# Assignment 3 Task A

Deep Learning and Neural Networks

Yizhen Dai (s2395479),
Anne Liebens (s1838458),
H.C. (Bram) Otten (s2330326)

September 27, 2021

## Contents



Note: the "B" part of this assignment – about autoencoders – is only enclosed as Jupyter notebook.

# 1 Introduction

The objective of this assignment is to experiment with sequence to sequence learning implementations that can perform addition of three-digit numbers represented as single strings, such as "123+456".

We will compare different input formats, as well as examine the effectiveness of different models for our task. Sutskever, Vinyals, and Le (2014) from Google suggested that reversing the input would help the performance of Long Short Term Memory(LSTM), which is a recurrent neural network (RNN) architecture – we will test whether this is the case with our tasks as well.

## 1.1 RNN layers in Keras

For this task, we are using RNN models for its ability to handle a sequences of inputs by using outputs of internal states in handling new inputs. This is especially convenient in tasks like the prediction of the next word in a sentence – the words that occured previously make certain words more or less likely (Britz 2015). This property is also advantageous to us as the model needs to memorize the numbers both before and after the "addition" sign.

The architecture used for the assignment comes from an example given by Keras Team (Marmiesse 2019), which provides an implementation of an RNN for sequence to sequence learning that has successfully been applied to translation tasks (Sutskever, Vinyals, and Le 2014) and executing simple computer programs, including computing a sum based on string input (Zaremba and Sutskever 2014). We stuck with the categorical cross entropy loss and Adam optimizer from the example.

Three types of Keras layers (*Recurrent layers* 2019) are used in RNN models:

- **Fully-connected RNN (SimpleRNN) layer**: It provide a fully connected RNN where output is reused as input. This is a basic implementation of a recurrent neural network. However, SimpleRNN generally does

not perform well on long-range dependencies. The previous input has low impact due to vanishing gradients. To combat this issue, Gate Recurrent Unit (GRU) and LSTM are proposed.

- **LSTM layer**: Like SimpleRNN, an LSTM is able to retrieve old information to use in the current task. The LSTM has the advantage that it is able to learn long-term dependencies, where the RNN is only able to work with short-term dependencies. Like the RNN, the LSTM has a chain like architecture where modules of the network repeat itself and take as input the output of the previous model. A regular RNN is very simple, its models only consist of one layer. However, during backpropagation the problem of exploding or vanishing gradients occurs, which is undesirable. The LSTM modules contain four layers and are able to overcome this problem (Hochreiter and Schmidhuber 1997). The key property of the LSTM is the cell state, which represents the flow through the module from input to output and in between it interacts with outputs from the hidden state to transform the input into the desired output. The hidden state contains the layers of the LSTM and this is where most of the computations take place. The first layer is the forget gate that decides which parts of the cell state are not going to be used. Next, there is the input gate, which decides which parts of the cell state will be updated. If desired, it is also possible to add new values to the cell state, which is the next step. Lastly, there is the output layer, which decides what information in the cell state is relevant as output and will thus be transferred to the next module (Olah 2015).
- **GRU layer**: This is a variation of the LSTM architecture. In this network, the forget gate is combined with the input gate into a so-called update gate. Additionally, the hidden state and the cell state are merged, which results in a simpler model (Olah 2015).

It is expected that the models with LSTM and GRU layers are more efficient learners than the models that utilize SimpleRNN layers. Therefore, we hypothesize that the performance of the LSTM and GRU models will be superior to the performance of the SimpleRNN models. The simplicity of the GRU compared to the LSTM might work to its advantage, as it might not be as prone to overtraining as the LSTM model, which has more parameters. However, a simpler model is not always better, as it could lead to an inability to learn the task. Therefore, we are not sure how the GRU models will perform compared to the LSTM models in our case.

## 1.2 The seq2seq problem

The task we are dealing with is a seq2seq problem, thus we are using an RNN Encoder-Decoder. It features two RNNs – one as an encoder and the other one as a decoder pair (Cho et al. 2014). The illustration of this architecture for our addition task is shown in Figure 1.

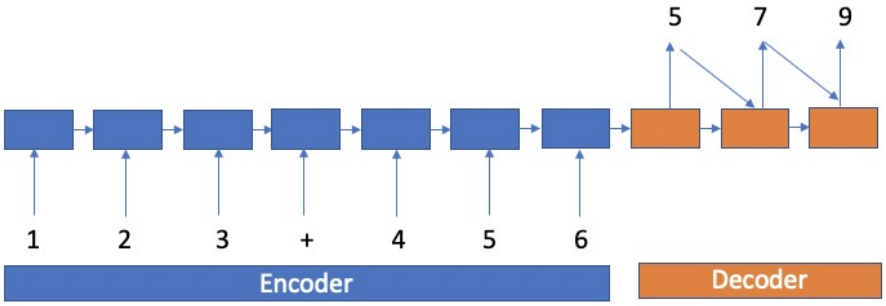


Figure 1: The illustration of Encoder-Decoder sequence to sequence model for the example input of "123+456" with target output of "579"

We can implement Encoder-Decoder RNN directly using Keras.

The Encoder is a stack of RNN units. We are using SimpleRNN, GRU or LSTM for our analysis. In our addition task, the input sequence is a sequence of digits and/or plus sign as text string from the question. The output of Encoder is a fixed-size vector, which serves as the Decoder input. The example Keras code is shown below.

```
model = Sequential()
model.add(LSTM(...))
```

The Decoder is also a stack of RNN units. It transforms the fixed-sized output from the Encoder into the desired sequence. In our case, a Dense layer is used as the output for the network. To compute the output sequence, same weights are applied in each time step by wrapping the Dense layer in the TimeDistributed wrapper. The example Keras code is shown below.

```
model.add(LSTM(..., return_sequences=True))
model.add(TimeDistributed(Dense(...)))
```

However, the sizes of encoder input and decoder output are inconsistent. The Encoder output is a 2-dimensional matrix while the Decoder input is a 3-dimensional matrix. Here comes the RepeatVector layer, which repeats the 2-dimensional matrix to create a 3-dimensional one. In our case, the RepeatVector layer allowes us to specify the length of the output vector, i.e. the max number of output digits. The example Keras code is shown below.

```
model.add(RepeatVector(...))
```

## 1.3 Loss Function and Metrics

Categorical Crossentropy, which is the crossentropy loss between the true labels and predicted categories, is used in our models since there are 10 digits as classes.

The 0/1-accuracy is tracked as metric, but it is not very informative: it denotes the accuracy *per bit*, which will usually be orders of magnitude larger than the accuracy on a string of 11 bits (it would be an accomplishment to get the former below 0.5). Note that the loss function used during training is very different from those used in (common sense) evaluation. For example, the (categorical entropy) loss of predicting a 9 where there should be a 0 early in the outcome number may not be large while the difference with the true outcome is.

To evaluate the model performance on the additions task of three-digit numbers represented as single strings, three metrics were computed on the complete set (i.e., the data set with all possible addition of three-digit numbers):

- total percentage of errors (i.e., wrong evaluations or 0/1-loss expressed as percentage);
- mean square error (MSE: $\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$);
- mean absolute error (MAE: $\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$)

## 1.4 Structure of our experiments

We firstly followed the tutorial provided by Marmiesse (2019). The structure in the tutorial consists of one LSTM layer as the Encoder and one LSTM layer as the Decoder (The LSTM layer will be replaced with SimpleRNN or GRU in further experiments). The size of the Encode output (i.e. the hidden vector) is $(N, 128)$, where $N$ is the data set size. The RepeatVector layer is used to transform the $N \times 128$ matrix to a $k \times N \times 128$ one, which is used as the input of the Decoder. The timestep we use here is $k$, which is the number of digits for addition results. $k$ is 4 for decimal input, and 11 for bit input.

There are four tasks, each requiring a different form of input. Tasks 1 and 2 work for normal sums and reversed sums. In Task 1 these sums are presented to the model in decimal form, while in Task 2 a bit representation is used. Tasks 3 and 4 make use of pairs of digits, where hundreds, tens and ones are paired together, for example, “123+456” is represented as “142536”. Empty spaces that occur due to a number being smaller than 100 are padded with spaces in the front of the input string.

The set up of the models we experimented are shown in Table 1. The results will be discussed in details in Section 2.

| | Input Type (string) | Input Presentation | Padding | RNN Layer |
|---|---|---|---|---|
| pre-Task 1 | Decimal Number | Normal/Reverse | Space in front | SimpleRNN/LSTM/GRU |
| pre-Task 2 | Binary Number | Normal/Reverse | Space in front | SimpleRNN/LSTM/GRU |
| pre-Task 3 | Decimal Pairs | Normal/Reverse | Space in front | SimpleRNN/LSTM/GRU |
| pre-Task 4 | Binary Pairs | Normal/Reverse | Space in front | SimpleRNN/LSTM/GRU |

Table 1: The Model Setup for Preliminary Experiments Based on Marmiesse (2019). Note: "pre-"="preliminary"

To improve the results, we add an additional RNN layer in the Decoder. Additionally, the padding method was modified: instead of padding with spaces for the whole string, zeroes were used for each number in the string. For example, " 123+45" will be modified to "123+045" as input. While training time was kept equal (200 epochs), the batch size was decreased to 64. The adam optimizer and categorical crossentropy loss were still used. The motivation will be discussed in Section 2.

The set up of our proposed models are shown in Table 2. We will elaborate on the improved results and their interpretations in Section 3,4,5 and 6.

| | Input Type (string) | Input Presentation | Padding | RNN Layer |
|---|---|---|---|---|
| Task 1 | Decimal Number | Normal/Reverse | 0's in front of each number | SimpleRNN/LSTM/GRU |
| Task 2 | Binary Number | Normal/Reverse | 0's in front of each number | SimpleRNN/LSTM/GRU |
| Task 3 | Decimal Pairs | Normal/Reverse | 0's in front of each number | SimpleRNN/LSTM/GRU |
| Task 4 | Binary Pairs | Normal/Reverse | 0's in front of each number | SimpleRNN/LSTM/GRU |

Table 2: The Proposed Model Setup

## 2 Preliminary experiments

The models were built using the architecture provided by Marmiesse (2019) and were trained for 200 epochs using Keras and the TensorFlow backend with a batch size of 128. The adam optimizer is used in combination with categorical crossentropy loss. The networks are trained on a random sample of 50000 sums (which is 5% of the approximately $10^6$ possible sums). After training on 50000 examples, the models were tested on all cases ($10^6$ with digits in task 1 and 3, $2^{20}$ with bits in pre-Task 2 and 4). The results for models in pre-Task 1 and 2 are shown in Table 3.

| | Non-Reverse | | | Reverse | | |
|---|---|---|---|---|---|---|
| | SimpleRNN | LSTM | GRU | SimpleRNN | LSTM | GRU |
| Task 1: decimal digit representation | | | | | | |
| Error % | 16.140 | 1.680 | .821 | 18.150 | 0.079 | 0.505 |
| MSE | 137103 | 29161 | 11097 | 68911 | 1359.740 | 4039.230 |
| MAE | 22.540 | 4.817 | 1.907 | 14.840 | 0.196 | 0.698 |
| Task 2: bit digit representation | | | | | | |
| Error % | 97.120 | 46.860 | 48.260 | 81.840 | 5.653 | 5.108 |
| MSE | 6522 | 1978 | 2046 | 11204 | 27.860 | 11.850 |
| MAE | 33.530 | 10.100 | 9.938 | 33.750 | .353 | 0.165 |

Table 3: Error measures on models for first two pre-tasks with space padding.

The best performed models with the lowest error percentages for each pre-task are shown in Table 4.

| | pre-Task 1 | pre-Task 2 | pre-Task 3 | pre-Task 4 |
|---|---|---|---|---|
| Best Model | LSTM Reverse | GRU Reverse | GRU Reverse | SimpleRNN Reverse |
| Error % | 0.079 | 5.107 | 0.008 | 50.535 |
| MSE | 1359.739 | 11.846 | 0.324 | 796.842 |
| MAE | 0.196 | 0.165 | 0.004 | 1.730 |

Table 4: Models with lowest error percentages.

It should be noted that while the GRU reverse model performed relatively well for pre-Task 2, the other models did not perform comparably: the LSTM reverse model had 5.652% errors, but after that the next best performing model is the LSTM regular model with 44.857% errors. Because not all models were able to perform well, we decided to apply some enhancements.

We started off with appending spaces as in the provided example. We have also gathered all results with these space-appending models. The results for task 1 and 2 – where they are quite different from zero-prepending – are shown in table 3. Some of our observations may actually only hold for the space-padding, usually worse versions. These are more dependent on an element of memory, or modulating network state based on states a few elements of the input back.

The results of our proposed models will be discussed in the following sections.

## 3 Task 1

For this task the input is provided in decimal representations, namely in the form of "123+456". Additionally, the model is trained to work with reversed input in the form of "654+321". If a number consists of fewer than three digits, zeroes are prepended to make its length three. A similar process makes all outputs four digits long.[1]

In order to verify the performance of the model, it was tested on all 1000000 three digit sums. Table 5 shows the results after training and testing models with the three different types of layers described in Section 1.

[1] The "+" is thus superfluous; it is always at the same position in the input.

| | Non-Reverse | | | Reverse | | |
|---|---|---|---|---|---|---|
| | SimpleRNN | LSTM | GRU | SimpleRNN | LSTM | GRU |
| Task 1: decimal digit representation | | | | | | |
| Error % | 0.444 | 0.004 | 0.012 | 0.007 | 0.001 | 0.005 |
| MSE | 251.920 | 0.061 | 0.484 | 0.333 | 0.150 | 0.135 |
| MAE | 0.373 | 0.001 | 0.005 | 0.003 | 0.002 | 0.002 |
| Task 2: bit digit representation | | | | | | |
| Error % | <0.001 | .03947 | 0.0110 | 0.001 | 0.000 | 0.012 |
| MSE | 225.526 | 148.740 | 31.232 | 1.455 | 0.000 | 204.934 |
| MAE | 0.371 | 1.221 | 0.372 | 0.022 | 0.000 | 0.779 |
| Task 3: decimal digit pairs representation | | | | | | |
| Error % | 1.110 | 0.003 | 0.038 | 0.003 | 0.002 | 0.001 |
| MSE | 202.203 | 0.092 | 0.127 | 0.012 | 0.081 | 0.009 |
| MAE | 0.719 | 0.001 | 0.004 | <0.001 | 0.004 | 0.0002 |
| Task 4: bit digit pairs representatio | | | | | | |
| Error % | 0.526 | 1.415 | 0.107 | 0.002 | 0.011 | 0.084 |
| MSE | 2468.809 | 34.078 | 12.341 | 3.910 | 0.102 | 1.563 |
| MAE | 3.067 | 0.354 | 0.085 | 2.575 | 0.002 | 0.029 |

Table 5: Error measures on models for all the tasks with zero paading.

Table 5 shows that the model built with LSTM layers performs best out of the three when applied to both types of decimal input, as it has the lowest percentage of errors on the test set, with the GRU being second best. The model built with SimpleRNN layers performs worst on both input types, however, it does show the most improvement when switching from regular decimal input to reversed decimal input. The fact that the LSTM is so much better than the SimpleRNN could be due to the fact that the LSTM is not constrained to short-term dependencies and is thus able to learn more efficiently. As the GRU is an extension of the LSTM, this reasoning also applies to the superiority of the GRU models compared to the SimpleRNN models. It is also noted that MSE is more sensitive to the model performance. It discriminates different models more than Error Percentage and MAE, i.e. visibly deviates between different models.

The differences in performance between the regular decimal models and the reverse decimal models are quite high, for example, the Error Percentage of SimpleRNN decreased by 63 times, while the only difference is that the input is reversed. It is consistent with our expectation. The possible reason is that reversed decimal representation provides shorter dependence between the input sums and their answers.

## 4 Task 2

Task 2 was very similar to the Task 1. However, for this task the model from Marmiesse (2019) was modified to work with a regular and reverse bit representation of integers. The two input numbers are each ten bits long, and the outputs consist of eleven bits. These lengths are forced by prepending with zeroes as in Task 1. The models were trained with a batch size of 256 for 200 iterations (though SimpleRNN can take fewer), on 50000 sums and evaluated on a random subset of 100000 of the ($2^{20}$) sums of 10 bit numbers.[2] The results for the experiments for this task are found in Table 5.

Overall, the MSEs of Task 2 are quite high in comparison to the ones in Task 1, indicating that there are some large errors that have more effect on the MSE. The bit digit representation might help improve the accuracy, at the expense of making possible huge mistakes.

The models with bit input perform better than decimal input for SimpleRNN models with lower Error Percentage. Without the ability to learn the long sequence, SimpleRNN takes advantage of smaller input and output sizes.

Reversing the input for LSTM model significantly improve the accuracy . This is consistent with the above-mentioned paper from Google.

## 5 Task 3

In this task, the input of the network is also altered from Task 1. Instead of providing a sum as a string, the input is presented as pairs of digits, where the hundreds, tens, and ones are paired together. This means that "123+456" is now represented as "142536". The input can also be reversed for "362514". The networks are trained for 200 epochs on 50000 sum with a batch size of 64. The results are shown in Table 5.

[2] The reason for this smaller test set is a late detection of inconsistent results.

The results show that overall, the input in pairs performs better than the input in regular sequence in Task 1. The improvement may come from the fact that the related digits (such as the digits both on hundred position) are closer to each other in Task 3, leading to shorter dependence. Also, the decrease in input string length (since we get rid of the plus sign ("+")) may also help.

Also, the reverse representation allows the models to learn more efficiently, especially for SimpleRNN model with drastic improvements in both the error percentage and MAE criteria. This improvement is also seen in Task 1 that the reverse decimal representation works better for the models.

# 6 Task 4

This task blends the previous two: input consists of pairs of bit digits. "10010+1011" is represented by the five pairs (1,0), (0,1), (0,0), (1,1), and (0,1). As also seen in the other tasks, the models are trained with a batch size of 64 for 200 iterations on 50000 sums, and evaluated on all paired sums of ten bit strings. The results are shown in Table 5.

Similar to the decimal representation, the pairs representation is not advantageous than the regular sum representation for bit representation. Interestingly, the SimpleRNN model is able to significantly outperform the LSTM and GRU models on the reversed input, even though its MSEs and MAEs are still higher.

# 7 Conclusion

For this assignment we experimented with SimpleRNN, LSTM and GRU networks and their ability to predict the outcome of sums. These sums were represented in different ways: regular decimal representation, bit representation, a paired decimal representation and a paired bit representation. Additionally, the models were also trained and tested on reversed versions of these representations.

## 7.1 Decimal or Bit

Generally, the models with decimal input outperform the ones with bit input. Longer sequences from bit representation may increase the issue of vanishing gradients, giving the models a hard time tuning the parameters in the previous layers.

One excpetion is SimpleRNN model, for which the big digit representation works much better. This may due to that SimpleRNN cannot reply on the long dependence but takes the advantage of small input and output vector size.

## 7.2 Pair or not pair the digits

Pairing, counter-intuitively, cannot help improve the model performance, even for bit representation. One of the reasons might be that pairing disrupts the pattern that could have been learned by the RNN.

## 7.3 Reverse or not reverse the input

In our case, reversing input generally improves the model performance. This is consistent with the paper presented by Sutskever, Vinyals, and Le (2014). They proposed reversing input for RNN. It was pointed out that reversing the input words but not the target sentences was one of the key tricks for their success on handling long sentences. There is no complete explanation, but one of the reasons may be that the reversed input order introduces shorter term dependencies, which simplifies the optimization problem. After reversal, the beginning of the source string is closer to the beginning of the target sentence. As a result, "the problem's minimal time lag is greatly reduced".

## 7.4 SimpleRNN, GRU or LSTM

Overall, it was found that the SimpleRNN models tended to perform worst, which was expected as the LSTM and GRU are supposed to be improved versions of the SimpleRNN. The performance difference between the LSTM and the GRU was inconsistent, meaning that one did not perform better than the other on all or most tasks. The GRU is a simplified extension of the LSTM and it was expected that its simplicity could work both in its advantage, as it was less likely to overfit, and to its disadvantage, as it might not be able to properly learn the task. It seems that both these extremes in performance were demonstrated in this assignment, indicating that implementation of the GRU should be done with care and after careful consideration of the task that it is supposed to learn to ensure that this task is a good fit for the GRU network.

Also, it was found that SimpleRNN performs best in Task 2 (Reverse or not) and Task 4 (Reverse). Future research could focus on why SimpleRNN performs better on digit representation.

### 7.5 Error %, MAE or MSE

The MSEs of the models vary greatly. However, this measure does not necessary define the true quality of the model, as it was found that in some cases models with lower MSE's had higher 0/1 error than models with higher MSE's. This is an interesting finding that might be investigated further as it would seem that models with lower MAE's should be able to quickly reduce their error percentage when trained further, as they are already able to consistently predict both accurately and precisely. It could also be considered a slightly unexpected finding as intuitively it is not straightforward to think that the model that is less accurate is actually to predict the true value more often.

MSE, with a squared calculation, is more sensitive to large errors than MAE. If this is much bigger than MAE it may indicate the existence of a few significantly bad misclassifications. Therefore, if we care about avoiding huge mistakes, we may prefer MSE as the metric.

## References


Britz, D. (2015). *Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs*. URL: http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/ (visited on 09/17/2015).

Cho, K., B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio (2014). "Learning phrase representations using RNN encoder-decoder for statistical machine translation." In: *arXiv preprint arXiv:1406.1078*.

Hochreiter, S. and J. Schmidhuber (1997). "Long short-term memory." In: *Neural computation* 9.8, pages 1735–1780.

Marmiesse, G. de (2019). *Addition RNN*. URL: https://github.com/keras-team/keras/blob/master/examples/addition_rnn.py (visited on 01/24/2019).

Olah, C. (2015). *Understanding LSTM Networks*. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/ (visited on 08/27/2015).

*Recurrent layers* (2019). Retrieved from: https://keras.io/layers/recurrent/. (Visited on 04/20/2019).

Sutskever, I., O. Vinyals, and Q. V. Le (2014). "Sequence to sequence learning with neural networks." In: *Advances in neural information processing systems*, pages 3104–3112.

Zaremba, W. and I. Sutskever (2014). "Learning to execute." In: *arXiv preprint arXiv:1410.4615*.