# Deep Learning in NLP

**Homework 9**

## Contents



## 1. Prologue

This homework has a more free-form structure and it is meant to help you learn how to design your own deep learning models and apply them to novel tasks. While this project is meant as a homework, we will use the remaining practical sessions to work on it, to discuss (on Webex most likely) the design of the model, the implementation details, etc. The remaining topics (MTL, pre-trained embeddings, batching) will still be covered on Github and/or Twitch.

## 2. Introduction

The overall goal of the homework is to implement a toy machine translation system based on a simple variant of the *RNN encoder-decoder* architecture. This document provides a specification of the model, so don't worry if this term doesn't mean anything to you yet.

In the remaining of the document, we call an input to the system a *source sentence* and an output of the system a *target sentence*. We denote the source sentence as:

$$\mathbf{x} = (x_1, x_2, \ldots, x_n) \tag{1}$$

where $n$ is the source sentence length and each $x_i \in X$, where $X$ is the *vocabulary* (set of words) of the source language, and the target sentence as

$$\mathbf{y} = (y_1, y_2, \ldots, y_m) \tag{2}$$

where $m$ is the target sentence length and each $y_i \in Y$, where $Y$ is the vocabulary of the target language. We thus assume that both sentences are already *tokenized*.

# 3. Dataset

As a dataset you can create a toy parallel dataset of several simple (source sentence, target sentence) pairs. Once the prototype of the model is complete, you may consider switching to a larger dataset (e.g. `https://github.com/multi30k/dataset` or `http://www.manythings.org/anki/`) in order to polish and optimize the model and its implementation.

# 4. Specification

The encoder-decoder architecture consists of two modules:

- Encoder: module which reads a source sentence and (in its simplest form) transforms it to a single vector
- Decoder: module which takes the vector produced by the encoder and unfolds it to a target sentence

The two modules are described separately below.[1]

## 4.1. Encoder

The encoder[2] is a neural module which takes a source sentence $\mathbf{x}$ on input and encodes it to a vector representation. There are different ways to implement the encoder, here is one similar to what we have already seen during the course and which you can replace later with a more advanced solution if you wish:

1. Embed each input word $x_i$ as a dense vector representation $e(x_i) \in \mathbb{R}^d$ of size $d$
2. Apply an LSTM to the resulting sequence $(e(x_i))_{i=1}^n$

[1] Appendix A provides an *attention*-based extension of the encoder-decoder model. You can optionally implement it at the end once the basic version of the model works.

[2] Not to be confused with the `Encoder` class we use for replacing words/characters/tags/... with ints.

3. Retrieve the last element of the LSTM output

The second step can be defined in terms of the following recurrent formula:

$$(h_i, c_i) = \texttt{LSTMCell}(e(x_i), (h_{i-1}, c_{i-1})) \tag{3}$$

where $h_0, c_0$ are trainable parameters and `LSTMCell` is an LSTM computation cell (with its own set of trainable parameters). The result of the encoder, which represents the entire source sentence, is ~~$h_n$~~ $(h_n, c_n)$ (see step 3).

## 4.2. Decoder

The decoder serves to unfold the vector representation produced by the encoder – ~~$h_n$~~ $(h_n, c_n)$ – to a sequence of words. The decoder can be implemented as a recurrent process which, given the already predicted words $\hat{y}_0, \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{i-1}$, tries to predict the next word $\hat{y}_i$. The process ends when either a special *end-of-sentence* marker is produced, or when a maximum number of iterations is reached (to avoid infinite recurrence).

The task is to implement the decoder as a variant of LSTM[3] based on the following recurrent formula:

$$(h'_i, c'_i) = \texttt{LSTMCell}([e'(\hat{y}_{i-1}); h_n; c_n], (h'_{i-1}, c'_{i-1})) \tag{4}$$

where:

- ~~$h_n$~~ $(h_n, c_n)$ is the representation of the source sentence produced by the encoder
- ~~$h'_0 = h_n$~~ $(h'_0, c'_0) = (h_n, c_n)$, i.e. the hidden/cell states are initialized to the summary of the source sentence (note the use of $'$ to distinguish the decoder-related symbols $h'_i, c'_i, \ldots$ from those related to the encoder $h_i, c_i, \ldots$)
- ~~$c'_0$ is a trainable parameter of the decoder~~
- $\hat{y}_0$ is a special *beginning-of-sentence* marker
- $e'(y)$ is the embedding vector of target word $y$
- $[v; w; \ldots]$ represents the concatenation of vectors $v, w, \ldots$

The next word $\hat{y}_i$ is then predicted by selecting the word from the target language vocabulary $Y$ which maximizes the following probability distribution:

$$P(y_i) = \texttt{softmax}(U_o h'_i + V_o e'(\hat{y}_{i-1}) + W_o[h_n; c_n]) \tag{5}$$

where $U_o, V_o, W_o$ are trainable parameters. Put differently, the formula within `softmax` produces a vector of scores, one score per word in the target language, and the word with the highest score is selected.

[3] In the original paper a custom, more sophisticated variant of RNN was proposed for decoding; here we propose to use the LSTM gating mechanism, already implemented in PyTorch, for simplicity.

## 4.3. Training

The model can be trained to minimize either the cross-entropy loss or the negative log-likelihood of the predicted distributions $P(y_i)$ w.r.t. the actual target words $y_i$. However, to facilitate the process, $\hat{y}_{i-1}$ is often replaced by $y_{i-1}$ in Eq. 4 in order to make the training process more stable. Put differently, the next word $\hat{y}_i$ is predicted based on the word $y_{i-1}$ that should have been predicted in the previous step, even when $y_{i-1} \neq \hat{y}_{i-1}$. Of course this does not apply during decoding proper, where $y_{i-1}$ is unknown.

## 4.4. Evaluation

Evaluation of MT systems is beyond the scope of this homework. You can implement a regular accuracy function and plug it into the training procedure as a sanity check to make sure that the resulting model does not underfit, but be aware that accuracy is a very poor measure of translation quality. You should rather enrich the model with a higher-lever translation method, e.g.

```
def translate(self, sent: List[Word]) -> List[Word]:
    ...
```

(where `Word = str`) and test it manually. Do not expect an amazing translation quality, though, this is just a toy system! It could be nevertheless extended with more advanced features, e.g. attention (see App. A), BERT encoder, etc.

# A. Attention-based model

This section describes an optional extension of the encoder-decoder architecture based on the mechanism of *attention*.[4]

## A.1. Encoder

In the attention-based variant of the architecture, the encoder produces a sequence of contextualized embeddings $(h_i)_1^n$ rather than a single vector. Put differently, the last step of the encoder (see Sec. 4.1) can be discarded, since the output of encoding should contain the (contextualized) vector representations of all input words. Besides, one may use a BiLSTM, or even BERT to produce $(h_i)_1^n$.

## A.2. Decoder

On the decoder side:

$$(h'_i, c'_i) = \texttt{LSTMCell}([e(\hat{y}_{i-1}); \hat{h}_i], (h'_{i-1}, c'_{i-1})) \tag{6}$$

where $\hat{h}_i$ is a *projection* of the encoded source sentence $(h_1, \ldots, h_n)$ on the target position $i$ along a latent word-level alignment function $\alpha$:

$$\hat{h}_i = \sum_{j=1}^{n} \alpha_{ij} h_j \tag{7}$$

The alignment function itself is defined as:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{n} \exp(e_{ik})}, \tag{8}$$

where

$$e_{ij} = a(h'_{i-1}, h_j). \tag{9}$$

$a$ is an *alignment model* which scores how well the inputs around position $j$ and the output at position $i$ match. In practice, $a$ can be defined as:[5]

$$a(h'_{i-1}, h_j) = v_a^\top \tanh(W_a h'_{i-1} + U_a h_j), \tag{10}$$

where $v_a, W_a, U_a$ are all learnable parameters of the model.

In the alignment-based model, the next word $\hat{y}_i$ is predicted by selecting the word which maximizes the following probability distribution:

$$P(y_i) = \texttt{softmax}(U_o h'_i + V_o e'(\hat{y}_{i-1}) + W_o \hat{h}_i) \tag{11}$$

[4] As proposed in this paper, if you are curious.

[5] See this paper for other ways of implementing the alignment model $a$.