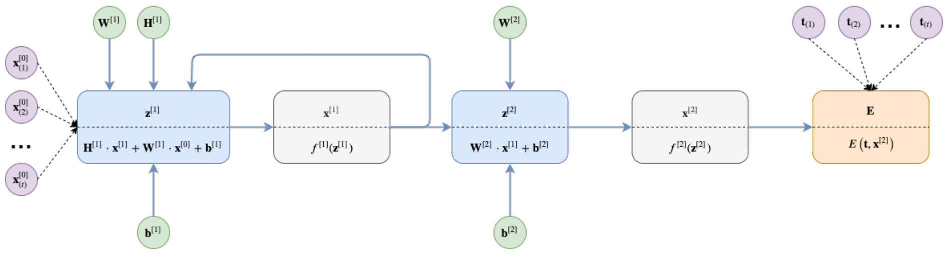
Deep Learning (IST, 2021-22)

Practical 9: Recurrent Neural Networks

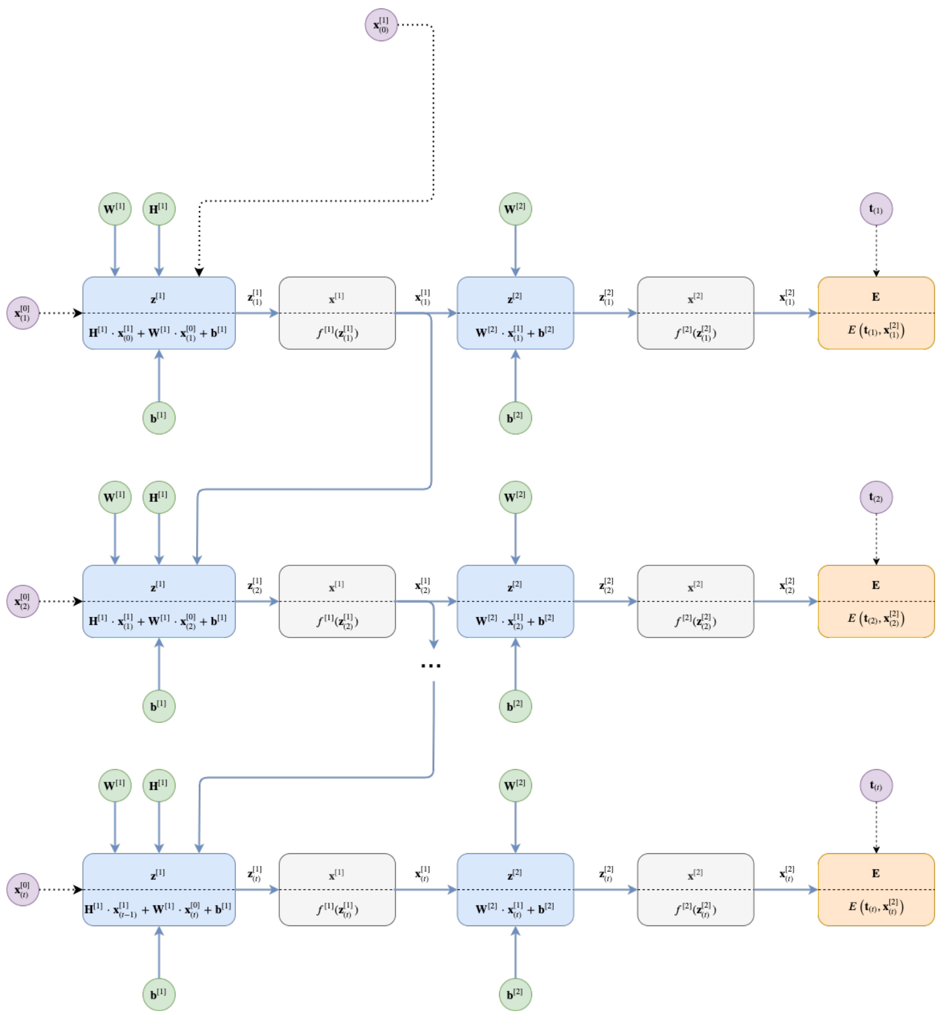
Pedro Balage, Gonçalo Faria, Luis Sa-Couto, Andreas Wichert, André Martins, Gonçalo Correia

Introduction

Below we present a simple recurrent network that maps a sequence of inputs to a sequence of targets. In this example, we use only one hidden layer that has a feedback connection.

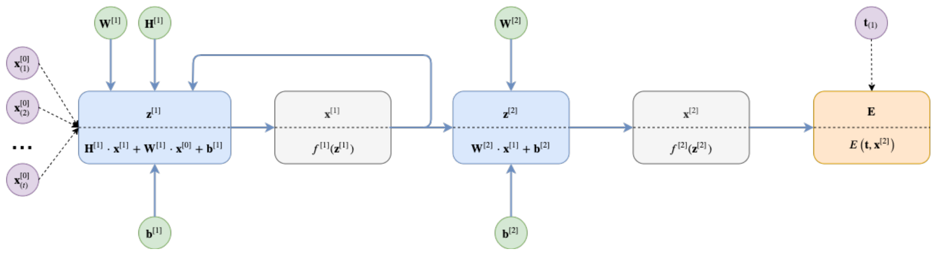


When one thinks about how to apply backpropagation to learn the parameters on such a network it may not seem obvious at first. However, if we unfold the network's function graph, the recurrent network becomes much like a regular one.

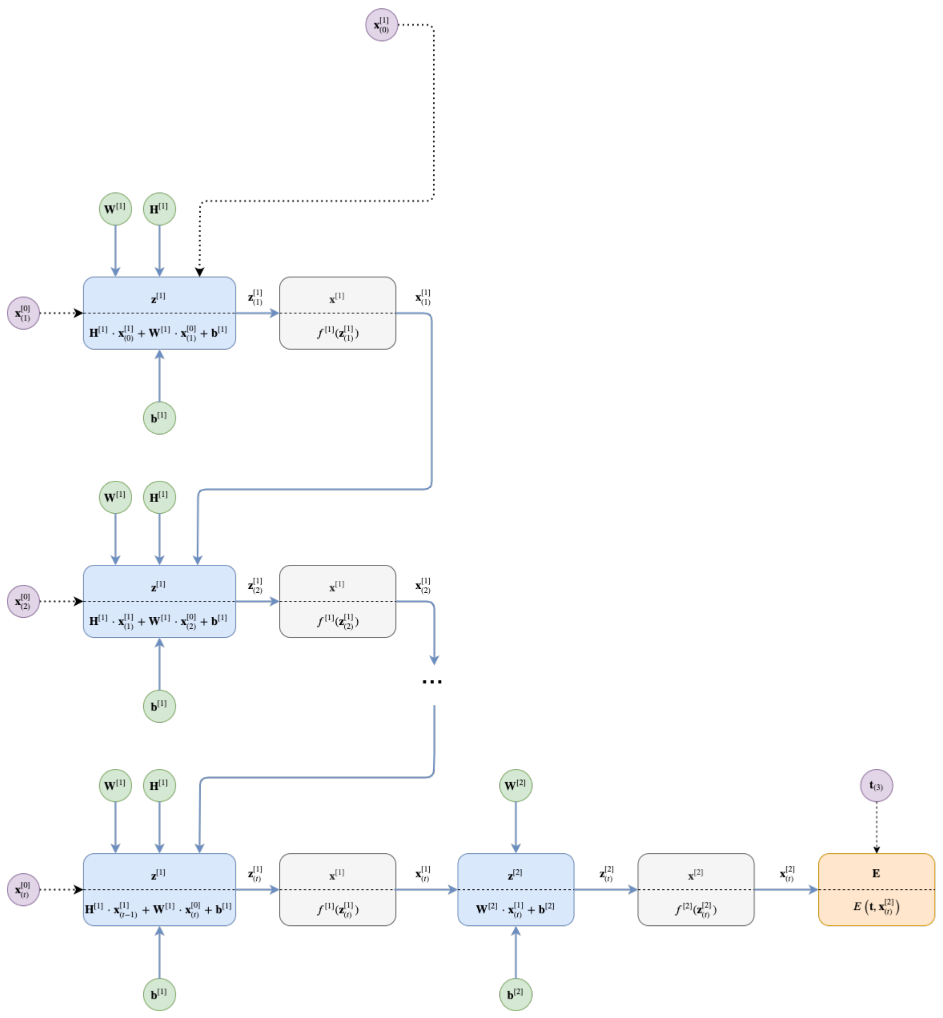


Unfolding corresponds to making copies of the network structure to match the total number of sequence elements. This implies that between copies of the network the weights are shared! This means that $\mathbf{W}^{[1]}$ at the bottom of the unfolded graph is the same as the $\mathbf{W}^{[1]}$ at the top. This weight sharing will influence the way we apply the backward pass. Namely, to update a parameter, we need to sum accross all paths that lead to all copies of that parameter. A requirement of this approach is that we need to initialize $\mathbf{x}_{(0)}^{[1]}$ to feed as hidden state to the first copy of the network.

As an example, we used a many-to-many recurrent network and showed unfolding corresponds to making copies of its structure for every timestep. However, many other architecture templates exist. For example, we could have a many-to-one network. In this case, we will see that sometimes what we have to replicate is just the recursion part. For example, for the network:



the unfolding would be done as follows:



Note that the recursion is copied throughout all timesteps but since there is only one output, we only place an output layer after the last timestep.

Pen-and-Paper Exercises

The following questions should be solved by hand. You can use, of course, tools for auxiliary numerical computations.

Question 1

Consider a recurrent network with one hidden layer to solve a **many-to-many** (sequence tagging) task. Take into account that:

- The recurrent connections happen between the hidden layers
- The hidden activation function is the hyperbolic tangent
- The output activation function is softmax
- The error function is the cross-entropy between output and target

1. Write down the model equations for forward propagation.

Solution:

Let

- $\boldsymbol{x}_1, \dots, \boldsymbol{x}_T$ denote the inputs,
- $\boldsymbol{h}_1, \dots, \boldsymbol{h}_T$ denote the hidden states,
- $\boldsymbol{p}_1, \dots, \boldsymbol{p}_T$ the label probabilities computed by the model,
- $y_1, \dots, y_T$ denote the gold outputs, $\boldsymbol{e}_{y_1}, \dots, \boldsymbol{e}_{y_T}$ their one-hot vector representations,
- $\ell_1, \dots, \ell_T$ the cross-entropy loss incurred at each time step,
- $\mathbf{W}_{hh}$ the recurrent (hidden-hidden) matrix, $\mathbf{W}_{xh}$ the input-hidden matrix, $\mathbf{W}_{hy}$ the hidden-output matrix, $\boldsymbol{b}_h$ the bias in the hidden layer, and $\boldsymbol{b}_y$ the bias in the output layer.

For all $t \in \{1, \dots, T\}$, the forward equations are:

$$\begin{aligned}
\boldsymbol{z}_t^{[1]} &= \mathbf{W}_{hh}\boldsymbol{h}_{t-1} + \mathbf{W}_{hx}\boldsymbol{x}_t + \boldsymbol{b}_h \\
\boldsymbol{h}_t^{[1]} &= \tanh(\boldsymbol{z}_t^{[1]}) \\
\boldsymbol{z}_t^{[2]} &= \mathbf{W}_{yh}\boldsymbol{h}_t + \boldsymbol{b}_y \\
\boldsymbol{p}_t &= \text{softmax}(\boldsymbol{z}_t^{[2]}) \\
\ell_t &= \log[\boldsymbol{p}_t]_{\boldsymbol{y}_t} = -\boldsymbol{e}_{\boldsymbol{y}_t} \cdot \log \boldsymbol{p}_t.
\end{aligned}$$

2. Perform a forward propagation for the following sequence of length 2:

$$[\boldsymbol{x}_1, \boldsymbol{x}_2] = \left[\begin{pmatrix} 4 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 8 \\ 2 \\ 0 \end{pmatrix} \right]$$

With targets (gold outputs):

$$[\boldsymbol{e}_{y_1}, \boldsymbol{e}_{y_2}] = \left[\begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right]$$

Initialize all weights and biases to 0.1, using 3 units per hidden layer, initializing the hidden state to all zeros.

Solution:

Initializing the parameters, we have:

$$\mathbf{W}_{hx} = \begin{pmatrix} 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 \end{pmatrix}$$

$$\boldsymbol{b}_h = \begin{pmatrix} 0.1 \\ 0.1 \\ 0.1 \end{pmatrix}$$

$$\mathbf{W}_{hh} = \begin{pmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{pmatrix}$$

$$\mathbf{W}_{yh} = \begin{pmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{pmatrix}$$

$$\boldsymbol{b}_y = \begin{pmatrix} 0.1 \\ 0.1 \end{pmatrix}$$

$$\boldsymbol{h}_0 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Now, we can use the equations from the previous question to apply forward propagation. Let us start with timestep $t = 1$:

$$\begin{aligned}
\boldsymbol{z}_1^{[1]} &= \begin{pmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 \end{pmatrix}\begin{pmatrix} 4 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.1 \\ 0.1 \end{pmatrix} \\
&= \begin{pmatrix} 0.5 \\ 0.5 \\ 0.5 \end{pmatrix} \\
\boldsymbol{h}_1 &= \tanh\begin{pmatrix} 0.5 \\ 0.5 \\ 0.5 \end{pmatrix} \\
\boldsymbol{z}_1^{[2]} &= \begin{pmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{pmatrix}\tanh\begin{pmatrix} 0.5 \\ 0.5 \\ 0.5 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.1 \end{pmatrix} \\
&= \begin{pmatrix} 0.3\tanh 0.5 + 0.1 \\ 0.3\tanh 0.5 + 0.1 \end{pmatrix} \\
\boldsymbol{p}_1 &= \operatorname{softmax}\begin{pmatrix} 0.3\tanh 0.5 + 0.1 \\ 0.3\tanh 0.5 + 0.1 \end{pmatrix} \\
&= \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}
\end{aligned}$$

Now we can apply the same logic to the second timestep.

$$
\begin{aligned}
\boldsymbol{z}_2^{[1]} &= \begin{pmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{pmatrix} \tanh \begin{pmatrix} 0.5 \\ 0.5 \\ 0.5 \end{pmatrix} + \begin{pmatrix} 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 \end{pmatrix} \begin{pmatrix} 0 \\ 8 \\ 2 \\ 0 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.1 \\ 0.1 \end{pmatrix} \\
&= \begin{pmatrix} 1.1 + 0.3 \tanh 0.5 \\ 1.1 + 0.3 \tanh 0.5 \\ 1.1 + 0.3 \tanh 0.5 \end{pmatrix} \\
\boldsymbol{h}_2 &= \tanh \begin{pmatrix} 1.1 + 0.3 \tanh 0.5 \\ 1.1 + 0.3 \tanh 0.5 \\ 1.1 + 0.3 \tanh 0.5 \end{pmatrix} \\
\boldsymbol{z}_2^{[2]} &= \begin{pmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{pmatrix} \tanh \begin{pmatrix} 1.1 + 0.3 \tanh 0.5 \\ 1.1 + 0.3 \tanh 0.5 \\ 1.1 + 0.3 \tanh 0.5 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.1 \end{pmatrix} \\
&= \begin{pmatrix} 0.3 \tanh \left(1.1 + 0.3 \tanh 0.5\right) + 0.1 \\ 0.3 \tanh \left(1.1 + 0.3 \tanh 0.5\right) + 0.1 \end{pmatrix} \\
\boldsymbol{p}_2 &= \operatorname{softmax} \begin{pmatrix} 0.3 \tanh \left(1.1 + 0.3 \tanh 0.5\right) + 0.1 \\ 0.3 \tanh \left(1.1 + 0.3 \tanh 0.5\right) + 0.1 \end{pmatrix} \\
&= \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}
\end{aligned}
$$

Programming Exercises

The following exercises should be solved using Python, you can use the corresponding practical's notebook for guidance.

Question 1

1. Implement the forward propagation from Question 1 in NumPy.

Question 2

One of the capabilities of an RNN is to model language generation. In this exercise, we will use PyTorch to implement a RNN cell and use it to generate random person names.

1. Complete the PyTorch code for the forward pass of the RNN.
2. Using the previous functions, train a model for 1,000 epochs using negative log-likelihood, a hidden size of 128 and a learning rate of 0.0005. Report the training loss curve.
3. Explore how to generate random names. Implement sampling from the output distribution over letters and also top-k sampling. Experiment with different values for k (e.g., 1, 2, 3, etc.). Did you get good results? Discuss the results you got.

   Solution:

   In order to avoid this problem we may want to explore the use of regularizers and also sampling among the top 2 or top 3 best letters in the sequence decoding `(topk(2), topk(3))` instead of always selecting the best one.

Question 3

Now it's time to try Recurrent Neural Networks on real data and compare it with other approaches previously seen in our practical sessions.

Let's use PyTorch and RNNs to perform sentiment analysis with the IMDB database.

1. Using the RNN implementation from PyTorch, train a sentiment classifier for 5 epochs using an embedding layer with dimension 128 and hidden layer with dimension 256. Use the `Adam` optimizer and the `BCEWithLogitsLoss`. Report the final accuracy.

2. Change the network architecture to use a bidirectional LSTM and report the new results. Discuss how would you improve the results even more.

   Suggestion: You may want to run different experiments in Google Colab to have a faster execution using a GPU environment.

   Solution: Using a bidirectional LSTM require some changes in the code as the output from both directions need to be concatenated.

   In order to improve the results, other strategies may be considered:

   - Start with pre-trained embeddings
   - Use a padded sequence
   - Add a dropout
   - Change the hyperparameters (number of layers; layer sizes, etc)