Structure Prediction

So far, we've studied this scenario:

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

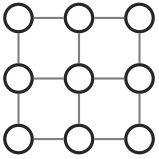



In the next part of class,
we study **structured outputs**.

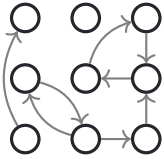


# Reminder: Kinds of Structure

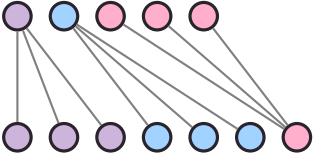
Sequence



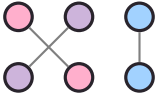
Grid



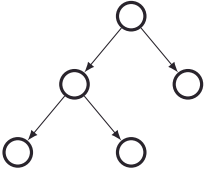
Graph



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?

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

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.

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)

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

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

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 $\text{score}(y) = a_y$.

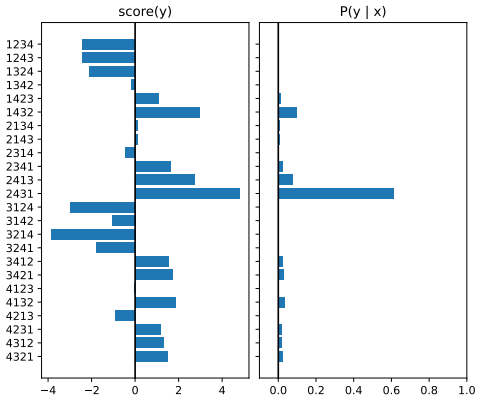
Can we generalize this to structured $\mathcal{Y}$?

# 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.

# The Road Ahead

In the rest of the class, we shall cover a wide range of structured output tasks:

- Sequence labelling
- Sequence segmentation
- Alignments between sequences;
- Assignments and permutations
- Grid / graph labelling

While there is no general-purpose structure prediction algorithm, we shall learn a formalism that will get you far.

# Structure Prediction

# 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).

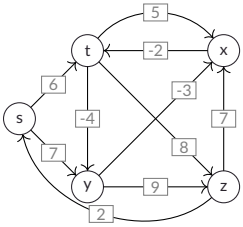
So, we must actually make use of its structure.

# 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.

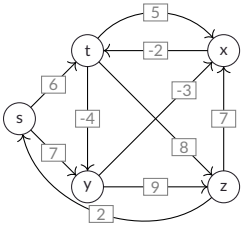


# Recap: Graphs

**Definition 1: Weighted directed graph**

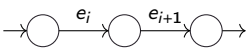
A weighted directed graph is $G = (V, E, w)$ where:

- $V$ 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 $v_i = u_{i+1}$.

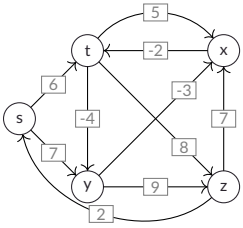


# Recap: Graphs

**Definition 1: Weighted directed graph**

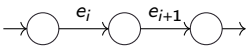
A weighted directed graph is $G = (V, E, w)$ where:

- $V$ 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: $A = e_1 e_2 \ldots e_k$, 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 $v_i = u_{i+1}$.



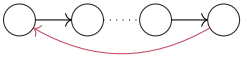

The weight of a path is the sum of arc weights: $w(A) = \sum_{e \in P} w(e)$.

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

# Directed Acyclic Graphs

**Definition 3: Cycle**

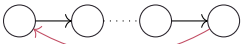
A cycle is a path $e_1 e_2 \ldots e_k$ wherein the last edge $e_k$ points to the node from which the first edge $e_1$ departs.



# Directed Acyclic Graphs

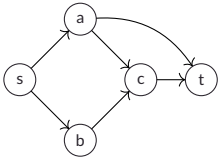
**Definition 3: Cycle**

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)**

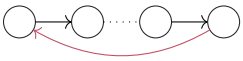
A DAG is a directed graph that contains no cycles.



# Directed Acyclic Graphs

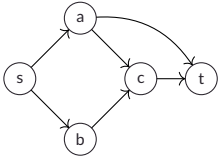
**Definition 3: Cycle**

A cycle is a path $e_1 e_2 \ldots e_k$ 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.




**Definition 4. Topological ordering**

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

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

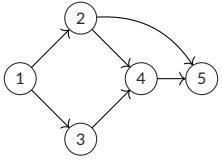
$G$ is a DAG if and only if $G$ admits a topological ordering.
Rough intuition: "backward" edges against the ordering $\iff$ cycles.

# Structure Prediction

# Paths In DAGs

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

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

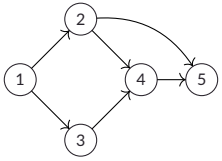


# Paths In DAGs

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:

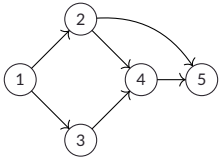
- $\text{score}(y) = w(y)$
- $\text{argmax}_{y \in \mathcal{Y}_n} w(y)$
- $\log \sum_{y \in \mathcal{Y}_n} \exp w(y)$

# Paths In DAGs

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 $\mathcal{Y} = \mathcal{Y}_n$.



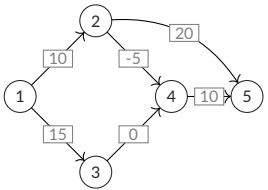

Important things to compute:

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

Later, I'll show you some structured problems that can be usefully reduced to paths in a DAG, and some that cannot.

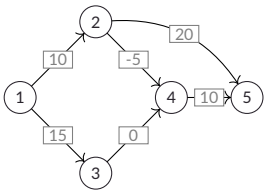
# Max-Scoring Path

- 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. .

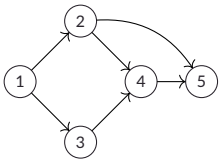


# Max-Scoring Path

- 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.



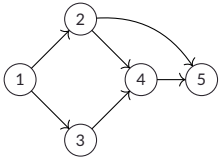
# Dynamic Programming Recurrence




**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 $i$:

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

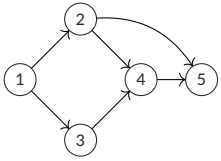
Define predecessors of $i$ as $P_i := \{j \in V : ji \in E\}$.

**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 $ji$.

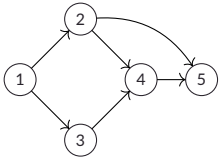
In other words: if $y \in \mathcal{Y}_i$ then $y = y'^\frown ji$ for some $j \in P_i$ and some $y' \in \mathcal{Y}_j$.

**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 $y' \in \mathcal{Y}_j$.

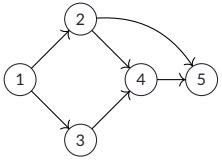
**Proposition: DP recurrence for max**

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 $y' \in \mathcal{Y}_j$.

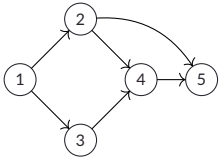
**Proposition: DP recurrence for max**

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

# 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 $y' \in \mathcal{Y}_j$.

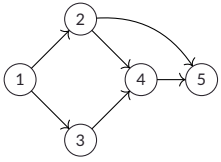
**Proposition: DP recurrence for max**

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

# 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**

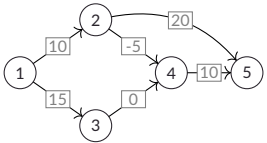
For any $i > 1$, the best path from 1 to $i$ is the best among the extensions of the best path to the predecessors of $i$:

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

Proof:

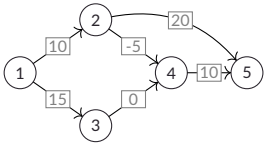
$$\begin{aligned} m_i &:= \max_{y \in \mathcal{Y}_i} w(y) \\ &= \max_{j \in P_i} \max_{y' \in \mathcal{Y}_j} \left(w(y') + w(ji)\right) \\ &= \max_{j \in P_i} \left( \max_{y' \in \mathcal{Y}_j} (w(y')) + w(ji) \right) \\ &= \max_{j \in P_i} \big(m_j + w(ji)\big) . \end{aligned}$$

# 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



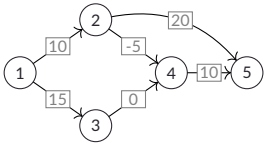

$m_i = \max_{j \in P_i} \left(m_j + w(ji)\right)$ holds for any graph; but we would chase our own tail forever.

**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




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

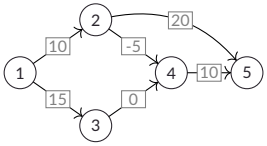
(So, we may compute $m_1, \ldots, m_n$ in order.)

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

# 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.**

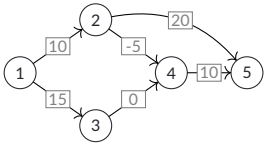
A path acheiving maximal weight is made up of the edges $j^\star i$, where $j^\star$ is the node selected by the max at each iteration.

**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




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

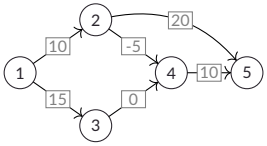
**General Viterbi algorithm for DAGs**

**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**
$m_i \leftarrow \max_{j \in P_i} \big(m_j + w(ji)\big)$
$\pi_i \leftarrow \arg\max_{j \in P_i} \big(m_j + w(ji)\big)$

*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




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

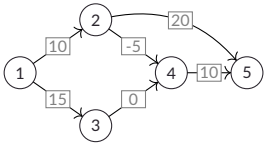
**input:** Topologically-ordered DAG
$G = (V, E, w)$, $V = \{1, \ldots, n\}$
**output:** maximum path weights $m_1, \ldots, m_n$.

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

*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**

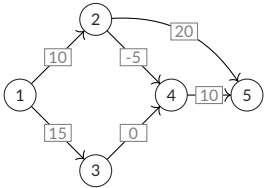
**input:** Topologically-ordered DAG
$G = (V, E, w)$, $V = \{1, \ldots, n\}$
**output:** maximum path weights $m_1, \ldots, m_n$.

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 $n$:

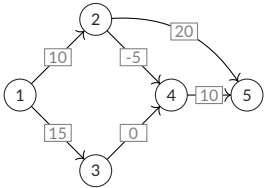
$$\Pr(y) = \frac{\exp(w(y))}{\sum_{y' \in \mathcal{Y}_n} \exp(w(y'))}$$

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

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 $n$:

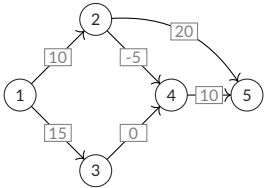
$$\Pr(y) = \frac{\exp(w(y))}{\sum_{y' \in \mathcal{Y}_n} \exp(w(y'))}$$

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

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 $n$:

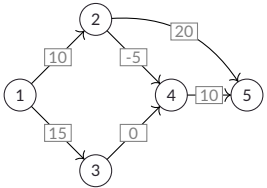
$$\Pr(y) = \frac{\exp(w(y))}{\sum_{y' \in \mathcal{Y}_n} \exp(w(y'))}$$

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

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.**

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

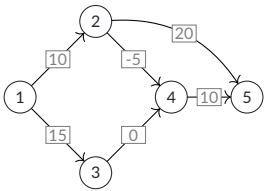
Compare with the DP recurrence for max:

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

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?

Why are these two algorithms so similar?

Deriving the DP recurrences was almost identical.

# Spot A Pattern?

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?

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

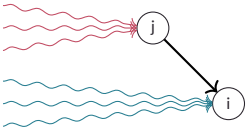
  $$\mathbb{E}_{\Pr(Y)}[F(Y)]$$

  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 $ji$:

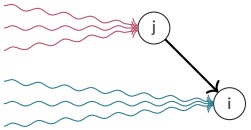
$\Pr(ji|y \text{ ends in } i) =$



# Sampling: One Arc At A Time

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

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

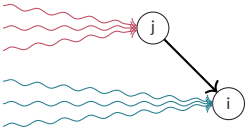


# Sampling: One Arc At A Time

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

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

$$= \frac{\exp(w(ji)) \sum_{y' \in \mathcal{Y}_j} \exp(w(y'))}{\sum_{y \in \mathcal{Y}_i} \exp(w(y))}$$



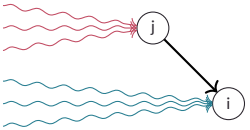
# Sampling: One Arc At A Time

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

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

$$= \frac{\exp(w(ji))\sum_{y'\in\mathcal{Y}_j} \exp(w(y'))}{\sum_{y\in\mathcal{Y}_i} \exp(w(y))}$$

$$= \exp(w(ji) + q_j - q_i)$$

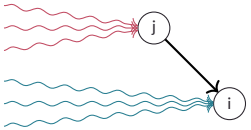


# Sampling: One Arc At A Time

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

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

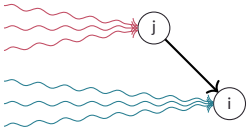
All paths end in $n$, so draw the final arc $jn$ first.



# 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.

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

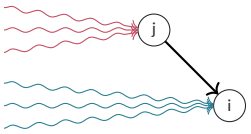
Probability that the last arc of a path ending in $i$ is $ji$:

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

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. $p_j = \exp(w(ji) + q_j - q_i)$
  $y \leftarrow ji^\frown y$
  $i \leftarrow j$

# Conclusions

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.

# Structure Prediction

# 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

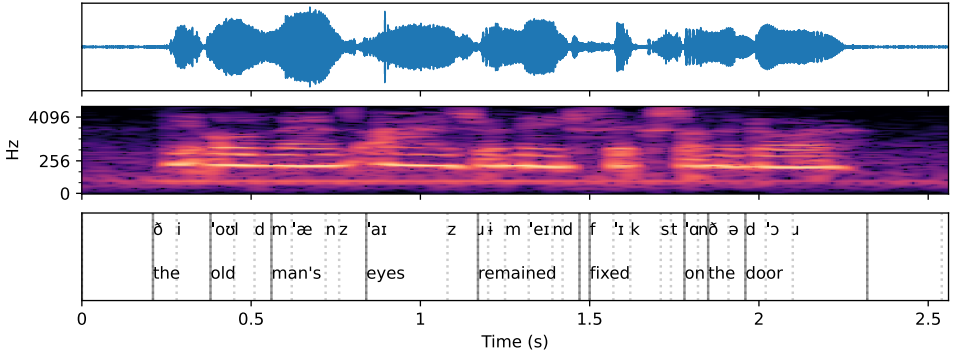
| | the | old | man | the | boat |
|---|---|---|---|---|---|
| $\boldsymbol{y}_a$ | det | adj | noun | det | noun |
| $\boldsymbol{y}_b$ | det | noun | verb | det | noun |

# 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)

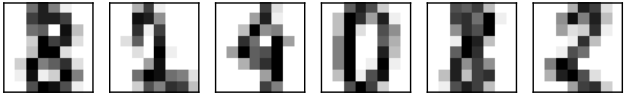


# Sequence Tagging

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, Speech and Language Processing. https://web.stanford.edu/~jurafsky/slp3/8.pdf ©Jurafsky and Martin

# POS Tagging Evaluation

Evaluation: sequence-level accuracy

$$\frac{\sum_{i=1}^{N_{\text{valid}}} \boldsymbol{y}^{(i)} = \hat{\boldsymbol{y}}^{(i)}}{N_{\text{valid}}}$$

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

$$\frac{\sum_{i=1}^{N_{\text{valid}}} \sum_{j=1}^{n^{(i)}} y_j^{(i)} = \hat{y}_j^{(i)}}{\sum_{i=1}^{N_{\text{valid}}} n^{(i)}}$$

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 |

# 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:

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

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

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

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

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

$\boldsymbol{A} =$

| | 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.

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 Transitions Between Tags

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.)

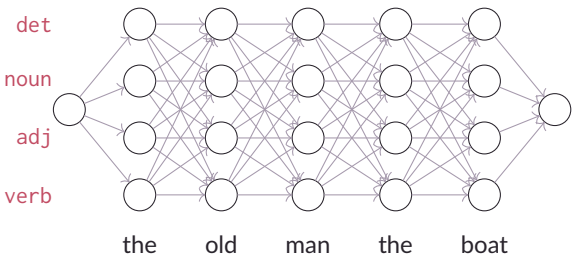
The transition scores $\boldsymbol{T} \in \mathbb{R}^{K \times K}$:

- 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:

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

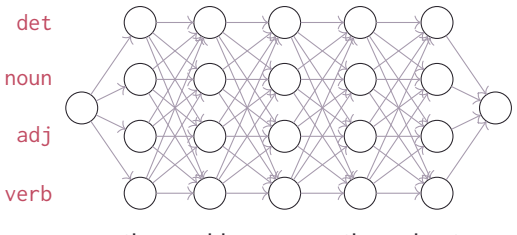
$$E = \{(j-1, c') \to (j, c) : j \in [2, n], c, c' \in [K]\} \cup \{s \to (1, c) : c \in [K]\} \cup \{(n, c) \to t : c \in [K]\}$$

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

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

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.


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.


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 | $\text{score}(y_a) = 25$ |
| $y_b$ | 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

$\alpha =$

| | 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:

$A =$

| | 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 |

| | 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

$\alpha =$

| | 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:

$A =$

| | 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 |

| | 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

$\alpha =$

| | 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 |

$$\log Z \approx 26.885$$

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

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 |

| | 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

$\alpha =$

| | 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 |

$$\log Z \approx 26.885$$

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

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

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 |

| | 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

$\boldsymbol{\alpha}$ =

| | 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 |

$$\log Z \approx 26.885$$

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

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

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

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 |

# 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
   $-\log P(\boldsymbol{y} \mid \boldsymbol{x}) = -\text{score}(\boldsymbol{y}) + \log \sum_{\boldsymbol{y}' \in \mathcal{Y}} \exp \text{score}(\boldsymbol{y}')$.

This probabilistic model is often known as a Linear-Chain Conditional Random Field.

(Historically, Linear-Chain CRFs didn't use neural net scorers, but the math doesn't change. Today I prefer to teach it this way.)

# 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

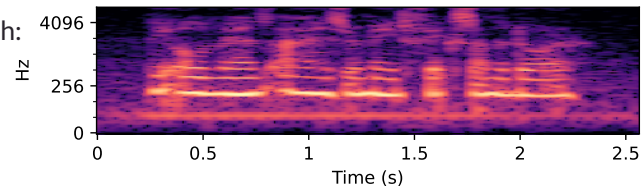
The rod cutting problem: We have a rod of length $n$ units, and we can cut it at every marker. What cuts to make to maximize the total value of the resulting pieces?


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:



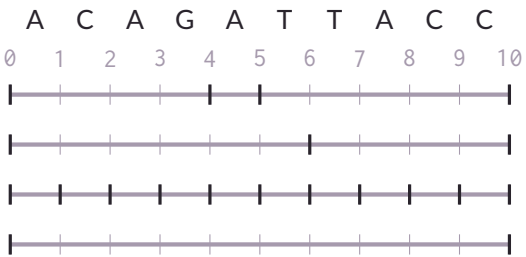
# Representing and scoring segmentations

# Representing and scoring segmentations

# Representing and scoring segmentations

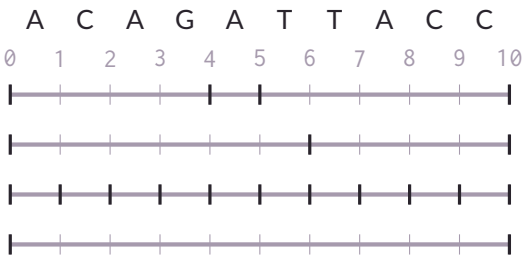

| cuts | segments |
|---|---|
| [4,5] | [(0,4), (4,5), (5,10)] |
| [6] | [(0,6), (6,10)] |
| [1,2,...,9] | [(0,1), . . . , (9,10)] |

# 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)] |

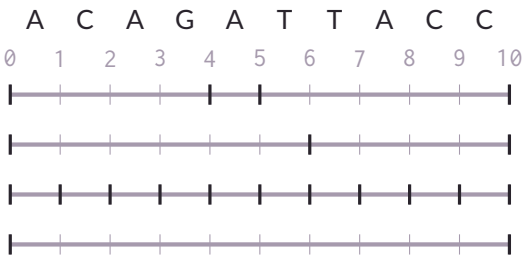
# Representing and scoring segmentations




| 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?

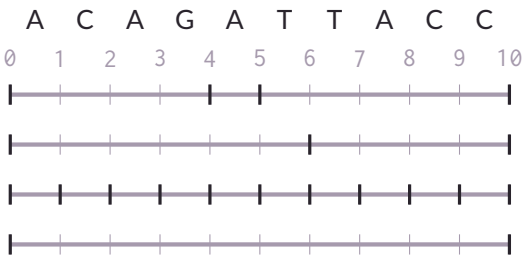
# Representing and scoring segmentations




| 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*?

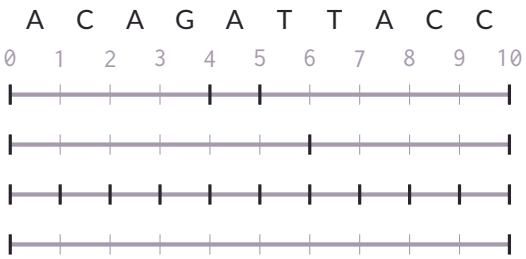
# Representing and scoring segmentations




| 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)] | $a_{0,10}$ |

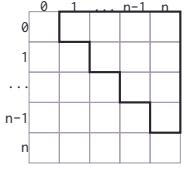
- How many possible segments?
- How many possible *segmentations*?
- Scoring: assign a score to every possible segment $(i, j)$.

# Representing and scoring segmentations

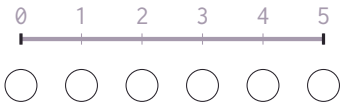



| cuts | segments | score |
|---|---|---|
| [4,5] | [(0,4), (4,5), (5,10)] | $a_{0,4} + a_{4,5} + a_{5,10}$ |
| [6] | [(0,6), (6,10)] | $a_{0,6} + a_{6,10}$ |
| [1,2,...,9] | [(0,1), ..., (9,10)] | $a_{0,1} + a_{1,2} + \ldots + a_{9,10}$ |
| [] | [(0, 10)] | $a_{0,10}$ |

- 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:

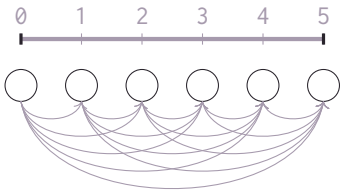


# Dynamic programming: DAG formulation




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

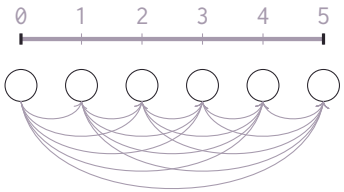
# Dynamic programming: DAG formulation




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

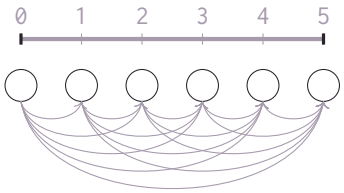



Nodes: one per fencepost. $V = \{0, 1, \ldots, n\}$.

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

Topologic order?

# Dynamic programming: DAG formulation



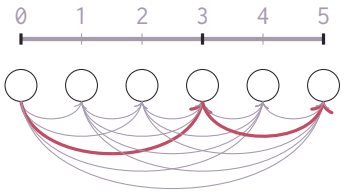

Nodes: one per fencepost. $V = \{0, 1, \ldots, n\}$.

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



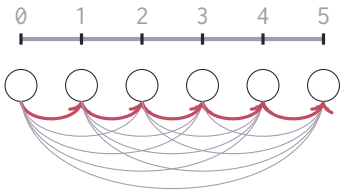

Nodes: one per fencepost. $V = \{0, 1, \ldots, n\}$.

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



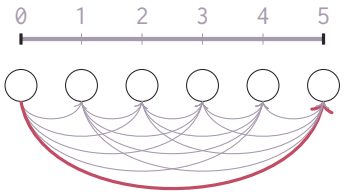

Nodes: one per fencepost. $V = \{0, 1, \ldots, n\}$.

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



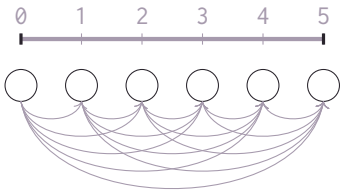

Nodes: one per fencepost. $V = \{0, 1, \ldots, n\}$.

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.

# Evaluation


True segments: $\boldsymbol{y} = [(0, 3), (3, 5), (5, 6), (6, 11)]$

A few possible predictions:
$\hat{\boldsymbol{y}}_a = [(0, 11)]$

$\hat{\boldsymbol{y}}_b = [(0, 1), (1, 2), \ldots, (10, 11)]$

$\hat{\boldsymbol{y}}_c = [(0, 3), (3, 5), (5, 11)]$

# Evaluation


True segments: $\boldsymbol{y} = [(0, 3), (3, 5), (5, 6), (6, 11)]$

A few possible predictions:
$\hat{\boldsymbol{y}}_a = [(0, 11)]$

$\hat{\boldsymbol{y}}_b = [(0, 1), (1, 2), \ldots, (10, 11)]$

$\hat{\boldsymbol{y}}_c = [(0, 3), (3, 5), (5, 11)]$

The number of predicted and true segments differ.

A common way to evaluate in this scenario is:

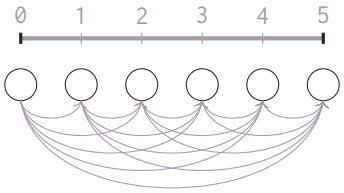
$$\text{precision} = \frac{\text{n. correctly predicted segments}}{\text{n. predicted segments}}$$

$$\text{recall} = \frac{\text{n. correctly predicted segments}}{\text{n. true segments}}$$

$$F_1 = \frac{2PR}{P + R}$$

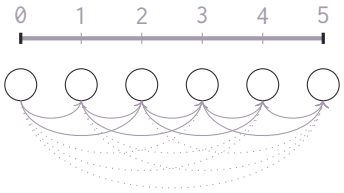
More advanced metrics can partially reward overlaps.

# Extension 1: bounded segment length



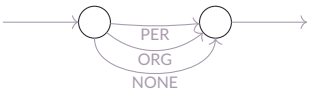

- can be much faster if we limit segment lengts to $L \ll n$.
- in terms of the DAG: discard edges $ij$ where $j - i > L$
- exercise: how does this impact the complexity of Viterbi?

# 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.

# 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.