RHEINISCH-WESTFÄLISCHE TECHNISCHE HOCHSCHULE AACHEN

Neuroinspired Computing (Prof. Dr. Abigail Morrison)

Deterministic Recurrent Neural Networks

Seminar Thesis

Marco Bischoff (Matr.-Nr. 370222)

February 23, 2024

Contents

1 Introduction

Machine learning has seen a rapid evolution in recent years, with neural networks emerging as a powerful tool for a wide range of applications. Initially inspired by the structure and function of the human brain, artificial neural networks have evolved into a diverse family of models, each tailored to specific tasks and data types. In particular, Recurrent Neural Networks (RNNs) have revolutionized the processing of sequential data, enabling tasks such as language modeling, machine translation, and speech recognition. In this seminar thesis, we explore the foundational concepts, architectures, and applications of RNNs, with a particular focus on the Long Short-Term Memory (LSTM) network and its role in addressing the challenges of training RNNs.

1.1 Feedforward Neural Networks

Before delving into the realm of RNNs, it is essential to understand the foundational concepts of feedforward neural networks, which form the basis for more complex models such as RNNs. Feedforward neural networks, also known as multilayer perceptrons, are a class of neural networks that process input data through a series of interconnected layers. Each layer consists of a set of nodes, or neurons, which perform a weighted sum of the inputs followed by the application of an activation function. The output of each layer serves as the input to the next layer, ultimately producing a final output. The process of training a feedforward neural network involves adjusting the weights of the connections between neurons to minimize the difference between the predicted and actual outputs, typically using the backpropagation algorithm and gradient descent.

This poses a limitation on the types of data that can be processed by feedforward neural networks, as they are designed to operate on fixed-size input vectors and produce fixed-size output vectors. This constraint makes them ill-suited for tasks involving sequential data, where the length of the input and output sequences may vary. For example, in natural language processing, the length of a sentence can vary significantly, making it challenging to process using a feedforward neural network. Additionally, feedforward neural networks lack an internal state, meaning they do not have the ability to remember information from previous inputs. This makes it difficult for them to capture the sequential dependencies present in many real-world datasets. To address these limitations, we turn to Recurrent Neural Networks.

2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a class of neural networks that are designed to handle sequential data by maintaining an internal state and processing input sequences one element at a time. Unlike feedforward neural networks, which process the entire input at once, RNNs process the input elements sequentially, allowing them to capture dependencies over time. This makes them

well-suited for tasks such as language modeling, machine translation, and speech recognition, where the order of the input elements is crucial for understanding the data.

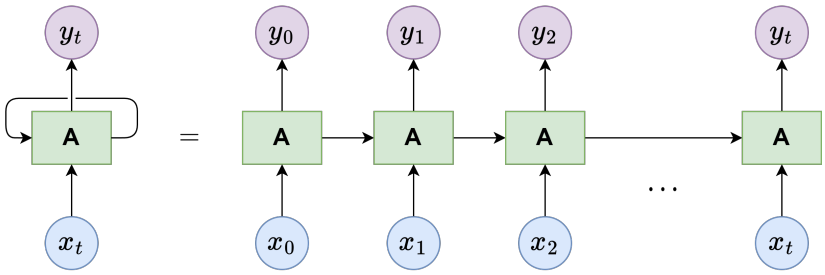

Figure 1: A recurrent neural network unrolled into a sequence of layers. The input sequence $x_0, x_1, \ldots, x_t$ is shown in blue, the network layer in green and the output sequence $y_0, y_1, \ldots, y_t$ in purple.

In its simplest form, the structure of an RNN is the same as a feedforward neural network, with the addition of a feedback loop that allows the network to maintain an internal state. This internal state is updated at each time step, allowing the network to remember information from previous inputs. The output of the network at each time step is influenced not only by the current input but also by the internal state, which captures the context from previous inputs. This means that, instead of having multiple layers with separate weights, RNNs have a single layer that acts as a dynamical system, changing over time. The repeating module in an RNN can be unrolled into a sequence of interconnected layers, allowing the network to process sequences of arbitrary length, as shown in Figure 1. This unrolling reveals that RNNs are intimately related to sequences and lists, making them a natural architecture for processing data of this form.

2.1 Categories of Applications

The applications of RNNs can be categorized based on the length of the input and output sequences and the relationship between them, as shown in Figure 2. For example, for tasks such as image classification, where the input and output sequences are of fixed length, RNNs can be used in a manner similar to feedforward neural networks (one-to-one). They can also be used for tasks such as image captioning, where the input is a fixed-size image and the output is a sequence of words (one-to-many). Additionally, RNNs can be used for tasks such as sentiment analysis, where the input is a sequence of words and the output is a single value (many-to-one). Tasks, where both the input and output are sequences of varying lengths (many-to-many), can be categorized into two subtypes. For example, in machine translation, the input sequence is processed first, and then the output sequence is generated. In contrast, for video classification, the output is generated simultaneously with the input, allowing the

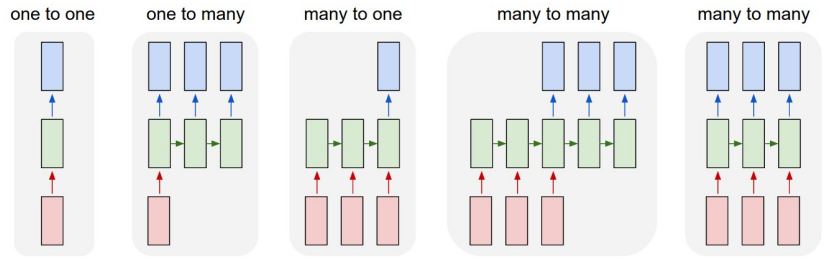

Figure 2: Ways of applying RNNs to different types of data. Both the input and output can be a fixed-size vector or a sequence of vectors. If both are sequences, the input and output vectors can either be regarded as pairs or as unrelated sequences of different lengths. [5]

network to use the previous frames as context. This demonstrates the flexibility of RNNs in handling a wide range of tasks involving sequential data.

2.2 Internal Structure

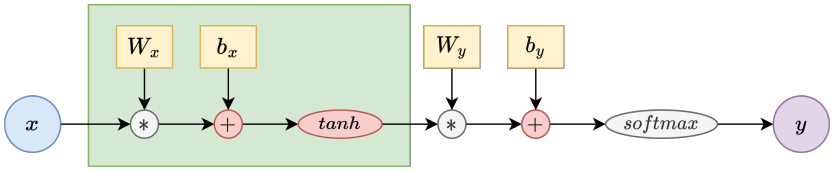

Figure 3: The internal structure of a feedforward neural network with a single layer. The input vector x is transformed by a linear transformation $Wx+b$ and then passed through an activation function. The output can, for example, be processed with another linear transformation and a softmax function to produce a probability distribution over classes. The vector y is the output of the network. In a multi-layer network, the components in the green box would be repeated for each layer.

To understand the internal structure of an RNN, we compare a single layer feedforward NN with a single layer RNN. As shown in Figure 3, a feedforward NN transforms the input vector x by a linear transformation and an activation function to produce the output of a single layer. In contrast, as shown in Figure 4, an RNN first concatenates the input vector x with the previous state vector h_{t-1}, then transforms the concatenated vector in the same way as a feedforward NN. The resulting state vector h_t then serves as the input to the next time step. For the first iteration of the RNN, the previous state vector is typically initialized to a vector of zeros. However, it can also be learned as part of the training process, which can boost the performance of the network in some cases.

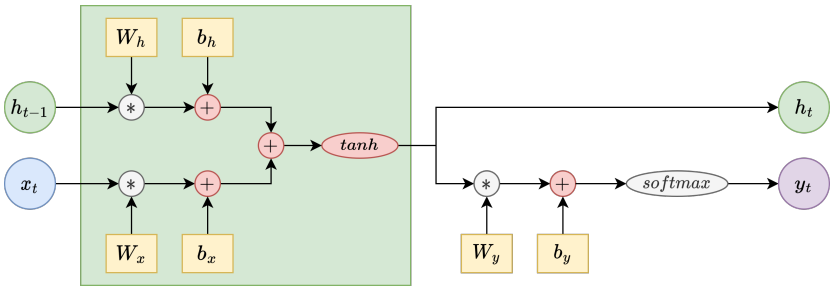

Figure 4: The internal structure of a recurrent neural network with a single layer. The input vector x and the previous state vector h_{t-1} are concatenated and transformed by a linear transformation and an activation function to produce the new state vector h_t. To produce the output vector y_t, the state vector can, for example, be transformed by another linear transformation and a softmax function.

2.3 Backpropagation

The training of RNNs is typically performed using the backpropagation through time (BPTT) algorithm, which is an extension of the backpropagation algorithm used to train feedforward neural networks. As an example, consider a single layer feedforward NN, like the green box in Figure 3. Let

- x be the input vector,
- W the weight matrix,
- b the bias vector,
- y the output vector,
- t the target vector (ground truth)
- σ the activation function, and
- $L = \frac{1}{2}\sum_{i=1}^{n}(t_i - y_i)^2$ the loss function (e.g. mean squared error).

We first do a forward pass to compute the output $y = \sigma(Wx+b)$ and compare it to the target t by the loss function L. Then, we do a backward pass to compute the gradient of the loss function with respect to the weights and biases of the network. Here, we use the chain rule of calculus:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial W} \tag{1}$$

$$= \frac{\partial L}{\partial y}\frac{\partial y}{\partial (Wx+b)}\frac{\partial (Wx+b)}{\partial W} \tag{2}$$

$$= -(t-y)\cdot\sigma'(Wx+b)\cdot x^T \tag{3}$$

The gradient for the bias vector $\frac{\partial L}{\partial b}$ can be computed similarly. For a network with multiple layers, the gradients are computed layer by layer using the chain rule, starting from the output layer and moving backwards through the network. In the final step, the weights and biases are updated in the direction that minimizes the loss: $W \leftarrow W - \alpha \frac{\partial L}{\partial W}$ and $b \leftarrow b - \alpha \frac{\partial L}{\partial b}$, where α is the learning rate. This is usually done using an optimization algorithm such as stochastic gradient descent.

The BPTT algorithm extends this approach to RNNs by unrolling the network into a sequence of interconnected layers, as shown in Figure 1. The forward pass is performed as usual, with the output of each time step serving as the input to the next time step. The backward pass then accumulates the gradients over the entire sequence and updates the weights and biases accordingly. This allows the network to capture the dependencies between the input elements and learn from the entire sequence, rather than just the current input. The key difference between BPTT and regular backpropagation is that the weights of the RNN are shared across time steps, meaning that the same weights are used at each time step. Also, the number of time steps is not given by the number of layers, but by the length of the input sequence. For long sequences, this can lead to problems such as vanishing or exploding gradients, which can make training difficult.

2.4 The Problem of Long-Term Dependencies

One of the key challenges in training RNNs is the problem of long-term dependencies. To illustrate this, consider a simplified RNN [7] that takes an input sequence $x_0, x_1, \ldots, x_n$ and produces a sequence of outputs/hidden states $h_0, h_1, \ldots, h_n$. It is defined by the function F for each time step $t \in \{0, 1, \ldots, n\}$:

$$h_t = F(h_{t-1}, x_t) = W_h \tanh h_{t-1} + W_x x_t + b \tag{4}$$

Then the gradient with respect to the hidden state at time step t is given by:

$$\nabla_h F(h_{t-1}, x_t) = W_h \mathrm{diag}(\tanh'(h_{t-1})) \tag{5}$$

For the backward pass, we need to compute the gradient of the loss function, which in this case is given by

$$\partial L = \nabla_h L(h_n, x_1, \ldots, x_n) \cdot \sum_{t=1}^{n} \prod_{k=n-t+1}^{n} \nabla_h F(h_{k-1}, x_k) \tag{6}$$

where each term in the sum is the gradient of the current layer. The sum can be written as:

$$\begin{aligned} &\sum_{t=1}^{n} \prod_{k=n-t+1}^{n} \nabla_h F(h_{k-1}, x_k) \\ &= \nabla_h F(h_{n-1}, x_n) \\ &+ \nabla_h F(h_{n-1}, x_n) \cdot \nabla_h F(h_{n-2}, x_{n-1}) \\ &+ \nabla_h F(h_{n-1}, x_n) \cdot \nabla_h F(h_{n-2}, x_{n-1}) \cdot \nabla_h F(h_{n-3}, x_{n-2}) \\ &+ \ldots \end{aligned} \tag{7}$$

With Equation (5) and Equation (7), we can see that the gradient for time step t is predominantly influenced by W_h^{n-t+1}. If the largest singular value of W_h is less than 1, the gradient will vanish as t increases. If it is greater than 1, the gradient will explode [7]. This makes it difficult for the network to learn long-term dependencies, as the gradients become too small or too large to be useful. This is known as the problem of long-term dependencies, and it is a fundamental limitation of standard RNNs.

Multiple approaches have been proposed to address this problem, for example, clipping the gradient to prevent it from becoming too large [7]. Let $\nabla_W L$ be the gradient of the loss function and ϵ a small constant. Then the clipped gradient ∇ is

$$\nabla = \begin{cases} \nabla_W L & \text{if } ||\nabla_W L|| < \epsilon \\ \epsilon \cdot \frac{\nabla_W L}{||\nabla_W L||} & \text{otherwise} \end{cases} \tag{8}$$

This prevents the gradient from becoming too large, but it does not address the problem of vanishing gradients. Another approach is to use ReLU activation functions [3], which have the advantage of not saturating for positive inputs because their derivative is 1. However, the problem of vanishing gradients still remains for negative inputs. Moreover, there were methods proposed to initialize the weights of the network in a way that prevents the gradients from vanishing or exploding [6]. While these methods can mitigate the problem of long-term dependencies to some extent, they do not provide a complete solution. A more effective approach is to use Long Short-Term Memory (LSTM) networks, which were specifically developed to address the problem of vanishing and exploding gradients in RNNs.

3 Long Short-Term Memory Networks

Long Short-Term Memory (LSTM) networks are a special type of RNN that are designed to learn long-term dependencies more effectively than standard RNNs. They were introduced by Hochreiter and Schmidhuber in 1997 [4] and have since become a popular choice for a wide range of applications, including language modeling, machine translation, and speech recognition. The key idea behind LSTMs is the use of a memory cell, which allows the network to store and access information over long time scales. In this section, we explore the architecture and operation of LSTMs, as well as their applications and variants.

3.1 Architecture

The architecture of an LSTM network is based on a repeating module that contains four interacting layers, as shown in Figure 5. The key to LSTMs is the cell state C_t, which runs straight down the entire chain with only minor linear interactions. This allows information to flow along the cell state unchanged, making it easy for the network to remember information over long time scales. The cell state is regulated by structures called gates, which are composed of

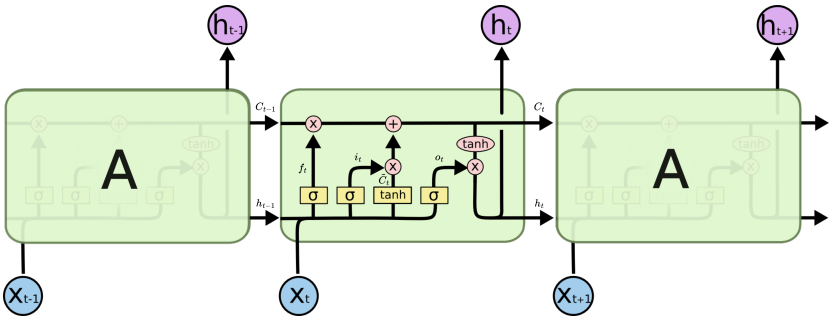

Figure 5: The repeating module of an LSTM network. Each yellow box represents a neural network layer with its activation function and the pink circles represent pointwise operations. The lines merging denote concatenation, while a line forking denotes its content being copied. The output is denoted by h_t.

neural network layers and pointwise operations. The gates allow the network to control the flow of information into and out of the cell state, enabling it to remember or forget information as needed.

3.1.1 Forget Gate

The first step in the operation of an LSTM is to decide what information to forget from the cell state. This is done by a sigmoid layer called the forget gate layer, which takes the previous hidden state h_{t-1} and the current input x_t as input and outputs a number between 0 and 1 for each element in the cell state C_{t-1}. A value of 0 indicates that the corresponding element should be forgotten, while a value of 1 indicates that it should be retained. This allows the network to selectively forget information from the cell state as needed, enabling it to discard irrelevant information and focus on the most important elements.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{9}$$

3.1.2 Input Gate

The next step is to decide what new information to store in the cell state. This has two parts. First, a sigmoid layer called the input gate layer decides which values to update. Next, a tanh layer creates a vector of new candidate values $\tilde{C}_t$ that could be added to the state. The input gate layer outputs a number between 0 and 1 for each element in the cell state, indicating how much of the new candidate values should be added to the state. This allows the network to selectively update the cell state with new information, enabling it to incorporate relevant information from the current input.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{10}$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{11}$$

3.1.3 Update Cell State

The next step is to update the old cell state C_{t-1} into the new cell state C_t. The previous steps have already decided what to do, so the network just needs to actually do it. First, the old state is multiplied by the forget gate f_t, forgetting the things that were decided to forget earlier. Then, the new candidate values $\tilde{C}_t$ are added to the state, scaled by how much the input gate decided to update each state value.

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \tag{12}$$

3.1.4 Output Gate

Finally, the network needs to decide what to output. This output will be based on the cell state, but will be a filtered version. First, a sigmoid layer called the output gate layer decides what parts of the cell state should be output. Then, the cell state is put through a tanh function to push the values to be between -1 and 1, and multiplied by the output of the sigmoid gate. The result is the output of the network at the current time step h_t, which can be used for further processing or as the final output of the network.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{13}$$

$$h_t = o_t \cdot \tanh(C_t) \tag{14}$$

3.2 Variants

LSTMs have been the subject of extensive research, leading to the development of several variants that aim to improve their performance and address specific challenges. One popular variant, introduced by Gers and Schmidhuber in 2000 [2], adds peephole connections to the gates, allowing them to look at the cell state. This allows the gates to take into account the current state of the cell when making decisions, enabling the network to better control the flow of information.

Another variation is to use coupled forget and input gates, which allows the network to decide what to forget and what to add new information to in a single step. This can simplify the operation of the network and make it easier to learn long-term dependencies.

Additionally, there are variants of LSTMs that use different activation functions, such as the Gated Recurrent Unit (GRU) [1], which uses a simpler structure with fewer gates and has been shown to perform well on a wide range of tasks. These variants demonstrate the flexibility of LSTMs and their ability to be adapted to different applications and challenges.

References

[1] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation, September 2014.

[2] Felix Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to Forget: Continual Prediction with LSTM. *Neural computation*, 12:2451–71, October 2000.

[3] Xavier Glorot, Antoine Bordes, and Y. Bengio. Deep Sparse Rectifier Neural Networks. In *Journal of Machine Learning Research*, volume 15, January 2010.

[4] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-term Memory. *Neural computation*, 9:1735–80, December 1997.

[5] Andrej Karpathy. The Unreasonable Effectiveness of Recurrent Neural Networks. https://karpathy.github.io/2015/05/21/rnn-effectiveness/, 2015.

[6] Siddharth Krishna Kumar. On weight initialization in deep neural networks, 2017.

[7] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning*, pages 1310–1318. PMLR, May 2013.

A Benutzerdokumentation

B Introduction