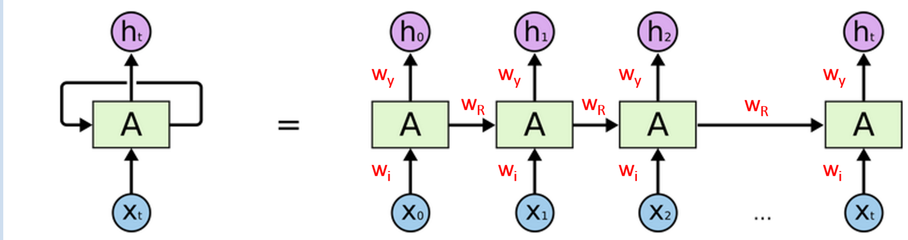
An Autocompletion Algorithm using Recurrent Neural Networks

Marvin Klaus[1], Daniela Schacherer[2], Sebastian Bek[3]
[1]*Heidelberg University, M.Sc. Applied Computer Science*
[2]*Heidelberg University, M.Sc. Applied Computer Science*
[3]*Heidelberg University, M.Sc. Applied Computer Science*

Introduction

A classical feedforward network expects data that are independent from each other as well as idenpendently and identically distributed. It follows that there is also no relation between the current and the previous output of the network. However there are some scenarios where the previous output is needed in order to get the new output. This is for example the case when dealing with sequences like text, genomes or numerical time series data. Here a meaningful prediction can only be made based on the current and one or multiple previous outputs. In such cases **recurrent neural networks** (RNN) are the method of choice.

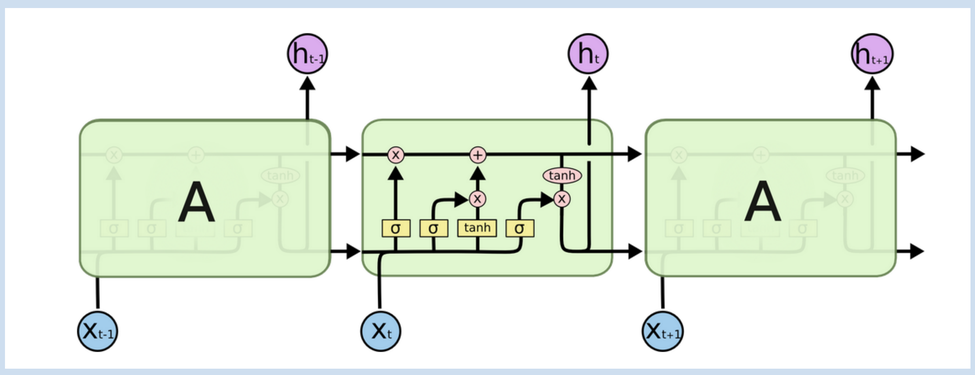


Hereby each of the unrolled network states A receives the current input x_t as well as the input at time $t-1$ and computes the ouput considering both.
Long term dependencies:
RNNs like the one above can memorize the information from the timepoint directly before the current time. But there are also situations where the information from even earlier is needed for the prediction, which is where **long short term memory networks (LSTMs)** come into play.

LSTM Architecture

For our purpose we decided to use long short term memory networks (LSTMs). In comparison to simple RNNs, LSTMs have a more complex repeating module each consisting of four interacting layers. The horizontal line passing the module is the central part of an LSTM, also called the cell state C. There are three gates that allow to introduce or remove information from the cell state:

- What to forget: $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$
- Values to add: $\tilde{C}_t = tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$
- What to add: $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$
- The new cell state: $C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$
- Module output h_t:
  $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$
  $h_t = o_t \cdot tanh(C_t)$



Requirements and Concept

Requirements
We used the programming language python and pytorch as the framework to implement our neural network.

Concept
We used an extract from Dan Brown's book Origin as training set for our model. We decided to use a relatively small training set that allows training on a CPU. Our workflow consists of the following steps:

- **Data preprocessing and construction of the training set**: We splitted our text into slices of a predefined length (e.g. 30 characters) spacing the sequences by a predefined offset (e.g. 4 characters). The data were than converted into an One-Hot encoding.
- **Training of the RNN**: For training we used an RNN consisting of an LSTM architecture with 128 neurons, followed by a fully connected layer with as many neurons as unique characters exist in our text. Finally a softmax layer is applied to the output.
- **Prediction**: The trained network was then used to predict the next character given a sequence of characters. Based on this network we predicted the following character.

Conclusion

One of the main challenges during this project was to define the network architecture, loss function and optimizer as well as good values for the various parameters like e.g. the learning rate.

- Loss function: As we needed a categorical class measure we decided to use the nn.CrossEntropyLoss which combines a LogSoftMax and the Negative Log Likelihood Loss in one single class.
- Optimizer: The Adam optimizer that uses the Adam algorithm, an extension of the stochastic gradient algorithm, provided in our case the best results.
- Parameters: By separate testing we found a value of 0.001 up to 0.0075 for the learning rate, a batch size of 16 and a hidden size of 128 to be suitable to our scenario.

RESULTS

CrossEntropy Loss during training