Ludwig-Maximilians-Universität München
Department of Statistics

Master's Thesis

Sequence to Sequence Models: Knowledge Tracing with Deep Learning

Author:
Johannes Nawrath

Supervisor:
Prof. Dr. Christian Heumann

April 29, 2021

Statutory Declaration

I herewith declare that I have composed the present thesis myself and without use of any other than the cited sources and aids. Sentences or parts of sentences quoted literally are marked as such. Other references regarding the statement and scope are indicated by full details of the publications concerned. The thesis in the same or similar form has not been submitted to any examination body and has not been published. This thesis was not, even in part, used in another examination or as a course performance. Furthermore, I declare that the submitted written bound copies of the present thesis and the version submitted on a data carrier are consistent with each other in contents.

Augsburg, 29.04.2021

Johannes Nawrath

Abstract

This thesis consists of two parts. At first, various methods of modeling sequential data with neural networks are presented. The idea is to show the development of sequential deep learning models over time. For this purpose, the problems of early approaches are highlighted in order to better understand the methods in more modern solutions. More precisely, the thesis starts off with simple recurrent neural networks and moves on to gated recurrent units. Furthermore, attention mechanisms are discussed, which ultimately result in the transformer architecture.

The second part of the thesis demonstrates that modern sequential deep learning models can be applied successfully to the task of knowledge tracing, the ability to model the knowledge of students over time, as they interact with coursework. Various model configurations are trained on real world data from almost 400K students as part of a competition on Kaggle. The models achieve state-of-the-art results and trump other machine learning algorithms, which cannot exploit the sequential data structure well enough.

Contents

List of Figures

List of Tables

1 Introduction

According to the UNESCO Institute for Statistics, there have been about 258 million children, who did not attend school in 2018. These children can not enjoy the personalized learning experience of a well educated teacher, who knows the individual strengths and weaknesses of each student. An approach to tackle these challenges in education could be a fully automated learning system, based on large scale data bases and innovative algorithms. Such systems would dramatically reduce the cost of education and ease the access to personalized lectures and exercises. In addition to that, the Covid-19 outbreak in 2020 forced schools to shut down all around the globe and industrialized countries like Germany learned the hard way, how beneficial an online tutoring system could for keeping up the educational standard (Schipp 2021). A key element for these systems is a task called knowledge tracing; the ability to accurately model students knowledge over time, such that the learning material can be adjusted to the individual needs of a student.

From a statistical modeling perspective, the progress of a user in an online tutoring system generates sequential data, where each step is resembled by an exercise and its corresponding response. The overall goal of knowledge tracing models is to predict, whether a user will answer a specific question correctly, given a history of previous question-response pairs. Since the available history varies widely among different users, a key aspect of such models is the ability to handle sequences of variable length. This is a feature, most machine learning algorithms do not support inherently. For this reason, the thesis focuses on sequential deep learning models, that can process variable length inputs.

The structure of the thesis is as follows. Section 2 gives a short introduction to time series modeling with tabular machine learning algorithms. Section 3 covers the fundamentals of recurrent neural networks and its modern generalizations, such as long short-term memory cells (LSTM's) and gated recurrent units (GRU's). Furthermore, the encoder-decoder architecture is described, that allows to model sequence to sequence mappings of arbitrary and varying lengths. Attention mechanisms are introduced as an improvement to these models and chapter 4 extends the idea to self-attention and the well known Transformer architecture. Chapter 5 gives a short review of knowledge tracing models and the theory is finally applied on real world data (Chapter 6), as part of participation in the *Riiid AIEd Challenge 2020* hosted on Kaggle[1].

[1]Kaggle, a subsidiary of Google LLC, is an online community of data scientists. Companies can post problems on the platform and machine learning practitioners compete to find state of the art solutions, which are typically rewarded with cash prizes.

2 Machine Learning with (Time) Series

Sequential data is ubiquitous in many applications, that include but are not limited to human speech recognition, weather forecasting or stock market analysis. The common characteristic of these examples is, that the data is observed at a finite number of known (time) steps. More precisely, sequential data consists of a sequence of positions $t_1, ..., t_T$, e.g days, months or simply the position of a word in a sentence, and a sequence $x(t_1), ..., x(t_T)$, where $x(t_i)$ can be a vector of arbitrary dimension, representing the observation at position t_i. In the example of weather forecasting, the $x(t_i)$ might consist of meteorological data, like the minimum and maximum temperature, the amount of rainfall etc. Throughout this thesis, the notation is slightly abbreviated and it is simply written $\mathbf{x} = x_1, ..., x_T$, to refer to a sequence with measurements at position $t_1, ..., t_T$.

There are two fundamentally different ways, in which sequential data arises (Löning et al. 2019):

- **(Multivariate) time series data**: Two or more variables are observed over time, representing different kinds of measurements within a single experimental unit. E.g. the daily closing price of all stocks in the S&P 500.
- **Panel data**: Multiple independent instances of the same kinds of measurements are observed, e.g. time series from multiple patients in a hospital.

For multivariate time series, the data is highly correlated and no i.i.d. assumption can be made. The panel data setup corresponds to independent instances of multivariate time series and therefore an i.i.d. assumption among different instances is plausible. But note, that each instance might be a multivariate time series itself, for which the assumption does not hold.
The amount of different time series generating scenarios is mirrored by the amount of learning tasks applicable to such data. Two common tasks are (Löning et al. 2019):

- **Time series regression/classification**: N i.i.d. training instances of feature-label pairs $(\mathbf{x}_i, y_i), i = 1, ...N$ are observed. Where $\mathbf{x}_i = x_1, ..., x_T$ is a series of values and for regression tasks $y_i \in \mathbb{R}$ is a scalar value. For classification y_i takes a value from a finite set of categories. The goal is to learn a predictor f that can accurately predict $\hat{y} = f(\mathbf{x}_*)$ for a new input sequence $\mathbf{x}_*$
- **Supervised/Panel forecasting:** N i.i.d training instances $(\mathbf{x}_i, \mathbf{y}_i), i = 1, ..., N$ are observed, where $\mathbf{x}_i = x_1, ..., x_T$ and $\mathbf{y}_i = y_1, ..., y_T$ are both, possibly multivariate, time series. The task is to learn a forecaster f, that can make accurate temporal forward predictions $\hat{y}_i^{T+1} = f(\mathbf{x}_i, \mathbf{y}_i)$, where $\hat{y}_i^{T+1}$ is the next value in the series $\mathbf{y}_i$.

Time series regression/classification is basically the same as the usual tabular supervised learning task, with the difference that the input features are time series. In contrast

to that, panel forecasting trains a model that predicts future values of a time series, conditioned on its history and additional input features, which can be time series as well.

The panel forecasting setup can be reduced to the simpler problem of time series regression/classification by iterating over the output periods (Bontempi et al. 2012). For two sequences $y_1, ..., y_T$ and $x_t, ..., x_T$ the goal is to learn a model f, such that for every time step $t = 1, ..., T$:

$$\hat{y_{t+1}} = f(y_1, ..., y_t, x_1, ..., x_t) \tag{2.1}$$

Which is the same formulation as in the time series regression/classification task, with $y_1, ..., y_t, x_1, ..., x_t$ as the multivariate time series and y_{t+1} as the target value. However, there are two important differences: first, the i.i.d. assumption does not hold among different time steps t in (2.1) and second, the sequence lengths of the source sequences $y_1, ..., y_t, x_1, ..., x_t$ grows with t. Hence, the amount of features varies at each step.

The violated i.i.d. assumption requires specialized validation strategies. A popular method is rolling window cross validation (Hyndman & Athanasopoulos 2018, Chapter 3), which ensures that the training set only consists of observations, which have been made prior to the observations in the test set. More sophisticated procedures can be found in (de Prado 2018, Chapter 7). These methods, share one similarity: They require multiple models trained on different subsets of the training data. This makes such approaches impracticable in the field of deep learning, due to the enormous computational effort to train a single model, and are therefore out of scope of this thesis.

Typically, machine learning algorithms are designed to process an arbitrary, but fixed amount of input features. This leads to a problem with the formulation in (2.1), where the amount of input features grows with the sequence length. To overcome this problem, one needs to extract a fixed amount of features from the variable length input sequences. An easy approach is to simply use the last n steps of the sequence as input features, i.e.

$$\hat{y}_t = f(y_{t-1}, ..., y_{t-n}). \tag{2.2}$$

In general, arbitrary engineered features can be used. For example moving averages over different periods or other statistics of the input sequence. More sophisticated approaches (Fulcher & Jones 2016) extract thousands of such features automatically. The information is then compressed with dimension reduction methods and used as the models input.

3 Recurrent Neural Networks

The ideas of the previous section can be applied directly to feed forward neural networks. For example one could train a model with the values of time steps $(t-1), ..., (t-n)$ as input nodes and step t as the target value. However, such a model could only generalize to sequences of length n. For tasks with approximately same sequence lengths, this could be handled by padding the sequences to the same size. But for tasks such as knowledge tracing (section 5), where the lengths vary between one and several thousand time steps, this is not an appropriate approach. Another drawback is, that there are separate parameters for every input position and the model would have to learn patterns independently for every position.

Recurrent neural networks (RNN's) use a more natural approach to model sequential data. Instead of training a model that processes a fixed sized sequence all at once, RNN's intend to model transitions from one time step to the next. For this, a model f is learned, that takes as input only the current time step $x_t \in \mathbb{R}^d$ and a fixed size vector $h_{t-1} \in \mathbb{R}^u$, called the hidden/latent state, such that

$$h_t = f_\theta([x_t; h_{t-1}]). \tag{3.1}$$

For a finite amount of time steps, this recursion can be unfolded by applying the function f repeatedly. For example for $t = 3$:

$$\begin{aligned} h_3 &= f_\theta([h_2; x_3]) \\ &= f_\theta([f_\theta([h_1; x2]); x_3]) \\ &= f_\theta([f_\theta([f_\theta([h_0; x_1]); x_2]); x_3]) \end{aligned} \tag{3.2}$$

Where h_0 is a predefined initial state, that is usually set to 0.

Figure 1 shows the computational graph of equation (3.1) and its unrolled version in (3.2). The recursion is displayed by a loop in the graph that can be unfolded, such that every node is associated with one particular time step.

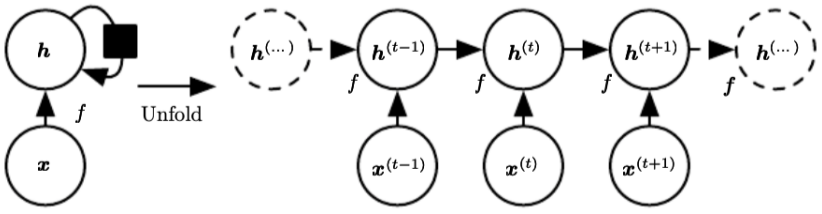

Figure 1: Unfolding an RNN (Goodfellow et al. 2016, Chapter 10)

This means, that the hidden state after t steps can be represented in two ways: By a function u^t of the whole past input sequence $x_t, x_{t-1}, ..., x_2, x_1$ and an initial state h_0 or by a function f of the previous hidden state h_{t-1} and the current input x_t:

$$h_t = u^t(x_t, x_{t-1}, ..., x_2, x_1) = f(h_{t-1}, x_t) \tag{3.3}$$

Note, that the function u^t is different for every time step but can be factorized into repeated application of the function f. This introduces two major advantages (Goodfellow et al. 2016, Chapter 10):

1. The model is specified in terms of transitions from one hidden state to another. This way, the input size is the same for all time steps.

2. The same function f with the same parameters is used at every time step.

This makes it possible to train a single model, that is shared across all time steps and therefore generalizes to sequences of arbitrary length. Another benefit of this form of parameter sharing is, that it requires far less training samples than training a model u^t for every time step.(Goodfellow et al. 2016, Chapter 10)

With this concept of parameter sharing, the forward pass of a recurrent neural network can be formalized. For this, a reasonable representative example is used, where a input sequence x is mapped to a corresponding output sequence o and a function L measures the loss between o and the ground truth sequence y. In the following, the output sequence is assumed to be binary and the loss function is set to the binary cross entropy. This is a fairly general example and could be interpreted as predicting the probability for rain, given a sequence of meteorological data or as the task of predicting, whether the price of a stock moves upwards or downwards, given a history of past financial information.
Let $x_i \in \mathbb{R}^d$ be a d-dimensional input at every time step and $y_i \in \{0, 1\}$ the binary output. Furthermore, $W \in \mathbb{R}^{k \times h}$ is the hidden to hidden weight matrix, $U \in \mathbb{R}^{k \times d}$ is the input to hidden weight matrix and $V \in \mathbb{R}^{1 \times k}$ are the hidden to output weights. $b \in \mathbb{R}^k$ and $c \in \mathbb{R}$ are biases applied before each activation function. The forward pass of such a recurrent neural network can then be summarized with a set of update equations, that are applied at each time step:

$$\begin{aligned} a_t &= b + Wh_{t-1} + Ux_t \\ h_t &= tanh(a_t) \\ o_t &= c + Vh_t \\ \hat{y_t} &= \sigma(o_t) \end{aligned} \tag{3.4}$$

The loss L for the whole sequence of length τ can then be computed as the sum of the

losses L_t over all time steps:

$$\begin{aligned} & L(\{y_1, ..., y_\tau\}, \{\hat{y_1}, ..., \hat{y_\tau}\}) \\ & = \sum_{i=1}^{\tau} \mathrm{L}_t(y_t, \hat{y_t}) \\ & = \sum_{i=1}^{\tau} y_i log(\hat{y_i}) + (1 - y_i) log(1 - \hat{y_i}) \end{aligned} \tag{3.5}$$

By computing the gradient of 3.5 with respect to the weight matrices and bias vectors, the model can be trained using gradient descent and back-propagation trough time (Werbos 1990).

3.1 Backpropagation through Time

The purpose of this section is to provide a high level understanding for computing the gradients in recurrent neural networks. There are several algorithms for this task, for example *Real Time Recurrent Learning* (Williams & Zipser 1989) and *Backpropagation Through Time* (BPTT; Werbos (1990)). The following focuses on BPTT, because it is a straight forward application of the back-propagation algorithm to the unfolded graph of the recurrent network.

The back-propagation algorithm is obtained, by recursively applying the chain rule of calculus, to calculate the derivative of a function that is formed as a composition of other functions. Let $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$ be two vectors and $g : \mathbb{R}^m \to \mathbb{R}^n$, $g : \mathbb{R}^n \to \mathbb{R}$ two real-valued functions. For $y = g(x)$ and $z = f(y)$, the chain rule yields the derivative of z with respect to x:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial y_i} \tag{3.6}$$

Consider a simple feed forward network with two input nodes, two hidden nodes and an output node f. For the input-to-hidden weights $w_{i,j}$, the hidden-to-output weights u_i, and the biases b_i and c the forward pass for a single input example (x_1, x_2) is calculated with

$$
\begin{aligned}
z_{in,1} &= w_{1,1}x_1 + w_{2,1}x_2 + b1 \\
z_{in,2} &= w_{2,1}x_1 + w_{2,2}x_2 + b2 \\
z_{out,1} &= \sigma(z_{in,1}) \\
z_{out,2} &= \sigma(z_{in,2}) \\
f &= u_1 z_{out,1} + u_2 z_{out,2} + c
\end{aligned}
\tag{3.7}
$$

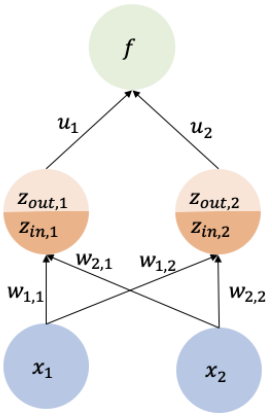

Figure 2: A simple feed forward network

Where σ is the logistic sigmoid function. Given a ground truth value y, the loss (L-2 loss) can be computed with respect to the networks output f:

$$
L(y, f(x)) = \frac{1}{2}(y - f(x))^2 \tag{3.8}
$$

The derivatives of L with respect to the weights $w_{i,j}$, u_j and biases b_i, c yield the information, in which direction a change in those parameters would affect the total loss. Since the forward pass in (3.7) is nothing else than a composition of multiple functions, the gradients can be obtained, by recursively applying the chain rule from (3.6):

$$
\begin{aligned}
\frac{\partial L(y, f(x))}{\partial f} &= y - f \\
\frac{\partial L(y, f(x))}{\partial u_i} &= \frac{\partial L(y, f(x))}{\partial f}\frac{\partial f}{\partial u_i} = (y - f) z_{out,i} \\
\frac{\partial L(y, f(x))}{\partial c} &= \frac{\partial L(y, f(x))}{\partial f}\frac{\partial f}{\partial c} = (y - f) \\
\frac{\partial L(y, f(x))}{\partial w_{i,j}} &= \frac{\partial L(y, f(x))}{\partial f}\frac{\partial f}{\partial z_{out,j}}\frac{\partial z_{out,j}}{\partial z_{in,j}}\frac{\partial z_{in,j}}{\partial w_{i,j}} \\
&= (y - f) u_j \sigma(z_{in,j})(1 - \sigma(z_{in,j})) w_{i,j} \\
\frac{\partial L(y, f(x))}{\partial b_j} &= \frac{\partial L(y, f(x))}{\partial f}\frac{\partial f}{\partial z_{out,j}}\frac{\partial z_{out,j}}{\partial z_{in,j}}\frac{\partial z_{in,j}}{\partial b_j} \\
&= (y - f) u_j \sigma(z_{in,j})(1 - \sigma(z_{in,j}))
\end{aligned}
\tag{3.9}
$$

Note, that some of these expressions, $\sigma(z_{in,j})$ for instance, have already been computed during the forward-pass. This means those values can be cached when feeding the data through the network and can then be plugged in when computing the gradient. The term back-propagation refers to the fact, that the gradients are computed backwards, starting at the output layer. Consider the gradient for the input to hidden weights $w_{i,j}$.

The terms for calculating these four gradients share two common expressions:

$$\begin{aligned} \delta_j &= (y - f) u_j \sigma(z_{in,j})(1 - \sigma(z_{in,j})) \\ \frac{\partial L(y, f(x))}{\partial w_{i,j}} &= \delta_j w_{i,j} \end{aligned} \tag{3.10}$$

Therefore, the gradients can be computed very efficiently, by caching the intermediate result δ_j and plugging it in multiple times. For this toy example this is only a minor advantage, but for large networks with hundreds of hidden units and dozens of layers this advantage is huge.
On the other hand, if the gradients were computed forwards, starting from the input layer, the gradient for $w_{1,1}$ and $w_{2,1}$ would be calculated with

$$\begin{aligned} \frac{\partial L(y, f(x))}{\partial w_{1,1}} &= \left(\left(\left(\frac{\partial z_{in,1}}{\partial w_{1,1}} \frac{\partial z_{out,1}}{\partial z_{in,1}} \right) \frac{\partial f}{\partial z_{out,1}} \right) \frac{\partial L(y, f(x))}{\partial f} \right) \\ \frac{\partial L(y, f(x))}{\partial w_{2,1}} &= \left(\left(\left(\frac{\partial z_{in,j}}{\partial w_{2,1}} \frac{\partial z_{out,1}}{\partial z_{in,1}} \right) \frac{\partial f}{\partial z_{out,1}} \right) \frac{\partial L(y, f(x))}{\partial f} \right) \end{aligned} \tag{3.11}$$

This way, there would be no common expressions, that could be reused multiple times and the computation of the gradients would be too inefficient to train large neural networks.

This very same procedure can be applied to the unfolded computational graph of a recurrent neural network. Reconsider the forward pass of the RNN described by the equations in 3.4. The goal is to derive the gradients of the loss function $L(\{y_1, ..., y_\tau\}, \{x_1, ..., x_\tau\})$ in 3.5 with respect to the weight matrices W, U, V and the biases b, c.

In the following, the gradients are derived as in Goodfellow et al. (2016), starting at the loss L_t of each output node:

$$\frac{\partial L}{\partial L_t} = \frac{\partial}{\partial L_t} \sum_{i=1}^{\tau} L_t(y_t, \hat{y_t}) = 1 \tag{3.12}$$

The gradients of each output node o_t can be computed:

$$\frac{\partial L}{\partial o_t} = \frac{\partial L}{\partial L_t} \frac{\partial L_t}{\partial \hat{o_t}} = (y_t - o_t) \tag{3.13}$$

At the final time step τ, the gradient with respect to h_τ is simply

$$\nabla_{h_\tau} L = V^T \frac{\partial L}{\partial o_\tau}, \tag{3.14}$$

because the only descendent of h_τ is o_τ. From there the gradients with respect to the hidden states at time steps $t < \tau$ can be obtained by back-propagating the gradients

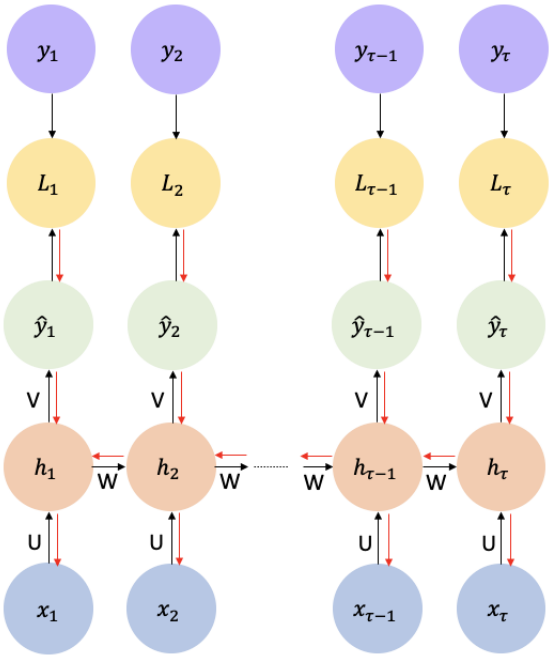

Figure 3: Backpropagation through time

through time from $t = \tau - 1$ to $t = 1$. The difference to the gradient in 3.14 is, that for $t < \tau$ the descendents of h_t are o_t and h_{t+1}. The gradient is thus given by

$$\begin{aligned} \nabla_{h_t} L &= \left(\frac{\partial h_{t+1}}{\partial h_t}\right)^T (\nabla_{h_{t+1}} L) + \left(\frac{\partial o_t}{\partial h_t}\right)^T (\nabla_{o_t} L) \\ &= W^T diag(1 - (h_{t+1})^2)(\nabla_{h_{t+1}} L) + V^T (\nabla_{o_t} L). \end{aligned} \tag{3.15}$$

Where $diag(1 - (h_{t+1})^2)$ is a diagonal matrix of the hidden states at $t + 1$, that is the Jacobian matrix of the hyperbolic tangent. With these results, the gradients for the weight matrices and bias vectors can be computed as the sum of the gradients at each time step:

$$
\begin{aligned}
\nabla_c L &= \sum_t \left(\frac{\partial o_t}{\partial c} \right) \nabla_{o_t} L = \sum_t \nabla_{o_t} L \\
\nabla_b L &= \sum_t \left(\frac{\partial h_t}{\partial b_t} \right) \nabla_{h_t} L = \sum_t diag(1 - (h_t^2)) \nabla_{h_t} L \\
\nabla_V L &= \sum_t \left(\frac{\partial L}{\partial o_t} \right) \nabla_{v_t} o_t = \sum_t (\nabla_{o_t} L) h_t^T \\
\nabla_W L &= \sum_t \sum_i \left(\frac{\partial L}{\partial h_t^{(i)}} \right) \nabla_{w_t} h_t^i = \sum_t diag(1 - (h_t)^2)(\nabla_{h_t} L) h_{t-1}^T \\
\nabla_U L &= \sum_t \sum_i \left(\frac{\partial L}{\partial h_t^{(i)}} \right) \nabla_{U_t} h_t^{(i)}
\end{aligned}
\tag{3.16}
$$

3.2 Modern Architectures for Recurrent Neural Networks

Recurrent neural networks are a powerful model class for sequential data. Similar to the universal approximation theorem for feed forward networks (Hornik et al. 1989), RNN's can approximate any measurable sequence-to-sequence mapping to arbitrary accuracy (Hammer 2000). This means, that in theory RNN's can learn dependencies among positions in the input and output sequence, no matter how far the distance between those positions. In practice however, it is a difficult task to learn such long-term dependencies with gradient descent and back-propagation through time (Bengio et al. 1994). The reason for this is that the gradients tend to either explode or vanish, the further they are propagated back through time. Modern architectures use gating mechanisms within the hidden state updates, to overcome this problem. In this section two popular gated recurrent units are introduced: long short-term memory models and gated recurrent units.

3.2.1 Vanishing and Exploding Gradients

Consider the following unfolded recurrent network, that produces a single output after processing a sequence of length τ.

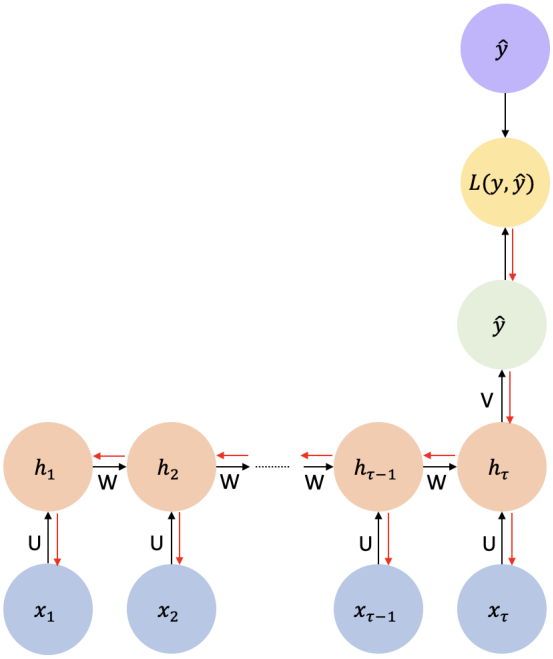

Figure 4: Error signal in an RNN

To train this network, the loss-gradient with respect to the weight matrices U, W and V has to be computed. At one step during the recursive application of the chain-rule, the gradient with respect to the first hidden state output h_1 will be required:

$$\nabla_{h_1} L = \frac{\partial L}{\partial h_\tau} \frac{\partial h_\tau}{\partial h_{\tau-1}} \cdots \frac{\partial h_2}{\partial h_1} \tag{3.17}$$

with

$$h_{t+1} = f(b + W h_{t-1} + U x_t), \tag{3.18}$$

where f is a differentiable activation function, usually the logistic sigmoid function or the hyperbolic tangent. The gradient in 3.17 is then:

$$\begin{aligned}
\frac{\partial h_\tau}{\partial h_{\tau-1}} &= diag(f'(W h_{\tau-1} + W x_\tau + b))W = D_{\tau-1} W \\
\frac{\partial h_{\tau-1}}{\partial h_{\tau-2}} &= diag(f'(W h_{\tau-2} + W x_{\tau-1} + b))W = D_{\tau-2} W \\
&\vdots \\
\frac{\partial h_2}{\partial h_1} &= diag(f'(W h_1 + W x_2 + b))W = D_1 W \\
\frac{\partial L}{\partial h_1} &= \frac{\partial L}{\partial h_\tau} D_{\tau-1} D_{\tau-2} ... D_1 W^{\tau-1}
\end{aligned} \tag{3.19}$$

Therefore, the gradient at any time step t will contain the term W^{t-1}. In the one-

dimensional case, where W is a scalar value, it is clear that W^{t-1} will explode for $W > 1$ and vanish for $W < 1$, for large values of t. In case W is a $k \times k$ dimensional matrix, suppose there exists an eigendecomposition, such that $W = Q\Lambda Q^{-1}$, where Λ is diagonal matrix of eigenvalues. The term W^{t-1} can then be simplified to

$$W^{t-1} = Q\Lambda^{t-1}Q^{-1}. \tag{3.20}$$

Since Λ is a diagonal matrix with values λ_i, Λ^{t-1} is also diagonal with values λ_i^{t-1}, for $i = 1, ..., k$. If $\lambda_{max} = \max_i \lambda_i$ is larger than 1, the gradient in 3.19 will grow exponentially. On the other hand, if $\lambda_{max} < 1$, all diagonal values λ_i will shrink to zero and thus the gradient will vanish for long term dependencies.

A effective method to avoid exploding gradients is called gradient clipping (Pascanu et al. 2013). This mechanism rescales the gradient∇_Θ, if its norm exceeds a certain threshold:

$$\nabla_\Theta = \begin{cases} \frac{threshold}{||\nabla_\Theta||}\nabla_\Theta, & || \nabla_\Theta ||> threshold \\ \nabla_\Theta, & else \end{cases} \tag{3.21}$$

This is a simple to implement and computationally efficient method, with one additional hyper-parameter, the threshold. A good heuristic for setting this parameter is to calculate a sufficient larger number of gradients and look at statistics of the average norm (Pascanu et al. 2013).

The problem of vanishing gradients, however, can not be solved that easily and a lot of research in that area has been made (e.g. Hochreiter (1998), Bengio et al. (1994)). Modern approaches to this problem use modified recurrent architectures such as long short term memory (Hochreiter & Schmidhuber 1997) models and gated recurrent units (Cho et al. 2014), which will be highlighted in the next sections.

3.2.2 Long Short-Term Memory

Long short term memory networks (LSTM's; Hochreiter & Schmidhuber (1997)) were one of the earliest approaches to address the difficulty of learning long term dependencies. LSTM's introduce gated memory cells for the recurrent units, that allow the model to decide, whether inputs from previous hidden states should be remembered or ignored. While the original architecture did not include a forget gate, in the following the widely used implementation in Gers et al. (2000) is described. A LSTM-cell consists of an input gate, a forget gate and an output gate, followed by some element-wise operations. A major difference, to the simple recurrent cell in (3.4) is the introduction of an additional variable, the cell state s_t. In contrast to the hidden outputs h_t at each step, the cell states are purely internal and are not passed to subsequent layers of the network.

Let $x_t \in \mathbb{R}^k$ be the input, $h_t \in \mathbb{R}^d$ the hidden state and $c_t \in \mathbb{R}^d$ the cell state at time step t, with $h_0 = c_0 = 0$. For learnable weight matrices and bias vectors $W_j, U_j, b_j, j \in \{f, i, o, s\}$, the update equations at each time step are:

$$\begin{aligned} f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \\ i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\ o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \\ \tilde{c}_t &= \tanh(W_s x_t + U_s h_{t-1} + b_s) \\ c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\ h_t &= o_t \odot \tanh(c_t) \end{aligned} \tag{3.22}$$

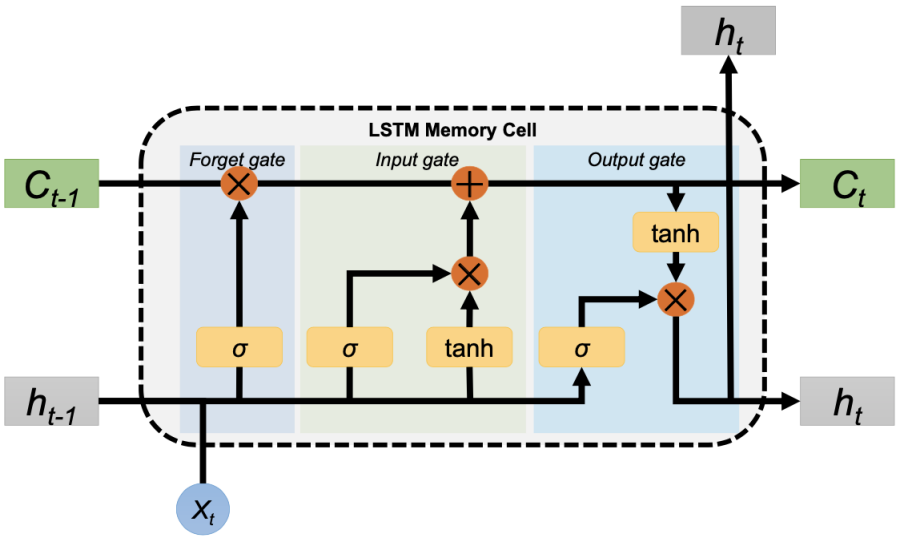

Figure 5: Long Short-Term Memory Cell (graphic from Fan et al. (2020))

There are three gates, the forget gate $f_t \in \mathbb{R}^d$, the input gate $i_t \in \mathbb{R}^d$ and the output gate $o_t \in \mathbb{R}^d$. These gates are small feed-forward networks itself, that operate on the previous hidden state h_{t-1} and the current input x_t. Each has its own set of weight matrices and bias vectors.

The forget-gate learns which components of the previous cell-state c_{t-1} should be forgotten. Mathematically, this is achieved by multiplying every dimension of $c_{t-1} \in \mathbb{R}^d$ with the output of the forget gate $f_t \in \mathbb{R}^d$. Since the forget gate uses a sigmoid activation, the values are between 0 and 1, where 0 means that the cell state at this position is fully removed, and 1 means the state is kept unchanged.

After the decision, which old information should be removed from memory, the next step focuses on the new information that should be added to the models state. For this, the new cell state candidate $\tilde{c}_t$ is computed by a feed forward network, that uses the

hyperbolic tangent as activation, i.e. the output values are between -1 and 1.
Depending on the last hidden state h_{t-1} and the current input x_t the forget gate decides, which information from $\tilde{c}_t$ should be added to the new cell state c_t. The procedure is the same as in the forget gate: The output of the input gate are values between 0 and 1, that indicate how much of $\tilde{c}_t$ is actually used for the new cell state c_t.
The final output of the LSTM h_t is the hyperbolic tangent of the new cell state c_t, that is filtered by the output gate o_t. This is again implemented by a element-wise product with values between 0 and 1.

3.2.3 Gated Recurrent Units

Gated recurrent units (GRU's) were introduced by Cho et al. (2014) and have become a widely used alternative to LSTM networks (e.g. Ravanelli et al. (2018)). Similar to LSTM's, GRU's control the flow of information with gating mechanisms. The key difference is, that GRU's do not rely on an internal cell state s_t. Therefore GRU's have less parameters than LSTM's and can be trained faster.

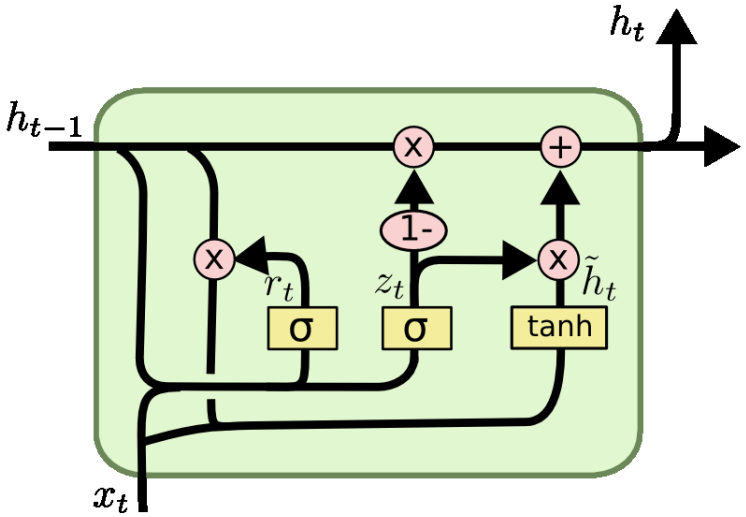

Figure 6: Gated Recurrent Unit (graphic from (Jabreel & Moreno (2019))

Let $x_t \in \mathbb{R}^k$ be the input and $h_t \in \mathbb{R}^d$ the hidden state at time step t, with $h_0 = 0$. For learnable weight matrices and bias vectors $W_j, U_j, b_j, j \in \{z, r, h\}$, the update equations at each time step are:

$$\begin{aligned} z_t &= \sigma(W_z x_t + U_z h_{t-1} + b_z) \\ r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r) \\ \tilde{h}_t &= \phi(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h) \\ h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \end{aligned} \tag{3.23}$$

The candidate vector $\tilde{h}_t$ for a new hidden state is obtained, similar as in the simple RNN, by a feed forward network of the previous hidden state h_{t-1} and the current input x_t. The difference is, that the reset gate r_t decides which information of h_{t-1} should be removed, by an element-wise multiplication with values between 0 and 1. The final output is then a element-wise convex combination between the previous hidden state h_{t-1} and the candidate vector $\tilde{h}_t$. Where each coefficient of the convex combination is the output of the update gate z_t, a feed forward network with a sigmoid activation.

3.3 RNN Applications

The concepts discussed so far cover the basic building blocks of recurrent neural networks, that are necessary to process sequences of variable length. Furthermore, the modification of the recurrent cells with gating mechanisms enables RNN's to learn long term dependencies. These models can already be used in a wide range of tasks, e.g. document classification or time series forecasting. However, they are still limited to two cases: either they produce an output for every time step of the input sequence or they produce only a single output for the whole sequence. Additionally, the flow of information is limited to one direction: The input sequence is processed from left to right.

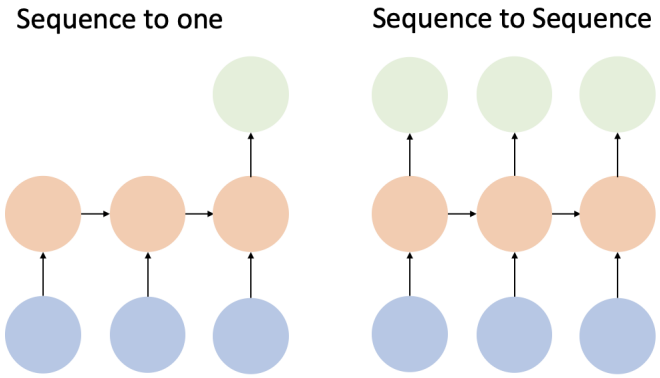

Figure 7: Different types of RNNs

3.3.1 Bidirectional RNN's

The recurrent neural networks covered so far, have one thing in common: The hidden state output at each time step is computed in a causal order, i.e. the output at step t only depends on inputs up to step t. Depending on the task however, this might not be the best approach. For example, consider a model for speech recognition (Graves et al. 2013). A correct interpretation of the current sound may be difficult, due to indistinct pronunciation or an ambiguous meaning of a specific word. Humans can easily bridge such uncertainties by inferring the correct interpretation from future context. Bidirec-

tional recurrent neural networks were invented by Schuster & Paliwal (1997) to address this issue.
As the name suggests, bidirectional RNN's generalize recurrent neural networks to operate in two directions. For this there are two separate RNN's. One processes the inputs in the forward direction from x_1 to x_τ, computing a sequence of hidden state $h_1, ..., h_\tau$ and the other one in the backward direction from x_τ to x_1 computing another sequence of hidden states $\tilde{h_\tau}, ..., \tilde{h_1}$. Both sequences of hidden states are then merged by e.g. concatenation and the output at each step contains information about the whole sequence.

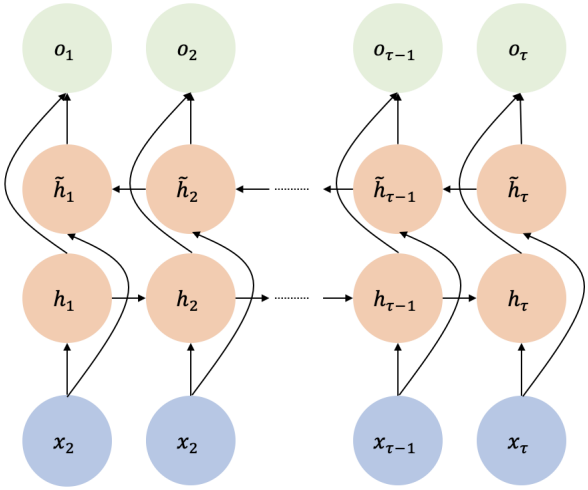

Figure 8: Bidirectional RNN

3.3.2 Encoder-Decoder

Cho et al. (2014) introduced the so called Encoder-Decoder architecture, that makes it possible to condition a variable length sequence on yet another variable length sequence. The architecture consists of two components. The encoder maps a variable length sequence to a fixed sized vector and the decoder takes this fixed sized vector and generates the output sequence from it. A popular application for such models is *neural machine translation*, since the amount of words in a sentence may vary among different languages.

The encoder consists of an RNN, that reads a input sequence $x_1, ..., x_T$, and generates a hidden state at each step. Eventually, the last hidden state h_T depends on the whole input sequence and can be interpreted as a fixed sized summary, also called the context vector c, of the variable length input sequence. The decoder consists of another RNN, that is trained to generate the output sequence $y_1, ..., Y_{T'}$, solely based on the context vector and the already predicted values.

There are several options, how a sequence can be generated from the context vector

c. Figure 9 illustrates the approach proposed by Sutskever et al. (2014). The context vector is used as the initial state of the decoder RNN and the input at the first position is a special start token ST. Note, that this requires the context vector dimension and the hidden state dimension to be equal.

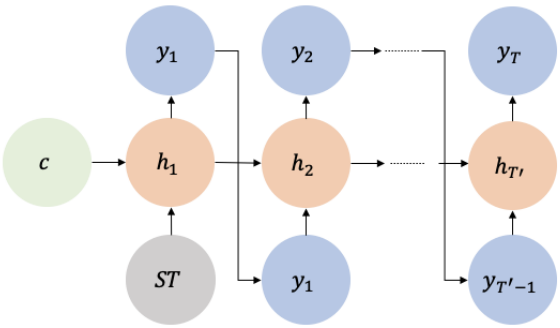

Figure 9: Generating sequences with an RNN 1

A different method (Cho et al. (2014); figure 10) is to use the context vector as additional input at every position. This can for example be achieved by a simple concatenation of the context vector and the input at the current position. Note, that for concatenation the dimensions of context vector and input vectors have to match.

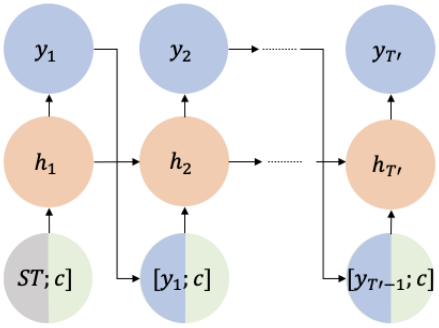

Figure 10: Generating sequences with an RNN 2

The main difference between both methods is, that the second approach uses the context vector at every position. This leaves room for improvement by adjusting the context vector at every time step.

3.3.3 Attention-Mechanisms

The psychological concept of attention refers to a bottleneck in the amount of data, the human brain can process at a time. In human vision for example, less than 1% of the visual input data can enter this bottleneck. The reason why this is not an obstacle, is that humans can easily shift their attention to different things for different tasks. (Zhaoping 2014, Chapter 1)

The idea of attention mechanisms in neural networks (Bahdanau et al. 2015) is similar: The network has access to a huge amount of time steps, from which it selectively chooses those, which are relevant to make the next prediction.

Reconsider the Encoder-Decoder architecture from the previous section. From a probabilistic perspective, such a model learns the conditional distribution of a variable length sequence $y = y_1, ..., y_T$ given another variable length sequence $x = x_1, ..., x_{T'}$:

$$p(y_1, ..., y_{T'}|x_1, ..., x_T) \tag{3.24}$$

It is important to note, that the lengths T and T' of these sequences may differ. The Encoder-RNN compresses the input sequence x to a context vector c of fixed size. The Decoder-RNN is trained to predict the next entry y_i of the target sequence, given the context vector c and the predictions $y_1, ..., y_{i-1}$ made so far.
The use of a fixed sized representation of a variable length input sequence introduces a bottleneck in the network architecture, that makes it difficult to capture all the relevant information of long sequences (Goodfellow et al. 2016, Chapter 12). Cho et al. (2014) and Sutskever et al. (2014) could partially overcome this problem, by increasing the model size in terms of parameters and the context vector dimension. This approach however, is not very efficient and does not solve the problem in a fundamental way. Bahdanau et al. (2015) proposed attention mechanisms to address this issue.

Let $x = x_1, ..., x_T$ and $y = y_1, ..., y_{T'}$ be the input and target sequence, respectively. The encoders output is then $enc(x) = h_1, ..., h_T$. Using the method from Figure 10, the attention mechanism at every decoding time step i works as follows:

1. Initialize the decoders hidden state s_0 (e.g. with the last hidden state of the encoder)

2. Compute an alignment score between the previous decoder hidden s_{i-1} state and every position of the encoders output $\mathrm{h}_1, ..., h_T$:

$$score_{ij} = align(s_{i-1}, h_j) \tag{3.25}$$

3. Normalization of the scores with the softmax function:

$$\alpha_{ij} = \frac{exp(score_{ij})}{\sum_{k=1}^{T} exp(score_{ik})}, \tag{3.26}$$

4. The context vector at decoding step i is then obtained by a weighted sum of the

encoder outputs h_j with the normalized scores α_{ij}:

$$c_i = \sum_{j=1}^{T} \alpha_{ij} h_j \tag{3.27}$$

5. The input to the decoder at step i is then the concatenation of the context vector c_i and the previous decoder output, along with the decoders previous hidden state.

In the original paper (Bahdanau et al. 2015), the alignment model in step 2 is a simple feed forward network with a hyperbolic tangent activation

$$align(s_{i-1}, h_j) = \sum \tanh(W_s s_{i-1} + W_h h_j). \tag{3.28}$$

where the sum reduces the output to a scalar value, i.e. a score that represents the relevance of the source sequence position j with respect to the target position i. This type of attention is often referred to as additive attention or Bahdanau-style attention.

4 The Transformer

The models in Chapter 3 relied on recurrent neural networks, that sequentially process a sequence of inputs $x_1, ..., x_T$ and output a sequence of hidden states $h_1, ..., h_T$, that capture the relevant information. Furthermore, attention mechanisms were introduced to allow the model to search for relevant information in the hidden states when making predictions. Vaswani et al. (2017) proposed a novel architecture, that dispenses recurrent neural networks entirely, by relying solely on attention mechanisms.

4.1 Self-Attention

The attention mechanism in section 3.3.3 was used to align positions in a source sequence with positions in a corresponding target sequence. Self-attention (sometimes called intra-attention; Cheng et al. (2016)) refers to the very same mechanism, that is used to align different positions within a single sequence.

For a sequence $x = x_1, ..., x_T$, the self-attention mechanism computes a vector of normalized scores of the same length T for every time step $i = 1, ..., T$ of the input sequence. Resulting in a $T \times T$ matrix A, where the entries α_{ij} are weights, that represent how well position i aligns with position j:

$$A = \begin{pmatrix} \alpha_{11}, \alpha_{12}, ..., \alpha_{1T} \\ \vdots \\ \alpha_{T1}, \alpha_{T2}, ..., \alpha_{TT} \end{pmatrix} = \begin{pmatrix} softmax(f(x_1, x_1), f(x_1, x_2), \cdots, f(x_1, x_T)) \\ \vdots \\ softmax(f(x_T, x_1), f(x_T, x_2), \cdots, f(x_T, x_T)) \end{pmatrix} \tag{4.1}$$

f can be an arbitrary function, that maps two same sized vectors to a scalar value. In practice, this function should be easy to compute and somehow represent the similarity between two vectors. It is important to note, that the $T \times T$ matrix A in 4.1 is obtained by repeated application of the same function f on different positions of the input sequence.
The matrix A can then be multiplied with the input sequence x, such that

$$Ax = \begin{pmatrix} \alpha_{11}x_1 + \alpha_{12}x_2 + \cdots + \alpha_{1T}x_T \\ \vdots \\ \alpha_{T1}x_1 + \alpha_{T2}x_2 + \cdots + \alpha_{TT}x_T \end{pmatrix} \tag{4.2}$$

is of the same length as the original input sequence $x_1, .., x_T$, and each position contains information about the whole sequence.

This approach allows to process sequences of variable length such as the recurrent neural network. However, self-attention works in a fundamentally different way. For RNN's the basic idea is to model the transition between two consecutive positions in a sequence for which the same parameters can be used at every time step. Self-attention shares the same function as well, but instead of focusing on transitions, the similarity between two arbitrary positions is used to produce an output, that incorporates positions from the whole sequence. More precisely, the output at every position is a weighted sum of all other sequence positions, with weights obtained by a model f.

This reflects still a difference to unidirectional RNN's, where the output at position t does only depend on earlier positions $1, ..., t-1$. Fortunately, the solution for this problem is quite simple. Every score $f(x_i, x_j)$ represents the similarity between position i and j. Therefore, positions that are not allowed to attend to each other can be masked out by setting $f(x_i, x_j) = -\infty$ and the $softmax$ operation will push the corresponding weights α_{ij} to zero. This allows the model to maintain a causal order in the output sequence.

4.2 Multi Head Attention

Vaswani et al. (2017) rephrased attention mechanisms with the concept of *query key* and *value*, which has been widely adopted by many researchers and commercial software implementations. An attention function can then be described as mapping a query and a set of key-value pairs to an output. The output is a weighted sum of the values, with weights obtained by a compatibility function of the query and keys.

The particular attention function used in the Transformer is called *scaled dot product attention*. This mechanism takes matrices of queries $Q \in \mathbb{R}^{T \times d_k}$, keys $K \in \mathbb{R}^{S \times d_k}$ and values $V \in \mathbb{R}^{S \times d_v}$ as input. In terms of sequential data, the rows represent time steps and the columns represent the multidimensional observations at every position. An alignment score is computed, that determines how well the i-th row of Q matches the j-th row of K, by computing the dot products of all rows in Q with all rows in K. These scores are then divided by $\sqrt{d_k}$ and normalized with the softmax function. The final output is then the values weighted by the normalized scores:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{4.3}$$

This is essentially the same as in Luong et al. (2015), except for the scaling factor $\frac{1}{\sqrt{d_k}}$. Vaswani et al. (2017) suspect, that the dot product can grow large in magnitude for large values of d_k, pushing the softmax into regions with extremely small gradients. A more probabilistic view is the following: Assume, that the entries of q and k are independent random variables with mean 0 and variance 1. The dot product $\sum_{i=1}^{d_k} q_i k_i$ then

has a variance of d_k. Thus, the scaling factor counteracts the growing variance.

The idea of scaled dot product is extended to a mechanism called *multi-head attention* (Vaswani et al. 2017). For this, the key, value and query vectors are linearly projected to h separate vectors of dimension $\tilde{d_k}$ for queries and keys and dimension $\tilde{d_v}$ for the values. On each of these projections the scaled dot product attention is performed in parallel, yielding h output values of dimension $\tilde{d_v}$. These outputs are then concatenated and again linearly projected.
Let $Q \in \mathbb{R}^{T \times d_q}$, $K \in \mathbb{R}^{S \times d_k}$ and $V \in \mathbb{R}^{S \times d_v}$ be the queries, keys and values. For learnable weight matrices $W_Q^i \in \mathbb{R}^{d_q \times \tilde{d_q}}$, $W_K^i \in \mathbb{R}^{d_k \times \tilde{d_v}}$, $W_V^i \in \mathbb{R}^{d_v \times \tilde{d_k}}$ and $W_{out} \in \mathbb{R}^{h\tilde{d_v} \times d_v}$ the multi-head attention is computed as

$$\begin{aligned} MultiHead(Q,K,V) &= Concat(head_1, ..., head_h)W_{out}, \\ \text{with } \; head_i &= Attention(QW_Q^i, KW_K^i, VW_V^i). \end{aligned} \tag{4.4}$$

The intuition of multi headed attention is, that each attention head can specialize to specific tasks. For example, one head could focus on the immediate neighborhood, while another one captures long term dependencies. In practice, the dimension of each attention head is coupled to the amount of heads, such that the computational effort stays roughly the same.

4.3 Transformer Architecture

The transformer architecture (Vaswani et al. 2017) is similar to the recurrent Encoder-Decoder with attention mechanisms described in section 3.3.2 and 3.3.3, with the difference, that RNN's are replaced with self-attention. The encoder processes an input sequence $x = x_1, ..., x_T$ of variable length and outputs a sequence of the same length $h_1, ..., h_T$. Each h_i depends on x_i and, depending on the masking strategy, arbitrary additional time steps $x_j, j \in \{1, ..., T\}$. Note, that this is a difference to RNN's, where each h_i depends on all previous time steps (or the whole sequence in case of bidirectional RNN's). Another difference is, that the there are (possibly) multiple attention mechanisms, that align the encoders output with the target sequence.

The encoder of the transformer is composed of N identical layers. Each layer consists of a multi-head self-attention mechanism followed by a fully connected feed forward network. Both sub-layers are followed by a normalization layer and skip-connections

(He et al. 2016) are employed around them. The output of the encoder is computed as:

$$\begin{aligned} \tilde{att} &= MultiHead(X, X, X) \\ att &= Normalization(\tilde{att} + X) \\ ffn &= feedforward(att) \\ output &= Normalization(att + ffn) \end{aligned} \tag{4.5}$$

For the first layer, the input is the matrix $X \in \mathbb{R}^{T \times d}$ containing the input sequence, i.e. one row represents one d-dimensional time step. And for the subsequent layers, the input is the output of the previous layer. The multi-head attention is computed as in (4.4). *Normalization* corresponds to *Layer Normalization* as in Ba et al. (2016). In contrast to *Batch Normalization* (Ioffe & Szegedy 2015), which is commonly used in computer vision models, the normalization is applied for each example in a batch independently. The feed-forward network is a simple fully connected network with one hidden layer.

Likewise, the decoder consists of N identical layers, which are slightly more complex. Self-attention is computed along the decoders input sequence. The output is fed as query to another multi-head attention mechanism with key and value from the encoders output:

$$\begin{aligned} \tilde{att}1 &= MultiHead(Y, Y, Y) \\ att1 &= Normalization(\tilde{att}1 + Y) \\ \tilde{att}2 &= MultiHead(att1, enc_{out}, enc_{out}) \\ att2 &= Normalization(\tilde{att}2 + att1) \\ ffn &= feedforward(att2) \\ output &= Normalization(ffn + att2) \end{aligned} \tag{4.6}$$

The input to the first layer is the input sequence $y_1, ..., y_T$ and the subsequent layers take the output of the previous layer as input. The key and value in the second attention mechanism are identically set to the encoder output in every layer. This means, that the final output of the decoder layer would be just a weighted sum of the encoders output, if it weren't for the skip connections. This gives the residual connections a greater meaning than just stabilizing the gradients.

4.3.1 Positional Encoding

In contrast to recurrent neural networks, the transformer architecture does not explicitly model the absolute or relative positions in the input sequences. At each time step, the transformer maps the query against all keys and produces a weighted sum of the corresponding values. This means, that a random permutation of the input positions

would result in the exact same output values, except for that permutation. To overcome this problem, the positional information about every time step has to be encoded in the input sequence. There are two major approaches for creating such *positional encodings*: Learned encodings, that are trained jointly with the model and fixed encodings, that are computed with a predefined function of the absolute position in the sequence.
Vaswani et al. (2017) propose a fixed encoding of sine and cosine functions of different frequencies, that are:

$$\begin{aligned} PE(pos, 2i) &= sin(\frac{pos}{10000^{2i/d_{model}}}) \\ PE(pos, 2i+1) &= cos(\frac{pos}{10000^{2i/d_{model}}}) \end{aligned} \tag{4.7}$$

where *pos* is the absolute position in the sequence and i corresponds to the dimension of the input sequence. These encodings are then simply added to the models input sequences.

5 Knowledge Tracing

Computer-assisted education promises a personalized learning experience, which is independent of well educated teachers or nearby educational institutions. Especially for the 258 million children, that were not attending school in 2020, such systems would reduce the minimal requirements for education to a reliable internet connection. In addition to that, the covid-19 pandemic highlighted, how beneficial such systems are even in industrialized countries that struggled with keeping up the educational standard, when schools were forced to shut down.
Knowledge Tracing is the task of modeling students knowledge over time, as they interact with coursework. More precisely, such models predict how well a student will perform on a specific task, given the history of already answered questions. Accurate models would allow to allocate content based on the individual needs, such that tasks that are too easy for the student can be skipped and tasks that are too hard can be delayed until they fit the students knowledge.

5.1 Bayesian Knowledge Tracing

Bayesian Knowledge Tracing (BKT; Corbett & Anderson (1995)) was a popular approach for modeling students knowledge over time and has a long history of being actively used in computer assisted tutoring systems (Yudelson et al. 2013). BKT models students knowledge as a set of binary variables, where each variable represents the understanding of a single concept/skill. BKT assumes, that once a concept is learned, it can not be forgotten. Besides the probability of knowing a concept, BKT also models the probability of answering a question, that belongs to an unknown concept, correctly, as well as the probability for a wrong answer on a known concept. Summarized, there are four parameters for each skill concept k (Yudelson et al. 2013):

1. The probability of knowing the skill a priori $p(L_0)^k$
2. The probability for a transition from *skill is not known* to *skill is known* $p(T)^k$
3. The probability of making a mistake, although the skill is known $p(M)^k$
4. The probability of applying an unknown skill correctly $p(G)^k$

For a students response at time step t on concept k, the probability $P(L_t^k)$ that a student knows the concept k is computed according to Bayes rule. For a correctly answered question it is

$$P(L_t^k \mid correct_t^k) = \frac{P(L_t^k)(1 - P(M^k))}{P(correct_t^k)} \tag{5.1}$$

and for an incorrect response it is

$$P(L_t^k \mid incorrect_t^k) = \frac{P(L_t^k)P(M^k)}{P(incorrect_t^k)}. \tag{5.2}$$

The students knowledge of concept k is updated as the sum of the probability that the concept was known beforehand and the probability that the recent interaction made the student learn the skill:

$$P(L_t^k) = P(L_{t-1}^k \mid interaction_{t-1}^k) + (1 - P(L_{t-1}^k \mid interaction_{t-1}^k))P(T^k) \tag{5.3}$$

The probability, that a student answers a question on concept k at time step t correctly is

$$P(correct_t^k) = P(L_t^k)(1 - P(M^k)) + (1 - P(L_k^c))P(G^c). \tag{5.4}$$

There are several extensions to this model. For instance, the use of student-specific parameters (Yudelson et al. 2013) or the estimation of individual problem difficulty (Pardos & Heffernan 2011). However, with or without such extensions, bayesian knowledge tracing suffers from several difficulties. First, it may be unrealistic to represent students knowledge with binary variables. Second, the meaning of the hidden variables and their mappings onto exercises can be ambiguous, rarely meeting the model's expectation of a single concept per exercise. (Piech et al. 2015)

5.2 Deep knowledge Tracing

With the advances in deep learning architectures, increasing computational power and larger datasets, neural networks started to outperform traditional models, like bayesian knowledge tracing, and became the new standard for this task (Choi et al. 2020). The advantage of neural networks (NN's) over the hidden Markov model from the previous section is, that NN's use a high-dimensional and continuous representation of the latent state instead of the binary variables used in bayesian knowledge tracing. The rich representation and the ability to detect dependencies over many time steps make neural networks a pefect fit for the task of knowledge tracing.

Formally, the activity of a student is recorded as a sequence of interactions $I_1, ..., I_n$, where $I_i = (Q_i, R_i)$ is a tuple of question and response information. $Q_i = (q_i^1, ..., q_i^K)$ is a tuple itself and refers to the question information at step i. Each q_i^j is a categorical or continuous feature, that holds meta-information about the specific question. These features can be as general as a certain subject area, that is shared across many questions or as specific as a unambiguous id assigned to each question. Similar, each $R_i = (r_i^1, ..., r_i^L)$ is a tuple of response related features, such as the time the student spent on solving the exercise and, of course, the students response itself. The aim is to

predict a binary outcome at each time step, indicating, whether the student answered the current question correctly.

The first approaches to deep knowledge tracing used simple recurrent neural networks and LSTM's (Piech et al. 2015) and were extended to bidirectional networks with attention mechanisms (Liu et al. 2021). These networks process the sequence of user-interactions $I_1, ..., I_n$ with bidirectional LSTM's yielding sequences of the same length as output. An attention mechanism aligns the question information with those outputs before predicting the probability for a correct answer. Most recent architectures dispense with recurrent networks and rely on the transformer architecture (Choi et al. 2020, Shin et al. 2021).

6 Riiid AIEd Challenge 2020

Riiid! is a South Korean Company that disrupted the education market by its AI driven tutor solution (*Riiid* 2014). In 2017 they launched *Santa TOEIC* (Test of English for International Communication), an AI driven tutor solution. Since then more than one million South Korean students have been attracted and a substantial amount of data has been gathered and published (Youngduck et al. 2020). In 2020 they hosted the *Riiid! Answer Correctness Prediction* Challenge on Kaggle[2] looking for innovative algorithms for modeling students knowledge over time. More specifically the overall goal is to develop a model, that is capable of predicting whether a user answers a given question correctly based on the user's historical performance. This Chapter captures the participation in the Challenge using the previously discussed theory for sequential deep learning models.

6.1 Dataset Description

The dataset provided by the host of the competition consists of real world data gathered from user interactions with *Santa TOEIC*, a tutoring service, which aims to prepare students for the TOEIC (Test of English for International Communication) Listening and Reading Test. The dataset provides the same information as an actual education app would have for predicting whether a user would answer a given question correctly: The students interaction history, the performance of other students for the same question and additional meta-information about the questions.

The host provided the following dataset description:

- *timestamp*: The time in milliseconds between this user interaction and the first event completion from that user.
- *user-id*: ID code for each user.
- *content-id*: ID code for the user interaction.
- *content-type-id*: 0 if the event was a question being posed to the user, 1 if the event was the user watching a lecture.
- *task-container-id*: ID code for the batch of questions or lectures. For example, a user might see three questions in a row before seeing the explanations for any of them.Those three would all share a task-container-id
- *user-answer*: the user's answer to the question, if any. Read -1 as null, for lectures.
- *answered-correctly*: if the user responded correctly. Read -1 as null, for lectures

[2]The official description of the challenge can be found here: `https://www.kaggle.com/c/riiid-test-answer-prediction`

- *prior-question-elapsed-time*: The average time in milliseconds it took a user to answer each question in the previous question bundle, ignoring any lectures in between. Is null for a user's first question bundle or lecture. Note that the time is the average time a user took to solve each question in the previous bundle.
- *prior-question-had-explanation*: Whether or not the user saw an explanation and the correct response(s) after answering the previous question bundle, ignoring any lectures in between. The value is shared across a single question bundle, and is null for a user's first question bundle or lecture. Typically the first several questions a user sees were part of an onboarding diagnostic test where they did not get any feedback.

The meta information for the questions is:

- *question-id*: Foreign key for the train/test content-id columns, when the content type is question (0).
- *bundle-id*: Code for which questions are served together.
- *correct-answer*: The answer to the question. Can be compared with user-answer column to check if the user was right.
- *part*: The relevant section of the TOEIC test.
- *tags*: One or more detailed tag codes for the question. The meaning of the tags will not be provided, but these codes are sufficient for clustering the questions together.

The data contains roughly 100M answered questions from $\sim$ 400K different users. From a sequential modeling perspective the data can be treated as 400K i.i.d. sequences of variable length, where each step is one user interaction. A single interaction refers to either a watched lecture video or to the responses to a bundle of up to five questions. The interactions, which solely contained lectures, were filtered out of the data, because the use of this information did not contribute to the models performance. There are a total of 13523 different questions, split into 9765 question-bundles, with a maximum of five questions per bundle. Each question belongs to one of the 7 parts of the TOEIC-Test and has a variable amount of tags attached, that provide additional meta information. All questions are multiple choice, with exactly one correct answer.

Figure 11 illustrates the user interactions: The student is presented a variable amount of questions at one time step and the next time step contains the responses for those questions. *Elapsed time* represents the average time the user spent on solving the exercises, which is limited to 5 minutes. The timestamp provided in the dataset refers to *the time in milliseconds between this user interaction and the first event completion*

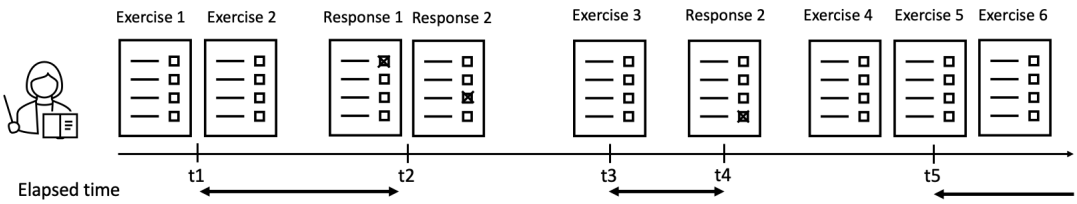

Figure 11: User interactions over time

from that user. In terms of Figure 11, the timestamps would be $timestamp_1 = t2$, $timestamp_2 = t4 - t2$ and $timestamp_3 = t6 - t2$.

As can be seen in Figure 12, the sequence lengths vary between 1 and 10000. However, only 4% of the users have more than 1000 interactions (note the logarithmic scale in figure 12). Figure 13 shows the average correctness for the j-th interaction across all users. A clear trend is visible, that the correctness increases with the amount of questions a user has answered. However, there is still an uncertainty whether the students improve their knowledge or the questions become easier.

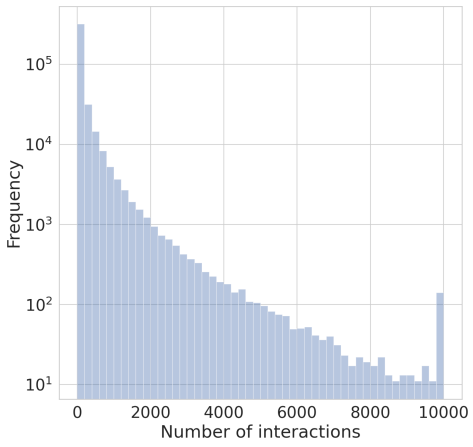

Figure 12: Histogram of sequence lengths

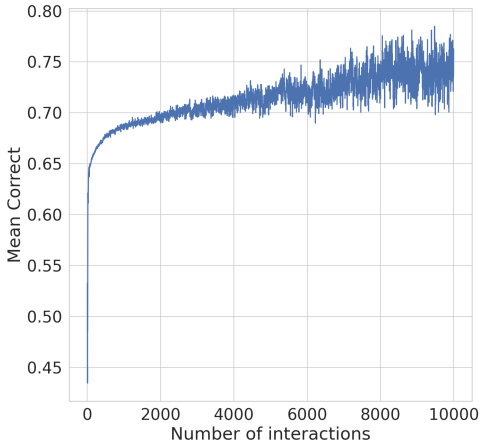

Figure 13: Average correctness over time

6.2 Problem Definition

The overall goal of the challenge was to develop a model, that takes a variable length sequence of user interactions $I_1, I_2, ..., I_T$ as input, where $I_i = (R_i, Q_i)$ is a tuple of question and response information corresponding to the i-th interaction, and predicts the probabilities for answering each question in the next question bundle correctly.

The fact that the predictions have to be made for a variable amount of questions

at each step distinguishes the problem from Choi et al. (2020) and Pandey & Karypis (2019). When training a sequential model this difference has to be treated carefully in order to avoid data leakage. Data leakage refers to a mistake in a predictive model, in which the model is trained on information that would not be available when run in a productive environment. In the usual time series setup it is sufficient, to ensure that the output at step t does only depend on inputs from step 1 to t. However, this is not satisfactory for the problem at hand.

position	Bundle	Question features	Prior Response Features	Target
1	1	Q_1	PAD	r_1
2	2	Q_2	R_1	r_2
3	2	Q_3	R_2	r_3
4	3	Q_4	R_3	r_4

Table 1: Exemplary input sequence

Table 1 illustrates the format of the data for a user that answered four questio. Every position in the sequence contains the current question information and the response information from the previous question. At the first position in the sequence the prior response feature is set to a padding value and the probability for a correct answer has to be predicted with only the question information. At position 2 the available information is the question from the first position and its corresponding response (Q_1, R_1), the question features from the current question Q_2 and additionally the question features from the next position Q_3, because position 2 and 3 belong to the same question bundle. For position 3 the exact same features as for position 2 can be used. Note that the response R_2 to question Q_2 can not be used to predict r_3, because R_2 becomes available not until all questions in the bundle have been answered.
Summarized, the model has to predict the probability of a correct answer for every question in the bundle, before seeing the response for any of them. I.e. the probabilities of interest are:

1. $P(r_1 \mid Q_1)$
2. $P(r_2 \mid (Q_1, R_1), Q_2, Q_3)$
3. $P(r_3 \mid (Q_1, R_1), Q_2, Q_3)$
4. $P(r_4 \mid (Q_1, R_1), (Q_2, R_2), (Q_3, R_3), Q_4)$

More generally, when predicting the targets for a question bundle ranging from position i to $i+s$, the model can attend to question features from position 1 to $i+s$ and to response features from position 1 to $i-1$. Every additional dependency between input and output would cause a leak of information during training.

6.3 Preprocessing

Each user history of length $T \in \mathbb{N}$ contains a sequence of question information $Q_1, Q_2, ..., Q_T$, and a sequence of response information $R_1, R_2, ..., R_{T-1}$. The features used for Q_i are:

- *Container-id*: categorical feature with 10000 categorie
- *Question-id*: categorical feature with 13000 categories
- *Question part*: categorical feature with 8 categories
- *timediff*: numerical feature

and the response related features R_i are:

- *Prior answer correct*: categorical feature with 2 categories
- *Prior question had explanation*: categorical feature with 10000 categories
- *Prior question elapsed time*: numerical feature

Some features, like the tags and correct answers of each question, were excluded from the models. This keeps the amount of features small, while preserving as much information as possible.
Since most of the features are categorical, not much preprocessing is necessary. *Prior question elapsed time* is technically a numerical feature. However the exploratory analysis revealed, that it only takes 2800 distinct values $et_1, et_2, ..., et_{2800}$. Therefore, these values are used to create bins $(-\inf, et_1], (et_1, et_2], ..., (et_{2799}, et_{2800}]$, each representing one category. The models input for that feature is then the category of its corresponding bin. The numerical feature *timediff* is computed as the normalized logarithm of the first order difference of the timestamps x_i, i.e.. $\textit{timediff} = \frac{log(x_t - x_{t-1}+1)}{\textit{timediff}_{max}}$. Where the $+1$ is necessary to avoid taking the logarithm of 0 and dividing by $\textit{timediff}_{max}$, the maximum value of $log(x_t - x_{t-1} + 1)$ across the whole dataset, normalizes the values between -1 and 1.

6.4 Sampling Strategy

For sequential deep learning models, the sampling strategy is an important aspect. There are two classes of such models one needs to differentiate. Models that process the input sequence as a whole (e.g. bidirectional RNN's) do not maintain a causal order and can only learn from the label at the last position in the sequence. On the other hand, models, which ensure that the output at step t does only depend on inputs up to step t (e.g. Transformers), can learn from every position in a single sequence.
To get the most out of the data, every labeled time step in every user sequence should be used to train the models. For a user history of length T, Transformers can be trained with a single input sequence $I_1, ..., I_T$ containing the whole history. For a bidirectional RNN however, one would need T distinct sequences $(I_1), (I_1, I_2), ..., (I_1, I_2, ..., I_T)$.

In theory, this approach would be sufficient. In practice however, the sequences had to be truncated to fit the computational resources. Therefore, each sequence of length T is split into $\lceil \frac{T}{S} \rceil$ sub-sequences of length W by sliding a window of size W, starting at the right end of the sequence, S steps to the left. Remaining elements in the last window are pre-padded to length W. The input-pipeline fills a buffer of 100000 such sequences and then randomly samples elements from it, replacing selected sequences with new sequences. Perfect shuffling would require a buffer size as large as the whole dataset. Due to the amount of data this was not possible. However, the order in which the user histories are read from file is randomized and the buffer is large enough to break the correlations in sequences from the same user. One input sample is then a sequence of tuples $I_1, ..., I_W = (Q_1, PAD), (Q_2, R_1), ..., (Q_W, R_{W-1})$ and the corresponding targets $r_1, ..., r_W$ indicate whether the questions were answered correctly.
Consider a sequence of interactions $I = I_1, I_2, I_3, I_4, I_5, I_6$. The selected sub-sequences for given W and S are:

$W = 3$ $S = 1$	$W = 3$ $S = 2$	$W = 3$ $S = 3$
• $s_1 = I_4, I_5, I_6$ • $s_2 = I_3, I_4, I_5$ • $s_3 = I_2, I_3, I_4$ • $s_4 = I_1, I_2, I_3$ • $s_5 = PAD, PAD, I_1$	• $s_1 = I_4, I_5, I_6$ • $s_2 = I_2, I_3, I_4$ • $s_3 = PAD, I_1, I_2$	• $s_1 = I_4, I_5, I_6$ • $s_2 = I_1, I_2, I_3$

Table 2: Sampling strategy

This approach covers a wide range of different sampling strategies, by adjusting just two parameters. Increasing W usually improves the models performance, since the model can attend to a longer history of answered questions. S controls the granularity

of the sequences presented to the model. The larger the value, the fewer sub-sequences are generated from each user history. Thus, the value should be chosen as small as possible. Note that S is only active at training time, since during inference the window is always slid by one step.

6.5 Input-representation

The input pipeline streams sequences of question information $Q = Q_1, Q_2, ..., Q_W$, where each time step t is a four-dimensional vector:

$$Q_t = \begin{pmatrix} Containerid_t \\ Questionid_t \\ Questionpart_t \\ timediff_t \end{pmatrix}^T \in \mathbb{R}^4 \tag{6.1}$$

The sequence of response information $R = R_1, ..., R_{W-1}$ is a three-dimensional vector at every time step:

$$R_t = \begin{pmatrix} Prioranswercorrect_t \\ Priorquestionhadexplanation_t \\ Priorquestionelapsedtime_t \end{pmatrix}^T \in \mathbb{R}^3 \tag{6.2}$$

For each categorical feature in Q_t and R_t an embedding is trained. I.e. for every feature k a mapping $E_k : X_k \to \mathbb{R}^{d_{model}}$ assigns a latent vector of size $d_{model} = 128$ to every category of feature k. For the numerical feature a single learnable vector e of size d_{model} is multiplied with the feature value.

$$\begin{aligned} \tilde{Q}_t^e &= \begin{pmatrix} E_1(Containerid_t) \\ E_2(Questionid_t) \\ E_3(Questionpart_t) \\ timediff_t * e \end{pmatrix}^T \in \mathbb{R}^{4 \times d_{model}} \\ \tilde{R}_t^e &= \begin{pmatrix} E_4(Prioranswercorrect_t) \\ E_5(Priorquestionhadexplanation_t) \\ E_6(Priorquestionelapsedtime_t part_t) \end{pmatrix}^T \in \mathbb{R}^{3 \times d_{model}} \end{aligned} \tag{6.3}$$

Finally, the dimension of each Q_t^e and R_t^e is reduced to a vector of size d_{model} by taking the sum over the first axis.

$$\begin{aligned} Q_t^e &= \sum \tilde{Q}_t^e \in \mathbb{R}^{d_{model}} \\ R_t^e &= \sum \tilde{R}_t^e \in \mathbb{R}^{d_{model}} \end{aligned} \tag{6.4}$$

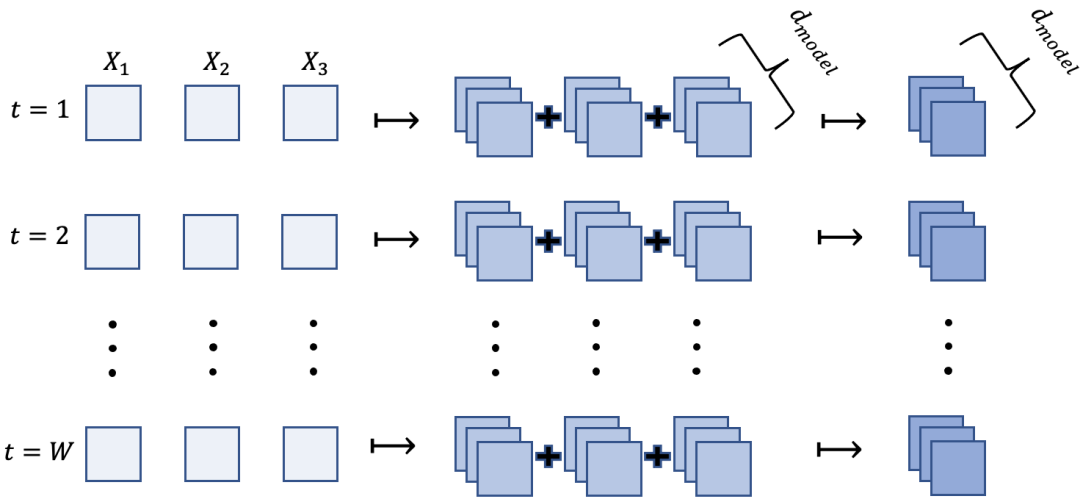

Figure 14: Input representation

This approach has a few downsides. E.g. consider the feature *prioranswercorrect*. It is definitively an overkill to represent a binary feature with a vector of size 128. A more intuitive approach would be to carefully think about the complexity of each feature and choose a more appropriate embedding dimension and then concatenate the features to get the final embedding. This advantage however would only spare a few hundred parameters at most, which seems negligible in contrast to the millions of other parameters that have to be trained anyway. Using the same embedding dimension for every feature allows for more flexibility, when adjusting the overall dimension of the model. Moreover, this is the approach suggested in Choi et al. (2020) and Shin et al. (2021).
In the following, $Q^e = Q_1^e, ..., Q_W^e$ and $R^e = R_1^e, ..., R_{W-1}^e$ refers to a sequence of question and response embeddings, respectively.

6.6 Model Architectures

In this section two approaches for tracing students knowledge over time are presented: One is based on bidirectional recurrent neural networks and the other one on the Transformer model. The architectures are inspired by (Choi et al. 2020) and adjusted to the present dataset. Both approaches take the previously described sequences of question and response embeddings as input and predict the probability that a given question

will be answered correctly. The main difference between the architectures is, that the bidirectional recurrent network generates outputs only for the last question bundle in the input sequence, whereas the transformer is truely a sequence to sequence model, generating an output for each position in the input sequence.

6.6.1 Bidirectional GRU

The idea of the network-architecture (similar as in Liu et al. (2021)) is to use bidirectional GRU-layers, that extract the information, which is relevant to represent the students knowledge from a sequence of question-response pairs. The output is of the same length and each position could be interpreted as the knowledge obtained by this user interaction with respect to all other interactions, future and past. An attention mechanism maps then the questions from the target bundle against the output of the GRU layers and retrieves those outputs, which align well with the current target question, to make the final prediction for a correct answer.

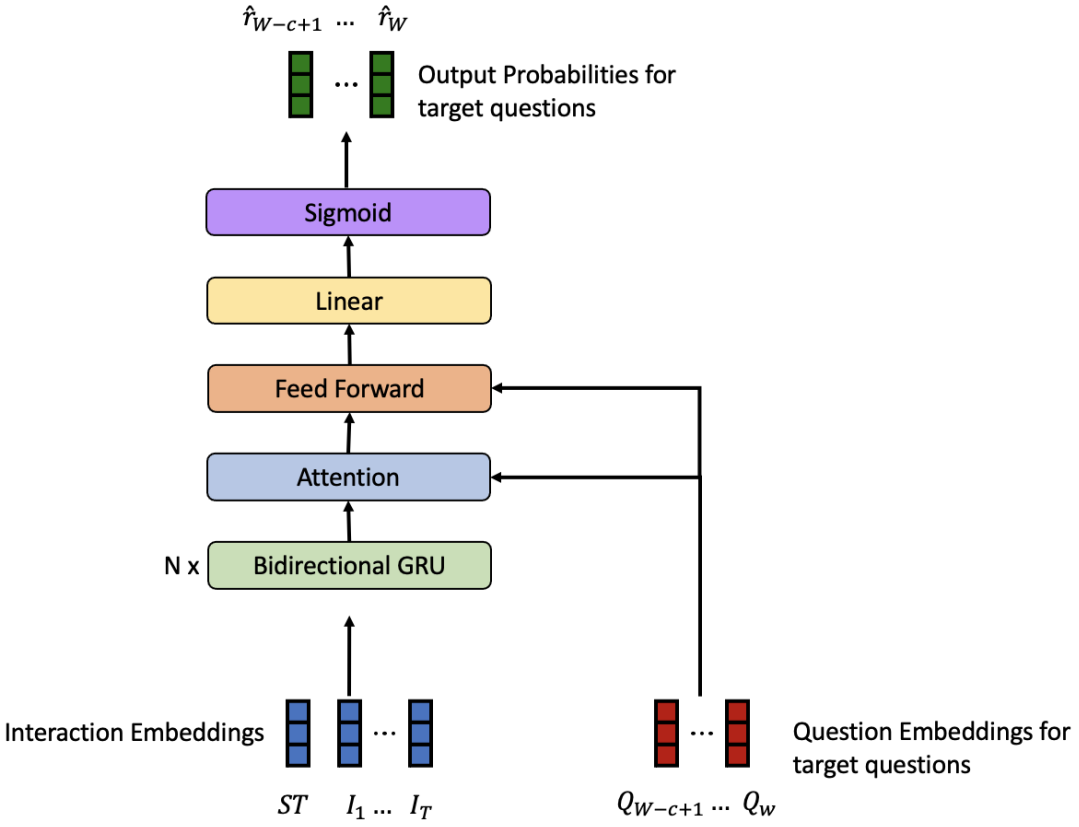

Figure 15: Bidirectional Recurrent Network with Attention

The model takes as input the sequences of question and response embeddings of dimension $d_{model} = 128$, as described in section 6.5, where the response information at step t corresponds to the question information at step $t - 1$. Additionally there is a third sequence $B_1, ..., B_W$, that indicates the question bundle of each question. This

sequence is used to determine the size of the last question bundle:

$$c = \sum_{i}^{W} \mathbb{1}_{[B_W = B_i]} \tag{6.5}$$

Where $\mathbb{1}_{[B_W=B_i]}$ is 1 if $B_W = B_i$ and else 0.
The question sequence is then truncated to c target questions $Q_{target} = Q_{W-c+1}, ..., Q_W$ that all share the same question bundle. In case $c > 1$ this leads to the problem, that the last $c-1$ responses refer to the target questions $Q_{W-c+1}, ..., Q_{W-1}$ and can not be used to predict the probabilities that the questions in Q_{target} will be answered correctly. Therefore, these responses are overwritten with a padding-value:

$$R_t = \begin{cases} PAD & \text{if } W - t > c, \\ R_t & else \end{cases} \tag{6.6}$$

After this step, every possibility for data leakage is eliminated, because any response information regarding the questions in Q^{target} is removed. Note, that for the simplest case with a bundle of size $c = 1$, it holds $W - t \leq c$ for all $t = 1, ...W$ and it is not necessary to overwrite any values.
The sequences of response and question information are then aligned and summed up to a sequence of interaction embeddings $I^e = ST, Q_1 + R_2, ..., Q_{W-1} + R_W$, where ST is a learnable start-token, and fed into a stack of $N = 2$ bidirectional GRU's (encoder):

$$\begin{aligned} \overrightarrow{out} &= GRU_{forward}(\overrightarrow{I^e}) \\ \overleftarrow{out} &= GRU_{backward}(\overleftarrow{I^e}) \\ gru^{out} &= [\overrightarrow{out}; \overleftarrow{out}] \end{aligned} \tag{6.7}$$

For each layer there are two distinct GRU's, each with d_{model} units, that process the sequence forward and backward. The outputs of the forward and backward GRU are then concatenated along the last axis, such that the final output gru^{out} is of shape $(W-1, 2d_{model})$. The first layer takes the sequence of interaction embeddings and the subsequent layers use the output from the previous layer as input.
An attention mechanism is then used to align each question in Q_{target} with the output-sequence of the bidirectional GRU. For every question in Q_{target} the output is a weighted sum of the encoders output, where the weights are large, for positions that are similar to the target question and small for positions that are dissimilar:

$$att_i = \sum_{j=1}^{W-1} \alpha_{ij} gru_j^{out} \tag{6.8}$$

The index $i = 1, ..., c$ refers to the sequence of target questions Q^{target} and the index $j = 1, ...W - 1$ refers to the output sequence of the GRU layers.

The weights α_{ij} are normalized scores, which are computed with an alignment model, that has weight matrices $W_q \in \mathbb{R}^{d_{model} \times d_{model}}$ and $W_v \in \mathbb{R}^{2d_{model} \times d_{model}}$:

$$\begin{aligned} score_{ij} &= tanh(Q_i^{target} * W_q + gru_j^{out} * W_v) \\ \alpha_{ij} &= \frac{exp(score_{ij})}{\sum_{k=1}^{W} exp(score_{ik})} \end{aligned} \tag{6.9}$$

Each output of the attention mechanism att_i is concatenated with its corresponding target-question Q_i^{target} and a fully connected feed forward network is applied to every position i. This network has a single hidden layer with $2d_{model}$ units and a $relu$ activation function:

$$fc^{out} = fullyconnected([att; Q^{target}]) \tag{6.10}$$

Finally a linear layer reduces the dimension to a single output for each question in Q^{target} and the sigmoid function is applied to squeeze the outputs between 0 and 1, i.e. the probability that the question will be answered correctly:

$$p = sigmoid(linear(fc^{out})) \tag{6.11}$$

The forward pass as described here, only operates on a single example. In practice, these computations are done simultaneously for batches of $B = 8192$ sequences of length $W = 300$ and the following loss is computed:

$$loss(x) = \frac{1}{B} \sum_{n=1}^{B} \sum_{i=0}^{c_n} BCE(y_{n,W-i}, p_{n,W-i}) \tag{6.12}$$

Where c_n is the amount of questions, that are in the same question bundle as the question at position W and $BCE(y_{n,W-i}, p_{n,W-i})$ is the binary cross entropy for the target and predicted values of the $(W-i)$-th question in the n-th input sequence.
The following hyper-parameters yielded reasonable results:

Number of encoder layers	$N = 2$
Model dimension	$d_{model} = 128$
Sequence length	$W = 300$
Step size of the sliding window	$S = 1$
Batchsize	$B = 8192$
learning-rate	$\alpha = 0.0005$

Table 3: Hyper-parameters: Bidirectional GRU

6.6.2 Transformer

The model uses the vanilla Transformer architecture, as described in Chapter 4. The Encoder processes the sequence of user interactions and generates an output of the same length. In contrast to the bidirectional GRU, a masking mechanism keeps the output in causal order with respect to the question bundles. The decoder takes the sequence of questions as input, attends on its own and on the encoder's output to predict at each step, whether the question was answered correctly.

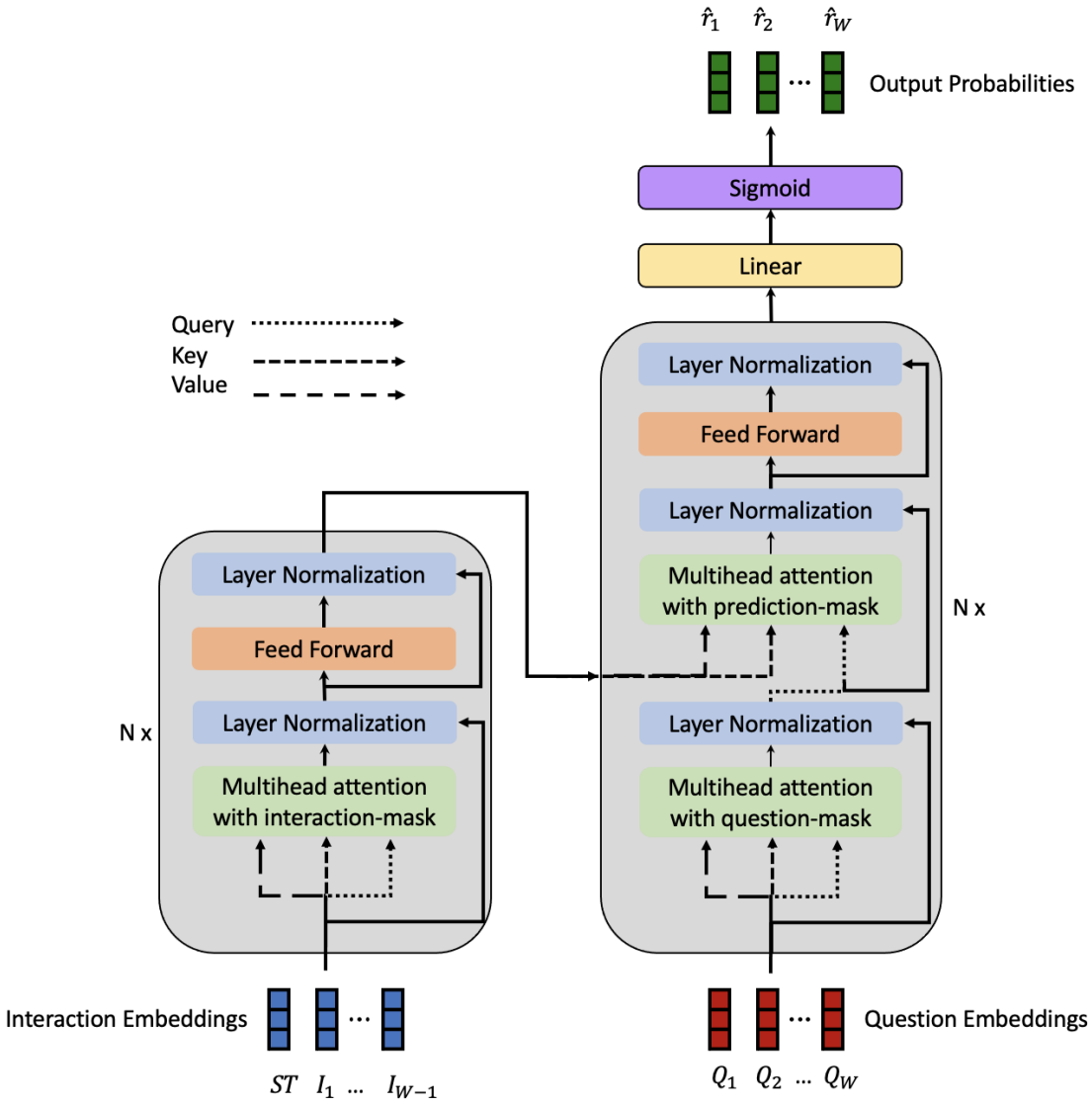

Figure 16: Transformer model for Knowledge Tracing

The decoder uses two separate masks, where one prevents the questions from attending to future question bundles and the other one determines, which positions from the encoders output can be used for each prediction. Loosely speaking the encoder extracts the students knowledge from the history of interactions and the decoder learns what

kind of knowledge concept a given question requires and looks in the encoders output how profound the users knowledge is.

The model takes the same input as the bidirectional GRU: the sequences of question and response embeddings of dimension $d_{model} = 128$, as described in section 6.5, where the response information at step t corresponds to the question information at step $t-1$. Additionally there is a third sequence $B_1, ..., B_W$, that indicates the question bundle of each question. The response and question sequences are again aligned and added to a sequence of interaction embedding $I^e = ST, Q_1 + R_2, ..., Q_{W-1} + R_W$, where ST is a learnable start token embedding.

Encoder

Each encoder layer is composed of two sub-layers. A multi-head self-attention mechanism, where each interaction can attend to interactions from the same and from previous question bundles. Note, that interactions at position t can attend to future positions, if the interaction at step $t+1$ comes from the same question bundle. The other sub-layer is a fully connected feed-forward network, applied to every position independently. There are residual connections around both sub layers followed by a layer normalization. Dropout is used as regularization to avoid overfitting:

$$\begin{aligned} attn &= dropout(MHA(x, x, x, interaction\text{-}mask)) \\ out1 &= layernorm(attn + x) \\ fc &= droput(fullyconnected(out1)) \\ out2 &= layernorm(fc + out1) \end{aligned} \tag{6.13}$$

The input to the first layer is the sequence of interaction embeddings I^e and in the subsequent layers the input is the output from the previous layer. The *interaction-mask* can be computed from the sequence of bundle-ids as described below.

Decoder

The Decoder receives the encoders output and the sequence of question embeddings as input. A multi-head self-attention mechanism is applied to the input-sequence and the output is then used as query in another multi-head attention mechanism, where key

and value are the encoders output:

$$\begin{aligned} attn &= dropout(MHA(x, x, x, question\text{-}mask)) \\ out1 &= layernorm(attn + x) \\ attn2 &= dropout(MHA(out1, encoder\text{-}out, prediction\text{-}mask)) \\ out2 &= layernorm(attn2 + out1) \\ fc &= droput(fullyconnected(out2)) \\ enc^{out} &= layernorm(fc + out2) \end{aligned} \tag{6.14}$$

In the first attention mechanism, the *question-mask* prevents questions from attending to questions from future question bundles. This is essentially the same as the *interaction-mask*) in the encoder, but shifted for the start token ST, which is not necessary for the sequence of questions, since there is always at least one question. The *prediction-mask* in the second attention mechanism ensures, that the output from the first block can only attend to interactions from previous bundles. Note that questions can not attend to the encoders output at the same or earlier positions, if they come from the same question bundle.

Masking

There are three different attention blocks, one in the encoder and two in the decoder. Each block operates with a different mask. The Interaction mask is used in the encoder, the question and prediction mask in the decoder. Given a sequence of question bundles $b_1, ..., b_W$, the masks are computed as follows:

1. initialize a $W \times W$ matrix M with ones below the diagonal and zeros otherwise
2. compute the matrix A, where $a_{i,j} = \begin{cases} 1, & if\ b_i = b_j \text{ and } j > i \\ 0, & else \end{cases}$.
3. compute the matrix $B = S * A^T * S^T$, where S is a shift matrix with ones on the sub-diagonal and zeros otherwise.
4. the *question-mask* is $M + A$
5. the *interaction-mask* is $S * CM * S^T$
6. the *prediction-mask* is $M - B$

The model was trained with batches of size $B = 128$ and sequences of length $W = 300$. The gradients are computed on the last $S = 100$ positions of each sequence. Where S is the step size of the sliding window as in section 6.4. Therefore, every prediction has a sufficiently long history to attend on when making predictions, just as in a productive environment. Another reason is the overlap in the generated sequences, which occurs if

$S < W$. Truncating the loss function to the last S steps ensures that the gradient is not computed multiple times on the same target, which would cause an overrepresentation of some questions.
The gradients are computed according to the following loss:

$$loss(x) = \frac{1}{B} \sum_{n=1}^{B} \sum_{i=W-S}^{W} BCE(y_{n,i}, p_{n,i}) \tag{6.15}$$

Where $BCE(y_{n,i}, p_{n,i})$ is the binary cross entropy of the target and predicted values of the i-th question in the n-th sequence.

Number of encoder/decoder layers	$N = 2$
Model dimension	$d_{model} = 128$
number of attention heads	$h = 8$
Sequence length	$W = 300$
Step size of the sliding window	$S = 100$
Batchsize	$B = 128$
learning-rate	$\alpha = 0.0005$

Table 4: Hyper-parameters: Transformer

6.7 Training and Evaluation

The models were trained on kaggle notebooks, using one cloud TPU v3-8 (see section 6.7.3 below) with tensorflow and python. The Adam optimizer (section 6.7.2) was used throughout all experiments and the target metric is the area under the receiver operator characteristic curve (section 6.7.1). An ablation study evaluates the models for different hyper-parameter configurations (section 6.7.4)

6.7.1 Evaluation Metric

Binary classifiers, as the models described above, predict the probability that the target takes the value zero or one. However, most evaluation metrics rely on contingency tables, that are built on binary predictions, not probabilities. For this one needs to choose a threshold c and probabilities above the threshold are treated as positives and values below as negatives.

	Target Positive	Target Negative
Predicted Positive	# True Positive (TP)	# False Positive (FP)
Predicted Negative	# False Negative (FN)	# True Negative (TN)

Table 5: Contingency table

Two popular metrics are for example the

- False positive rate: $FPR = \frac{FP}{FP+TN}$
- and true positive rate: $TPR = \frac{TP}{TP+FN}$.

Note, that these metrics depend on the chosen threshold and are therefore ambiguous within a single model. For example, if one sets the threshold to $c = 1$, i.e. every prediction belongs to the negative class, there would be not a single positive prediction and it would hold $FPR = TPR = 0$. On the other hand, a threshold of $c = 0$ would set every prediction to the positive class and the metrics would be $FPR = TPR = 1$. In general, the true/false positive rate increases monotonically as the threshold decreases.

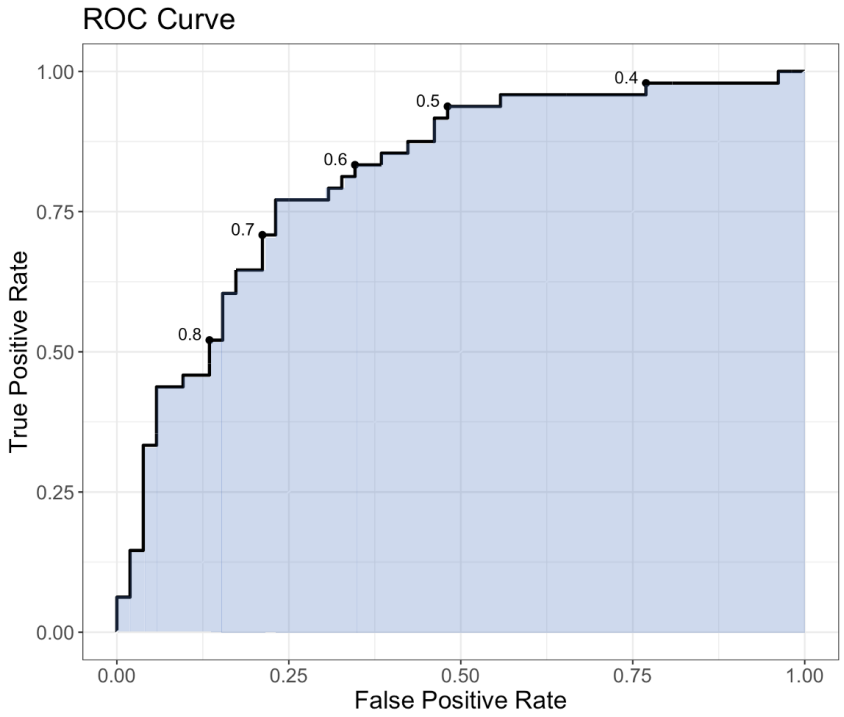

Figure 17: Receiver operator characteristic curve and AUC

Different values of the thresholds can be used, to plot the true positive rate against the false positive rate, resulting in the so called receiver operating characteristic (ROC) curve. Figure 17 displays an exemplary ROC-curve, where some thresholds are highlighted. The information of this curve can be compressed into a single scalar value, by computing the area under the curve (AUC), which is displayed in blue in Figure 17. The possible values of the AUC range from 0 to 1, where 1 represents a perfect classifier and 0.5 an uninformed classifier. From a probabilistic view, the AUC is equivalent to the probability, that the classifier will rank a randomly chosen positive example higher than a randomly chosen negative example.(Fawcett 2006)

6.7.2 Optimization

Both models were trained with the Adam-optimizer as it is "*computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data/parameters*" (Kingma & Ba 2015). The name Adam is derived from adaptive moment estimation and aims to combine the advantages of two earlier proposed methods: *Adaptive Subgradient Methods* (AdaGrad; Duchi et al. (2011) and *Root Mean Square Propagation* (RMSProp, Tieleman & Hinton (2012)). AdaGrad maintains an adaptive per-parameter learning rate and is especially suited for sparse gradients. RMSProp also uses an adaptive learning rate for each parameter, that is based on the running average of the gradients magnitude.
The Adam optimizer requires first-order gradients and computes adaptive learning rates for each parameter from the first and second moments of the gradients. The algorithm requires 4 parameters:

- The learning rate α
- The exponential decay rates for the moment estimates β_1, β_2
- A small constant for numerical stability ϵ

The moment vectors are initialized with $m_0 = 0$ and $v_0 = 0$. For a stochastic objective function $f(\theta)$, i.e a neural network with weights θ, the gradient update at each step t is computed as follows:

$$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1}) \tag{6.16}$$
$$m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{6.17}$$
$$v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{6.18}$$
$$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t) \tag{6.19}$$
$$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t) \tag{6.20}$$
$$\theta_t \leftarrow \theta_{t-1} - \alpha \hat{m}_t / \sqrt{\hat{v}_t + \epsilon} \tag{6.21}$$

In 6.16 the gradients with respect to the objective function at step t are computed. The first and second biased moments are updated in 6.17 and 6.19, where g_t^2 indicates the elementwise square. The moment estimates are corrected for the bias in 6.19 and 6.20. Finally, the parameters are updated according to the step size α in 6.21.
The parameters were set to $\beta_1 = 0.9$ and $\beta_2 = 0.999$, as recommended in (Kingma & Ba 2015). For the learning rate a simple schedule was used, that increases linearly over 4000 warm-up steps to $\alpha = 0.0005$. It was found, that this guarantees a stable training for both, the recurrent and transformer model and for a wide range of different model sizes. The so called Noam learning rate schedule, proposed in Vaswani et al. (2017), which decays the learning rate proportionally to the inverse square root of the step number after the warm-up steps, did not work out very well and either resulted in a

very long training process, or a unpredictable rapid increase in the loss, from which the model could only recover very slowly or not at all.

6.7.3 Hardware

All computations were done using python and tensorflow on kaggle-notebooks. The available hardware was a single NVIDIA TESLA P100 GPU and a Cloud TPU-v3-8. The hardware usage was limited to 30 hours each per week. While TESLA P100 GPU's are already very powerful devices, the TPU were more than 5 times as fast. Therefore all models were trained solely on TPU's.
These TPU's have a total of 8 cores and were trained with an synchronous distribution strategy. This means, that all trainable variables are replicated across the cores and are kept in sync using *all-reduce* algorithms (Patarasuk & Yuan 2009): Each core computes the gradients for $\frac{1}{8}$ of the batch and the parameters are updated according to the mean of those 8 gradients.

6.7.4 Ablation Study

In this section multiple configurations of the presented models are fit to the data, to get some insights how certain components affect the models performance. All scores are obtained through the submission API of the challenge, which ensures that the model submits a prediction before seeing the answer for it, eliminating the possibility of data leakage. The API streams a total of 2.5 million questions and requires the predictions to finish within 9 hours using a single Tesla P100 GPU.
If not stated otherwise, the parameters are the same as in table 4 and 3. The models were trained on $92mio$ questions and a holdout data-set with $6mio$ questions was used to determine the models performance during training. The models were trained until the AUC stopped improving on the holdout data-set and the weights from the best epoch were restored. The models are then fine-tuned with a learning rate ten times smaller than the initial learning rate. And again, training was stopped, when the AUC stopped improving on the validation set. This procedure consistently improved the models performance by a small margin.

For the Transformer model three hyper-parameters are considered: the number of encoder and decoder layers N, the model dimension d_{model}, and the amount of attention heads in every attention mechanism h. The dropout rate was set to 0.1.

S	N	d_{model}	h	AUC	$params$
100	2	64	8	0.793	$1.9 * 10^6$
100	2	128	8	0.801	$4.1 * 10^6$
10	2	128	8	0.803	$4.1 * 10^6$
100	4	128	8	0.801	$4.8 * 10^6$
100	2	256	8	0.803	$9.5 * 10^6$
100	2	256	16	0.802	$9.5 * 10^6$
100	3	256	8	0.803	$10.8 * 10^6$
100	4	256	8	0.803	$12.1 * 10^6$

Table 6: Results Transformer

As illustrated in table 6, the results are similar throughout all hyper parameter configurations. The smallest model with $d = 64$ performed significantly worse than the others. The sampling strategy used a sequence length of $W = 300$ for all models during training and for inference. The step size S of the sliding window did not effect the transformers model in a meaningful way. Reducing it to 10, which results in roughly 10 times as many training examples, increased the score only by 0.002. This demonstrates the sample efficiency of the Transformer.

For the recurrent network the hyper-parameters are the amount of recurrent layers N and the model dimension d_{model}. The dropout rate was set to 0.1:

S	N	d_{model}	AUC	$params$
1	2	256	0.795	$9.2 * 10^6$
1	2	128	0.798	$4.0 * 10^6$
5	2	128	0.786	$4.0 * 10^6$
1	3	128	0.799	$4.3 * 10^6$

Table 7: Results Recurrent Network

Overall, the recurrent models performed slightly worse than the transformer. Most importantly, it was necessary to set the step size S of the sliding window to 1, to achieve similar results as the transformer. In concrete terms, this means the models had to be trained on ≈ 92 million distinct sequences of length $W = 300$, which blew up the training time compared to the transformer. More precisely, the training took approximately 10 hours (2 hours for the Transformer). This is also the reason, why only a small amount of different parameter configurations has been trained.

6.8 Winning solution

The winning solution (Jeon 2021) used a truncated version of the transformer encoder, that reduces the computational effort of the matrix multiplication of query and key. For a query of L time steps, the complexity of QK is $O(L^2)$. Because of that, the computational effort is to large for very long sequences. The winning solution solved this problem, by using only the transformer-encoder with a single hidden layer. The query is truncated to only the last position in the sequence. I.e. Q is of shape $(1, d)$ and K and V are of shape (L, d). Therefore, the attention mechanism computes only L dot products instead of L^2. The output of the multi-head attention is then of shape $(1, d)$. To maintain the length of the sequence, this output is added to every position of the original input sequence. Finally, a LSTM-layer is added on top and the last hidden state is fed through a feed forward network and the output is the probability of answering the question at the last position correctly. In fact, the sequence length used in the winning model is 1728 and a ensemble of 5 models scored as high as 0.820 on the leader board.

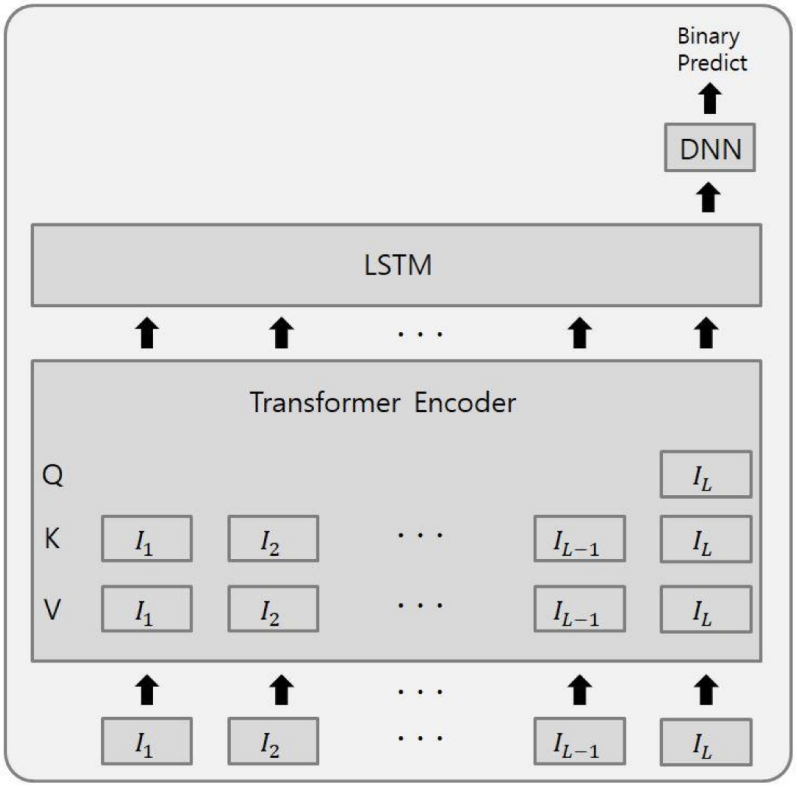

Figure 18: Winning solution (Jeon 2021)

7 Discussion and Outlook

The *Riiid! Answer Correctness Prediction Challenge 2020* demonstrated, that sequential deep learning methods are capable of detecting the complex patterns in the human learning process. Both, recurrent neural networks and transformer models dominated the final leader-board of the challenge. Other competitive methods relied on gradient boosting methods with dozens of carefully hand-crafted features. The problem with engineered features is, that they require comprehensive domain knowledge. For the task of knowledge tracing, this is not a severe issue, since everyone went through the process of learning something and can use such personal experiences to create features that might be useful. From a larger perspective however, the outcome of the challenge is yet another landmark in favor of deep learning models, that once more demonstrated the power of automated feature extraction.

For the task of knowledge tracing, a major weakness of the bidirectional recurrent model is the inefficiency during training. The output at every time step contains information about the whole input sequence and therefore the gradients could only be computed for the last time step of every training sequence. In contrast to that, the Transformer models maintain a causal order in the output sequence and the gradients could be computed for multiple time steps at once, resulting in a much faster training process. In concrete terms, the recurrent network required almost 100 times as many training-samples as the Transformer to achieve similar results. This could be counteracted, by a much larger batchsize that enlarged the models throughput. However, the training still took about 5 times as long. This was a hurdle especially in the early development phase, where one wants to try many different approaches.
The transformer model is well known for its highly parallelizable architecture. In combination with powerful TPU's, that are optimized for large scale matrix multiplications, competitive models could be trained in less than two hours. A drawback of the Transformer model is its quadratic complexity with respect to the sequence length. The model presented in section 6.6.2 was limited to a maximum sequence length of 300. However, the winning solution suggests, that longer sequences are the key for further improvements.

Since its introduction in 2017, transformer models have become the de-facto standard for many tasks. Especially in the field of natural language processing (NLP), RNN's have practically been replaced by transformer models. *Bidirectional Encoder Representation from Transformers* (BERT; (Devlin et al. 2019)) and *Generative Pre-trained Transformers* (GPT, (Radford & Narasimhan 2018)) are pre-trained models, that can be fine tuned to achieve state of the art results for many NLP tasks, such as question answering (Radford et al. 2019) or text summarization (Khandelwal et al. 2019). These results are particularly impressive, given the fact that the pre-training is done in an

unsupervised fashion.

Due to the enormous availability of text data and the computational efficiency of the transformer architecture, it has become possible to train models of unprecedented size with over 100 Billion parameters and the models and datasets are still growing, showing no signs of saturating performance. (Dosovitskiy et al. 2021)

In addition to that, Transformers have also been applied to computer vision tasks. Touvron et al. (2021) achieve state of the art results on the Imagenet dataset (Deng et al. 2009), relying on a convolution-free Transformer. Dosovitskiy et al. (2021) found out, that for large scale training (14M-300M images) the Transformer trumps the inductive bias of convolutional networks, which relies on locality and translation invariance.

8 Conclusion

This thesis covered the fundamentals of modern deep learning models for sequential data. Recurrent neural networks were introduced by bringing in loops to the computational graph of feed forward networks. It was demonstrated, that the backpropagation algorithm generalizes straight forward to the unfolded graph of RNN's. However, the problem of vanishing gradients makes it difficult for vanilla RNN's to capture long term dependencies. To address this issue, two approaches were presented: LSTM and GRU models. Both rely on gating mechanisms, that control the flow of information into and out of the recurrent cell.

The encoder-decoder architecture allows the modeling of arbitrary sequence to sequence mappings. For this, the encoder RNN compresses the information of the input sequence into a fixed size vector and the decoder RNN uses this vector to generate the target sequence. This approach however, suffers from a bottleneck that is introduced by the fixed length representation of variable length sequences. Attention mechanisms overcome this problem by aligning every position in the target sequence with positions from the source sequence.

The transformer architecture builds on top of the attention mechanisms used in encoder-decoder RNN's, but dispense completely with recurrence. The central component of the Transformer is called multi head self-attention. This mechanism implements the sequence to sequence mapping as weighted sums over linear projections of the input sequence. This method allows the model to directly look at any position in the input sequence, bridging arbitrary long distances. Since transformer models do not rely on sequential processing, they are highly parallelizable and can be trained efficiently on large scale datasets.

Finally, the discussed theory was applied to real world data as part of the *Riiid! Answer Correctness Prediction Challenge 2020*. It was demonstrated, that sequential deep learning models achieve state-of-the-art results in the field of knowledge tracing. The Transformer excelled with a rapid training process and accurate results. The RNN required much more training samples than the Transformer and thus more time to train. Ultimately the performance of both architectures was very similar. Furthermore, the winning solution suggests that it might be beneficial to merge recurrent layers and self attention mechanisms within the same model.

References

Ba, J., Kiros, J. & Hinton, G. E. (2016), 'Layer normalization', *arXiv* **1607.06450**.

Bahdanau, D., Cho, K. & Bengio, Y. (2015), Neural machine translation by jointly learning to align and translate. 3rd International Conference on Learning Representations.

Bengio, Y., Simard, P. & Frasconi, P. (1994), 'Learning long-term dependencies with gradient descent is difficult', *IEEE transactions on neural networks* **52**, pp. 157–66.

Bontempi, G., Ben Taieb, S. & Le Borgne, Y.-A. (2012), 'Machine learning strategies for time series forecasting', *Lecture Notes in Business Information Processing* **138**, pp. 62–78.

Cheng, J., Dong, L. & Lapata, M. (2016), Long short-term memory-networks for machine reading, *in* 'Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing', Association for Computational Linguistics, Austin, Texas, pp. 551–561.

Cho, K., van Merrienboer, B., Gulcehre, C., Bougares, F., Schwenk, H. & Bengio, Y. (2014), Learning phrase representations using rnn encoder-decoder for statistical machine translation, *in* 'Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)'.

Choi, Y., Lee, Y., Cho, J., Baek, J., Kim, B., Cha, Y., Shin, D., Bae, C. & Heo, J. (2020), 'Towards an appropriate query, key, and value computation for knowledge tracing', *arXiv* **2002.07033**.

Corbett, A. T. & Anderson, J. R. (1995), 'Knowledge tracing: Modelling the acquisition of procedural knowledge.', *User Modeling and User-Adapted Interaction* **4**(4), pp. 253–278.

de Prado, M. L. (2018), *Advances in Financial Machine Learning*, 1st edn, Wiley Publishing.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K. & Fei-Fei, L. (2009), Imagenet: A large-scale hierarchical image database, *in* '2009 IEEE conference on computer vision and pattern recognition', pp. 248–255.

Devlin, J., Chang, M.-W., Lee, K. & Toutanova, K. (2019), BERT: Pre-training of deep bidirectional transformers for language understanding, *in* 'Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies', Vol. 1, pp. 4171–4186.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J. & Houlsby, N. (2021), An image is worth 16x16 words: Transformers for image recognition at scale, *in* 'International Conference on Learning Representations'.

Duchi, J., Hazan, E. & Singer, Y. (2011), 'Adaptive subgradient methods for online learning and stochastic optimization', *Journal of Machine Learning Research* **12**, 2121–2159.

Fan, H., Jiang, M., Xu, L., Zhu, H., Cheng, J. & Jiang, J. (2020), 'Comparison of long short term memory networks and the hydrological model in runoff simulation', *Water* **12**(1).

Fawcett, T. (2006), 'An introduction to ROC analysis', *Pattern Recognition Letters* **27**(8), pp. 861–874.

Fulcher, B. & Jones, N. (2016), 'Automatic time-series phenotyping using massive feature extraction', *arXiv* **1612.05296**.

Gers, F., Schmidhuber, J. & Cummins, F. (2000), 'Learning to forget: Continual prediction with lstm', *Neural computation* **12**, pp. 2451–2471.

Goodfellow, I., Bengio, Y. & Courville, A. (2016), *Deep Learning*, MIT Press. `http://www.deeplearningbook.org`.

Graves, A., Mohamed, A.-r. & Hinton, G. (2013), Speech recognition with deep recurrent neural networks, *in* '2013 IEEE International Conference on Acoustics, Speech and Signal Processing', pp. 6645–6649.

Hammer, B. (2000), 'On the approximation capability of recurrent neural networks', *Neurocomputing* **31**(1), pp. 107–123.

He, K., Zhang, X., Ren, S. & Sun, J. (2016), Deep residual learning for image recognition, *in* '2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)', pp. 770–778.

Hochreiter, S. (1998), 'The vanishing gradient problem during learning recurrent neural nets and problem solutions', *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* **6**, pp. 107–116.

Hochreiter, S. & Schmidhuber, J. (1997), 'Long short-term memory', *Neural Computation* **9**(8), pp. 1735–1780.

Hornik, K., Stinchcombe, M. & White, H. (1989), 'Multilayer feedforward networks are universal approximators', *Neural Networks* **2**(5), pp. 359–366.

Hyndman, R. & Athanasopoulos, G. (2018), *Forecasting: Principles and Practice*, 2nd edn, OTexts, Australia.

Ioffe, S. & Szegedy, C. (2015), Batch normalization: Accelerating deep network training by reducing internal covariate shift, *in* 'Proceedings of the 32nd International Conference on Machine Learning', Vol. 37, pp. 448–456.

Jabreel, M. & Moreno, A. (2019), 'A deep learning-based approach for multi-label emotion classification in tweets', *Applied Sciences* **9**.

Jeon, S. (2021), 'Last query transformer rnn for knowledge tracing', *arXiv* **2102.05038**.

Khandelwal, U., Clark, K., Jurafsky, D. & Kaiser, L. (2019), 'Sample efficient text summarization using a single pre-trained transformer', *arXiv* **1905.08836**.

Kingma, D. P. & Ba, J. (2015), Adam: A method for stochastic optimization, *in* '3rd International Conference on Learning Representations'.

Liu, Q., Huang, Z., Yin, Y., Chen, E., Xiong, H., Su, Y. & Hu, G. (2021), 'Ekt: Exercise-aware knowledge tracing for student performance prediction', *IEEE Transactions on Knowledge and Data Engineering* **33**(1), pp. 100–115.

Löning, M., Bagnall, A. J., Ganesh, S., Kazakov, V., Lines, J. & Király, F. J. (2019), 'sktime: A unified interface for machine learning with time series', *arXiv* **1909.07872**.

Luong, T., Pham, H. & Manning, C. D. (2015), Effective approaches to attention-based neural machine translation, *in* 'Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing', Lisbon, Portugal, pp. 1412–1421.

Pandey, S. & Karypis, G. (2019), A self-attentive model for knowledge tracing, *in* 'EDM 2019 - Proceedings of the 12th International Conference on Educational Data Mining', pp. 384–389.

Pardos, Z. & Heffernan, N. (2011), Kt-idem: Introducing item difficulty to the knowledge tracing model, *in* 'In Proceedings of the 19th International Conference on User Modeling, Adaptation and Personalization', pp. 243–254.

Pascanu, R., Mikolov, T. & Bengio, Y. (2013), On the difficulty of training recurrent neural networks, *in* 'Proceedings of the 30th International Conference on International Conference on Machine Learning', Vol. 28, pp. 1310–1318.

Patarasuk, P. & Yuan, X. (2009), 'Bandwidth optimal all-reduce algorithms for clusters of workstations', *Journal of Parallel and Distributed Computing* **69**(2), pp. 117–124.

Piech, C., Bassen, J., Huang, J., Ganguli, S., Sahami, M., Guibas, L. J. & Sohl-Dickstein, J. (2015), Deep knowledge tracing, *in* 'Advances in Neural Information Processing Systems', Vol. 28, pp. 505–513.

Radford, A. & Narasimhan, K. (2018), 'Improving language understanding by generative pre-training', *OpenAI* .

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D. & Sutskever, I. (2019), 'Language models are unsupervised multitask learners'.

Ravanelli, M., Brakel, P., Omologo, M. & Bengio, Y. (2018), 'Light gated recurrent units for speech recognition', *IEEE Transactions on Emerging Topics in Computational Intelligence* **2**(2), pp. 92–102.

Riiid (2014), `https://www.crunchbase.com/organization/riiid`. Accessed: 2021-04-29.

Schipp, A. (2021), 'Frankfurter allgemeine zeitung', `https://www.faz.net/aktuell/gesellschaft/menschen/welche-probleme-es-beim-homeschooling-gibt-17149736.html`. Accessed: 2021-04-29.

Schuster, M. & Paliwal, K. (1997), 'Bidirectional recurrent neural networks', *IEEE Transactions on Signal Processing* **45**, pp. 2673–2681.

Shin, D., Shim, Y., Yu, H., Lee, S., Kim, B. & Choi, Y. (2021), Saint+: Integrating temporal features for ednet correctness prediction, *in* 'LAK21: 11th International Learning Analytics and Knowledge Conference', pp. 490–496.

Sutskever, I., Vinyals, O. & Le, Q. V. (2014), Sequence to sequence learning with neural networks, *in* 'Proceedings of the 27th International Conference on Neural Information Processing Systems', Vol. 2, pp. 3104–3112.

Tieleman, T. & Hinton, G. (2012), 'Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude', COURSERA: Neural Networks for Machine Learning.

Touvron, H., Cord, M., Douze, M., Massa, F., Sablayrolles, A. & Jegou, H. (2021), 'Training data-efficient image transformers & distillation through attention', *arXiv* **2012.12877**.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u. & Polosukhin, I. (2017), Attention is all you need, *in* 'Proceedings of the 31st International Conference on Neural Information Processing Systems', pp. 6000–6010.

Werbos, P. (1990), 'Backpropagation through time: what it does and how to do it', *Proceedings of the IEEE* **78**, pp. 1550–1560.

Williams, R. J. & Zipser, D. (1989), 'A learning algorithm for continually running fully recurrent neural networks', *Neural Computation* **1**(2), pp. 270–280.

Youngduck, C., Youngnam, L., Dongmin, S., Junghyun, C., Seoyon, P., Seewoo, L., Jineon, B., Chan, B., Byungsoo, K. & Jaewe, H. (2020), Ednet: A large-scale hierarchical dataset in education, *in* 'International Conference on Artificial Intelligence in Education', pp. 69–73.

Yudelson, M., Koedinger, K. & Gordon, G. (2013), Individualized bayesian knowledge tracing models, *in* 'International Conference on Artificial Intelligence in Education', pp. 171–179.

Zhaoping, L. (2014), *Understanding Vision: Theory, Models and Data*, Oxford University Press.