# Encoder-Decoder Models

Jindřich Libovický, Jindřich Helcl

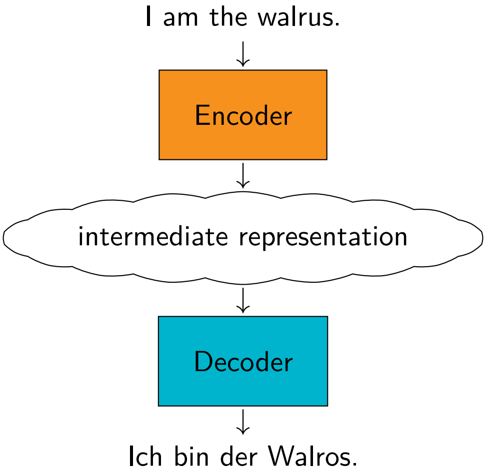
March 03, 2022

EUROPEAN UNION
European Structural and Investment Fund
Operational Programme Research,
Development and Education

Charles University
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics

# Model Concept

# Conceptual Scheme of the Model




Neural model with a sequence of discrete symbols as an input that generates another sequence of discrete symbols as an output.

- pre-process source sentence (tokenize, split into smaller units)
- convert input into vocabulary indices
- run the encoder to get an intermediate representation (vector/matrix)
- run the decoder
- postprocess the output (detokenize)

# Language Models and Decoders

# What is a Language Model

LM = an estimator of a sentence probability given a language

- From now on: sentence = sequence of words $w_1, \ldots, w_n$
- Factorize the probability by word
  i.e., no grammar, no hierarchical structure

$$\begin{aligned}\Pr(w_1, \ldots, w_n) &= \Pr(w_1) \cdot \Pr(w_2|w_1) \cdot \Pr(w_3|w_2, w_1) \cdot \cdots \\ &= \prod_i^n \Pr(w_i|w_{i-1}, \ldots, w_1)\end{aligned}$$

# What is it good for?

- Substitute for grammar: tells what is a good sentence in a language
- Used in ASR, and statistical MT to select more probable outputs
- Being able to predict next word = proxy for knowing the language
  - language modeling is training objective for word2vec
  - BERT is a masked language model

- **Neural decoder is a conditional language model.**

# $n$-gram vs. Neural LMs

**$n$-gram**
cool from 1990 to 2013

- Limited history = Markov assumption
- Transparent: estimated from $n$-gram counts in a corpus

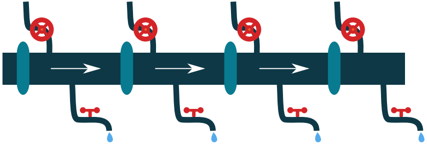
$$\mathsf{P}(w_i|w_{i-1}, w_{i-2}, \dots, w_{i-n}) \approx \sum_{j=0}^{n} \lambda_j \frac{c(w_i|w_{i-1}, \dots, w_{i-j})}{c(w_i|w_{i-1}, \dots, w_{i-j+1})}$$

---

**Neural**
cool since 2013

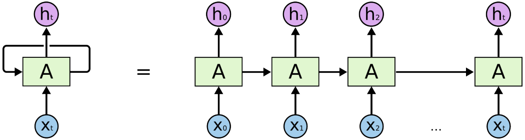
- Conditioned on RNN state which gather potentially unlimited history
- Trained by back-propagation to maximize probability of the training data
- Opaque, but works better (as usual with deep learning)

**RNN = pipeline for information**



In every step some information goes in and some information goes out.

Technically: A "`for`" loop applying the same function $A$ on input vectors $x_i$

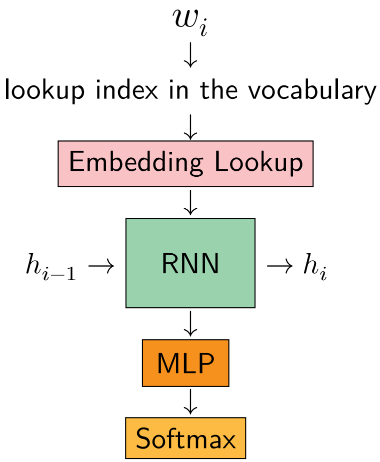



At training time unrolled in time: technically just a very deep network

Image on the right: Chris Olah. Understanding LSTM Networks. A blog post: `http://colah.github.io/posts/2015-08-Understanding-LSTMs`

# Sequence Labeling

- Assign a label to each word in a sentence.
- Tasks formulated as sequence labeling:
  - Part-of-Speech Tagging
  - Named Entity Recognition
  - Filling missing punctuation




---

MLP = Multilayer perceptron
$n \times$ layer: $\sigma\left(Wx + b\right)$

Softmax for $K$ classes with logits $\mathbf{z} = (z_1, \ldots, z_K)$:

$$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

# Detour: Why is softmax a good choice

Output layer with softmax (with parameters $W$, $b$) — gets categorical distribution:

$$P_y = \text{softmax}(\mathbf{x}) = \Pr(y \mid \mathbf{x}) = \frac{\exp\{\mathbf{x}^\top W\} + b}{\sum \exp\{\mathbf{x}^\top W\} + b}$$

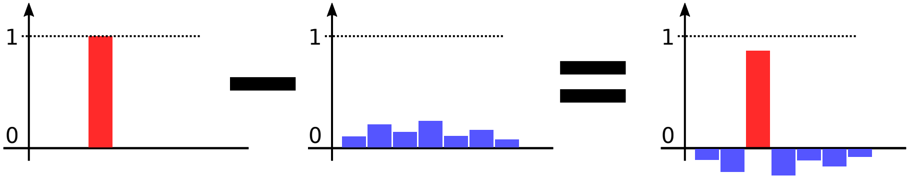
Network error = cross-entropy between estimated distribution and one-hot ground-truth distribution $T = \mathbf{1}(y^*) = (0, 0, \dots, 1, 0, \dots, 0)$:

$$\begin{aligned} L(P_y, y^*) = H(P, T) &= -\mathbb{E}_{i \sim T} \log P(i) \\ &= -\sum_i T(i) \log P(i) \\ &= -\log P(y^*) \end{aligned}$$

# Derivative of Cross-Entropy

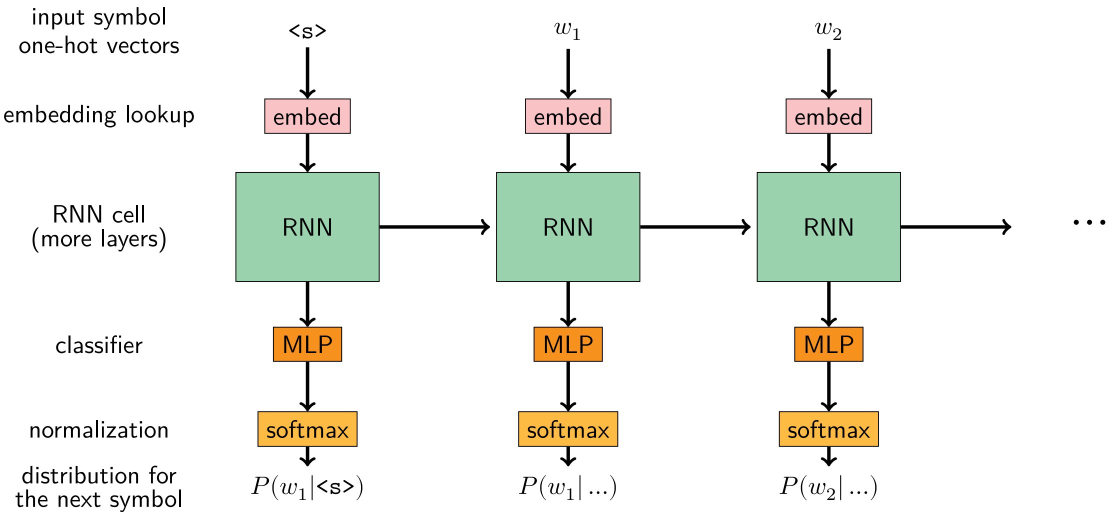
Let $l = \mathbf{x}^\top W + b$, $l_{y^*}$ corresponds to the correct one.

$$\begin{aligned}\frac{\partial L(P_y, y^*)}{\partial l} &= -\frac{\partial}{\partial l} \log \frac{\exp l_{y^*}}{\sum_j \exp l_j} = -\frac{\partial}{\partial l}\left(l_{y^*} - \log \sum \exp l\right) \\ &= \mathbf{1}_{y^*} + \frac{\partial}{\partial l} - \log \sum \exp l = \mathbf{1}_{y^*} - \frac{\sum \mathbf{1}_{y^*} \exp l}{\sum \exp l} = \\ &= \mathbf{1}_{y^*} - P_y(y^*)\end{aligned}$$

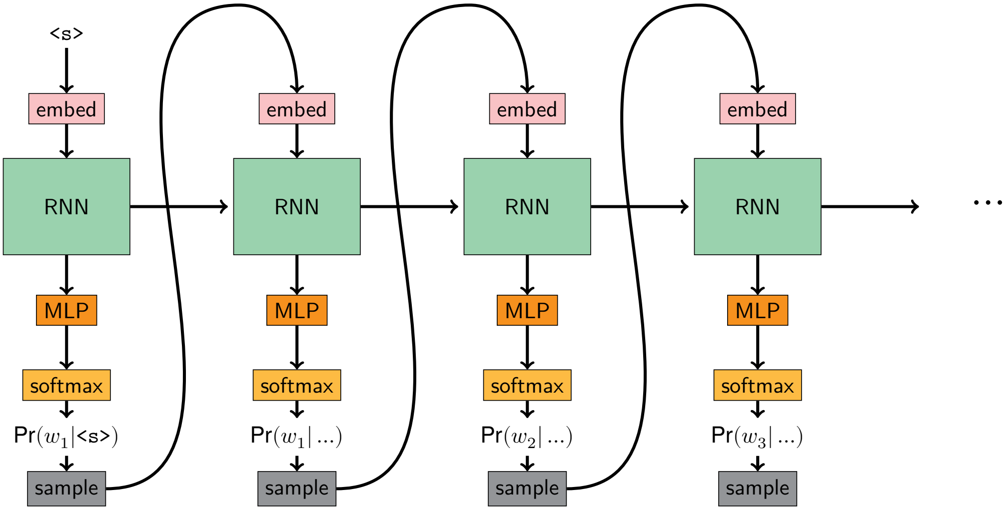



Interpretation: Reinforce the correct logit, suppress the rest.

# Language Model as Sequence Labeling



# Sampling from a Language Model



# Sampling from a Language Model: Pseudocode

```
last_w = "<s>"
state = initial_state
while last_w != "</s>":
    last_w_embeding = target_embeddings[last_w]
    state = rnn(state, last_w_embeding)
    logits = output_projection(state)
    last_w = vocabulary[np.random.multimial(1, logits)]
    yield last_w
```

# Training

Training objective: negative-log likelihood:

$$\mathsf{NLL} = -\sum_{i}^{n} \log \mathsf{Pr}\left(w_i | w_{i-1}, \ldots, w_1\right)$$

I.e., maximize probability of the correct word.

- Cross-entropy between the predicted distribution and one-hot "true" distribution
- Error from word is backpropagated into the rest of network unrolled in time
- Prone to exposure bias: during trainining only well-behaved sequences, it can break when we sample something weird at inference time

# Generating from a Language Model

Neural Machine Translation is

a new technology developed by a team at the University

a technology that uses neural networks and machine learning to

a powerful tool for understanding the spoken language.

(Example from GPT-2, a Tranformer-based English language model, screenshot from `https://transformer.huggingface.co/doc/gpt2-large`)

**Cool, but where is the source language?**

# Conditioning the Language Model & Attention

# Conditional Language Model

Formally it is simple, condition distribution of

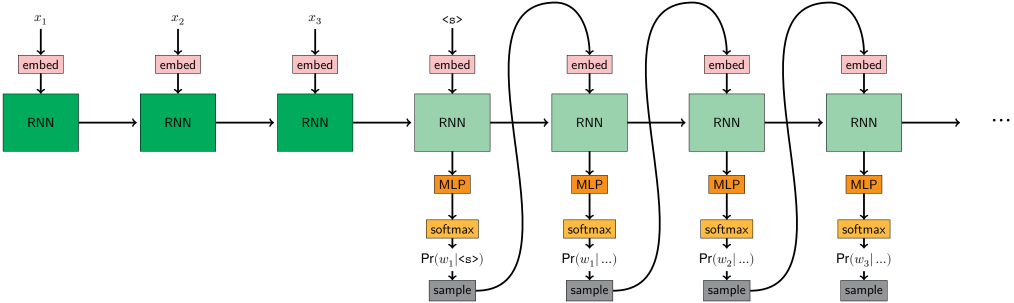
- target sequence $\mathbf{y} = (y_1, \ldots, y_{T_y})$ on
- source sequence $\mathbf{x} = (x_1, \ldots, x_{T_x})$

$$\Pr(y_1, \ldots, y_n | \mathbf{x}) = \prod_i^n \Pr(y_i | y_{i-1}, \ldots, y_1, \mathbf{x})$$

We need an *encoder* to get a representation of $\mathbf{x}$!

**What about just continuing an RNN…**

# Sequence-to-Sequence Model



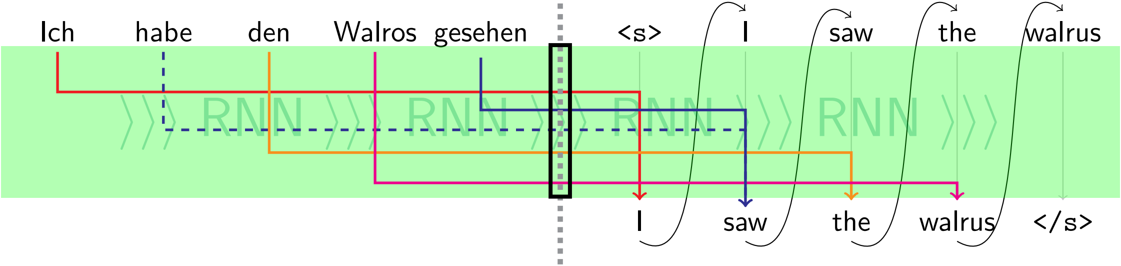

- The interface between encoder and decoder is a single vector regardless the sentence length.

# Seq2Seq: Pseudocode

```
state = np.zeros(rnn_size)
for w in input_words:
    input_embedding = source_embeddings[w]
    state = enc_cell(encoder_state, input_embedding)

last_w = "<s>"
while last_w != "</s>":
    last_w_embeding = target_embeddings[last_w]
    state = dec_cell(state, last_w_embeding)
    logits = output_projection(state)
    last_w = vocabulary[np.argmax(logits)]
    yield last_w
```

# Vanila Seq2Seq: Information Bottleneck




Bottleneck all information needs to run through.
A single vector must represent the entire source sentence.

**Main weakness and the reason for introducing the attention.**

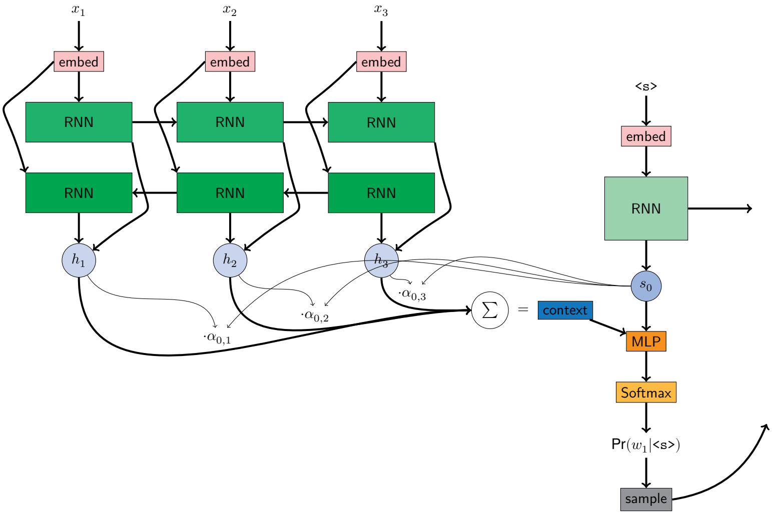
# The Attention Model

- Motivation: It would be nice to have variable length input representation
- RNN returns one state per word ...
- ...what if we were able to get only information from words we need to generate a word.

**Attention** = probabilistic retrieval of encoder states for estimating probability of target words.

**Query** = hidden states of the decoder
**Values** = encoder hidden states

# Sequence-to-Sequence Model With Attention




- Encoder = bidirectional RNN states $h_i$ ≈ retrieved values
- Decoder step starts as usual state $s_0$ ≈ retrieval query
- Decoder state $s_0$ used to compute distribution the over encoder states
- Weighted average of encoder states = context vector
- Decoder state & context concatenated MLP + Softmax predicts next word

**Inputs:**

decoder state $s_i$

encoder states $h_j = \left[\overrightarrow{h_j}; \overleftarrow{h_j}\right] \quad \forall i = 1 \dots T_x$

**Attention energies:**

$$e_{ij} = v_a^\top \tanh\left(W_a s_{i-1} + U_a h_j + b_a\right)$$

**Attention distribution:**

$$\alpha_{ij} = \frac{\exp\left(e_{ij}\right)}{\sum_{k=1}^{T_x} \exp\left(e_{ik}\right)}$$

**Context vector:**

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$

# Attention Model in Equations (2)

**Output projection:**

$$t_i = \mathsf{MLP}\left(s_{i-1} \oplus v_{y_{i-1}} \oplus c_i\right)$$

...attention is mixed with the hidden state
(different in differnt models)

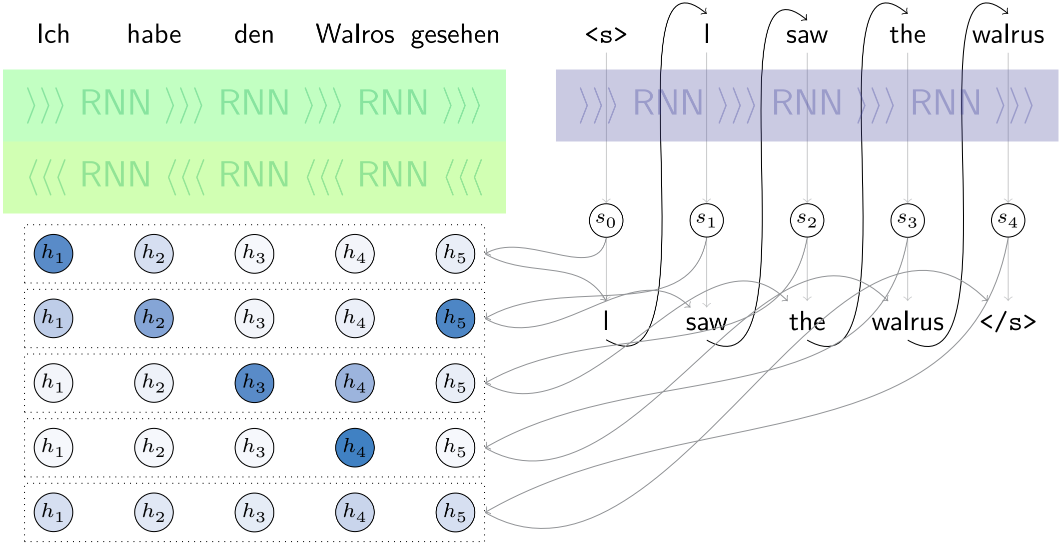
**Output distribution:**

$$p\left(y_i = k | s_i, y_{i-1}, c_i\right) \propto \mathsf{exp}\left(W_o t_i + b_k\right)_k$$

(usual trick: use transposed embeddings as $W_o$)

- Different version of attentive decoders exist
- Alternative: keep the context vector as input for the next step
- Multilayer RNNs: attention between/after layers

# Workings of the Attentive Seq2Seq model



# Seq2Seq with attention: Pseudocode (1)

```
state = np.zeros(emb_size)
fw_states = []
for w in input_words:
    input_embedding = source_embeddings[w]
    state, _ = fw_enc_cell(encoder_state, input_embedding)
    fw_states.append(state)

bw_states = []
state = np.zeros(emb_size)
for w in reversed(input_words):
    input_embedding = source_embeddings[w]
    state, _ = bw_enc_cell(encoder_state, input_embedding)
    bw_states.append(state)

enc_states = [np.concatenate(fw, bw) for fw, bw in zip(fw_states,
    reversed(bw_states))]
```

# Seq2Seq with attention: Pseudocode (2)

```
last_w = "<s>"
while last_w != "</s>":
    last_w_embeding = target_embeddings[last_w]
    state = dec_cell(state, last_w_embeding)
    alphas = attention(state, enc_states)
    context = sum(a * state for a, state in zip(alphas, enc_states))
    logits = output_projection(np.concatenate(state, context, last_w_embeding))
    last_w = np.argmax(logits)
    yield last_w
```

# Attention Visualization (1)

Image source: **?**, Fig. 3

# Attention Visualization (2)

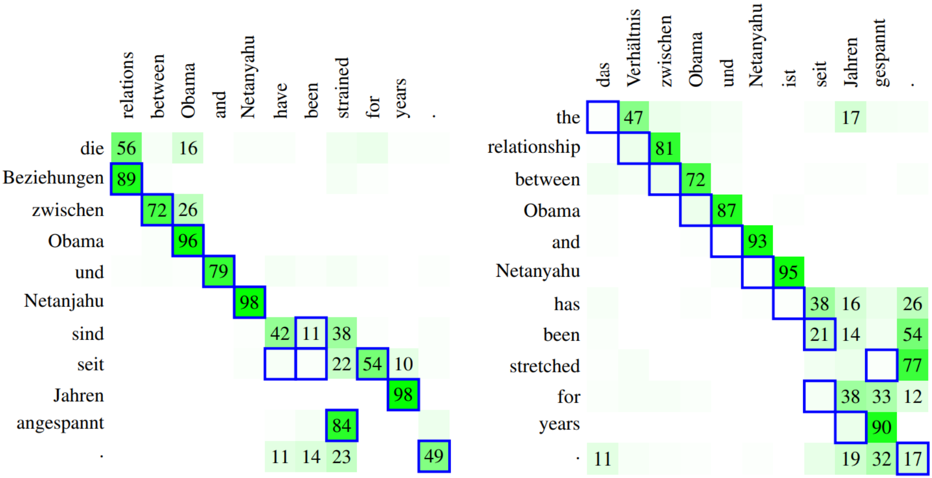



Image source: ?, Fig. 8

# Attention vs. Alignment

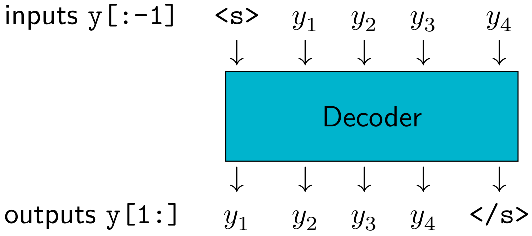
Differences between attention model and word alignment used for phrase table generation:

| **attention (NMT)** | **alignment (SMT)** |
|---|---|
| probabilistic | discrete |
| declarative | imperative |
| LM generates | LM discriminates |

Optimize negative log-likelihood of parallel data, backpropagation does the rest.

If you choose a right optimizer, learning rate, model hyper-parameters, prepare data, do back-translation, monolingual pre-training ...

Confusion: decoder inputs vs. output



# Inference

# Getting output

- Encoder-decoder is a conditional language model
- For a pair **x** and **y**, we can compute:

$$\Pr(\mathbf{y}|\mathbf{x}) = \prod_{i=1}^{T_y} \Pr(y_i|\mathbf{y}_{:i}, \mathbf{x})$$

- When decoding we want to get

$$\mathbf{y}^* = \operatorname*{argmax}_{\mathbf{y}'} \Pr(\mathbf{y}'|x)$$

☠ **Enumerating all $y^{'}$s is computationally intractable** ☠

# Greedy Decoding

In each step, take the maximum probable word.

$$y_i^* = \underset{y_i}{\text{argmax}}\,\text{Pr}\left(y_i | y_{i-1}^*, \ldots, \texttt{<s>}\right)$$

```
last_w = "<s>"
state = initial_state
while last_w != "</s>":
    last_w_embeding = target_embeddings[last_w]
    state = dec_cell(state, last_w_embeding)
    logits = output_projection(state)
    last_w = vocabulary[np.argmax(logits)]
    yield last_w
```

# What if...

This is a

platypus random end . </s>

25% 30% each

rather good sentence . </s>

24% 60% each

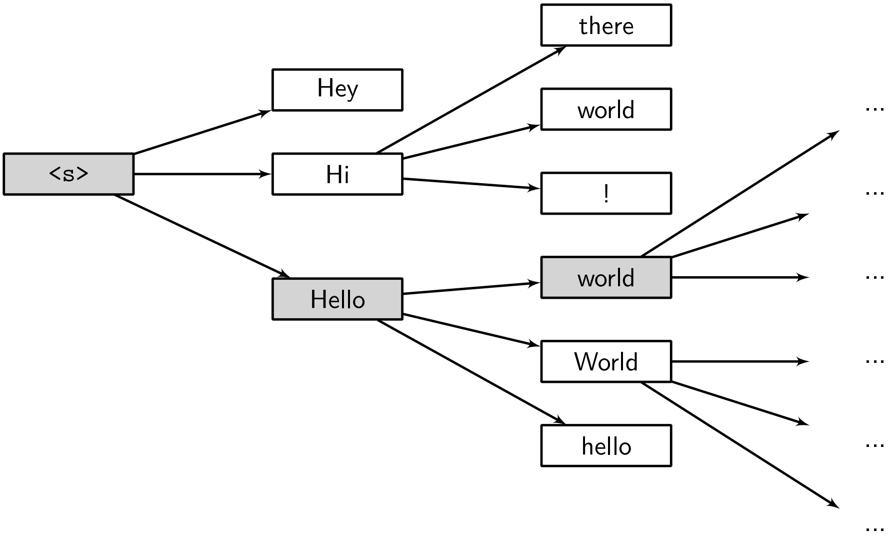
⚠ Greedy decoding can easily miss the best option. ⚠

# Beam Search

Keep a small $k$ of hypothesis (typically 4–20).

1. Begin with a single empty hypothesis in the beam.
2. In each time step:
   2.1 Extend all hypotheses in the beam by all (or the most probable) from the output distribution (we call these *candidate hypotheses*)
   2.2 Score the candidate hypotheses
   2.3 Keep only $k$ best of them.
3. Finish if all $k$-best hypotheses end with `</s>`
4. Sort the hypotheses by their score and output the best one.

# Beam Search: Example



# Beam Search: Pseudocode

```
beam = [(["<s>"], initial_state, 1.0)]
while any(hyp[-1] != "</s>" for hyp, _, _ in beam):
    candidates = []

    for hyp, state, score in beam:
    distribution, new_state = decoder_step(hyp[-1], state, encoder_states)
        for i, prob in enumerate(distribution):
            candidates.append(hyp + [vocabulary[i]], new_state, score * prob)

    beam = take_best(k, candidates)
```

# Implementation issues

- Multiplying of too many small numbers $\rightarrow$ float underflow
  need to compute in log domain and add logarithms
- Sentences can have different lengths

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| This | is | a | good | long | sentence | . | </s> | |
| 0.7 | $\times$ 0.6 | $\times$ 0.9 | $\times$ 0.1 | $\times$ 0.4 | $\times$ 0.4 | $\times$ 0.8 | $\times$ 0.9 | **= 0.004** |

| | | |
|---|---|---|
| This | </s> | |
| 0.7 | $\times$ 0.01 | **= 0.007** |

  $\Rightarrow$ use the geometric mean instead of probabilities directly

- Sorting candidates is expensive, assomptotically $|V| \log |V|$:
  $k$-best can be found in linear time, $|V| \sim 10^4 - 10^5$

# Final Remarks

# Brief history of the architectures

- **2013** First encoder-decoder model (?)
- **2014** First really usable encoder-decoder model (?)
- **2014/2015** Added attention (crucial innovation in NLP) (?)
- **2016/2017** WMT winners used RNN-based neural systems (?)
- **2017** Transformers invented (outperformed RNN) (?)

The development of achitectures still goes on...
Document context, non-autoregressive models, multilingual models, ...

# Summary

- Encoder-decoder architecture = major paradigm in MT
- Encoder-decoder architecture = conditional language model
- Attention = way of conditioning the decoder on the encoder
- Attention = probabilistic vector retrieval
- We model probability, but need heuristics to get a good sentence from the model

# References I