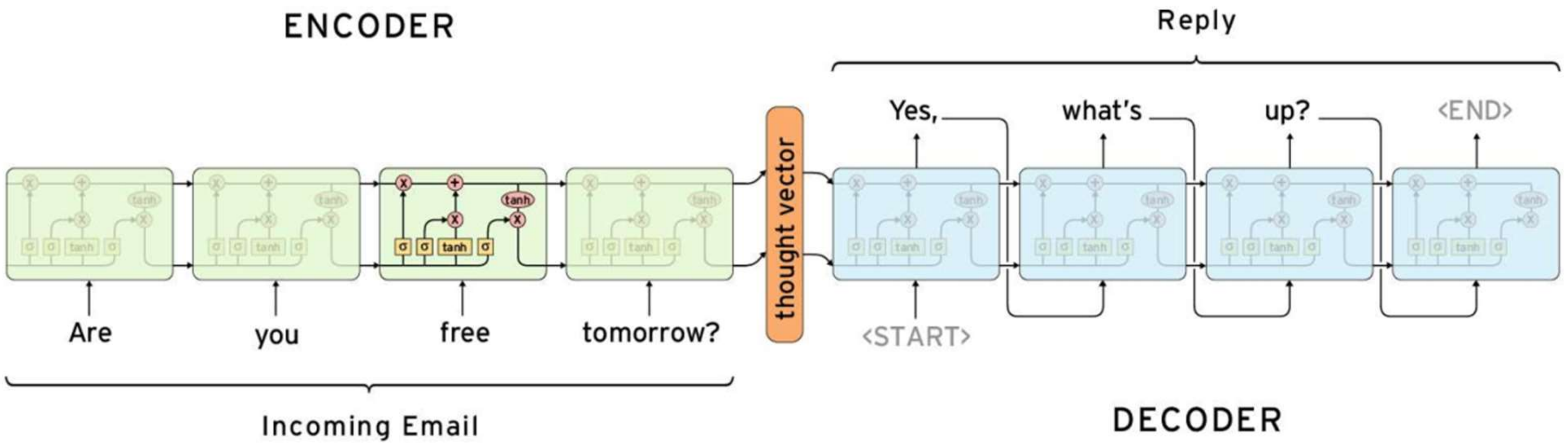
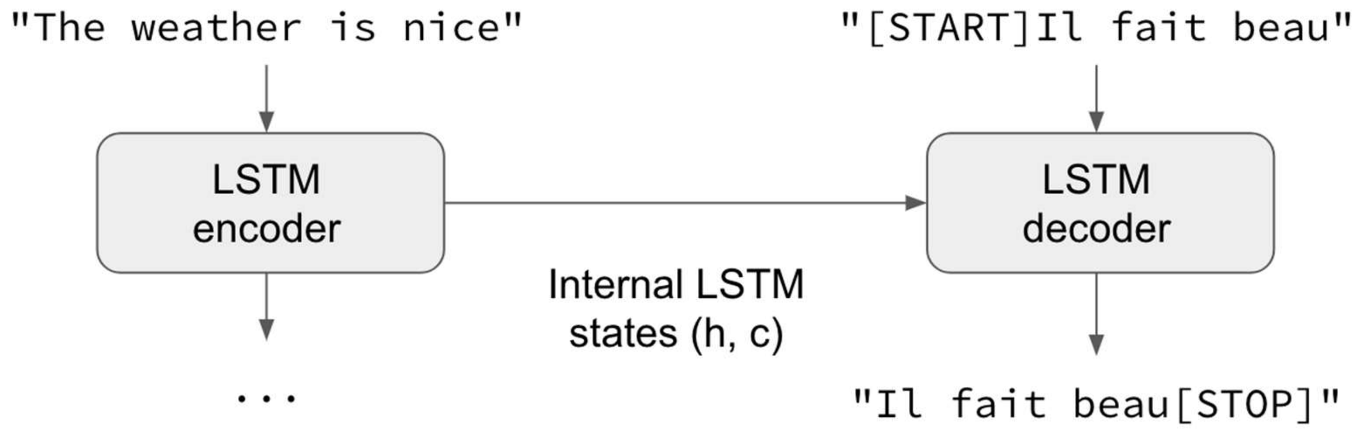
Week 7 – seq2seq model and subword tokenization

EGCO467 Natural Language and Speech Processing

sequence to sequence model (seq2seq)



seq2seq


"The weather is nice"
LSTM encoder
...
Internal LSTM states (h, c)
"[START]Il fait beau"
LSTM decoder
"Il fait beau[STOP]"

seq2seq

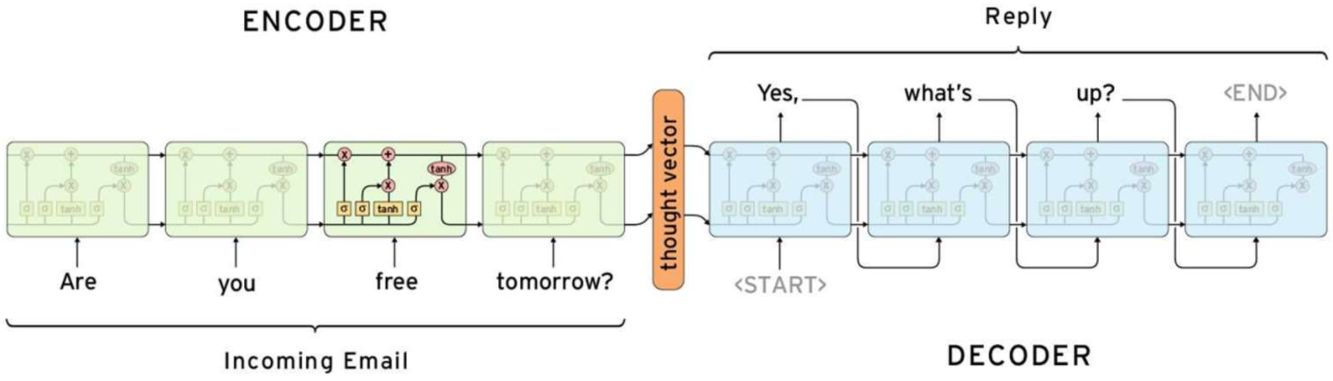
- machine translation
- POS/NER
- text summarization
- chatbot (free response)

data for seq2seq

- sentence pairs in source language and target language
- sentence and POS/NER tags
- conversation history (query-reply pairs) for chatbot
- long text and summary

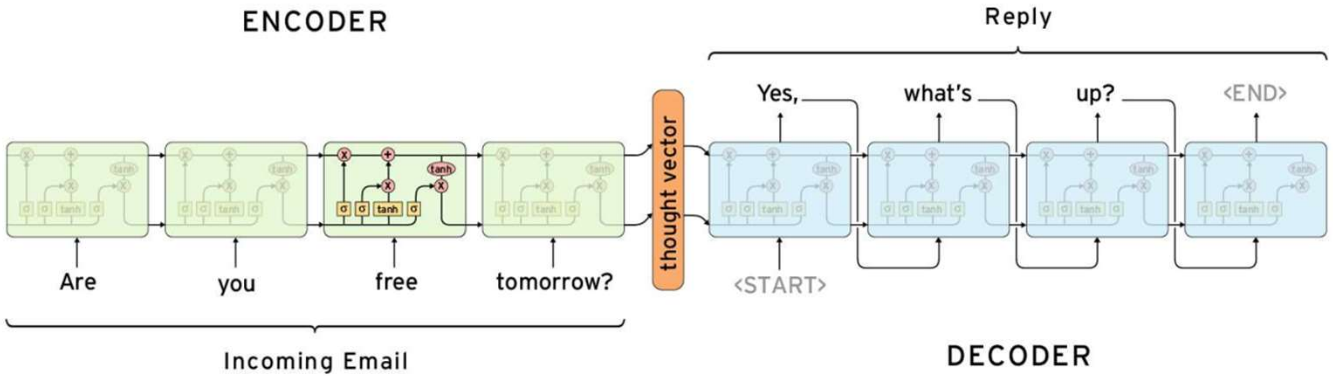
inference for seq2seq

- Run the encoder with seq1, save the final state of the encoder
- Initialize the decoder's initial state with the final state of the encoder
- Feed the decoder with the start sequence token
- Have the decoder predict the next token in the target (seq2)
- The predicted token becomes the input in the next timestep
- Repeat until decoder predict end of sequence tag or reached maximum
- sequence length



training seq2seq

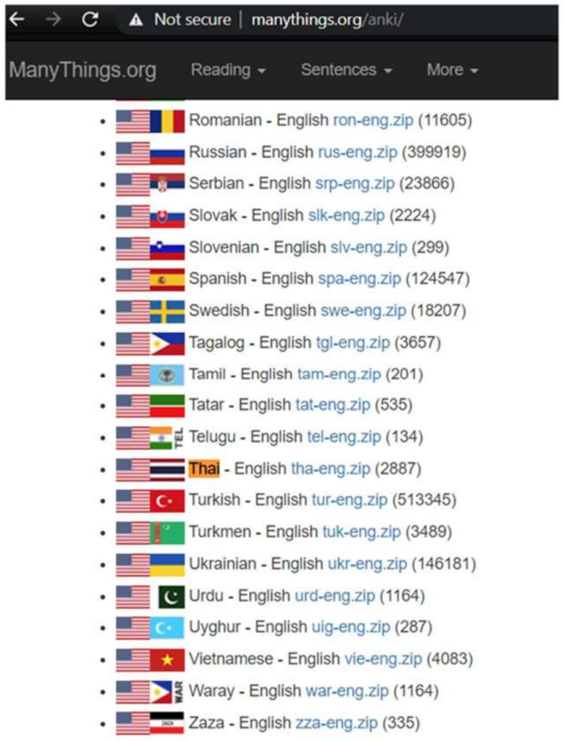
- encoder input: seq1
- decoder input: the corresponding seq2
- decoder target: seq2 shifted to the right one timestep
- (there is no target for encoder)
- the job of decoder is to predict the next token of seq2, given the final
- state of encoder and the current token of seq



Example

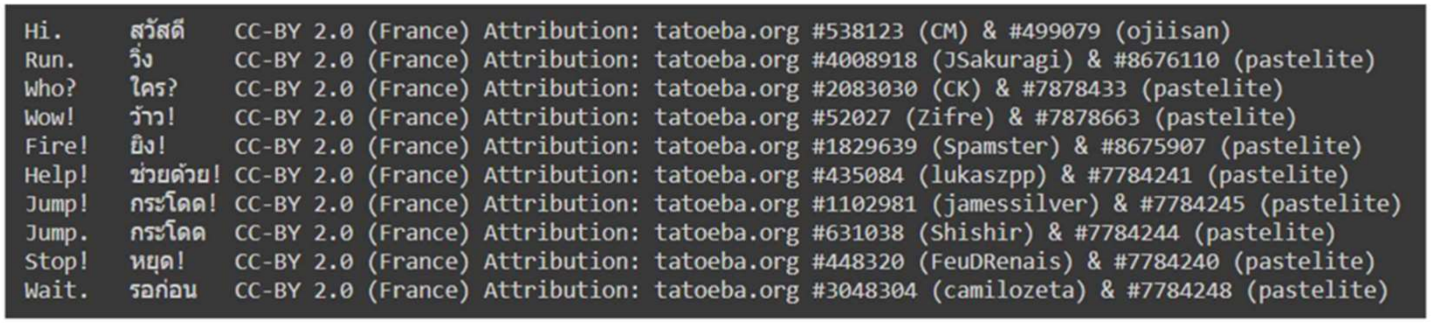
data

- http://www.manythings.org/anki/



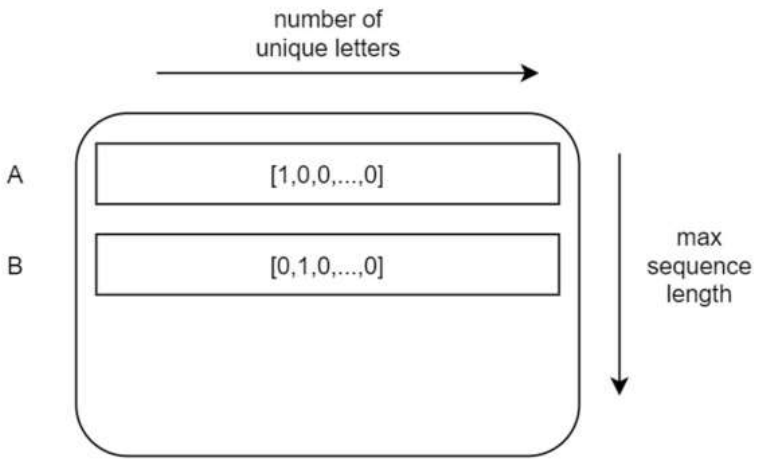
- Romanian - English ron-eng.zip (11605)
- Russian - English rus-eng.zip (399919)
- Serbian - English srp-eng.zip (23866)
- Slovak - English slk-eng.zip (2224)
- Slovenian - English slv-eng.zip (299)
- Spanish - English spa-eng.zip (124547)
- Swedish - English swe-eng.zip (18207)
- Tagalog - English tgl-eng.zip (3657)
- Tamil - English tam-eng.zip (201)
- Tatar - English tat-eng.zip (535)
- Telugu - English tel-eng.zip (134)
- Thai - English tha-eng.zip (2887)
- Turkish - English tur-eng.zip (513345)
- Turkmen - English tuk-eng.zip (3489)
- Ukrainian - English ukr-eng.zip (146181)
- Urdu - English urd-eng.zip (1164)
- Uyghur - English uig-eng.zip (287)
- Vietnamese - English vie-eng.zip (4083)
- Waray - English war-eng.zip (1164)
- Zaza - English zza-eng.zip (335)

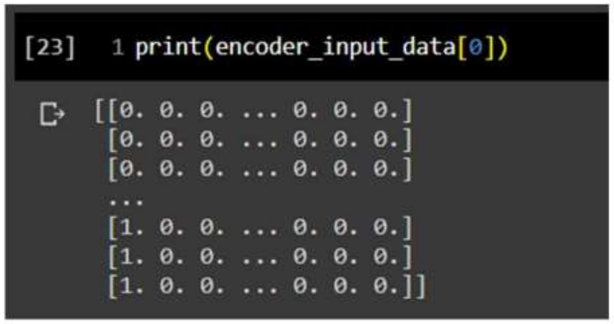
data



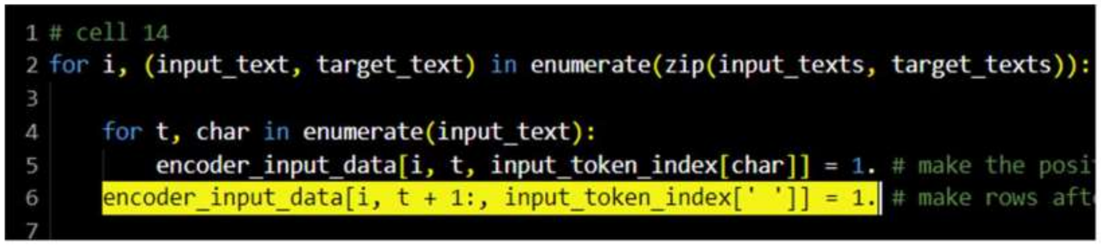
```
Hi.	สวัสดี	CC-BY 2.0 (France) Attribution: tatoeba.org #538123 (CM) & #499079 (ojiisan)
Run.	วิ่ง	CC-BY 2.0 (France) Attribution: tatoeba.org #4008918 (JSakuragi) & #8676110 (pastelite)
Who?	ใคร?	CC-BY 2.0 (France) Attribution: tatoeba.org #2083030 (CK) & #7878433 (pastelite)
Wow!	ว้าว!	CC-BY 2.0 (France) Attribution: tatoeba.org #52027 (Zifre) & #7878663 (pastelite)
Fire!	ยิง!	CC-BY 2.0 (France) Attribution: tatoeba.org #1829639 (Spamster) & #8675907 (pastelite)
Help!	ช่วยด้วย!	CC-BY 2.0 (France) Attribution: tatoeba.org #435084 (lukaszpp) & #7784241 (pastelite)
Jump!	กระโดด!	CC-BY 2.0 (France) Attribution: tatoeba.org #1102981 (jamessilver) & #7784245 (pastelite)
Jump.	กระโดด	CC-BY 2.0 (France) Attribution: tatoeba.org #631038 (Shishir) & #7784244 (pastelite)
Stop!	หยุด!	CC-BY 2.0 (France) Attribution: tatoeba.org #448320 (FeuDRenais) & #7784240 (pastelite)
Wait.	รอก่อน	CC-BY 2.0 (France) Attribution: tatoeba.org #3048304 (camilozeta) & #7784248 (pastelite)
```

encoder_input_data (cell 14)





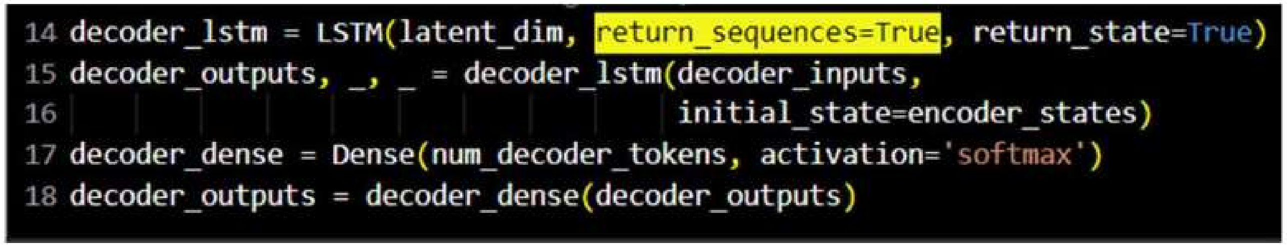
```
[23]  1 print(encoder_input_data[0])

[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [1. 0. 0. ... 0. 0. 0.]
 [1. 0. 0. ... 0. 0. 0.]
 [1. 0. 0. ... 0. 0. 0.]]
```



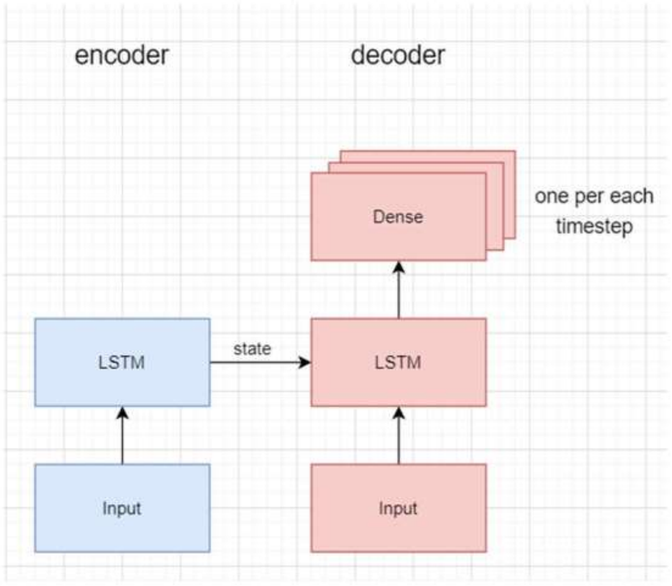
```
# cell 14
for i, (input_text, target_text) in enumerate(zip(input_texts, target_texts)):

    for t, char in enumerate(input_text):
        encoder_input_data[i, t, input_token_index[char]] = 1. # make the posi
    encoder_input_data[i, t + 1:, input_token_index[' ']] = 1. # make rows aft

```

cell 16

- normally we set return_sequence to False for the last LSTM layer that will connect to Dense
- but now we set it to True because we want to do softmax prediction on the whole sequence, not just one token



```
decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_inputs,
                                     initial_state=encoder_states)
decoder_dense = Dense(num_decoder_tokens, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)
```



Loss (cell 18)

- loss becomes multi-labels (one per token) prediction

$$-\sum_{j=1}^{M} y_j \log \hat{y}_j$$

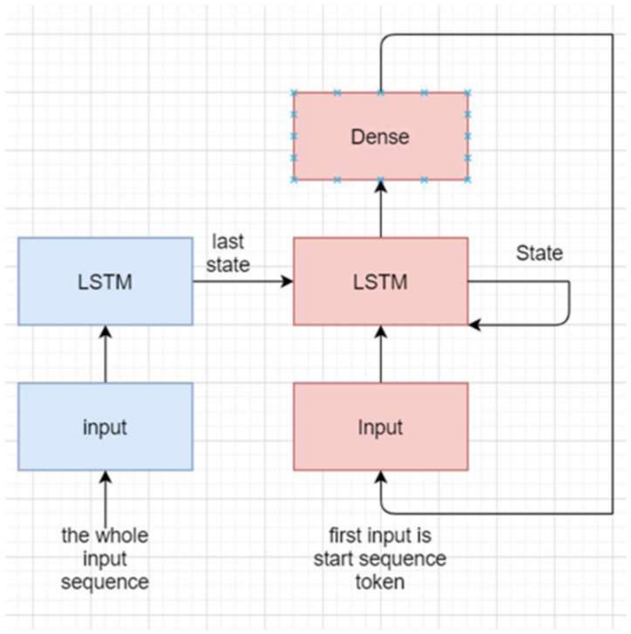
one tag per sequence

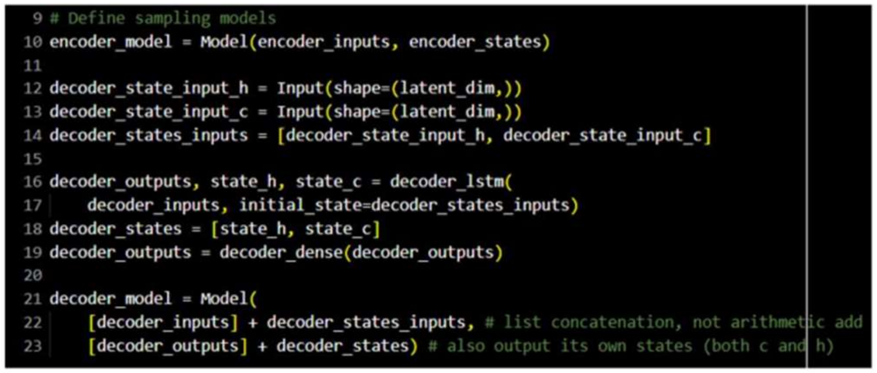
$$\Downarrow$$

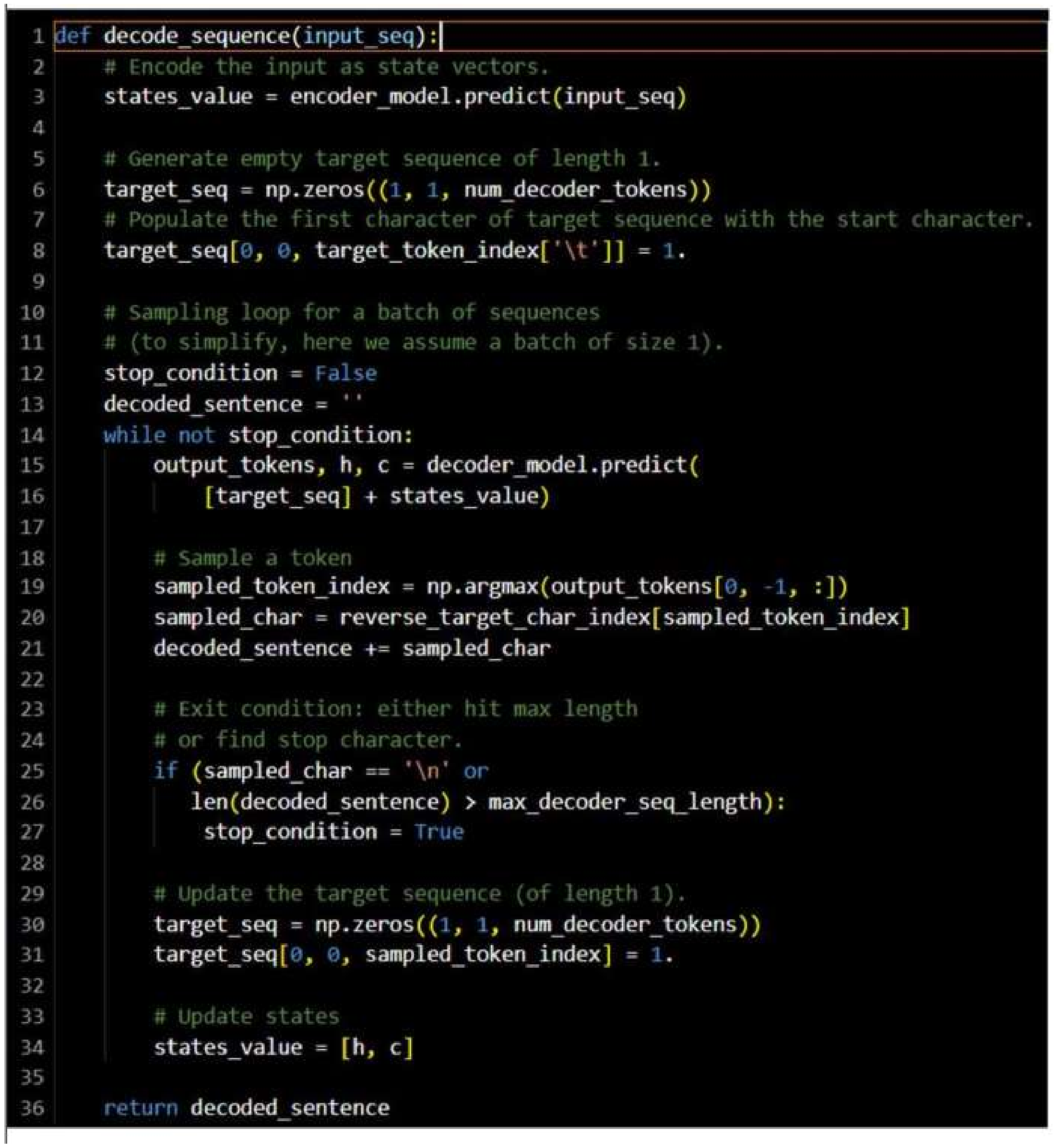
$$-\sum_{x \in \mathcal{X}} \sum_{j=1}^{M} y_j \log \hat{y}_j$$

one tag for every token in sequence

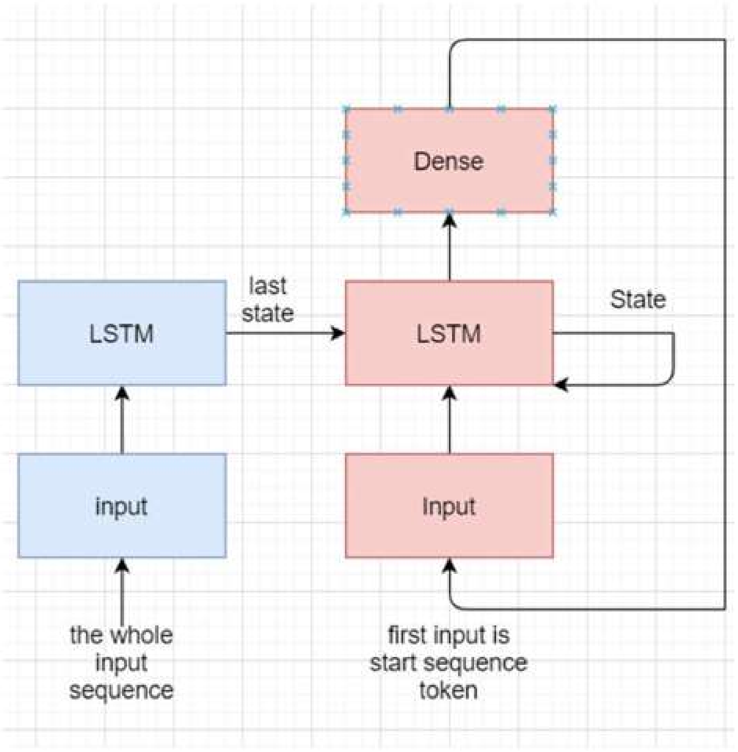
inference model




```
# Define sampling models
encoder_model = Model(encoder_inputs, encoder_states)

decoder_state_input_h = Input(shape=(latent_dim,))
decoder_state_input_c = Input(shape=(latent_dim,))
decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]

decoder_outputs, state_h, state_c = decoder_lstm(
    decoder_inputs, initial_state=decoder_states_inputs)
decoder_states = [state_h, state_c]
decoder_outputs = decoder_dense(decoder_outputs)

decoder_model = Model(
    [decoder_inputs] + decoder_states_inputs, # list concatenation, not arithmetic add
    [decoder_outputs] + decoder_states) # also output its own states (both c and h)
```



```python
def decode_sequence(input_seq):
    # Encode the input as state vectors.
    states_value = encoder_model.predict(input_seq)

    # Generate empty target sequence of length 1.
    target_seq = np.zeros((1, 1, num_decoder_tokens))
    # Populate the first character of target sequence with the start character.
    target_seq[0, 0, target_token_index['\t']] = 1.

    # Sampling loop for a batch of sequences
    # (to simplify, here we assume a batch of size 1).
    stop_condition = False
    decoded_sentence = ''
    while not stop_condition:
        output_tokens, h, c = decoder_model.predict(
            [target_seq] + states_value)

        # Sample a token
        sampled_token_index = np.argmax(output_tokens[0, -1, :])
        sampled_char = reverse_target_char_index[sampled_token_index]
        decoded_sentence += sampled_char

        # Exit condition: either hit max length
        # or find stop character.
        if (sampled_char == '\n' or
           len(decoded_sentence) > max_decoder_seq_length):
            stop_condition = True

        # Update the target sequence (of length 1).
        target_seq = np.zeros((1, 1, num_decoder_tokens))
        target_seq[0, 0, sampled_token_index] = 1.

        # Update states
        states_value = [h, c]

    return decoded_sentence
```



Subword Tokenization

Subword Tokenization

- More frequent words should be given unique ids
- Less frequent words should be decomposed into subword units that best retain their meaning
- Keep "wonderfully" as a single word since it appears often in dataset
- Split "structurally" into "structural" and "ly" since "structurally" is uncommon

Subword tokenization algorithms

- Byte-pair-encoding (BPE)
- Wordpiece
- Sentencepiece
- Unigram LM

BPE

- Borrow idea from information theory (Huffman encoding)
- Represent frequent words with fewer symbols
- Represent less frequent words with more symbols

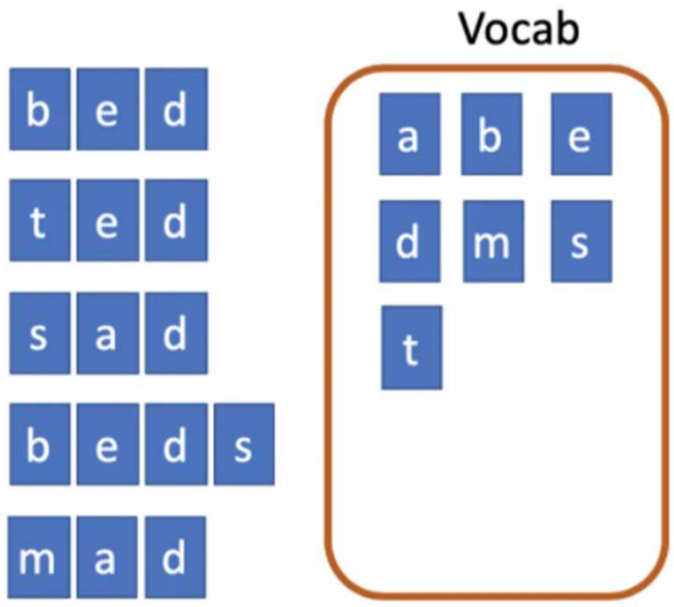
BPE

- Choose max vocabulary size.
- Split all words into unicode characters. Each unicode character corresponds to a symbol in the final vocabulary.
- While V < max vocab size
- Find the most frequent symbol bigram (pair of symbols)
- Merge those symbols to create a new symbol and add this to the vocabulary. This increases the vocabulary size by one.

Example

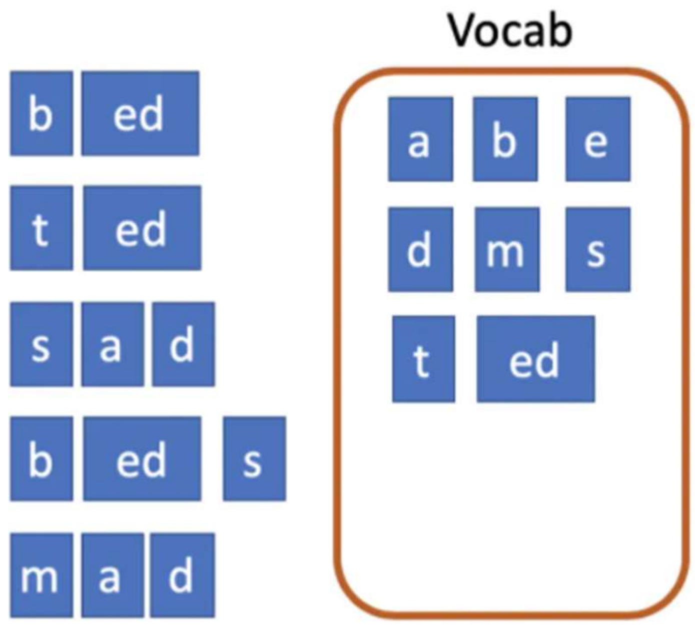
Corpus: "bed", "ted", "sad", "beds", and "mad"

Max vocab size = 10



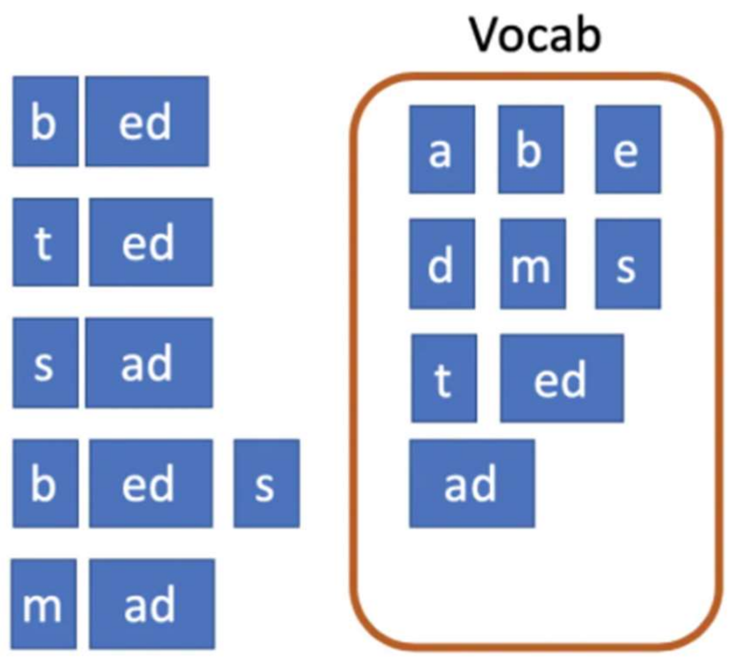
Example

- The most frequent symbol bigram is "ed" which appears 3 times. Merge these and add a new symbol to the vocabulary.



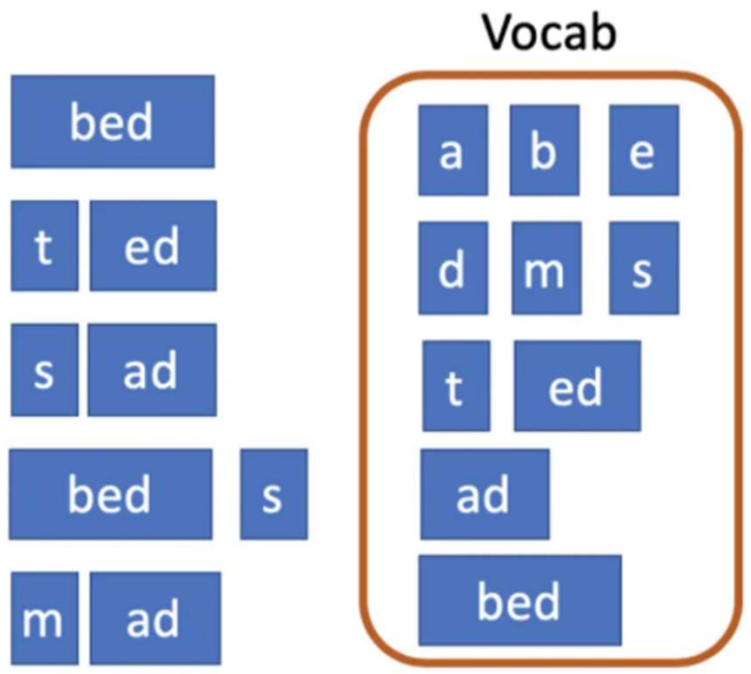
Example

- The next most frequent symbol (tied with 2 appearances) is "ad". Merge these and add the new symbol to the vocabulary once more. This brings the vocabulary size up to 9.



Example

- Finally, merge "b" and "ed" as this symbol pair also appears twice, bringing the vocabulary size to 10. Finished



Example

We want to distinguish between "ed" as a single word and the suffix "ed", so in reality we would represent the suffix as something like "##ed".

BPE need to tokenize beforehand to work. In English split with space, in languages with no space between words, need to use another tokenizer as preprocessing step.

Map unseen characters to an "UNKNOWN" token

Wordpiece

- Similar in concept to BPE
- WordPiece first initializes the vocabulary to include every character present in the training data
- does not merge not by frequency
- merges the bigram that, when merged, would increase the likelihood of a unigram language model trained on the training data.

Wordpiece

- recall unigram LM

$$p(w_1, w_2, ..., w_N) = \prod_{k=1}^{N} p(w_k) = \sum \log p(w_1) + \log p(w_2) + \cdots + \log p(w_n)$$

- assume that we merge w_1 and w_2

$$p(w_{12}, ..., w_N) = \prod_{k=1}^{N} p(w_k) = \sum \log p(w_{12}) + \cdots + \log p(w_n)$$

Wordpiece

- The change in the probability is

$$\log p(x, y) - \log p(x) - \log p(y) = \log \frac{p(x, y)}{p(x)p(y)}$$

- We choose the one where this change is maximum, so we're choosing the merge with maximum mutual information

$$\boxed{\mathrm{I}(X;Y) = \sum_{y\in\mathcal{Y}}\sum_{x\in\mathcal{X}} p_{(X,Y)}(x,y)\log\left(\frac{p_{(X,Y)}(x,y)}{p_X(x)\,p_Y(y)}\right), \quad \textbf{(Eq.1)}}$$

Unigram

- initializes its base vocabulary to a large number of symbols and progressively trims down each symbol to obtain a smaller vocabulary
- The natural choice is to use the union of all characters and the most frequent substrings in the corpus

$$P(\mathbf{x}) = \prod_{i=1}^{M} p(x_i),$$

$$\forall i \; x_i \in \mathcal{V}, \; \sum_{x \in \mathcal{V}} p(x) = 1,$$

$$\mathbf{x}^* = \arg\max_{\mathbf{x} \in \mathcal{S}(X)} P(\mathbf{x}),$$

Sentencepiece

- open-source implementation of wordpiece
- has many modes:
  - unigram
  - BPE
  - char
  - word (requires pre-tokenization)
- view <space> as _