Project 3

Edvard B. Rørnes* and Isak O. Rukan†
Institute of Physics, University of Oslo,
0371 Oslo, Norway
(Dated: December 2, 2024)

Abstracting very cool

CONTENTS

1. INTRODUCTION

2. THEORY

2.1. Recurrent Neural Networks

The Recurrent Neural Networks (RNNs) are a class of neural networks specifically designed to handle sequential data or data with temporal dependencies. Unlike traditional FeedForward Neural Networks, RNNs are capable of "remembering" information from previous time steps. This is done through the so called 'hidden state', which acts as a form of memory by retaining information about prior computations. The hidden state is essentially an array of data that is updated at each time step based on the input data and the previous hidden state. Alhough this enables RNN to access the temporal dependencies of the data at hand, it greatly increases the commputation time compared to that of the FFNN.

2.1.1. Structure

The RNN processes input sequentially, with information flowing step-by-step from the input to the output. The network consists of J neurons, each with associated weights and biases $\{\boldsymbol{w}_j, \boldsymbol{b}_j\}_{j\in J}$. These parameters govern the transformation of inputs to outputs and are conceptually similar to those in the FFNN.

The key difference between FFNN and RNN is that for each timestep t_i, the RNN maintains a hidden state $\boldsymbol{h}_i$ which acts as a memory of previous computations. The hidden state is initialized as the zero vector for each propagation through the network, updated at each step using both the current input and the hidden state from the previous (time) step. This update is parameterized by additional weights $\{\boldsymbol{w}_j^{(h)}\}_{j\in J}$, which give the contribution from the hidden state to the current update.

2.1.2. General Algorithm

Consider some general data ouput y, of shape (N, p_{out}) and some data input X, of shape (N, p_{in}), where N corresponds to the total amount of time points, and $p_{\text{out}},\ p_{\text{in}}$ the dimension of the output and input, respectively. The RNN then split the data into 'windows' of size N_W in time. Then for the i-th window, the output of the j-th neuron is computed as

$$\boldsymbol{h}_i^j = \sigma\left(\boldsymbol{h}_i^{j-1}\boldsymbol{w}_j + \boldsymbol{h}_{i-1}^j\boldsymbol{w}_j^{(h)} + \boldsymbol{b}_j\right), \tag{1}$$

where σ is some activation function and $\boldsymbol{h}_{i-1}$ the previous hidden state. For $i = 0$, the 'previous' hidden state $\boldsymbol{h}_{i-1j}$ is set to zero, and for $j = 0$, $\boldsymbol{h}_{ij-1}$ is simply the input X_i (note that this is the i-th *set* of inputs, not the input i-th from time point t_i).

To update the weights and biases we now look to investigate the error between y and the predicted output $\tilde{y}$, using some given loss function $L(y, \tilde{y})$,

$$L(y, \tilde{y}) = \frac{1}{N}\sum_{i=1}^{N} l(y_i, \tilde{y}_i), \tag{2}$$

where l is some error-metric. For some chosen learning rate η, the standard update rule for the weights is given by:

$$\boldsymbol{w}_j \to \boldsymbol{w}_j - \eta\frac{\partial L}{\partial \boldsymbol{w}_j}. \tag{3}$$

This transformation may be extended using optimization methods, aimed at handling exploding gradient, allowing for faster convergence, avoiding local minimas, etc. such as the root mean squared propagation (RMSprop) or adaptive moment estimation (Adam). We covered some of this in [**?**].

* e.b.rornes@fys.uio.no
† Insert Email

Compared to FFNN, computing the gradient of L with respect to the weights leads to a somewhat more complicated expression (we now omit the j-index for readability):

$$\nabla_{\boldsymbol{w}} L = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{w}} l(y_i, \tilde{y}_i) \tag{4}$$

$$= \frac{1}{N}\sum_{i=1}^{N} \nabla_{\tilde{y}_i} l(y_i, \tilde{y}_i) \cdot \nabla_{h_i} \tilde{y}_i \cdot \nabla_{\boldsymbol{w}} h_i. \tag{5}$$

The third factor, $\nabla_{\boldsymbol{w}} h_i$, is not present for the FFNN, and it leads to a recursion formula:

$$\begin{aligned} \nabla_{\boldsymbol{w}} h_i &= \nabla_{\boldsymbol{w}} \sigma + \nabla_{h_{i-1}} \sigma \cdot \nabla_{\boldsymbol{w}} h_{i-1} \\ &= \nabla_{\boldsymbol{w}} \sigma + \nabla_{h_{i-1}} \sigma \cdot \left(\nabla_{\boldsymbol{w}} \sigma + \nabla_{h_{i-2}} \sigma \cdot \nabla_{\boldsymbol{w}} h_{i-2}\right) + \ldots, \end{aligned} \tag{6}$$

and so on. For a more comprehensive explenation of this algorithm we refer to [**?**].

The recursion relation in (6) leads to a much greater computation time, compared to FFNN, as every gradient computation need an additional propagation through the entire network. Additionally, this dependency of each gradient on all hidden states can lead to gradients blowing up due to only (relatively) minor errors. There are multiple ways of resolving this issue. Perhaps the most obvious one is to simply truncate the amount of terms in (6), commonly referred to as 'truncated backpropagation through time' (see e.g. [**?**]). Apart from that it is an actual simplification of (6), it has the immediate consequence of ignoring long-term dependencies of the data, which in some cases is just the type of information you do not want your model to train on.

3. IMPLEMENTATION

4. DISCUSSION

5. CONCLUSION

Test bib [**?**]