Tutorial 05: Recurrent Neural Networks (26P)

The goal of this exercise is to understand and implement the forward and backward propagation of a simple recurrent neural network (RNN). We will do so by applying a RNN to the easy task of learning the binary addition.

Binary Addition Problem

The binary addition suits the area of application of RNNs as it is a sequence to sequence problem with a "temporal" context. The input sequence to our network will consist of two digits at each time step, i.e. a bit stream of two numbers that are supposed to be added. The output of the network at every time step will be the prediction of the network for the current bit of the summed number, i.e., the output will be a bit stream of the supposed added number.
Please submit your code by 25th Nov 23:59 to manuel.milling@informatik.uni-augsburg.de.
You can submit your solutions alone or in Teams of 2 (please indicate all names with the submission).
Note: The task descriptions are formulated in such a way, that the solution is as similar as possible to the solution of the previous exercise sheet. The task description therefore includes tensors with three axes, which might complicate some of the calculations. It is also reasonable to implement a solution that only uses matrices, i.e., tensors with two axes.

1 Binary Addition Data (5P)

Implement a method **generate_data(num_examples, num_bits)** that creates **num_examples** data points/ training examples for the binary addition problem with **num_bits** bits. The output should consist of two numpy arrays, one for the data of shape (**num_examples**, **num_bits**, 2) and one for the labels of shape (**num_examples**, **num_bits**, 1).
*Note: It is only necessary that the method works for 8 bit digits, so that the numpy method **unpackbits** can be used. If a different implementation is chosen, the shapes of the output of this method can be different as well.*

2 Recurrent Neural Network (2P)

Implement a class for a 1-hidden-layer RNN with simple fully connected cells and without any biases. The initialiser **__init__(self, n_input, n_hidden, n_out)** should initialise the weights of the network depending on the number of input (here 2) and output (here 1) features **n_input** and **n_out**, as well as the number of neurons in the hidden layer **n_hidden**.

3 Forward Propagation (6P)

Implement a class method **forward_propagation(self, X)** that performs the forward propagation of the RNN for a given data set **X** and returns the prediction of the network, which should have the same shape as the labels created before. Use the sigmoid activation at every layer. Save the results of every layer as class variables for the backpropagation.
Note: The binary notation of numbers usually starts with the bit representing the highest power of two. The addition in such a case has to be performed back to front. Figure out how matrix multiplications have to be applied for the tensors with 3 axes obtained from exercise 1 here.

4 Mean Square Error (1P)

Implement the mean squared error (loss function) as a method **mean_square_error(pred, Y)** considering the predictions **pred** of the network and the labels **Y**. The error of each sequence element of each training example should count equally.

5 Backpropagation Through Time (8P)

Implement a method **backprop_through_time(self, Y)** that executes the BPTT for the above mentioned neural network with the mean squared error as a loss function considering the network's predictions and the according labels **Y**. Save the components of the gradient in class variables of the same shape as the network's weights.
Note: Consider again the actual direction of the backpropagation and the use of matrix multiplications for tensors with 3 axes here.

6 Gradient step (2P)

Implement a class method **gradient_step(self, learning_rate)** that performs an optimisation step using the previously saved compoents of the gradient, as well as the learning rate **learning_rate**.

7 Accuracy (2P)

Implement a method **accuracy(pred, Y)** that determines the accuracy of the network by rounding the predictions **pred** to the closest integer numbers and comparing them to the labels **Y**.

8 Training Rountine (2P)

Create a training set of 100 data points and a test set of 10 data points. Train your network with 10 000 iterations and print loss and accuracy on training and test set every 100 training iterations.
Note: The train and test loss should be decreasing consistently and the accuracy should reach 100% quite fast.