Today: LSTMs, GRUs (vaguely)
Bidirectional RNNs
Deep RNNs
code examples: sentiment analysis for IMDB dataset

Problem (Vanishing gradients)

Simple RNN works for short sequences, but the hidden state acts as a primitive memory, so run into issues of gradient vanishing/exploding for longer sequences. This is because it attempts to retain all the information on the sequence prefix seen before.

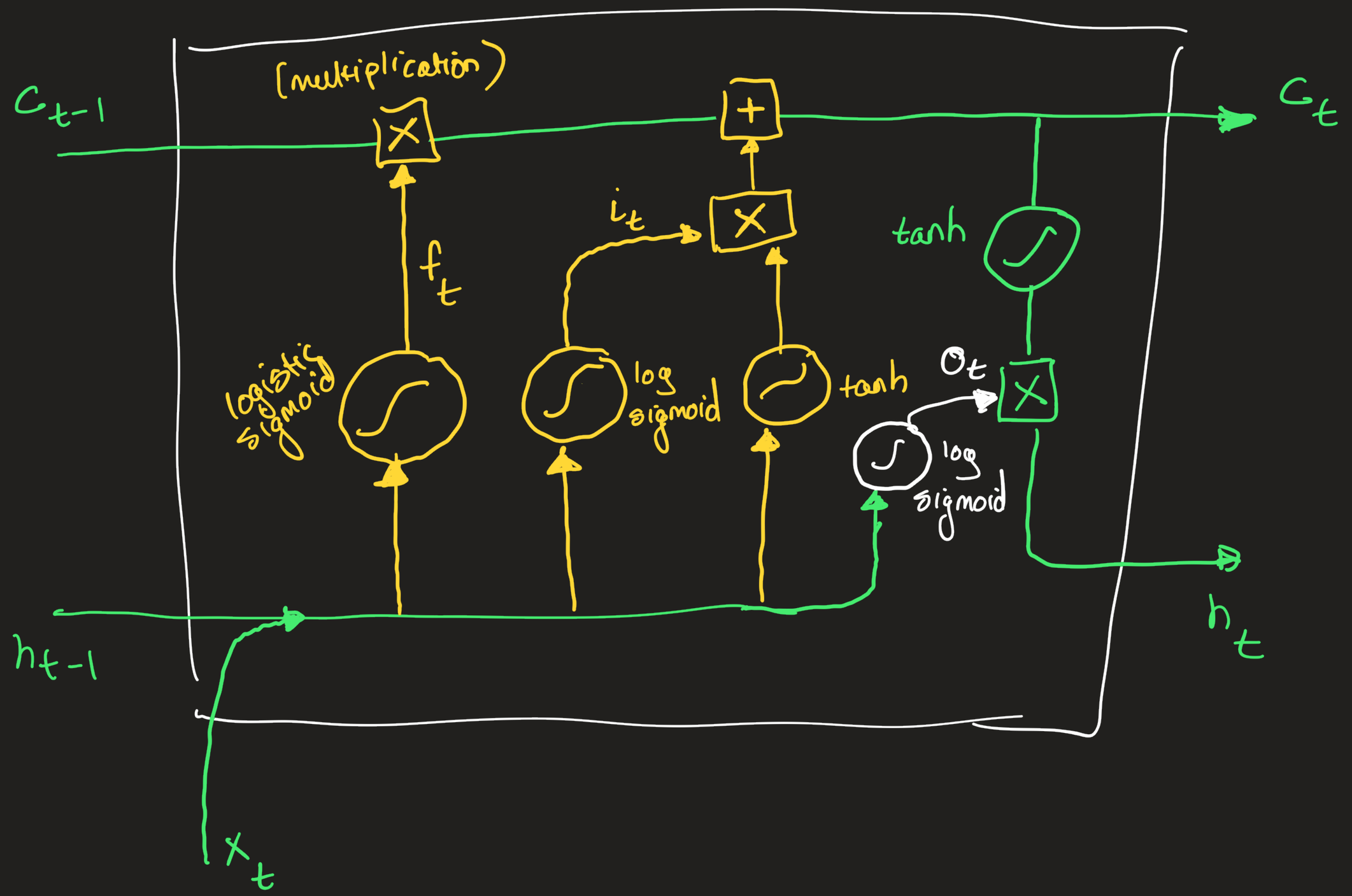
Resolution (in practice, by consensus, the most popular architectures)

LSTM (Long-Short Term Memories; 1997)
GRUs (Gated Recurrent Units; 2014)

focus on LSTMs: basic intuition is to selectively choose what you will remember or forget.

Introduce:
- cell state to augment our memory in addition to the hidden state, c_t (in cell state)
- forget gate — determine what we forget at each time, f_t
- input gate — determine what we remember in our cell state, i_t

Flow diagram for LSTM



$$i_t = \text{sigmoid}\left(\omega_{hi} h_{t-1} + \omega_{xi} x_t\right)$$

$$f_t = \text{sigmoid}\left(\omega_{hf} h_{t-1} + \omega_{xf} x_t\right)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tanh\left(\omega_{hc} h_{t-1} + \omega_{xc} x_t\right)$$

$$o_t = \text{sigmoid}\left(\omega_{ho} h_{t-1} + \omega_{xo} x_t\right)$$

$$h_t = o_t \odot \tanh\left(\omega_{ch} c_t\right)$$

Learn via backprop all the parameters for these gates

LSTMs are very popular. GRUs were introduced to simplify the LSTM architecture:

- have only a hidden state, no cell state
- only two gates: reset and update gates

Faster to train, but LSTMs are more popular.

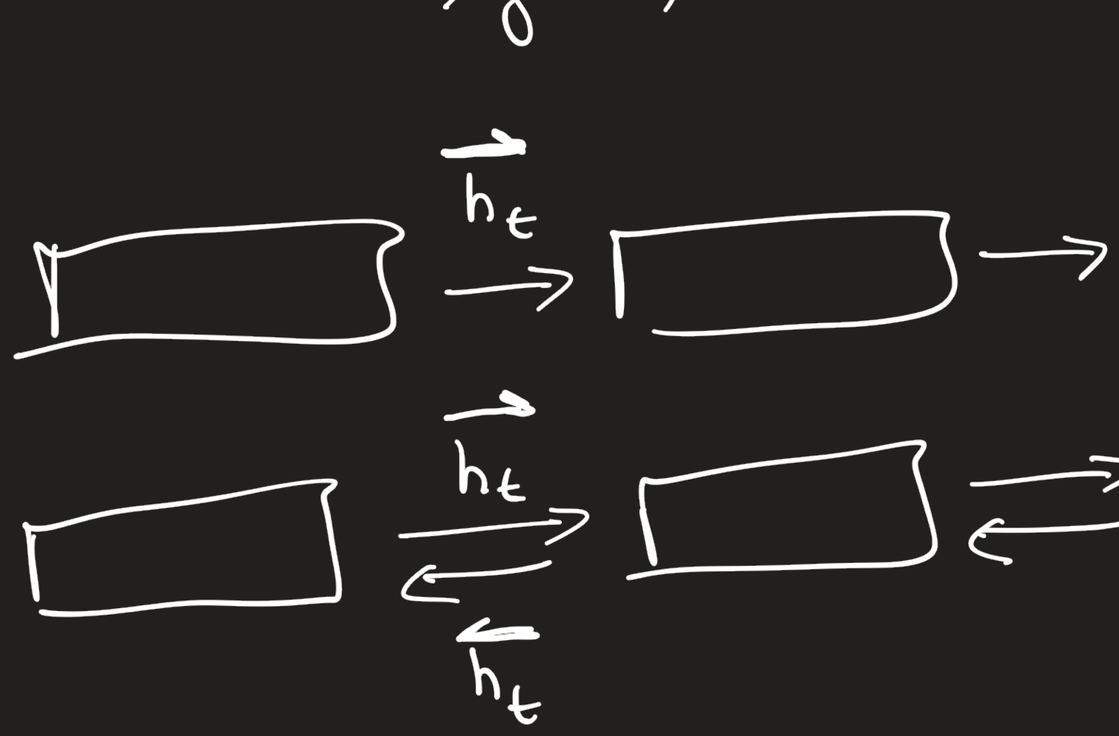
Bidirectional RNN

Very → how are you ? .

↑

well, good, etc.

The future can provide useful information as well.



standard RNN

bidirectional RNN

Bidirectional RNNs predict the current output $\hat{y}_t$ using the past and the future

$$\overrightarrow{h}_t = f(x_t, h_{t-1}, \overrightarrow{w}_h)$$

$$\overleftarrow{h}_t = f(x_t, h_{t+1}, \overleftarrow{w}_h)$$

$$h_t = \begin{bmatrix} \overrightarrow{h}_t \\ \overleftarrow{h}_t \end{bmatrix}$$

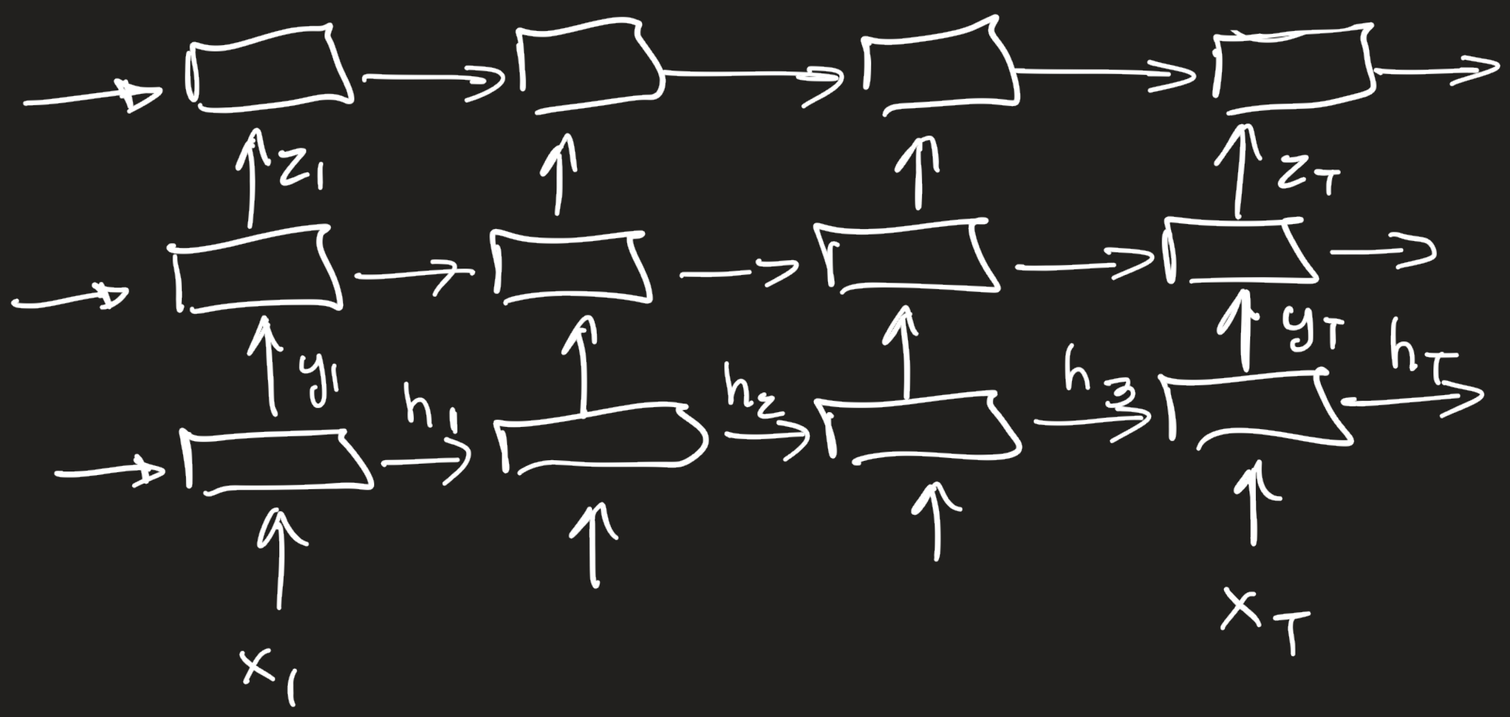
$$\hat{y}_t = g(h_t, w_o)$$

Useful for non-causal task where context is useful, e.g. NLP tagging task (NER, POS, etc.), change point detection in time series, translation

BTT needs to be modified to account for future dependencies

Deep RNNs

(typically used when "LSTMs" or "RNNs" are mentioned)



- train layer-wise, by computing all outputs from layer l so have inputs for layer $l+1$
- modify BTT to account for dependencies between layers

Sequence-to-Sequence Modeling

(2014 used for very good machine translation, popularized deep LSTM seq-to-seq model)

Sutskever et al. "Sequence to Sequence Learning w/ NNs"

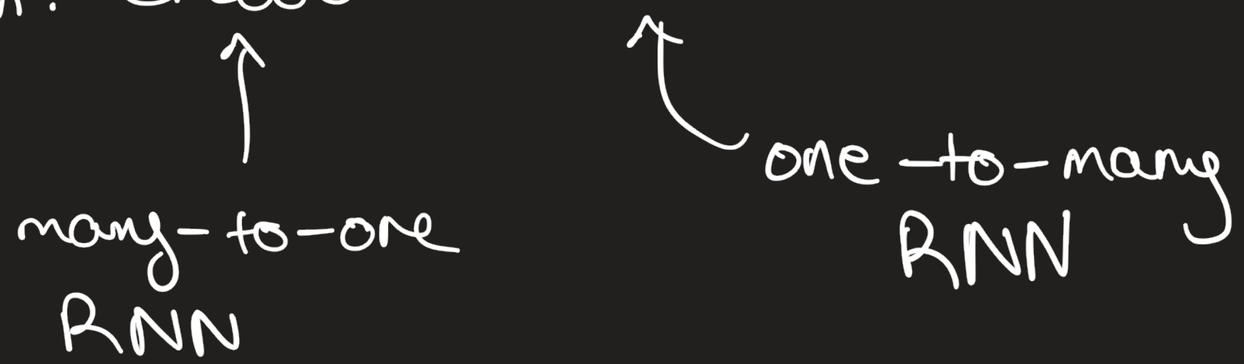
Goal: given input sequence of tokens $(x_1, \ldots, x_T)$

predict output sequence of tokens $(y_1, \ldots, y_{T'})$

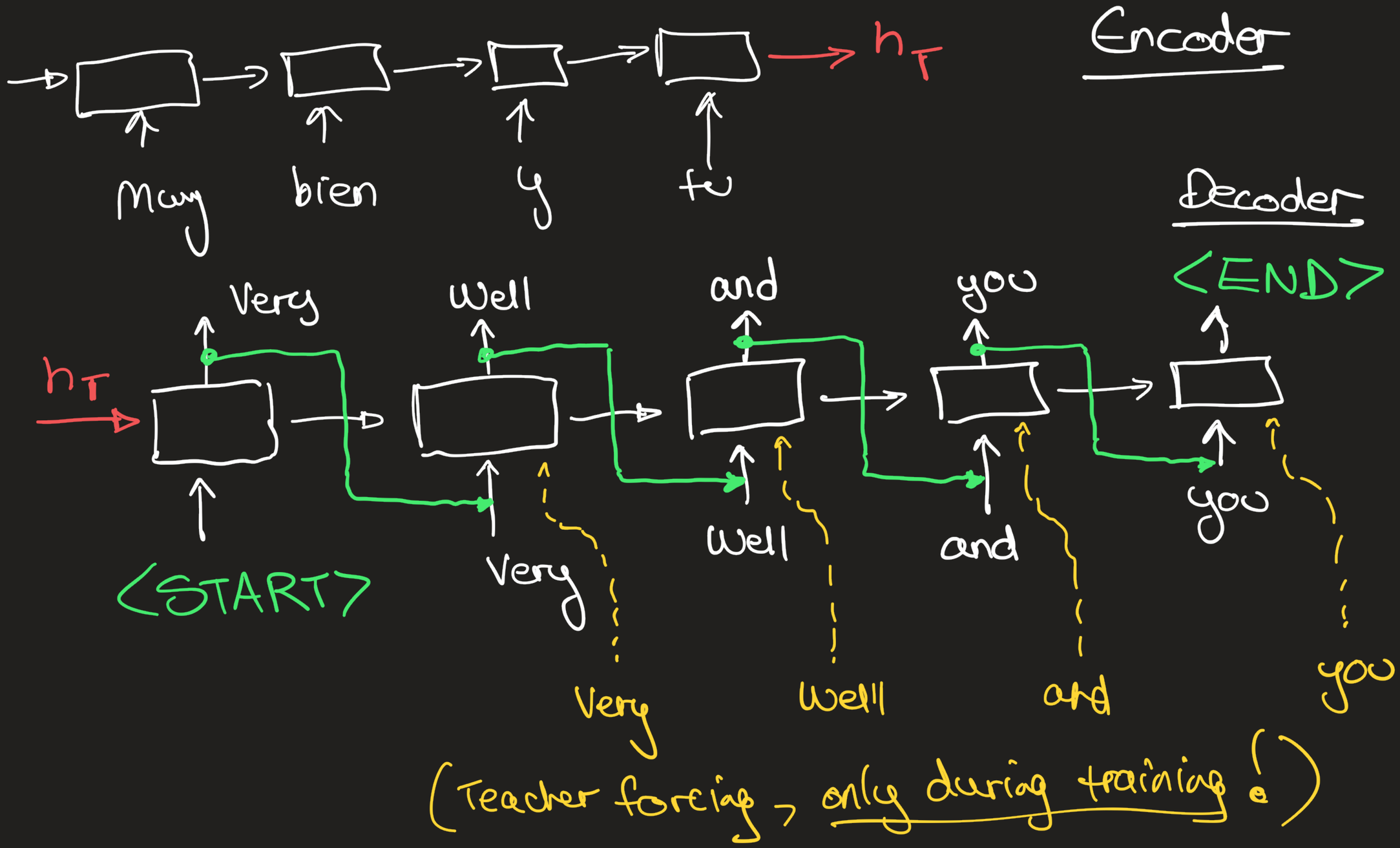
Canonical example: Machine Translation

Issue: T may be different than T' so the many-to-many design paradigm for RNNs may not fit

Soln: encoder-decoder architecture



Sequence-to-Sequence architecture

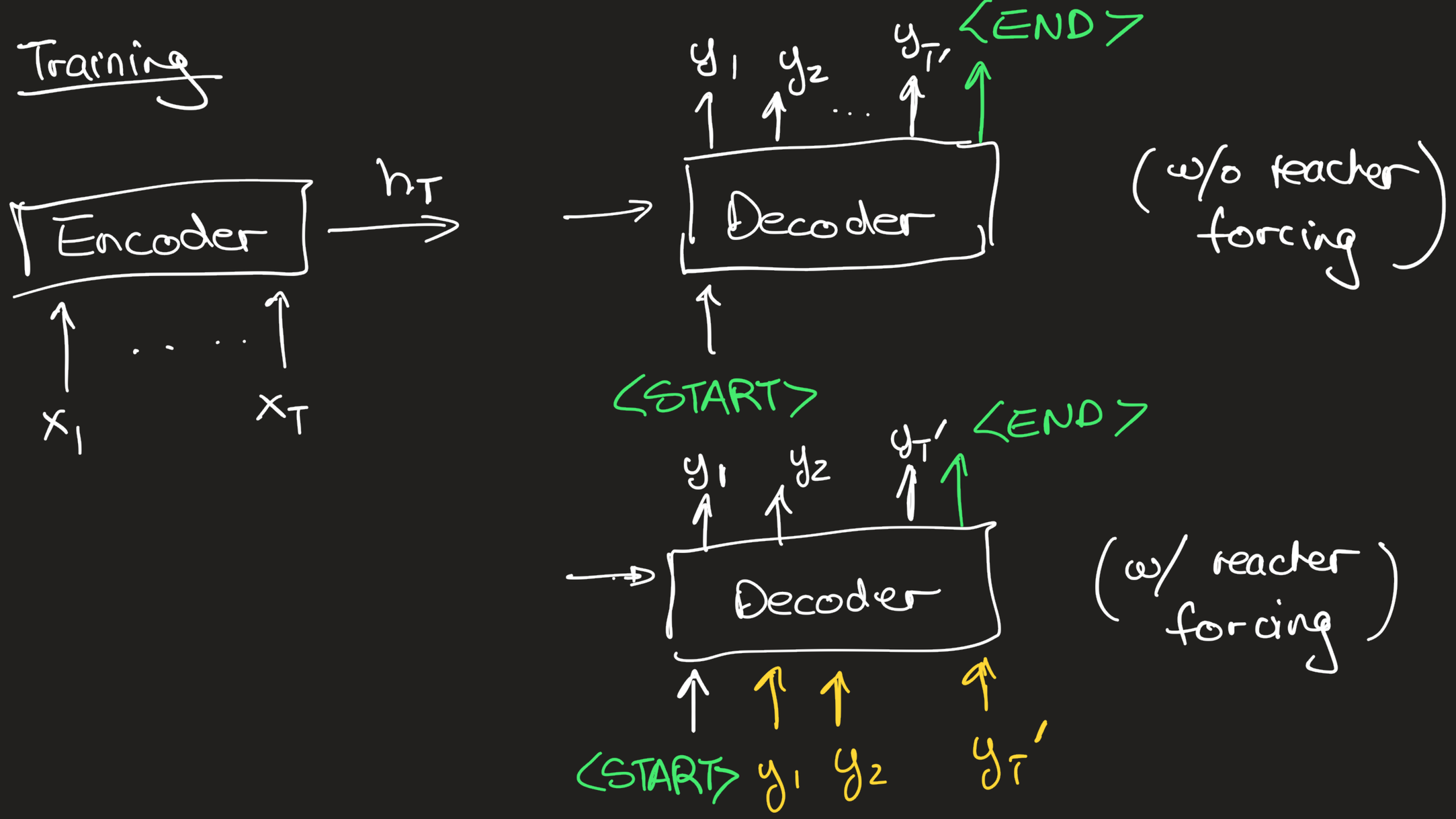


Training:

- train the encoder & decoder simultaneously (using BTT) so that given the input sequence $(x_1, \ldots, x_T)$ and output sequence $(y_1, \ldots, y_{T'})$:
  - encoder learns an informative h_T for decoding
  - decoder learns how to, given h_T and a special <START> token, generate the correct output followed by an <END> token.

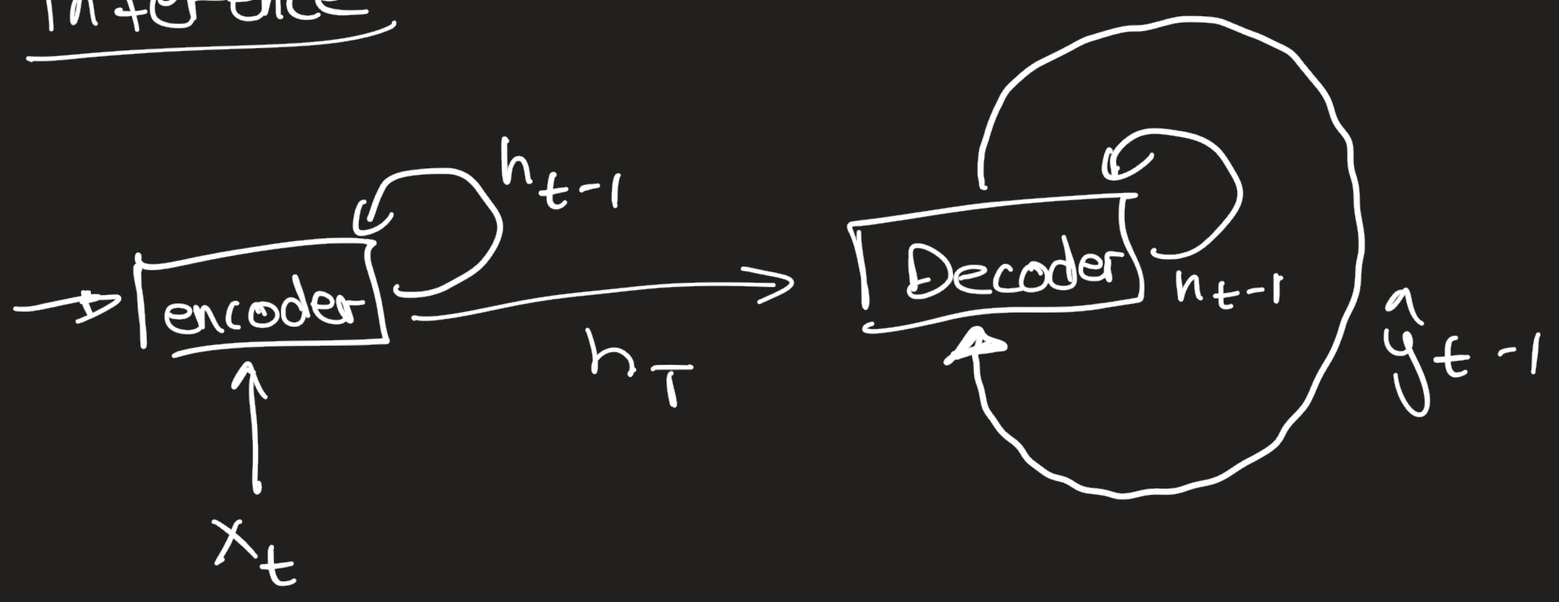
- Since the output $\hat{y}_t$ depends on $\hat{y}_{t-1}$ we can have very slow training if $\hat{y}_t$ is inaccurate at the start of the sequence, because the errors propagate. Instead, use Teacher forcing: use y_{t-1} (the ground truth) as the input to predict $\hat{y}_t$ at training time

Training



As our train inputs: $(x_1, \cdots, x_T)$ and $(\langle START \rangle, y_1, \cdots, y_{T'})$
and as training outputs: $(y_1, \cdots, y_{T'}, \langle END \rangle)$

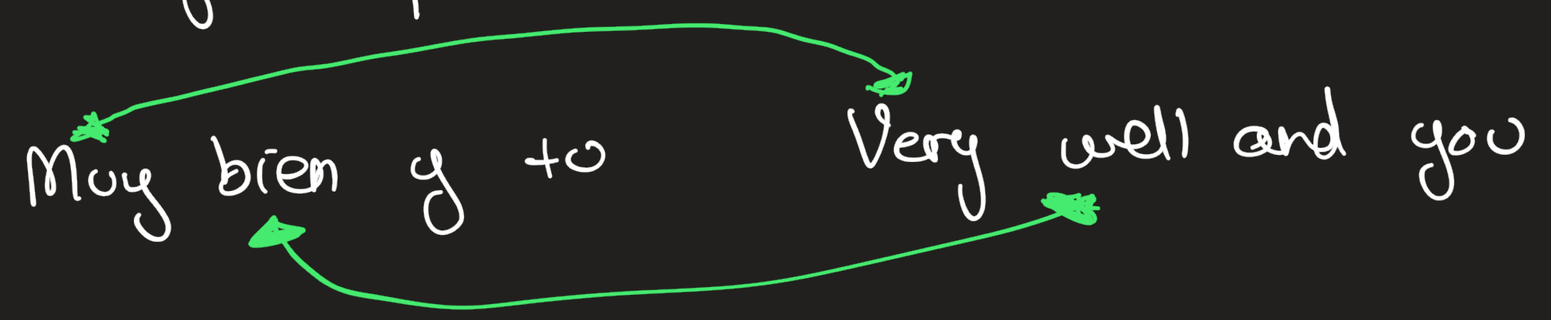
Inference

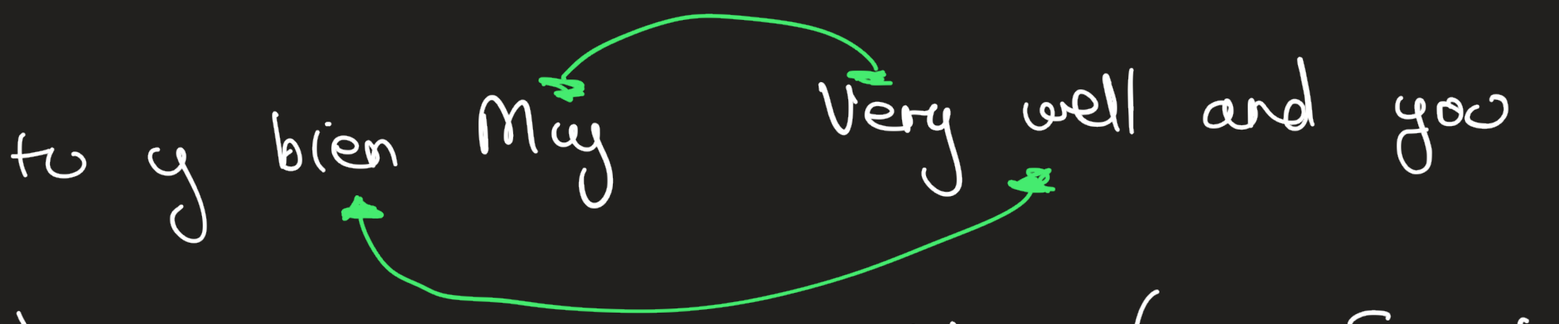


- Given input sequence $(x_1, \cdots, x_T)$ use encoder to obtain h_T
- initialize the decoder w/ hidden state h_T and feed it the <START> token
- At each time, feed $\hat{y}_{t-1}$ into the decoder to predict the next token and take $\hat{y}_t$ to be the most likely token

Details

- use deep LSTMs (GRUs) in the encoder and decoder
- reverse the order of the source sequence to introduce shorter length dependencies w/ the target sequence





- use "beam decoding" in practice (see Sutskever et al.)