Assignment 3.A: Adding Integers with Recurrent Networks

Wojtek Kowalczyk

28.03.2020

Introduction

Let us suppose that we want to develop a neural network that learns how to add two, at most 3-digit long, integers. For example, on an input string of 7 characters: '123+345' the network should return a sequence of 3 characters: '468' that represent the sum of 123+345. Clearly, we are not interested in "memorizing" the whole multiplication table (about 1 million strings); instead, we want the network to learn some general principles behind the addition operation (that we learn in elementary schools). Thus, we will train our network on a limited collection of examples: instead of using a million of possible examples we will use just a random sample of 50.00 examples (5%). Surprisingly(?), recurrent networks can learn this task quite well!

The purpose of this assignment is to investigate various network architectures (SimpleRNN, LSTM and GRU networks) and several ways of representing the input-output relation:

- "plain" (the string '123+345' is "read" from left to right),
- "reversed" (the input string is read from right to left (i.e., '543+321', while returning the same result, '468'),
- "binary plain" and "binary reversed" – the same as above but the input integers are represented by strings of 10 bits (while the result might require 11 bits).

Additionally, we might "help" the network to provide the input in a similar way we learn at school: pairs of digits, reading simultaneously from left to right or, in reverse order, from right to left. Thus, instead of using the input '123+345' (7 characters) we would use 3 pairs: (1,3), (2,4), (3,5), or, in the reverse variant: (3,5), (2,4), (1,3).

Your task is to experiment with 3 types of recurrent networks (SimpleRNN, LSTM, GRU) trained on 8 possible ways representing the "input-output" relation. To limit the training time, we will stick to adding integers between 0 and 999 (up to 3 digits in case of "decimal" representation) and between 0 and 1023 (up to 10 bits in case of binary representation).

This task seems to be huge! Fortunately, there is a very elegant implementation that you can use as a starting point for your experiments:

https://github.com/keras-team/keras/blob/master/examples/addition_rnn.py

This code was written in TF1.0 but it can be easily modified to work with TF2.0 – see the attached file.

Task 1: Study the code and get it running. When you understand the code and the way the input-output data is represented, extend the code by adding the ultimate error measure: the percentage of errors on a complete set of 1 million examples. Additionally, add two other error measures (to be applied to the complete set): MSE and MAE. Use the default number of 200 iterations. Report your findings on the first two tasks: decimal representation; "normal" and "reverse" order of digits.

Task 2: Modify your code to work on bit representation of integers. Repeat both experiments (with "normal" and "reverse" representations). Are the results better or worse? How would you explain it?

Task 3: Modify your code from **task 1** to work with "pairs of digits" representations (i.e., the sequence '123+345' (7 characters) should be represented by a sequence of 3 pairs: (1,3), (2,4), (3,5). Repeat both experiments with "normal" and "reverse" representations of input sequences ((1,3), (2,4), (3,5) becomes (3,5), (2,4), (1,3)).

Task 4: Finally, modify your code from **task 2** to work with "pairs of bits" representations (i.e., the sequence '10010+1011' could be represented by the 5 pairs: (1,0), (0,1), (0,0), (1,1), (0,1). Repeat both experiments with "normal" and "reverse" representations of input sequences.

Note that the provided script uses LSTM networks; however, it is easy to replace "LSTM" by "SimpleRNN" or "GRU" and repeat the experiments. Therefore, after finishing your experiments with LSTM networks, repeat them with SimpleRNN and GRU networks.

As usual, document your findings in a report that should contain the summary of results and their interpretation: are the results consistent with your expectations? How could you explain the differences in reported accuracies? Formulate some conclusions.

Hint: In Python, conversion from `integer` to `bit` representation can be implemented with help of the `format` function, for example:

```
[int(bit) for bit in '{:010b}'.format(287)]
```

converts the integer `287` into its 10-bit long representation: `[0, 1, 0, 0, 0, 1, 1, 1, 1, 1]`.