Neural Networks 2019 - 2020 Assignment 3A

Deadline May 24th 2020

1 Introduction

Predicting the future is a very important task in various contexts, e.g. when predicting stock prices in finance or anticipating a cars trajectory in autonomous driving systems. In this report, we will discuss so called recurrent neural networks (RNNs), a class of neural networks that is able to predict the future. In order to do so, they need to be able to work with sequential data such as stock prices, sentences, documents or audio samples of arbitrary length, which distinguishes them from previously discussed classes of neural networks, that worked with a fixed-size input. We will briefly discuss the fundamental concepts underlying RNNs and how to train them using backpropagation through time. Difficulties when training RNNs and how to solve them using different types of architectures are discussed in detail by means of an accompanying RNN implementation. In our implementation we demonstrate that a RNN is able to learn a simple arithmetic task like addition. More specifically, we demonstrate that a RNN is able to learn how to add two 3-digit long integers. For example, on an input string of 7 characters "123+456" the network should return the sequence "579". Instead of memorizing the whole multiplication table, the network is supposed to learn the general principles behind the addition operation in such a way that it is able to predict the correct answer for the addition. We will experiment with different architectures and several ways of representing the input-output relation in order to find the most efficient and accurate addition network. The different input-output relations will be discussed later on. We will start with a brief description of recurrent neurons and layers, followed by a comparison of the different types of RNNs before we implement the addition networks. Finally, performances will be presented and discussed.

2 Recurrent Neural Networks

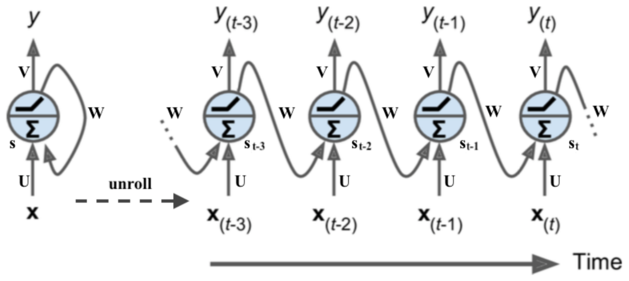

Figure 1: A recurrent neuron (left) unrolled through time (right). Géron 2019, p. 498.

Compared to the networks that we have discussed so far, the key difference in a RNN is that it also has connections that are pointing backwards. Therefore, the activations in a RNN are not only flowing in one direction from the input layer to the output layer, the output of a neuron is also sent back to itself. Consider a single recurrent neuron, at each time step t. These recurrent neurons receive not only the inputs $\mathbf{x}_{(t)}$ but also their own output from the previous time step $y_{(t-1)}$. In Figure 1, the simplest possible RNN consisting of one neuron is shown on the left, and its representation against the time axis is shown on the right. Representing a RNN against the time axis is also called *unrolling the network through time*, since it is the same recurrent neuron represented once per time step. Considering a layer of recurrent neurons, at each time step t, every neuron now receives two vectors, the input vector $\mathbf{x}_{(t)}$ and the output vector $\mathbf{y}_{(t-1)}$ from the previous time step. By placing all the inputs at time step t in an input matrix $\mathbf{X}_{(t)}$, the output of a recurrent layer for a whole mini-batch can be computed as [1]:

$$\begin{aligned}\mathbf{Y}_{(\mathbf{t})} &= \phi\left(\mathbf{X}_{(t)}\mathbf{W}_x + \mathbf{Y}_{(t-1)}\mathbf{W}_y + \mathbf{b}\right)\\ &= \phi\left(\left[\mathbf{X}_{(t)}\mathbf{Y}_{(t-1)}\right]\mathbf{W} + \mathbf{b}\right) \text{ with } \mathbf{W} = \begin{bmatrix}\mathbf{W}_x\\ \mathbf{W}_y\end{bmatrix}\end{aligned}$$

where

- $\mathbf{Y}_{(\mathbf{t})}$ is an $m \times n_{\text{neurons}}$ matrix containing the layer's outputs at time step t for each instance in the mini-batch and m is the number of instances and n_{neurons} is the number of neurons.
- $\mathbf{X}_{(t)}$ is an $m \times n_{\text{inputs}}$ matrix containing the inputs for all instances and n_{inputs} is the number of input features.
- $\mathbf{W}_x$ is a $n_{\text{inputs}} \times n_{\text{neurons}}$ matrix representing the connections weights for the inputs of the current time step.
- $\mathbf{W}_y$ is a $n_{\text{neurons}} \times n_{\text{neurons}}$ matrix representing the connection weights for the outputs of the previous time step.
- $\mathbf{b}$ is a vector of size n_{neurons} representing each neuron's bias term.
- ϕ is the activation function, typically the hyperbolic tangent (tanh) is prefered [1].

Thus, $\mathbf{Y}_{(t)}$ is a function of all the inputs since $t = 0$. Note that there are no previous outputs at $t = 0$, so they are typically assumed to be all zeros [1]. If the output of a recurrent neuron at time step t is a function of all the inputs from the previous time steps, one could say it has some form of *memory.* A *memory cell* is a part of a neural network that preserves some state across time steps. We denote a cell's state at time step t as $\mathbf{h}_{(t)}$, where $\mathbf{h}_{(t)}$ is a function of the inputs at time step t and its state from the previous time step:

$$\mathbf{h}_{(t)} = f(\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)}).$$

The cell's output $\mathbf{y}_{(t)}$ is a function of the previous state and the current inputs as well. For the simple cell we have discussed so far, the output is equal to the state. In order to train an RNN, the network is unrolled through time as in Figure 1 and then regular back-propagation is used. This procedure is called *backpropagation through time* (BPTT). In a forward pass through the unrolled network the outputs are calculated as described above. Then a cost function $C(\mathbf{Y}_{(0)}, \mathbf{Y}_{(1)}, ..., \mathbf{Y}_{(T)})$ (T is the max time step) is applied to evaluate the output sequence. The gradients of that cost function are then propagated backward through the unrolled network and the model parameters are updated according to these gradients.

Consequently, if we want to train an RNN on long sequences, the unrolled RNN becomes a very deep network and the problem of unstable gradients may occur. Moreover, a simple RNN will gradually forget the first inputs from long sequences. Solutions to prevent gradients from vanishing vary, but most commonly refer to appropriately initialize weights, make use of different activation functions or regularization approaches by putting constraints on weights. The problem of the short-term memory was solved by introducing another slightly more complex memory cell called *Long Short-Term Memory* (LSTM). Compared to the basic memory cell it performs much better and is able to detect long-term dependencies in the data. The key difference is that a LSTM cell splits its state into two vectors $\mathbf{h}_{(t)}$ and $\mathbf{c}_{(t)}$ for the short-term state and the long-term state respectively. Then, the network is supposed to learn what to store in the long-term state, which memories can be dropped and which memories should be added. At each time step t, the LSTM cell receives the current input vector $\mathbf{x}_{(t)}$, the previous short-term state $\mathbf{h}_{(t-1)}$ and the previous long-term state $\mathbf{c}_{(t-1)}$. The input vector $\mathbf{x}_{(t)}$ and the previous short-term state $\mathbf{h}_{(t-1)}$ are then fed into 4 different fully connected layers which work as *gate controllers*. They "control" which parts of the long-term state should be erased (forget gate), which parts of the input vector $\mathbf{x}_{(t)}$ should be added (input gate) to the long-term state and which parts of the long-term state should then be read and output (output gate) as $\mathbf{h}_{(t)}$ and $\mathbf{y}_{(t)}$. Thus, an LSTM cell is able to learn and recognize long-term patterns successfully.

Another memory cell that is also capable of capturing long-term patterns is the *Gated Recurrent Unit* (GRU). It is a simplified version of the LSTM cell which performs just as well. The difference between LSTM and GRU cells is that in GRU cells both state vectors are merged into a single vector $\mathbf{h}_{(t)}$ again. The gate controller work differently in such a way that a single gate controller controls the forget gate and the input gate and there is no output gate at all, meaning that the full state vector is output every time step. However, there is another gate controller that controls which part of the previous state will be considered in the computation of the output. In the following chapter, we will compare the performance of the different memory cells by implementing an addition RNN.

3 Arithmetic Addition RNN

We implemented a RNN that learns how to add two at most 3-digits long integers. As mentioned in the introduction, the network takes an input string of 7 characters, e.g. "123+456" and should return a sequence of characters representing the answer to the addition, here "579". We experimented with different input-output relations in order to find the most efficient and accurate addition RNN. Specifically, we tried four different approaches of encoding our strings:

1. As a string of integers, e.g. "123+456". This is the representation how one would type it into a computer.
2. As a string of binary numbers, e.g. "10010+1011". This could be more natural for a computer to process.
3. As pairs of integers, e.g. the string "123+456" would be represented as ((1,4),(2,5),(3,6)), this is close to how a human learns addition. This encoding makes it easy for the network to taking into account the place value ("thousands", "hundreds", "tens", a.s.o.). While this therefore should not necessarily be expected to improve the accuracy of our networks classifications, it could decrease the mean squared error and mean absolute error, as even incorrect classifications of one digit can lead to relatively close decoded numbers and thus a small error.
4. As pairs of binary data, which is the same as the above pairings only for the binary encoding.

Furthermore, we used each of the four methods above on normal and reversed strings. In the reversed order, "123+456" would thus be represented as "654+321". In natural language processing, reversing the input can lead to better results. This is largely due to the fact that key information might be contained in the first few words of a sentence and the network suffers from short-term memory loss. For these 8 different encodings, we compare the above mentioned three models: SimpleRNN, LSTM and GRU. Therefore, we compared 24 models in total. We trained each model on a random sample of size 50,000, where our validation set consisted of 5,000 samples. Generally, the accuracy we get on the validation set can be seen as a good approximation of the accuracy we expect to see on new data. However, in this task we already know all possible new data which is just the set of all possible additions with 3 + 3 digits ($10^{3\cdot 2}$) or 10 + 10 bits ($2^{10\cdot 2}$). Thus, if we take our models, which are trained on the 50k randomly selected samples and evaluate them on the corresponding complete set of possible additions, we can say something about the "true accuracy" of the models. The true accuracy is the percentage of correctly predicted results of adding two numbers with at most 3 digits. Additionally, we also measured the mean squared error (MSE) and the mean absolute error (MAE) which say something about how close the predictions of the model are to the true value on average. We first analyze the different models for each of the encodings, then compare the different encodings and finally conclude what combination of encodings and models provided the best results.

3.1 Addition with Integer Representation

Let us first start with looking at the training process of our 3 models and the integer representation. In Figure 2 we can see that all models reached an accuracy on the validation set of 0.9 within the first 25 epochs. The GRU and the LSTM reached a training accuracy of 1.00. The worst model was the SimpleRNN on the regular order. The slowest model was the GRU on encodings in the normal order.

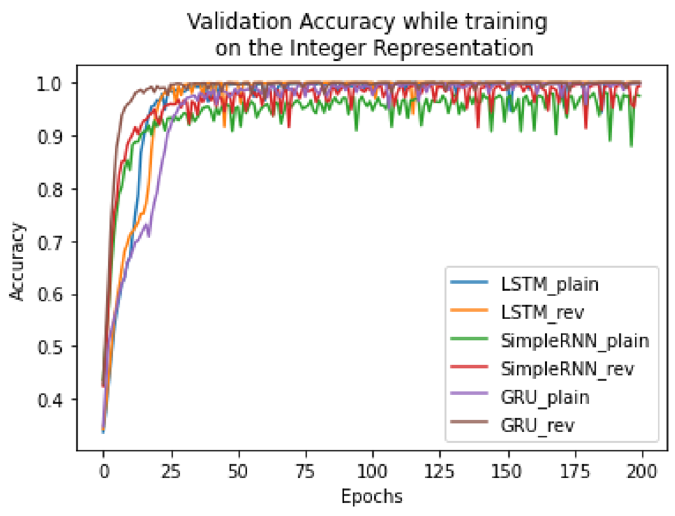

Figure 2: Validation Accuracy while training for 3 different RNNs, for plain and reversed data in the Integer Representation

We then evaluated the six models on the test set, consisting of all 1,000,000 combinations of additions of 3 + 3 digits. The results can be seen in Table 1. We can see that LSTM on the reversed data achieves the best accuracy with 0.999. The worst accuracy is achieved by the SimpleRNN with 0.84. From this first experiment it looks like memory-loss might adversely affect the SimpleRNN models, as LSTM and GRU perform significantly better. Furthermore, all models on the reversed data perform better than the models on the non-reversed data. This indicates that the more useful information is located in the beginning of the inputs. Indeed, the way the training data was generated, the useful information is located in the beginning. The length of the input strings is always 7 and for all the additions of numbers with less than 3 digits empty space is added until maximum length. For instance, the input string for the addition 2 + 3 is "2+3   ". In reversed order this string becomes

	LSTM_plain	LSTM_rev	SimpleRNN_plain	SimpleRNN_rev	GRU_plain	GRU_rev
ACC	0.991	**0.999**	**0.840**	0.958	0.986	**0.995**
MSE	18914	2601	82362	15667	34158	5744
MAE	3.074	0.391	16.623	3.393	5.003	0.957

Table 1: Results of training different RNNs on an Addition task with integer Representation. Rev are the models that are trained on the reversed representation.

" 3+2". Consequently, the useful information is now in the end and memory loss is not a problem anymore. Thus, the models can use the provided information more efficiently which ultimately leads to more accurate results.

3.2 Addition with Bit Representation

Now we look at another representation of the input. The integers used for the addition are now transformed into their binary representation. The example string "123+456" now becomes "0001111011+0111001000". The three different models are trained again on 50k randomly selected samples and their reversed order. The validation accuracy during training is shown in Figure 3.

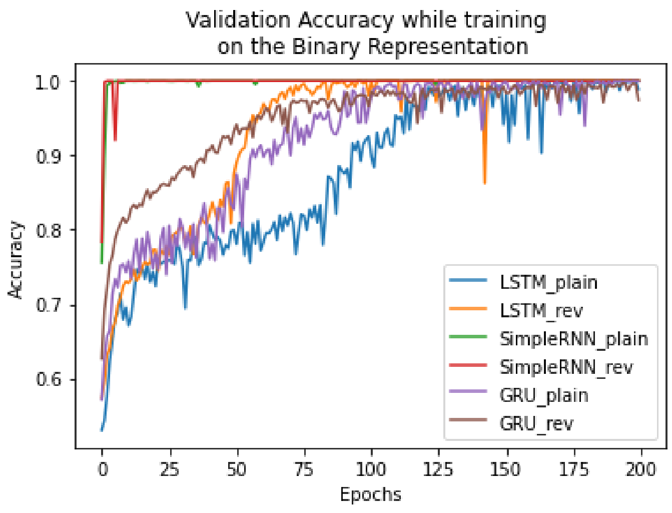

Figure 3: Validation Accuracy while training for 3 different RNNs, for plain and reversed data in the Bit Representation

The two SimpleRNN networks converge very fast. Learning the function seems to be relatively trivial for them. This is all the more surprising as the strings are longer in the bit-representation and we would therefore expect the Simple RNN's to suffer more from memory loss. In the Binary representation, the three different models seem to learn at vastly different speeds, with the SimpleRNN being the fastest and the LSTM with normal order being the slowest. Ignoring the effect of the order of encoding, this corresponds to the number of parameters that are trained, as the LSTM has about 200,000 parameters and the SimpleRNN about 50,000.

Again, the trained models are evaluated on the complete set of all possible additions. Here, the complete set of possible additions is of size $2^{20} = 1048576$ and represents all possible additions of integers up to 1023 in their 10-bit representation. The results are shown in Table 2. We observe that the LSTM and GRU models reached a test accuracy of 1 on the reverse order of the inputs, where the LSTM model performs slightly better than the GRU model when comparing the alternative error measures which are both 0.0 for the LSTM model. Thus, the LSTM model provides perfect predictions on the complete set. The models are performing better on the reverse order of inputs again , except for the SimpleRNN. This result is against our expectations and is most likely explained by memory loss. However, despite the fast convergence of the SimpleRNNs, they are outperformed by the accuracy of the other models eventually.

	LSTM_plain	LSTM_rev	SimpleRNN_plain	SimpleRNN_rev	GRU_plain	GRU_rev
ACC	0.998	**1.000**	0.996	0.943	0.960	**1.000**
MSE	4406.1	**0.0**	981.0	330220.1	47201.0	**0.4**
MAE	0.697	**0.000**	0.331	459.893	1.238	**0.004**

Table 2: Results of training different RNNs on an Addition task with Bit Representation.

3.3 Addition of Pairs of Digits

Next, we are going to look at addition of pairs of digits, here our example problem of 123+456 is encoded as (1,4),(2,5),(3,6). As mentioned earlier, this could help the model to add the hundreds together first, then add then add the tens, and last the ones. In theory, the encoding should thus make it significantly easier to get a result that is close to the target string.

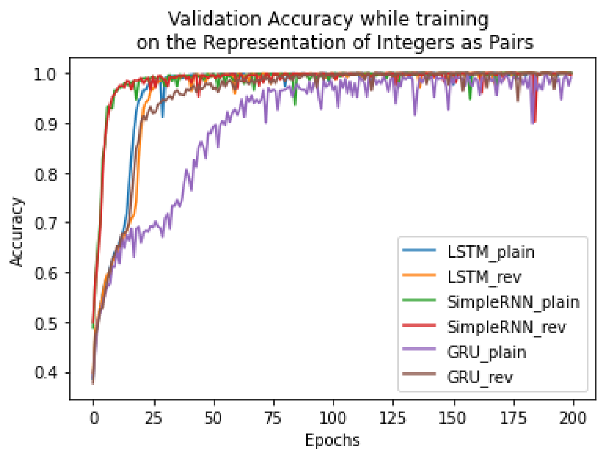

Figure 4: Validation Accuracy while training for 3 different RNNs, for plain and reversed data in the representation of paired integers.

In Figure 4 we can see that again our SimpleRNN significantly outpaced the other models. Interestingly, the GRU on the encoding of the normal order was significantly slower than the GRU on the reversed order. The reason for this effect is unclear to us, but could have to do with the model having a hard time to understand uninformative paddings. Furthermore, the GRU on the normal data seems to be relatively unstable as there continue to be large negative spikes even after 100 epochs of training.

In the results in Table 3 we can see that our results are extremely close to each other. Two models achieve a perfect accuracy of 1. Furthermore, by now we can confirm the general trend that the GRU and LSTM models outperform our simple RNN and that the reverse coding outperforms the normal encoding. However, it seems like the encoding as pairs helped the model to learn the task of addition more easily.

	LSTM_plain	LSTM_rev	SimpleRNN_plain	SimpleRNN_rev	GRU_plain	GRU_rev
ACC	0.998	**1.000**	0.996	0.998	0.960	**1.000**
MSE	4406.073	0.002	981.009	4382.8	4720.948	0.444
MAE	0.697	0.000	0.331	459.893	1.238	0.004

Table 3: Results of training different RNNs on an addition task with paired integer representation.

3.4 Addition of Pairs of Binary Numbers

Our final encoding tested was pairs of binary digits. Our example string ”10010+1011” would thus be represented as “(1,0), (0,1), (0,0), (1,1), (0,1)”. Addition in the base two system works exactly as in base 10. As our paired representation of integers outperformed the non-paired representation, and as our the binary representation achieved better results than the base 10 representation, we would expect this encoding to work best.

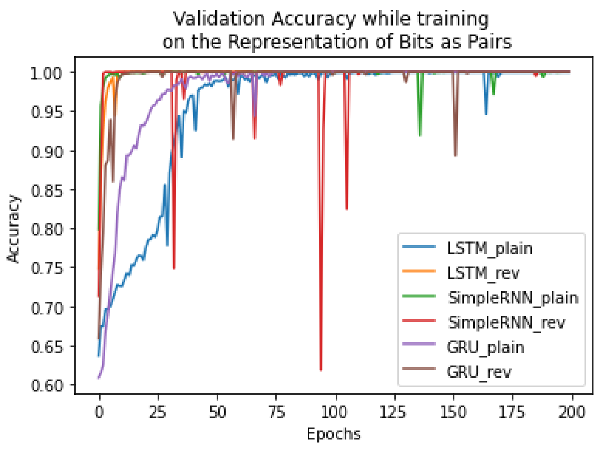

Figure 5: Validation Accuracy while training for 3 different RNNs, for plain and reversed data in the binary representation as pairs.

The first thing one notices when looking at Figure 5 are the huge negative spikes, most noticeably present in the SimpleRNN on the reverse order. The model shoots up to an accuracy of almost 1 after just a few epochs, and stays there for most of the time, but sometimes breaks out downwards. The exact reasons for this are unclear to us but could be related to the random batches used in training. As batches change randomly, there is a slight change of the model learning something about a batch that is not generalizable to the whole dataset. This explanation is supported by the fact that these spikes are not present in the training accuracy. Furthermore, it should be noted that when comparing the training process of the previous models the binary encodings exhibited larger negative spikes. When comparing the speed of convergence, we now see the LSTM and GRU, both on the regular ordered data, take more epochs until convergence than the other models.

	LSTM_plain	LSTM_rev	SimpleRNN	SimpleRNN_rev	GRU_plain	GRU_rev
ACC	**0.987**	**1.000**	**0.999**	**1.000**	**0.999**	**1.000**
MSE	15.382	0.021	78.717	0.027	1.523	0.117
MAE	0.268	0.000	0.140	0.001	0.033	0.003

Table 4: Results of training different RNNs on an addition task with paired binary representation.

Table 4 shows the results of our models from Task 4. The encoding as binary pairs indeed combined the previously noted advantages of the other models and outperforms them. We can see that the worst model, the LSTM on the non-reversed data, still achieved an impressive accuracy of 0.987 on the test set. All other models achieved an almost perfect or perfect accuracy of 1.

3.5 Discussion of Results

Let us briefly summarize some of the phenomenon that we saw in the four tasks. When comparing the different encodings, we saw that base 2 encodings outperformed base 10 encodings. Furthermore, encodings as pairs outperformed non-paired encodings. Consequently, the encoding of paired binary numbers performed the best. When comparing the models, we saw three patterns that were present in all tasks almost without exception: First, the reversed encoding achieved higher accuracy than the plain encoding. Secondly, LSTM and GRU achieved higher accuracies than the SimpleRNN. Thirdly, the SimpleRNN often converged quicker. Let us now outline some possible explanations for the omnipresence of these phenomena:

The first phenomenon, the superiority of the reverse coding, is perhaps the biggest surprise. As addition is commutative ($a+b=b+a$), the first three digits a are as important as the latter three digits b. However, as we padded our strings to always be the same length, we added uninformative spaces to the end. This changes the balance of information and might be the reason for the superiority of the reversed encodings.

The second phenomenon, the superiority of LSTM and GRU, is less surprising. LSTMs are more insensitive towards the gap-length of the input and thus can handle longer data. However, whether a string of length seven is counted as a long sequence is a matter of debate. Our results seem to indicate that LSTMs and GRUs show improvements of accuracy over Simple RNNs even on sequences of moderate length.

The third phenomenon, the quicker convergences of our SimpleRNN is the least surprising. This model only has 1/4 of the parameters of the LSTM models and 1/3 of the GRU models. This means that the model is able to find a local minimum significantly faster.

In conclusion, we can say that this assignment highlighted the potential of RNNs on sequence data and the advantages and disadvantages of different RNN architectures. While some results merely supported the theory, other results came as a surprise and are worth further investigation.

References

[1] Géron, Aurélien. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* O'Reilly Media, 2019.