# Neural Networks Assignment
# Character-based language model with Long Short-Term Memory

**Ngoc-Quan Pham** and **Kay Rotmann** and **Alex Waibel**
Karlsruhe Institute of Technology
{ngoc.pham, kay.rotmann, alex.waibel}@kit.edu

## 1 Introduction - Neural Language Model

A language model is an algorithm of learning (to approximate) a function capturing the statistical characteristics of the distribution of sequences of words or characters in a natural language. Probabilistic language models assign probabilities to any sequence, and are typically interpreted as guessing the next words/characters in a sequence.

The probability of any sequence with arbitrary length can be factorized into the product of multiple conditional probabilities:

$$\begin{aligned} &P(w_1 w_2 w_3 \dots w_{t-1} w_t) = \\ &P(w_1) \cdot P(w_2|w_1) \cdot P(w_|w_1 w_2) \\ &\dots P(w_t|w_1 w_2 \dots w_{t-1}) \end{aligned} \tag{1}$$

From Equation 1 we aim at estimating every conditional probability distribution. In order to do this, we would need several ingredients:

- A pre-defined **vocabulary** so that we can limit the scope of the distribution to be learned
- A strong function approximator, in this case we will investigate Vanilla Recurrent Neural Networks and Long Short-Term Memory Recurrent Neural Networks. The RNNs are naturally suitable for this task because equation 1 suggests that the input of the network can be arbitrarily long.

In the lecture we have covered these two network variations and showed that the Vanilla RNNs suffer from the problem of gradient vanishing and struggle to learn long-time dependencies between elements in the sequence. The Long Short-Term Memory (LSTM) was proposed as (one of) the solution(s) to make learning more efficient.

## 2 Task description

You are provided with the setup of a basic character-level language model - described in 1 using recurrent networks using Python with minimal requirement (only Numpy - a library for scientific computing specifically desgined for matrix manipulation).

**Materials** You are provided with an implementation of a Vanilla Elman Recurrent Neural Networks which follows the "Backpropagation through time" algorithm in Lecture 14. The code also shows (the most simple way) to load the dataset, and convert the inputs from characters to one-hot vectors which are the inputs of the network. The flow of each network step (unfolding) is described in figure 2. The network is then trained with Stochastic Gradient Descent (a variation which is **Adagrad** to help training converge faster).

**Goal** The goal of the assignment is to help you understand several concepts introduced in the lectures.

- **Back-propagation**: this algorithm allows us to compute the gradients efficiently in any (differentiable) neural networks. Here we introduced the Back-propagation through time which is the same algorithm applied for recurrent neural networks.
- **Recurrent Neural Networks**: character-based language model is a difficult task, in general because the network has to remember very long sequences. The LSTMs, if implemented correctly will show that it can learn much better than the Vanilla RNNs.
- **Note about LSTM**: Compared to the slide 53 of the lecture 'Recurrent Neural Networks', I suggest to first concatenate $x_t$ and

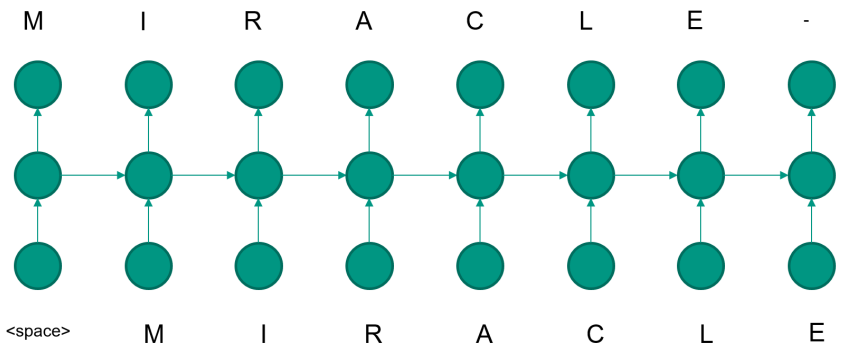


Figure 1: The operation of a Recurrent Neural Network learning a character-based language model. Each network step receives an input of a character in the history. The output layer of each step is a multinomial distribution of all characters in the vocabulary conditioned by the history.

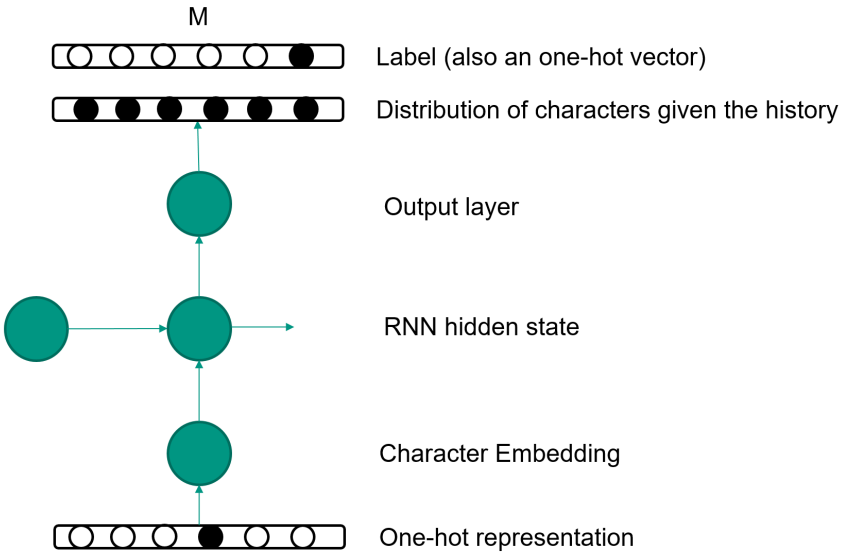


Figure 2: The operation of each network step in the unfolding process. Each character is represented by an one-hot representation (a vector with the size of the vocabulary with all zeros, only the index of the character is 1. It is then transformed (linearly) to a continuous-value vector (embedding) which is the input to the RNN network. The output of the network is fed into the Softmax function to get the distribution. Finally the loss of each step is Cross-Entropy between the output distribution and the label (a character represented with one-hot vector).

$h_{t-1}$ as one single input vector, so that we can also merge $W_x$ and $W_h$ as one single matrix with size $((dim_x + dim_h) * dim_h$. $dim_x$ is the dimension of the input $x$, and $dim_h$ is the dimension of the previous hidden $h$. The weight allocation in the template code will suggest this change, however you can also implement like this equation using 8 weight matrices (instead of 4 if we merge).

**Task** Implementation of a character-level language model using Long Short-Term Memory Recurrent Neural Networks. Since the data-loading process, gradient-checking and training is almost identical between two network variations, you only need to focus on implementing the **forward** and **backward** passes of the LSTMs. Inspired by the process of implementing any neural network in practice, your implementation should be able to sastify two requirements:

- Gradient-Checking. Back-propagation implementation, in general, depends on the correctness of the gradient computation based on each step of the chain rule. It is therefore crucial to verify the correctness of our returned gradients from the algorithm. The most efficient way to do this is to compare our analytical gradients (obtained by backpropagation) and the numerical gradients.

- Numerical gradients. Back-propagation is

the fast and efficient way to compute the gradients w.r.t the weights. We can compute the gradient of each weight using the basic equation in linear algebra:

$$\frac{dF(w)}{dw} = \frac{F(w+\delta) - F(x-\delta)}{2\delta} \quad (2)$$

Equation 2 suggests that for each weight $w$ in the network, we perform the forward pass twice. In each forward pass we change the weight $w$ by a small addition $\delta$. The difference between the loss values of these two runs gives us the numerical value of the gradient w.r.t this weight $w$ of the loss function. Of course this derivation method is very expensive compared to back-propagation, but it is easy to implement and typically serves as the providing the alternative values to check the correctness of a back-propagation implementation for a large network, in which errors are likely to happen.

- Training behaviour. A correct implementation of the neural network operation and learning algorithm will be shown in the training process. The loss function is computed for every sample which we want to reduce after each training step. The training progress is divided into two separated functions: "forward" and "backward" passes.

- Sampling. Since our network is a generative model we can check if it can learn properly based on trying to generate new samples. By generating new character sequences we will observe the learning progress of the network. Initially the network should generate almost random sequences, and then later on it manages to generate something that looks like the training data. The Vanilla RNN will be struggle to even generate correctly spelled words, while the LSTMs can generate more plausible sequences.

## 3 Submission guideline

You can find the code base here: **https://github.com/quanpn90/RNNAssignment** The repository contains the sample code for rnn (in the file "rnn.py") and the template for the LSTM in "lstm.py".

The data comes with "data/input.txt" which is by default used in 'rnn.py' and 'lstm.py', however you can also replace with any text file you'd lik to use. The file "rnn.py" will show how backpropagation is done (minimally complicated) and the "lstm.py" file contains the template guiding you how to implement the LSTM.

Your submission can be done in groups, with **maximum of 5** people. You can either send me the 'lstm.py' file or your github repository to my email **ngoc.pham@kit.edu** with the names and the KIT code of the group members.

Deadline: **8.8.2019; 23:59** (at the same day with the exam). I tried to extend the date but according to the KIT rule it cannot be after the exam.

The correct implementation will be worth 6 points in the final exam (10% of the total). Note that it only applies when your pre-added grade is adequate to pass.

Any question or discussion will be welcomed via email or Piazza. Thank you very much.