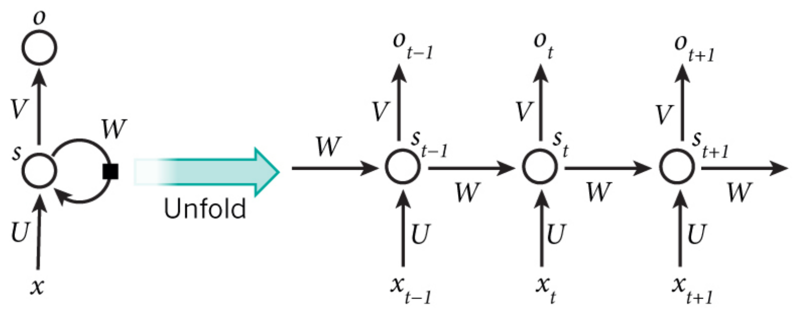
Backpropagation in Recurrent Neural Networks

1. Introduction

The goal of this assignment is to experiment with backpropagation on recurrent neural networks (RNN). Most of the code is already written, but you will have to implement yourself a few functions such as the backpropagation itself.

2. Recurrent neural networks (RNN)

You have seen that the multilayer perceptron can be interpreted as a computational oriented graph. In the multilayer perceptron, the graph does not contain any loops, so we sometimes call it a *feed-forward* network. On the contrary, in RNN, each neuron is taking its own output as input for the following step. The idea is to have a "memory" effect to be able to model sequential information.



The above diagram shows a RNN being unrolled (or unfolded) into a full network. By unrolling we simply mean that we write out the network for the complete sequence. For example, if the sequence we care about is a sentence of 5 words, the network would be unrolled into a 5-layer neural network, one layer for each word. The formulas that govern the computation happening in a RNN are as follows:

$$\begin{aligned} s_t &= f(Ux_t + Ws_{t-1}) \\ o_t &= \mathrm{softmax}(Vs_t) \end{aligned}$$

- x_t is the input at time step t. For example, x_1 could be a one-hot vector corresponding to the second word of a sentence.
- s_t is the hidden state at time step t. It's the "memory" of the network. s_t is calculated based on the previous hidden state and the input at the current step. The function f usually is a nonlinearity such as tanh or ReLU. s_{-1}, which is required to calculate the first hidden state, is typically initialized to all zeroes.
- o_t is the output at step t. For example, if we wanted to predict the next word in a sentence it would be a vector of probabilities across our vocabulary.

3. Defining a toy problem

Let's try to teach a RNN model how to add numbers. We will represent integers with a list of digits.

Example: 142857 = [0 0 0 0 1 4 2 8 5 7]

The input x will be a sequence of digits and each x_t is a single digit, one-hot encoded into a vector in $\mathbb{R}^{10}$.

Let's assume we pick a hidden layer size $H = 100$. You can think of the hidden layer size as the "memory" of our network. Making it bigger allows us to learn more complex patterns, but also results in additional computation. Then we have:

$$\begin{aligned} x_t &\in \mathbb{R}^{10} \\ o_t &\in \mathbb{R}^{10} \\ s_t &\in \mathbb{R}^{100} \\ U &\in \mathbb{R}^{100\times 10} \\ V &\in \mathbb{R}^{10\times 100} \\ W &\in \mathbb{R}^{100\times 100} \end{aligned}$$

To train our network we need a way to measure the errors it makes. We call this the loss function L, and our goal is find the parameters U, V and W that minimize the loss function for our training data. A common choice for the loss function is the **cross-entropy loss**. If we have N training examples and C classes then the loss with respect to our predictions o and the true labels y is given by:

$$L(y, o) = -\frac{1}{N}\sum_{n\in N} y_n \log o_n$$

The formula sums over our training examples and adds to the loss based on how off our prediction are. The further away y (the correct digit) and o (our predictions), the greater the loss will be.

4. Backpropagation Through Time (BPTT)

We want to find the parameters U, V and W that minimize the total loss on the training data. The most common way to do this is SGD, Stochastic Gradient Descent: We iterate over all our training examples and during each iteration we nudge the parameters into a direction that reduces the error. These directions are given by the gradients on the loss: $\frac{\partial L}{\partial U}, \frac{\partial L}{\partial V}, \frac{\partial L}{\partial W}$. SGD also needs a *learning rate*, which defines how big of a step we want to make in each iteration.

To calculate those gradients we use the backpropagation algorithm. In RNNs, this algorithm is called **Backpropagation Through Time (BPTT)**, because the parameters are shared by all time steps in the network, so the gradient at each output depends not only on the calculations of the current time step, but also the previous time steps.

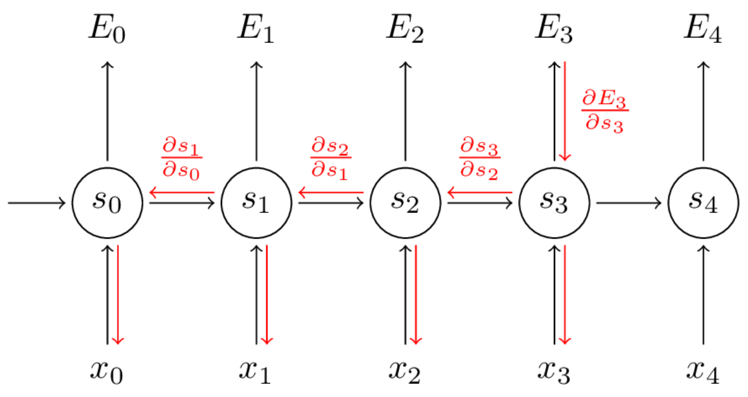
Let's recall the RNN model:

$$\begin{aligned} s_t &= \tanh(Ux_t + Ws_{t-1}) \\ \hat{y}_t &= \mathrm{softmax}(Vs_t) \end{aligned}$$

Exercise 1.

- Using the RNN model, the loss function, and the chain rule, derive the gradient expressions of $\frac{\partial L}{\partial U}, \frac{\partial L}{\partial V}, and \frac{\partial L}{\partial W}$.

Note that you need to sum up the contributions of each time step to the gradient. In other words, because W is used in every step up to the output we care about, we need to backpropagate gradients through the network all the way to $t = 0$:


E_0
E_1
E_2
E_3
E_4
$\frac{\partial E_3}{\partial s_3}$
$\frac{\partial s_1}{\partial s_0}$
$\frac{\partial s_2}{\partial s_1}$
$\frac{\partial s_3}{\partial s_2}$
s_0
s_1
s_2
s_3
s_4
x_0
x_1
x_2
x_3
x_4

Exercise 2.

- Fill the method `bptt` to implement the Backpropagation Through Time (BPTT) algorithm.
- Use the gradient checking utilities `run_gradient_check` to check if your function is correct.

Gradient Checking

Whenever you implement backpropagation it is good idea to also implement *gradient checking*, which is a way of verifying that your implementation is correct. The idea behind gradient checking is that derivative of a parameter is equal to the slope at the point, which we can approximate by slightly changing the parameter and then dividing by the change:

$$\frac{\partial L}{\partial \theta} \approx \lim_{h \to 0} \frac{J(\theta + h) - J(\theta - h)}{2h}$$

We then compare the gradient we calculated using backpropagation to the gradient we estimated with the method above. If there's no large difference we are good. The approximation needs to calculate the total loss for *every* parameter, so that gradient checking is very expensive.

5. Vanishing gradients

Exercise 3.

- Fill the method `bptt_viz` with your implementation of BPTT. This function takes an additional parameter `only_paths_of_size`, which makes the BPTT only use paths of length `only_paths_of_size` to compute $\frac{\partial L}{\partial W}$.
- Write a function to vizualize the mean absolute value of $\frac{\partial L}{\partial W}$, as a function of `only_paths_of_size`.
- Explain why we use a `bptt_truncate` parameter.
- Discuss the limit of RNN to model long distance interactions inside sequences.