Summary:

This thesis presents novel GSRec hybrid model for sequentional recommendation. With this model, we aim to tackle problem which ocurres in recommendation systems, like predict next item in dynamical nature of user behavior.

The GSRec model combines two sequential modeling methods: the SASRec transformer, which uses a self-attention layer to learn recent patterns in user interactions, and the Gated Recurrent Unit (GRU), a recurrent neural network which is used for learning long-range dependencies. Unlike other hybrid models that use these methods in parallel or depend on heavy preprocessing, GSRec organises them in a pipeline architecture, applying SASRec to process immediate user behavior and then passing those outputs into GRU to learn broader trends. This pipeline architecture reduces architectural complexity and also improves training efficiency.

To validate the performance of this GSRec model, the model was tested on two datasets: Amazon Beauty, representing sparse e-commerce data, and MovieLens 1M, a denser dataset of movie ratings. GSRec was evaluated using Hit Rate (HR@10), Normalized Discounted Cumulative Gain (NDCG@5, @10), and Mean Reciprocal Rank (MRR). Baselines for comparison included simple model PopRec, as well as more complex architectures like GRU4Rec, SASRec, and BERT4Rec.

Results of the experiments showed that GSRec model outperformed the GRU-only model (on sparse Beauty dataset) and the POP baseline across both datasets. Results on sparse dataset were lower than SASRec and BERT4Rec but not significantly, which indicates robustness of the GSRec model in areas where user data is limited. But, GSRec did not outperform SASRec or BERT4Rec in the denser ML-1M dataset, which is possibly because of the GRU's reprocessing of embeddings from SASRec, which may dilute attention patterns.

While the model does not outperform all of the state-of-the-art models, its efficient pipeline architecture shows a promising foundation for future hybrid recommendation systems. The thesis concludes by proposing few future directions how to improve GSRec, like the integration of feed-forward layers in the transformer part of model, the use of multi-layer GRUs to capture more complex dependencies.

Overall, GSRec contributes meaningfully to the field of sequential recommendation by showing how the combination of attention mechanisms and recurrent networks can lead to balanced performance across different types of user behavior data.

GSRec: A Hybrid Sequential Recommendation Model Combining GRU and SASRec

Master Thesis

Matej Eres

Aalborg University
Department of Computer Science
Selma Lagerløfs Vej 300
9220 Aalborg East

This thesis was created using Overleaf, which provided control over the document layout, mathematical formatting, and referencing. For implementing and training, we used Python with libraries such as PyTorch and Numpy, we also used Word and Excel for creating graphics and plots. Grammarly was used to fix spelling and other mistakes made during the process of writing this report. Also, ChatGPT was used to help with some minor code generation tasks for formatting code and debugging tasks related to state-of-the-art models for experiments. All decisions and conclusions shown in this thesis are our own.

Computer Science
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
GSRec: A Hybrid Sequential Recommendation Model Combining GRU and SASRec

Theme:
Scientific Theme

Project Period:
Spring Semester 2025

Project Group:
cs-25-mi-10-14

Participant(s):
Matej Eres

Supervisor(s):
Peter Dolog

Copies: 1

Page Numbers: 38

Date of Completion:
June 5, 2025

Abstract:

Sequential recommendation is key in modern recommender systems to capture user behavior in data. Two approaches for this have proliferated: neural networks and transformers. That is why we propose a new hybrid model, GSRec, for sequential recommendation, which combines the strengths of the SASRec transformer and the GRU neural network. We integrate them in a sequential pipeline to capture both short-term and long-term user preferences. SASRec is first applied to extract recent patterns in user data using an attention layer, followed by GRU to model longer-term dependencies. We evaluate GSRec on two datasets, Amazon Beauty and MovieLens 1M to see performance in comparison to state-of-the-art models such as POP, GRU4Rec, SASRec, and BERT4Rec. While BERT4Rec outperforms GSRec in dense datasets, our model shows robustness in sparser environments.

Acknowledgments

First and foremost, I would like to say thank you to my family for their love and support throughout this journey. Your belief in me, even from far away, always kept me going, and for that, I am forever grateful.

I would also like to extend my thanks to my dear friends Denis, Laura, Lea, Matea, and Josephine. Your constant motivation and support during the most challenging moments meant more to me than words can express. Whether through kind words, study help, or a small distraction when I needed it, you were always there for me. Each of you played a crucial role in helping me reach this milestone.

This achievement would not have been possible without every one of you. Thank you from the bottom of my heart.

Contents

Chapter 1

Introduction

In recent years, deep learning has played a crucial role in recommendation systems, specifically in capturing user behavior over a period of time. With the development of the internet, numerous digital platforms have emerged, such as music streaming services like Spotify, Apple Music, and Deezer; movie streaming platforms such as Netflix, Amazon Prime, and Disney+; and e-commerce sites like Amazon, Asos, and Zalando. To stay competitive, these platforms require models capable of capturing sequential user interactions, for example, what users bought or watched, what they liked or disliked based on their reviews, how long they listened to a particular song or podcast, and how their behavior evolved over time. Effectively capturing this information does not only mean understanding what a user likes right now but also how their preferences have changed. These are the reasons why sequential recommendation models are so important.

Two popular deep learning models used for sequential recommendation tasks are Recurrent Neural Networks (such as gated recurrent unit (GRU) and Long-Short Term Memory (LSTM)), and self-attention-based models like Transformers (such as SASRec and BERT4Rec) [25]. GRU and LSTM are great for learning long-term dependencies in sequences, but they can struggle with very recent or rapidly changing user behavior [11, 24]. On the other hand, attention mechanisms like those used in Transformers focus more on identifying which parts of a sequence are most relevant, making them powerful for learning short-term user interests [22]. Thanks to the strengths both of these approaches have, many recent models try to combine them to get the best of both worlds.

Several hybrid models have already been proposed in other fields that show the benefit of combining these algorithms. For example, Zheng et al.[24], introduced a GRU–Transformer model for predicting soil moisture levels. Their setup uses two parallel branches—one with GRU to model time dependencies and another with a Transformer to catch broader, contextual patterns. Li and Qian's [11] FDG-Trans model, designed for forecasting stock prices, first cleans up the data using signal decomposition and then feeds it through GRU, LSTM, and Transformer layers. While these models work well for their specific use cases, they are either tailored to structured, domain-specific data or require complex preprocessing steps. That makes them less ideal for general-purpose recommendation systems, where data is often less structured and changes fast.

In this project, we focus on sequential recommendation tasks in areas like movie and e-commerce platforms, where user interactions are unpredictable and often short-lived. We propose a simple but effective model that combines SASRec and GRU, not in parallel, but in a pipeline, which, as a result, thanks for this pipeline architecture, has faster training and lower complexity. The SASRec component comes first, using self-attention to capture recent user interests and behavior patterns. The output from SASRec is then passed into a GRU, which builds on that information to learn longer-term trends in user behavior. By doing this, the model first gets a clean, short-term representation of what the user is into right now, and then uses GRU to learn how that behavior fits into a bigger picture over time.

The paper is organised as follows: In Chapter 2, we present similar papers, which inspired this work. Chapter 3 provides an overview of the model we considered, including RNN variants and Transformers, and introduces our proposed hybrid model. In Chapter 4 we explain the experimental setup, including datasets, baselines and evaluation metrics. Chapter 5 presents and discusses the experimental results, comparing our model with existing approaches across different datasets. Finally, Chapter 7 concludes the paper and outlines potential directions for future work.

Chapter 2

Related Works

Over the past several years, there has been a vast amount of research on using recurrent neural networks to make recommendations across various domains such as finance, investment, music streaming, e-commerce, etc. Long short term memory (LSTM) being the most prominent and widely adopted model thanks to its structure of long-term and short-term cells. Those two cells captures different information, such as important information over longer period of time due to long-term cells and instant information thanks to short-term cells. Due to the complexity of data involved in making recommendations in domains such as e-commerce, finance, or movies, hybrid models are often used.

2.1 GAT4Rec

He et al. [5], proposed a sequential recommendation model based on a GRU and Transformer to capture both recent and long-term user interests. In GAT4Rec, the gated recurrent unit (GRU) models the latest k interactions to generate category information and uses the learned hidden layer to express users' intentions and filters the user's history before feeding it into a Transformer, while effective, this method relies on additional metadata such as item categories and applies GRU early in the sequence to guide filtering.

In contrast, our approach adopts the reverse order: we first use SASRec to capture short-term user interest through self-attention, then apply a GRU to learn long-term patterns from the attention-enhanced representations. This structure should allow our model to learn richer temporal dependencies

without needing additional data. Also, our framework is designed for wide applicability across domains such as movie and e-commerce recommendation, where auxiliary data may be limited.

2.2 FDG-Trans

Li and Qian [11], proposed a hybrid FDG-Trans model for stock price prediction that combines frequency decomposition, GRU, LSTM, and Transformer layers. Their model first applies Complete Ensemble Empirical Mode Decomposition (CEEMD) to decompose the raw, noisy financial time series data into trend and mode components, reducing noise and improving signal clarity. These decomposed series are then passed through LSTM and GRU layers to capture both temporal dependencies, followed by a Transformer to learn patterns over time and figure out which past steps are most important for the current one. While effective in the financial domain, this pipeline is adjusted specifically for noisy, high-frequency stock data and requires significant preprocessing.

In contrast, our approach is more lightweight and domain-flexible, designed for movie and e-commerce recommendation tasks. We first apply SASRec to extract short-term user behavior patterns using self-attention, then pass the resulting sequence into a GRU to learn longer-term trends. By reversing the layer order used in FDG-Trans and excluding decomposition, our method aims to capture temporal dynamics in a streamlined way that should adapt well to diverse datasets without requiring domain-specific signal processing. That way, our model works better across different recommendation systems where the data isn't that noisy, but user behavior keeps changing.

2.3 GRU-Transformer

Zheng et al. [24], introduced a GRU–Transformer hybrid model aims to predict soil moisture levels in root zones. Their model uses parallel branches, where GRU captures long-term sequence information and a Transformer module handles broader relationships within the input data. This structure was shown to be effective in capturing moisture dynamics at various soil depths and across different time frames. However, their approach is suited specifically to environmental data, especially in the agricultural domain, and relies on a parallel processing design.

In contrast, our GSRec model rather than processing GRU and Transformer outputs in parallel, uses a sequential pipeline: SASRec is applied first to capture short-term user patterns using self-attention, followed by GRU to model longer-term dependencies from the attention-refined sequences. This ordering is better suited for user-item interactions, where the immediate past strongly influences short-term preferences, and long-term trends evolve more gradually.

Chapter 3

Method

3.1 Neural network

In this section, we will discuss different neural networks that we considered while deciding on which neural network to use for this project.

3.1.1 Vanilla RNN

First, we looked into the most basic Vanilla Recurrent Neural Network. Vanilla RNN is the foundational architecture for processing sequential data. It is also known as Elman RNN, thanks to Jeff Elman, who introduced it back in 1990 [16] is used in natural language processing, speech recognition, and other sequence modeling tasks. Due to the loop architecture of Vanilla RNN, information can persist over time periods, which enables the neural network to capture temporal dependencies in a sequence. Mathematical representations of Vanilla RNN is [13] [7]:

$$h_t = \tanh(W_{xh} \cdot x_t + W_{hh} \cdot h_{t-1} + b_h)$$

where:

- x_t: input vector at time step t
- h_t: the new hidden state, which serves two roles:
  - The output of this time step
  - The input to the next RNN cell in the sequence
- h_{t-1}: hidden state from the previous time step
- W_{xh}, W_{hh}: weight matrices

- b_h: bias term
- tanh is an activation function (hyperbolic tangent) which quashes the input to a range between -1 and 1. Formula is:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

  where:

  - x: Input value
  - e: Euler's number

Figure 3.1 represents one cell of RNN with a representation of how RNN works. W_t serves at the same time as output of the current cell as well as input for the next cell.

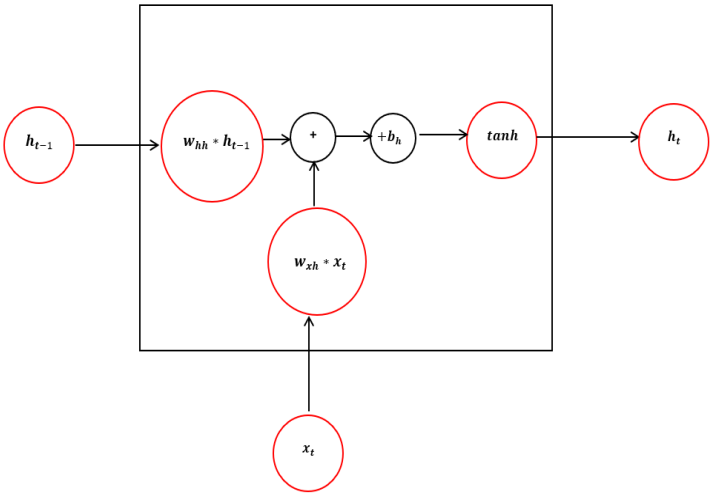

Figure 3.1: Representation of RNN cell [1]

Vanilla RNN is simple and easy to understand. It works well for short sequences, and since it processes one element at a time it can handle different length inputs. Naturally, since Vanilla RNN is the most basic type of RNN, it has major critical limitations [16].

One of the main issues is the vanishing gradient problem. During training, backpropagation through time is used to update weights. This involves applying the chain rule, which multiplies many derivatives together across

time. Since derivatives of activation functions such as tanh are always $<= 1$, gradient can decrease exponentially. If the gradient shrinks to 0, as a result, weights will stop updating, and the model won't be able to learn patterns from earlier in the sequence. Sometimes, gradients can grow excessively large during training. This happens when, during backpropagation through time, the chain rule causes the model to multiply several derivatives greater than 1. As a result, the gradient values increase exponentially, especially in long sequences. The problem arises from the weight matrices — especially the recurrent weight matrix W_{hh}. This happens if the weights in W_{hh} are large or they were badly initialized. As a result this often leads to numerical overflow [2] [15].

3.1.2 Long Short-Term Memory

The second neural network we considered is Long Short-Term Memory (LSTM). LSTM is a more advanced type of Recurrent Neural Network (RNN) introduced by Hochreiter and Schmidhuber in 1997 [6]. LSTM like vanilla RNN, is used for natural language processing, speech recognition, and other sequence modeling tasks. Like Vanilla RNNs, LSTM processes sequences of data one element at a time, maintaining a hidden state throughout the input sequence, constantly updating it with new relevant information while discarding information which is less important at each time step. However, LSTM has a more complex architecture, which is specifically designed to address the vanishing gradient problem[7].

LSTM uses a cell state in addition to a hidden state, and introduces three gates: input, forget, and output, that control the flow of information through the network. Figure 3.2 displays an architectural representation of LSTM.

Forget gate - The forget gate decides what information from the previous memory (C_{t-1}) should be discarded. This is important because not all past information is relevant. The gate takes the current input x_t and the previous hidden state h_{t-1}, and outputs values between 0 and 1. A value that is close to 1 means keep this, while a value that is close to 0 means forget this. This gate ensures that the LSTM can discard outdated or irrelevant information[8]. The formula for forget gate is [15]:

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

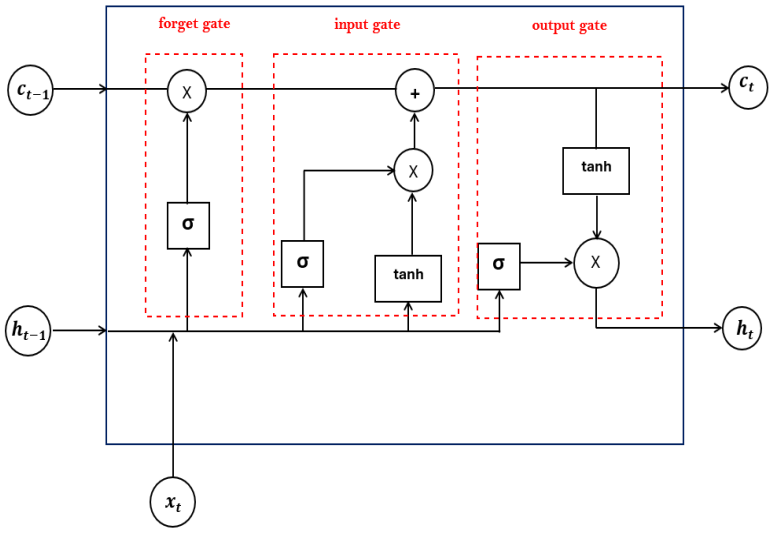

Figure 3.2: Representation of LSTM architecture [1]

where:

- f_t represents forget gate vector at time step t
- σ represents sigmoid activation function
- W_f represents weight matrix for the forget gate
- $[h_{t-1}, x_t]$ is concatenation of the previous hidden state h_{t-1} and current input x_t.
- b_f represents bias vector for the forget gate

Input gate - The input gate decides what new information should be added to the cell's memory. It works together with the candidate cell state, which calculates a potential update using a tanh activation. The input gate decides how much of this information should actually be added. This is what lets the LSTM learn new information while still remembering previous relevant information. This gives the model flexibility to update its memory without overwriting everything[8]. Formula for input gate is [15]:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$
$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

where:

- i_t: represents input gate activation (decides how much new information to let into the memory)
- $\tilde{C}_t$: represents new candidate values (proposed content to be added to memory)
- σ: represents sigmoid activation function used for gating
- tanh: represents hyperbolic tangent activation function
- W_i, W_C: Weight matrices for the input gate and candidate cell state.
- b_i, b_C: represents bias vectors for the input gate and candidate cell state
- C_t: updated cell state which, combines what to forget ($f_t \odot C_{t-1}$) and what new info to add ($i_t \odot \tilde{C}_t$)

Output gate - The output gate decides what part of the updated cell state (C_t) should be shown as the hidden state (h_t) for the current step. This is the part that gets passed on to the next LSTM cell and potentially used to make predictions. The gate applies a sigmoid activation to the current input and previous hidden state to calculate the output gate vector, whose values are in a range between 0 to 1. If value is closer to 1 it will allow information to pass through, while values near 0 will mostly not be passed through. The gate vector (o_t) is multiplied element-wise with the tanh activated cell state ($tanh(C_t)$), producing the new hidden state (h_t). This helps the model express useful information at each time step without dumping the entire memory[8]. Formula for the output gate is [15]:

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

$$h_t = o_t \odot \tanh(C_t)$$

where:

- o_t: represents output gate activation
- σ: represents sigmoid function
- W_o: represents weight matrix that transforms the input and previous hidden state for the output gate.
- $[h_{t-1}, x_t]$: represents a concatenated vector combining previous hidden state and current input.

- b_o: Bias term added to the gate computation to help adjust learning.
- h_t: represents final hidden state
- $\tanh(C_t)$: reduce the updated memory to a range between -1 and 1 before output.

LSTM reduces the vanishing gradient problem due to cell state. Cell state (C_t), is designed to flow mostly unchanged across time steps; it avoids repeated nonlinear squashing like tanh/sigmoid at every step (unlike hidden states in vanilla RNNs). LSTM, as the name suggests, remembers long-term dependencies better due to cell state (C_t) and forget, input, and output gates. LSTM is also more robust for longer sequences or complex dependencies since it selectively update memory rather than overwrites it through an input (update) gate [20] [8].

3.1.3 Gated recurrent unit

Gated Recurrent Unit (GRU) is a type of Recurrent Neural Network introduced by Cho et al. in 2014 [3]. It was designed as a simpler alternative to LSTM while still solving the vanishing gradient problem that affects vanilla RNNs. GRU has simpler architecture because it merges the input and forget gates into a single update gate, and it removes the separate cell state used in LSTM. GRU like LSTM is also well suited for sequence data[19]. Figure 3.3 represents GRU architecture.

As mentioned before GRU has two main gates:

Reset gate - The reset gate in a GRU controls how much of the previous hidden state h_{t-1} should be forgotten before creating the candidate hidden state $\tilde{h}_t$. When the reset gate is close to 0, it means that the model will ignore past information and focus mostly on the current input. This is especially useful when the network needs to reset its memory and focus on a sudden change in the sequence. While, if the gate is close to 1, it allows the full influence of the past state. The reset gate allows the GRU to be more flexible in deciding when to leave out old information[19]. Formula for the reset gate is [15]:

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$$

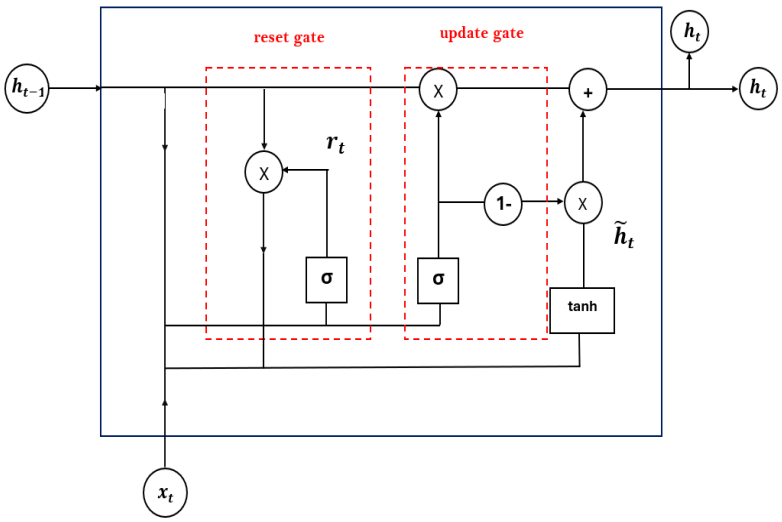

Figure 3.3: Representation of GRU architecture [1]

where:

r_t: represents reset gate values

σ: represents sigmoid activation function

W_r: represents reset gate weight matrix

$[h_{t-1}, x_t]$: represents concatenation of the previous hidden state and current input.

b_r: represents bias term for the reset gate.

Update gate - The update gate calculates how much of the previous hidden state should be carried forward into the current step. If the update gate z_t is close to 1, most of the previous state is kept, and little new information is added. If it's close to 0, the model overwrites the memory with the new candidate hidden state $\tilde{h}_t$. This gate helps the GRU decide how much to "remember" and how much to "update" at each time step[19]. It serves a similar purpose to the combined forget and input gates in LSTM.

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$$

where:

z_t: represents update gate values

W_z: represents update gate weight matrix

$[h_{t-1}, x_t]$: represents concatenation of the previous hidden state and current input.

b_z: represents bias term for the update gate.

Candidate hidden state ($\tilde{h}_t$) is the new potential content to add and is calculated using formula:

$$\tilde{h}_t = \tanh(W_h \cdot [r_t \odot h_{t-1}, x_t] + b_h)$$

where:

- $\tilde{h}_t$: represents candidate hidden state at time step t
- tanh: represents activation function
- W_h: represents weight matrix for generating the candidate hidden state
- $r_t \odot h_{t-1}$: represents element-wise multiplication.
- x_t: represents input at the current time step
- b_h: represents bias term for the candidate hidden state

Final hidden state (h_t)blends the previous hidden state and the candidate state using the update gate. Formula for final hidden state is:

$$h_t = (1 - z_t) \odot \tilde{h}_t + z_t \odot h_{t-1}$$

where:

- h_t: represents final hidden state
- $(1 - z_t)$: represents inverse of the update gate
- $\tilde{h}_t$: represents candidate hidden state
- z_t: represents update gate
- h_{t-1}: represents hidden state from the previous time step.

GRU as LSTM reduce the vanishing gradient problem due to the gate mechanisam (update and reset gate). Due to simpler architecture, GRU is trained faster, also less memory and computational power are needed. GRU, since it doesn't have a cell state like LSTM, offers less memory control [19] [18].

Selected neural network

To better understand how those three neural network process data we created a toy example in Table 3.1 where input is altering between 0s and 1s. In the table, it's visible that Vanilla RNN at the third timestamp decreases, which indicates that Vanilla RNN is not good for longer sequences and does not retain memory as good as LSTM and GRU, which are similar, with a small difference in favour of LSTM.

Time Step	Input	Vanilla RNN Output	LSTM Output	GRU Output
1	0	0.1	0.1	0.1
2	1	0.5	0.6	0.6
3	0	0.3	0.5	0.4
4	0	0.2	0.4	0.3
5	1	0.4	0.8	0.7

Table 3.1: Example of outputs of different RNN architectures at each time step

After comparison of Vanilla RNN, LSTM, and GRU, we decided to use GRU in this project because of its simplicity, efficiency, and strong performance on sequential recommendation tasks. Vanilla RNN was not a workable choice because it suffers heavily from the vanishing gradient problem and lacks gating mechanisms to manage memory over time, which is a problem we aim to solve in this project. LSTM solves this issue with gated architecture, including a separate memory cell. However, because of LSTM's complex architecture it has higher computational cost and longer training times. GRU, on the other hand, offers similar performance to LSTM while being faster and more lightweight. It uses fewer gates, trains more efficiently, and still has the ability to model long-term dependencies.

3.2 Transformers

In this section, we will discuss BERT4Rec and SASRec transformers. We will dive into their architecture, which tasks they solve, and what are advantages and disadvantages of both of those transformers, and make a conclusion on which we picked and why.

3.2.1 BERT4Rec

BERT4Rec is a Transformer-based model for sequential recommendation. It is based on the BERT model originally used in natural language processing, but it has been adapted for recommendation tasks by teaching it to predict masked items in a user's interaction history. Masked items are items which are randomly hidden (replaced with [MASK] token) in the sequence. During training the model tries to predict those masked items based on the surrounding context. For example, given sequence [x_1, x_2, x_3, x_4], the model might mask x_3 and be asked to predict it using the rest of the sequence. This approach helps the model learn contextual dependencies between items, no matter of their position. Unlike autoregressive models, which predict next value in a sequence based only on previous values, BERT4Rec uses bidirectional self-attention to model both past and future user behavior, giving it a more complete view of the sequence [21].

Mathematical Formulation:

The input to BERT4Rec is a sequence of user-item interactions such as:

$$X = [x_1, x_2, \ldots, x_T]$$

where x_t denotes the item user interacted with at position t, and fixed percentage of those positions are randomly masked during training. The [MASK] token is predefined and assigned a unique ID in the vocabulary. Sequence with masked items looks like:

$$\tilde{X} = [x_1, \texttt{[MASK]}, x_3, \ldots, x_T]$$

This masked sequence $\tilde{X}$ is passed through a stack of L Transformer encoder layers, where each of these layers is composed of multi-head self-attention and a feed-forward sublayer. For each layer l and position t, the hidden representation is computed as:

$$H_t^{(l)} = \text{LayerNorm}(H_t^{(l-1)} + \text{MultiHead}(H^{(l-1)}))$$

$$H_t^{(l)} = \text{LayerNorm}(H_t^{(l)} + \text{FFN}(H_t^{(l)}))$$

Where:

- $H_t^{(l)}$: represent hidden representation of item x_t at layer l

- MultiHead: represents multi-head self-attention function

- FFN: represent position-wise feed-forward network

- LayerNorm: represents layer normalization

BERT4Rec tackles some big challenges in sequential recommendation by using bidirectional self-attention and predicting masked items in a user's interaction history. Transformer encoder within BERT4Rec captures dependencies across the whole sequence, which helps it to learn from past and future interactions. The masking mechanism trains the model to guess missing items based on surrounding context. Because BERT4Rec doesn't rely strictly on item order, it performs nicely even when user history is partially missing. However, BERT4Rec has several limitations. It is not good for real-time recommendation systems, as it relies on future context, which isn't available in live scenarios. Also, the Transformer encoder is computationally expensive, specifically when processing long sequences, making training heavy. BERT4Rec requires careful tuning, including the masking strategy and hyperparameter selection [21], [14], [17].

3.2.2 SASRec

SASRec (Self-Attentive Sequential Recommendation) is a unidirectional Transformer based model which is particularly developed for sequential recommendation. It replaces traditional recurrent architectures like one in RNNs and GRUs with self-attention which allows it better parallelization and efficient modeling of both short and long range dependencies in user behavior. Compared to BERT4Rec, SASRec uses causal masking, which limits each item to attend only to past items which makes it suitable for realtime recommendation systems [9].

Mathematical Formulation: Let $M \in \mathbb{R}^{N \times d}$ be the item embedding matrix, where N is the total number of items and d is the embedding dimension (size of vector which is used to represent each item). Given a fixed padded sequence $s = [s_1, s_2, \ldots, s_n]$, we retrieve the input embedding matrix $E \in \mathbb{R}^{n \times d}$[1] where $E_i = M_{s_i}$.

Since self-attention model doesn't use any recurrent or convolutional model, positions of previous items are not known to model. To keep order of items in the matrix, positional embeddings $P \in \mathbb{R}^{n \times d}$ are added to item vectors using formula:

[1] n is maximum sequence length

$$\hat{\mathrm{E}} = \begin{bmatrix} \mathrm{M}_{s_1} + \mathrm{P}_1 \\ \mathrm{M}_{s_2} + \mathrm{P}_2 \\ \vdots \\ \mathrm{M}_{s_n} + \mathrm{P}_n \end{bmatrix}$$

where $\hat{E}$ are item embeddings and P are positional encodings.

To demonstrate this lets assume:

- Item vocabulary size $N = 5$ where items are: [A, B, C, D, E]
- Embedding dimension $d = 3$
- Input sequence: [B, D, A] which would have ID [1,3,0] and this is important since model can't process raw "A", "B", etc. inputs, instead it needs integer values which are IDs, to look up corresponding vectors in the embedding matrix.
- Max sequence length $n = 3$

Table 3.2: Vocabulary items and their corresponding embedded vector

Item	Embedding (size 3)
A	[0.1, 0.3, 0.5]
B	[0.2, 0.4, 0.6]
C	[0.5, 0.5, 0.5]
D	[0.3, 0.7, 0.2]
E	[0.9, 0.1, 0.2]

Embedded items [A, B, C, D, E,] are shown in Figure 3.2. Embedding of input sequence [B, D, A] is:

$$M = \begin{bmatrix} 0.2 & 0.4 & 0.6 \\ 0.3 & 0.7 & 0.2 \\ 0.1 & 0.3 & 0.5 \end{bmatrix}$$

Next thing needed for calculation is positional embedding matrix P. In SASRec, positional embeddings are learnable parameters, not fixed encodings. As Kang and McAuley explained in the original SASRec paper [9], using fixed positional encodings resulted in worse performance in their case.

Since max sequence length $T = 3$ and embedding dimension $d = 3$, then $P \in \mathbb{R}^{3\times3}$, it would look like:

Table 3.3: Example positional embedding matrix

Position	Positional Encoding
1	[0.01, 0.02, 0.03]
2	[0.04, 0.05, 0.06]
3	[0.07, 0.08, 0.09]

$$P = \begin{bmatrix} 0.01 & 0.02 & 0.03 \\ 0.04 & 0.05 & 0.06 \\ 0.07 & 0.08 & 0.09 \end{bmatrix}$$

Now since we have E matrix and positional embedding matrix P it can be calculated $\hat{E} = M + P$ and it would look like:

$$\hat{E} = \begin{bmatrix} 0.21 & 0.42 & 0.63 \\ 0.34 & 0.75 & 0.26 \\ 0.17 & 0.38 & 0.59 \end{bmatrix}$$

Each Transformer encoder layer within SASRec has two parts: self-attention layer and feed-forward layer. The input to the self-attention mechanism is the position-aware item embedding matrix $\hat{\mathrm{E}} \in \mathbb{R}^{n\times d}$, where $\hat{\mathrm{E}}_i = \mathrm{M}_{s_i} + \mathrm{P}_i$. Queries, Keys, and Values are computed as linear projections of $\hat{\mathrm{E}}$:

$$\mathrm{Q} = \hat{\mathrm{E}}\mathrm{W}_Q, \quad \mathrm{K} = \hat{\mathrm{E}}\mathrm{W}_K, \quad \mathrm{V} = \hat{\mathrm{E}}\mathrm{W}_V$$

where $\mathrm{W}_Q, \mathrm{W}_K, \mathrm{W}_V \in \mathbb{R}^{d\times d}$ are learned weight matrices. In l-th Transformer encoder layer self-attention output is calculated using formula:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}} + M\right)V$$

where:

- Q: Query vectors
- K: Key vectors
- V: Value vectors
- d: Dimensionality of vectors which is used for scaling so there wouldn't be any large value which would influence softmax
- M: represents causal mask matrix to ensure that model only attends to previous items where:

$$M_{i,j} = \begin{cases} 0 & \text{if } j \leq i \\ -\infty & \text{if } j > i \end{cases}$$

A feed forward layer has a purpose of looking at items one by one for further processing and transformation of its information. The feed-forward network (FFN) in SASRec is a small neural network applied independently to each item within the sequence. It transforms the output of the self-attention layer to better model relationships between item features which as a result helps the model learn more complex patterns in data. To calculate FFN this formula is used:

$$\text{FFN}(S_i) = \text{ReLU}(S_i W^{(1)} + b^{(1)})W^{(2)} + b^{(2)}$$

where:

- $W^{(1)}$, $W^{(2)}$ are d x d matrices
- $b^{(1)}$, $b^{(2)}$ are dimensional vectors
- S_i is output of the self-attention layer at position i ($S_i = Attention(Q, K, V)_i$)

Example: How FFN Works

Let the self-attention output be: $S_i = \begin{bmatrix} 0.6 & -0.2 \end{bmatrix}$

Let first layer weights ($W^{(1)}$) and bias ($b^{(1)}$) be:

$$W^{(1)} = \begin{bmatrix} 1 & -1 \\ 0.5 & 2 \end{bmatrix}, \quad b^{(1)} = \begin{bmatrix} 0.1 & 0.1 \end{bmatrix}$$

Now it can be calculated first part of the formula ($S_i W^{(1)} + b^{(1)}$), and that would be:

$$S_i W^{(1)} + b^{(1)} = \begin{bmatrix} 0.6 & -0.2 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 0.5 & 2 \end{bmatrix} + \begin{bmatrix} 0.1 & 0.1 \end{bmatrix} =$$

$$\begin{bmatrix} 0.6 - 0.1 & -0.6 - 0.4 \end{bmatrix} + \begin{bmatrix} 0.1 & 0.1 \end{bmatrix} = \begin{bmatrix} 0.6 & -0.9 \end{bmatrix}$$

After the first layer output is calculated now we need to use Rectified Linear Unit (ReLU) formula which is $ReLU(x) = max(0, x)$. Using ReLU formula, we get:

$$\text{ReLU}(\begin{bmatrix} 0.6 & -0.9 \end{bmatrix}) = \begin{bmatrix} 0.6 & 0 \end{bmatrix}$$

Let second layer weights ($W^{(2)}$) and bias ($b^{(2)}$) be:

$$W^{(2)} = \begin{bmatrix} 1 & 2 \\ -1 & 0.5 \end{bmatrix}, \quad b^{(2)} = \begin{bmatrix} 0.0 & 0.2 \end{bmatrix}$$

Now second layer output can be calculated:

$$\begin{bmatrix} 0.6 & 0 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ -1 & 0.5 \end{bmatrix} + \begin{bmatrix} 0.0 & 0.2 \end{bmatrix} = \begin{bmatrix} 0.6 & 1.2 \end{bmatrix} + \begin{bmatrix} 0.0 & 0.2 \end{bmatrix} = \begin{bmatrix} 0.6 & 1.4 \end{bmatrix}$$

Final output of the feed-forward network (FFN) is:

$$\text{FFN}(S_i) = \begin{bmatrix} 0.6 & 1.4 \end{bmatrix}$$

After the self-attention and FFN layers process the input sequence, the model ends up with a final representation for each time step, which is new vector for every position in the sequence. The prediction score for each position i and target item $\hat{y}_i$ is:

$$\hat{y}_i = F_i \cdot M_i$$

where $F_i = FFN(S_i)$ is output of feed-forward layer at positon i and M_i is the embedding of item i, which may be the true next item or a negative sample.

The model is trained with a binary cross-entropy loss:

$$L = -\sum_t \left[\log(\sigma(\hat{y}_i)) + \sum_{j \in N_t} \log\left(1 - \sigma(\hat{y}_j)\right) \right]$$

where:
- $\sigma()$ is the sigmoid function
- $\hat{y}_i$ is the score for the true item
- $\hat{y}_j$ is the score for a negative item j
- $\mathcal{N}_t$ is the set of negative items sampled at time t

Cross-entropy calculates how close the model's predicted probabilities are to the correct ones. It penalizes confident wrong guesses and rewards confident correct ones. In the case of SASRec, cross-entropy compares the predicted score of the actual next item (the positive class) to the scores of randomly selected non-interacted items (the negative class). Through this process of rewarding and penalizing, cross-entropy guides the model during training to learn attention weights and embeddings that help it better predict the next item in a user's sequence [4], [12].

SASRec has several advantages that make it suitable for the recommendation task. Since SASRec removes recurrence, it can process the entire input sequence in parallel, which makes it efficient in terms of training. Also causal masking mechanism ensures that each item can only attend to past interactions, not future ones. SASRec is suitable for real-time recommendation systems. This may also be viewed as a limitation of the SASRec model,

but for this project, which proposes a new model for a recommendation system, it is not a concern. Since SASRec relies on historical data, its effectiveness may be reduced if the data is sparse or limited [9].

Selected transformer

After comparing BERT4Rec and SASRec, we selected SASRec for several reasons. Although both SASRec and BERT4REc are made for recommendation tasks, because of difference in masking its noticeable that BERT4Rec is not suitable for this project since it relies on bidirectional attention. Prediction of masked item within BERT4Rec is appropriate for real time scenarios which is not case of our project. SASRec uses causal masking, which demands that items only look into history (past interactions) but not in the future which makes it appropriate for this project. Furthermore, SASRec is lighter in both model complexity and computational cost than BERT4Rec due to its unidirectional attention and causal masking.

3.3 Proposed model

The proposed GSRec hybrid model combines a previously selected GRU neural network and SASRec transformer to learn both short term and long term user preferences. Figure 3.4 shows a representation of the proposed model. Input for the model is the user's historical interaction sequence, which is fed into the SASRec module. Within SASRec each item is mapped to a sequence of dense embedding vector, and a positional embedding is added so the SASRec component can preserve the order of the interactions. After the item is embedded into a dense vector using a learnable embedding layer and positional embedding is added input goes through multiple Self-attention blocks. Those multiple blocks have the task of capturing short-term dependencies between items. The output of the SASRec component is a tensor of shape $[batch\_size, sequence\_length, hidden\_units]$, which represents contextualised embedding for each timestep in the sequence is then passed to GRU. The reason why we decided to use only the self-attention layer from SASRec is that feed-forward layer, which comes after the self-

attention layer, doesn't retain long-term dependencies as good as GRU, as a result of the GRU gating mechanism (update and reset gate).

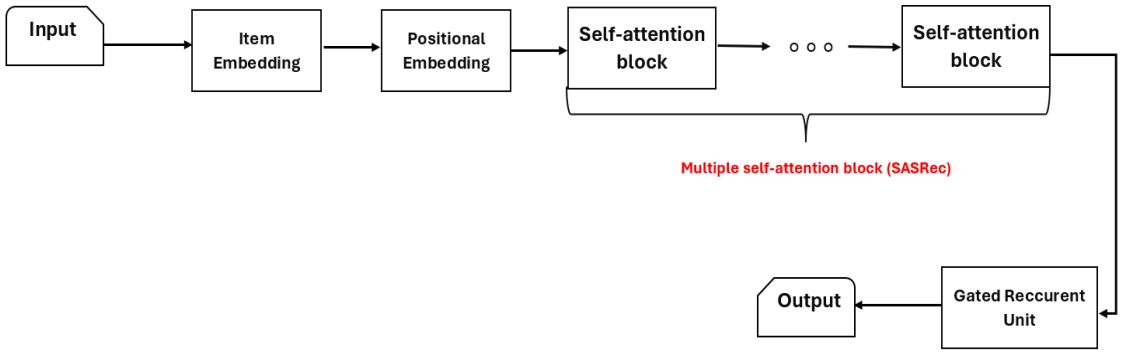

Figure 3.4: Representation of proposed model

One of the potential limitations of our design is that GRU receives only refined outputs from SASRec, but not raw item embeddings. Since in our design, SASRec had proposed that the self-attention layer focuses on short-term patterns, there is a possibility that it will suppress information which can be valuable for long term preference. This could limit the GRU's usefulness in learning deeper user behavior. However, we choose this pipeline architecture for its simplicity and computational efficiency.

Output of the self-attention layer is then passed into a GRU for learning long-term dependencies, as mentioned before. GRU takes contextualized embeddings from SASRec as input and processes them recurrently. While doing this GRU updates its internal memory so information across sequences are retained. From the GRU output, only the hidden layer at the last time stamp is selected, since it encapsulates the overall user's preference up to that point. Final hidden representation is then through a fully connected output layer, represented as a vector of size equal to a total number of items. This vector contains an unnormalized prediction score (logits) for all possible next items.

During training, the GSRec model uses BCEWithLogitsLoss as a loss function. BCEWithLogitsLoss combines Sigmoid function together with Binary Cross-Entropy loss into a single formula. It is most often used for binary classification tasks where each prediction is either 0 or 1. Sigmoid part is used to transform raw outputs (logits) into probabilities between 0 and 1 where Binary Cross Entropy loss is used to measure how well the predicted probability matches the true label. The formula for BCEWithLogitsLoss is:

$$\text{BCEWithLogitsLoss}(x, y) = -\left[y \cdot \log\left(\sigma(x)\right) + (1 - y) \cdot \log\left(1 - (\sigma(x))\right)\right]$$

where:

- x: represents the logit, which is the raw output of your model
- y: represents the target label, either 0 or 1.
- $\sigma(x)$: represents the sigmoid function applied to the logit to convert it into a probability

In our architecture, the loss function is applied at the end of the model pipeline directly after the output layer, and it is used to guide learning by penalising incorrect predictions.

To demonstrate how BCEWithLogitsLoss works let's say that the model predicts $x = 2.0$. The sigmoid function turns this into:

$$\sigma(2.0) = \frac{1}{1 + e^{-2.0}} \approx 0.88$$

Now, if true label is $y = 1$, the loss would be:

$$-\left[1 \cdot \log(0.88) + 0 \cdot \log(1 - 0.88)\right] = -\log(0.88) \approx 0.127$$

If true label is $y = 0$, the loss would be:

$$-\left[0 \cdot \log(0.88) + 1 \cdot \log(1 - 0.88)\right] = -\log(0.12) \approx 2.12$$

Chapter 4

Setup

In this chapter, we present the setup used to evaluate the performance of our proposed model. First, we will discuss the datasets used to train and test our model. Next, we will introduce baseline models, which are used so we can compare them with our model. We will then elaborate on the metrics we used to monitor our model's efficiency together with baseline models. Lastly, we will briefly describe the hyperparameters we used and how we selected them.

4.1 Dataset

For this project we used two publicly available datasets. Datasets are Amazon Beauty and MovieLens 1M (ml-1m). We picked those datasets since they represent two different domains: e-commerce (Amazon Beauty) and entertainment (MoviLense 1M), so we can test how our model works on different datasets.

The MovieLens 1M dataset consists of 1 million ratings collected from the MovieLens website. It is an often used dataset for movie recommendation. Dataset is consist of:

- Number of users: 6,040
- Number of items: 3,952
- Number of interactions: 1,000,209
- Average 165 interactions per user

The Amazon Beauty dataset is a subset of the larger Amazon product review dataset. It is also one of the common datasets used for recommendation tasks. Dataset is consist of:

- Number of users: 52,024
- Number of items: 57,289
- Number of interactions: 394,908
- Average of 7.6 interactions per user.

To prepare data for training we filter out users who have less than 3 interactions. We decided to do this because users with very short histories don't provide enough sequence data to meaningfully train the model, in this case the cold start problem can occur. The cold start problem refers to the challenge of making precise recommendations when there is not enough data about users or items [23]. For partitioning of data we used Leave-One-Out (LOO) partitioning. LOO partitioning split user's interaction sequence $S_u = \{s_1^u, s_2^u, \ldots, s_{|S_u|}^u\}$into three segments: the most recent interaction $s_{|S_u|}^u$ is reserved for testing, the second most recent $s_{|S_u|-1}^u$ is used for validation, and the remaining interactions $\{s_1^u, \ldots, s_{|S_u|-2}^u\}$ form the training sequence, where u refers to a specific user.

4.2 Baseline and hyperparameter values

To verify the effectiveness of our method, we compare it with the following representative baselines: To measure how good is our proposed model, we compare it with few different baselines:

- PopRec - this is a straightforward baseline that ranks items according to their popularity.
- GRU4Rec - we also want to observe the performance of our model in comparison with its components.
- SASRec - we are also using this one since it's a component in our hybrid model.
- BERT4Rec - we decided to use this transformer to see how our model performs compared to bidirectional transformer

4.3 Hyperparameters

For optimal performance of our GSRec model on datasets with different sparsity characteristics (number of users, items, and interactions), we tuned hyperparameters separately for both the Beauty and ML-1M datasets.

For the Beauty dataset, we set a batch size of 512 to stabilize gradient updates. In sparse dataset, small batch sizes can have many interactions with almost no data, so increasing batch size will have as a result that more data points are going to be included, and noise in gradient estimates will be somewhat smoothed out. Also, we set the learning rate to 0.0003, to allow more gradual updates and in that way prevent overfitting. The sequence length (maxlen) was set to 150, since this is a sparse dataset and it wasn't expected that user sequences were going to be long. Also, to reduce memory in case the user has less interaction than maxlen is set, padding will be added so sequence reach the size of maxlen. This padding will also be included when generating the attention matrix, so to reduce memory usage, we set maxlen to be 150. The dropout rate of 0.2 and L2 regularization (l2_emb) of 0.02 help control overfitting, which is crucial in sparse environments.

For ML-1M, we set batch size to 128. Smaller batches can introduce useful noise in denser datasets, since updates are gonna occur more often, and parameters are going to be updated more often than in the Beauty dataset, which will lead to faster learning. We increased the learning rate slightly

to 0.0005. A longer sequence length of 300 was chosen since the number of interactions increases here, and it is expected that each user have more interactions than in the Beauty dataset. The model size was also scaled up, with 500 hidden units to accommodate the increased data complexity.

4.4 Evaluation metrics

To evaluate our model, we used a couple metrics. Firstly we used two common Top-N metrics, NDCG@k and HitRate@k. HitRate@k measures the percentage of users whose ground truth item appears in the top k recommended items. Formula for HitRate@k is:

$$\text{HitRate@K} = \frac{1}{|U|} \sum_{u \in U} \mathbb{I}\left(i_u^* \in \text{Top-K}(u)\right)$$

where:

- $\mathcal{U}$ is the set of users
- i_u^* is the ground-truth item for user u
- Top-K(u) is the top K recommended items for user u
- $\mathbb{I}(\cdot)$ is the indicator function (if true then 1, otherwise 0)

Since we have only 1 test item for each user, then HitRate@k is equivalent to Recall@k, so that reason we didn't think to include Recall@k as evaluation metrics.

Normalized Discounted Cumulative Gain (NDCG) is a ranking quality metric. It measures the quality of a ranked list, how well are the top-K recommended items compared to the ideal order.

Besides those two metrics, we also used Mean Reciprocal Rank . Mean Reciprocal Rank (MRR@k) measures how early the first relevant item appears

in the ranked list of top-k recommendations. Higher results tell that the relevant item appeared higher in the ranking list. Unlike NDCG, which accounts for multiple relevant items and their positions, RNN only takes the first relevant item into account. Formula for MRR is:

$$\text{MRR@}K = \frac{1}{|U|} \sum_{s=1}^{|U|} \frac{1}{\text{rank}_u^K}$$

where:

- $|U|$ represents a total number of sequences
- rank_u^K represents the rank position of the ground-truth item for sequence u within the top-K recommended items.

Chapter 5

Experiments

In this chapter, we will provide results for our proposed model and compare it with different baselines described in section 4.2. The goal is to validate whether combining SASRec and GRU into a pipeline architecture results in better performance on sequential recommendation tasks accross different datasets. Results of experiments are displayed in Table 5.1. The primary objective of our experiments is to evaluate our proposed GSRec model in terms of capturing both short-term and long-term user preferences. With those experiments, we aim to answer the following questions:

1. Does GSRec outperform state-of-the-art models?
2. How does it compare to its components (SASRec and GRU4Rec)?
3. How well does GSRec perform on datasets with different characteristics?

1. Does GSRec outperform state-of-the-art models?

In this experiment, we compare the performance of POP, BERT4Rec, and our proposed GSRec model. We exclude SASRec and GRU4Rec from this comparison, as their performance is discussed in detail in the following experiment.

Table 5.1: Performance Comparison of Baseline Models and the Proposed Hybrid Model on Beauty and ML-1M Datasets

Datasets	**Metric**	POP	GRU4Rec	SASRec	BERT4Rec	GSRec
Beauty	HR@10	0.0823	0.2133	0.2450	0.2715	0.2357
	NDCG@5	0.0351	0.0725	0.1323	0.1387	0.1296
	NDCG@10	0.0442	0.1035	0.1517	0.1773	0.1349
	MRR	0.0347	0.0997	0.1488	0.1529	0.1348
ML-1m	HR@10	0.1283	0.5407	0.6429	0.6385	0.4212
	NDCG@5	0.0324	0.3058	0.4011	0.4162	0.2633
	NDCG@10	0.0588	0.3404	0.4392	0.4611	0.2841
	MRR	0.0459	0.2902	0.3543	0.3943	0.2209

When looking at Table 5.1, it is evident that GSRec consistently outperforms the POP baseline on both datasets, thanks to its more complex architecture and ability to capture short and long-term dependencies. However, BERT4Rec achieves the best performance overall. On the Beauty dataset, NDCG@10 value for GSRec is 0.1349, while BERT4Rec value is 0.1773, a small difference of around 4.2 percentage points. While on ML-1M dataset, GSRec's HR@10 value is 0.4212, which is much lower compared to BERT4Rec's value, which is 0.6385.

Even though GSRec theoretically benefits from combining SASRec (for short-term interest) and GRU (for long-term behavior), our design might introduce redundancy. Since SASRec already captures short-term dependencies through self-attention, passing its output to a GRU may have the consequence of reprocessing the same information without adding any value. In some cases, GRU could even blur the high-resolution patterns SASRec identifies, leading to a loss of predictive precision.

2. How does it compare to its components (SASRec and GRU4Rec)?

In this experiment, we compare the performance of GSRec with SASRec and GRU4Rec to see if our proposed model predicts better than its components. As shown in Table 5.1, GSRec outperforms GRU4Rec on the Beauty dataset across all four metrics. For example, on Beauty:

- HR@10: GSRec = 0.2357 vs. GRU4Rec = 0.2133
- NDCG@10: GSRec = 0.1349 vs. GRU4Rec = 0.1035
- MRR: GSRec = 0.1348 vs. GRU4Rec = 0.0997

This suggests that combining GRU with SASRec provides an improvement over using GRU alone, especially in sparse environments like Beauty.

But SASRec still outperforms GSRec in all four metrics. For example, in HR@10 SASRec value is 0.2450 compared to GSRec's 0.1991. While the difference isn't so big, it shows that adding GRU on top SASRec may not improve its strengths and could even make SASRec worse.

On the ML-1M dataset, both GRU4Rec and SASRec outperform GSRec. SASRec achieves the highest overall performance. When looking HR@10 SASRec values is 0.6429 while GSRec value is 0.4212. This suggests that the GRU component within GSRec does not effectively capture long-term dependencies on datasets that are more dense.

3. How well does GSRec perform on datasets with different characteristics?

We chose this experiment to see how our model performs on sparse (Beauty) and denser (ML-1m) datasets. Even though we tackled this experiment in the previous two experiments we want to elaborate on it more.

From the results in Table 5.1 its obvious that our model performs better on dense ML-1m dataset compared to sparse Beauty dataset. When looking sparse dataset its visible that our model outperformed GRU4Rec, which is its component. This indicates that through Self-Attention layer from SASRec, our model learns better than GRU4Rec does by itself. But, on the dense ML-1M dataset, GSRec does not perform as expected. Its Hit Rate@10 is only 42.12%, which is quite lower than for SASRec's 64.29% and GRU's 54.07%. This suggests that passing the output of SASRec into the GRU may

reduce the performance rather than improve it. One possible explanation is that in dense datasets such as ML-1m, recent user actions are highly predictive, and SASRec already models them effectively through the self-attention layer. When these embeddings are passed into a GRU, the GRU may reprocess them, which can result in worse signals. Also, the GRU's sequential processing may introduce noise or overwrite the weights which are already been learned by the attention mechanism within SASRec. In this case, the GRU does not add new information and instead acts as a bottleneck, degrading the overall performance of the model.

Chapter 6

Future Work

For future work, there are multiple things that we can try and explore to attempt to make our model better. Firstly, we can enhance the SASRec part of our GSRec model with Feed-Forward Layer, which we excluded in our arhitecture. Our model doesn't include FFL, but there is a possibility that this would enhance the performance of our model since FFL add non linearity after the attention layer. This nonlinearity would help with learning, since linear functions can't model complex patterns [10].

Another possible improvement that we can try is stacking multiple GRUs to create multi-layered GRU. This seem as an promising way also, since multiple GRUs can lead to better results, and the GSRec model will in that way be able to learn richer patterns in user history.

Eventhough GSRec was evaluated on Beauty and ML-1M, future experiments could include denser datasets with more complex user behavior patterns, such as: Netflix dataset.

Chapter 7

Conclusion

In this thesis, we proposed a new hybrid sequential recommendation model named GSRec which combines the SASRec and GRU in a sequential pipeline architecture. Our goal for this thesis was to create model which can efficiently capture both short-term and long-term user preferences with a fairly simple but efficient architecture.

Through experiments on two datasets, Amazon Beauty and MovieLens 1M we showed that our GSRec model performs competitively with state-of-the-art models in sparse dataset. Even though it does not outperform BERT4Rec in denser dataset, it shows competitive performance in comparison with state-of-the-art models on sparse dataset.

GSRec shows a promising approach for hybrid model architecture in sequential recommendation, particularly in its attempt to combine the strengths of self-attention and recurrent networks. While it does not surpass simpler models like SASRec in performance or complexity, the design emphasises the possible advantages of integrating various modeling strategies.

Future work may explore possible improvements such as including multi-layer GRUs or integrating feed-forward layers, also applying the model to more diverse datasets could improve how our proposed GSRec model works across different domains.

Bibliography

[1] Muhammad Abdullah. *Comparison and Architecture of LSTM, GRU and RNN. What Are the problems with RNN to process long sequences*. Access: 03-05-2025. 2023. URL: https://medium.com/@muhabd51/comparison-and-architecture-of-lstm-gru-and-rnn-1c9afe11b09f.

[2] Y. Bengio, P. Simard, and P. Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE Transactions on Neural Networks* 5.2 (1994), pp. 157–166. DOI: 10.1109/72.279181.

[3] Junyoung Chung et al. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling". In: *CoRR* abs/1412.3555 (2014). arXiv: 1412.3555. URL: http://arxiv.org/abs/1412.3555.

[4] Daniel Godoy. *Understanding binary cross-entropy / log loss: a visual explanation | Towards Data Science*. Access: 14-05-2025. 2018. URL: https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a/.

[5] Huaiwen He et al. "GAT4Rec: Sequential Recommendation with a Gated Recurrent Unit and Transformers". In: *Mathematics* 12.14 (2024). ISSN: 2227-7390. DOI: 10.3390/math12142189. URL: https://www.mdpi.com/2227-7390/12/14/2189.

[6] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.1 (1997). DOI: https://doi.org/10.1162/neco.1997.9.1.1. URL: https://www.bioinf.jku.at/publications/older/2604.pdf.

[7] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. 3rd. Online manuscript released January 12, 2025. 2025. URL: https://web.stanford.edu/~jurafsky/slp3/.

[8] Aamna Kamran. *Long Short-Term Memory (LSTM) - Comprehensive Guide to LSTMs in AI*. Access: 28-04-2025. 2024. URL: https://metaschool.so/articles/lstm-long-short-term-memory?utm_source=chatgpt.com.

[9] Wang-Cheng Kang and Julian J. McAuley. "Self-Attentive Sequential Recommendation". In: *CoRR* abs/1808.09781 (2018). arXiv: 1808.09781. URL: http://arxiv.org/abs/1808.09781.

[10] Balaji Kithiganahalli. *Understanding Transformers Intuitively: From Non-Linearity to Attention | by Balaji Kithiganahalli | May, 2025 | Medium | GoPenAI*. Access: 31-05-2025. 2025. URL: https://blog.gopenai.com/understanding-transformers-intuitively-from-non-linearity-to-attention-85493c5c74a5.

[11] Chengyu Li and Guoqi Qian. "Stock Price Prediction Using a Frequency Decomposition Based GRU Transformer Neural Network". In: *Applied Sciences* 13.1 (2023). ISSN: 2076-3417. DOI: 10.3390/app13010222. URL: https://www.mdpi.com/2076-3417/13/1/222.

[12] Gleb Mezentsev et al. "Scalable Cross-Entropy Loss for Sequential Recommendations with Large Item Catalogs". In: *18th ACM Conference on Recommender Systems*. ACM, Oct. 2024, 475–485. DOI: 10.1145/3640457.3688140. URL: http://dx.doi.org/10.1145/3640457.3688140.

[13] Nerdjock. *Recurrent Neural Network — Lesson 3: Working and Mathematics of Vanilla RNNs*. https://medium.com/@nerdjock/recurrent-neural-network-lesson-3-working-and-mathematics-of-vanilla-rnns-bf5b196e7194. Access: 06-05-2025. 2021.

[14] Thu Nguyen et al. "H-BERT4Rec: Enhancing Sequential Recommendation System on MOOCs based on Heterogeneous Information Networks". In: *IEEE Access* PP (Jan. 2024), pp. 1–1. DOI: 10.1109/ACCESS.2024.3462830.

[15] Christopher Olah. *Understanding LSTM Networks*. https://colah.github.io/posts/2015-08-Understanding-LSTMs/. Access: 06-05-2025. 2015.

[16] Neri Van Otten and Neri Van Otten. *Understanding Elman RNN — Uniqueness How To Implement In Python With PyTorch*. Access: 06-05-2025. 2023. URL: https://spotintelligence.com/2023/02/01/elman-rnn/#:~:text=Elman%20recurrent%20neural%20networks%20(RNNs)%20are%20a%20type%20of%20neural,at%20the%20next%20time%20step..

[17] Shaina Raza et al. *A Comprehensive Review of Recommender Systems: Transitioning from Theory to Practice*. 2025. URL: https://arxiv.org/abs/2407.13699.

[18] Shipra Saxena. *Introduction to Gated Recurrent Unit (GRU)*. Access: 06-05-2025. 2021. URL: https://www.analyticsvidhya.com/blog/2021/03/introduction-to-gated-recurrent-unit-gru/#h-applications-of-gated-recurrent-unit.

[19] Ravjot Singh. *GRU Explained: The Simplified RNN Solution for Sequential Data*. Access: 05-05-2025. 2025. URL: https://ravjot03.medium.com/gru-explained-the-simplified-rnn-solution-for-sequential-data-c706d0d149c5.

[20] Rishabh Singh. *Long Short-Term Memory (LSTM) Networks - Rishabh Singh - Medium*. Access: 29-04-2025. 2024. URL: https://medium.com/%40RobuRishabh/long-short-term-memory-lstm-networks-9f285efa377d.

[21] Fei Sun et al. "BERT4Rec: Sequential Recommendation with Bidirectional Encoder Representations from Transformer". In: *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. CIKM '19. Beijing, China: Association for Computing Machinery, 2019, 1441–1450. ISBN: 9781450369763. DOI: 10.1145/3357384.3357895. URL: https://doi.org/10.1145/3357384.3357895.

[22] Ashish Vaswani et al. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[23] Hongli Yuan and Alexander Hernandez. "User Cold Start Problem in Recommendation Systems: A Systematic Review". In: *IEEE Access* (Dec. 2023). DOI: 10.1109/ACCESS.2023.3338705.

[24] Wengang Zheng et al. "GRU–Transformer: A Novel Hybrid Model for Predicting Soil Moisture Content in Root Zones". In: *Agronomy* 14.3 (2024). ISSN: 2073-4395. DOI: 10.3390/agronomy14030432. URL: https://www.mdpi.com/2073-4395/14/3/432.

[25] Jiancong Zhu. "Comparative study of sequence-to-sequence models: From RNNs to transformers". In: (2023).