Music Recommendation Project

zhaiminghaobb

November 2024

Chapter 1

Model Explanation

1.1 Gated Recurrent Unit Model

1.1.1 History

Gated Recurrent Units (GRUs) are a type of recurrent neural network architecture introduced by Kyunghyun Cho and colleagues in 2014. GRUs use a gating mechanism to control the flow of information, enabling the model to retain or forget features as needed. By dynamically updating its hidden state, GRUs can effectively capture long-term dependencies in sequential data. With fewer parameters compared to some other architectures, GRUs have proven to be efficient and effective in tasks like polyphonic music modeling, speech signal modeling, and natural language processing. Their design highlights the importance of gating mechanisms for improving the performance of recurrent neural networks

1.1.2 Architecture

The following parameters play key roles in GRUs:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z)$$

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$$

$$\hat{h}_t = \phi(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)$$

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

Variables

- $x_t \in \mathbb{R}^d$: Input vector(data point)
- $h_t \in \mathbb{R}^d$: hidden state
- $\hat{h}_t \in \mathbb{R}^d$: Candidate hidden state
- $z_t \in (0,1)^d$: Update gate vector
- $r_t \in (0,1)^d$: Reset gate vector
- $W \in \mathbb{R}^{e \times d}, U \in \mathbb{R}^{d \times d}, b \in \mathbb{R}^d$: Parameter matrices and vector which will be learning during training.

Activation Functions

- σ: The original is a logistic function.
- ϕ: The original is a hyperbolic tangent.

Alternative activation functions are various based on different tasks, provided that $\sigma(x) \in [0,1]$ and $\phi \in [-1,1]$.

1.1.3 General Procedure

1. Starting with the Input Tensor X:

The input matrix **X** is called **Tensor**, which contains three dimensions: batch size B; sequence length T; and Input size d(as number of features)

$$\mathbf{X} \in \mathbb{R}^{B \times T \times d}$$

- Batch size (B) :Since model cannot take the whole data at once, therefore, we as model trainer, need to feed the model piece by piece. Each batch represents a small chunk of data that model consumes. If B = 2, then model would take two groups of data at the same time.

- Sequence Length: Number of steps that model considers as a cycle. If Sequence Length = 20, then the model would run it own operations(mostly upgrade parameters) for every 20 steps.

- Input Size: it means the number of features for one data point contains.

In the most deep learning training, batch size usually realted with model efficiency. Often when the data is large, the batch size tend to be large (32,64,128 etc), it means that the model would calculate these batches at the same time like parallel calculation. As RNN structured model, each time t, there are 32 pieces of data have been put into the model at the same time. Combining every piece, model will form a new tensor has shape (32,11). Each row indicates the first data point in each batch. At time 1, 32 data points have been put into the model, and calculated predicted values ($[\hat{y_1}, \hat{y_{33}}...]$).

2. Hidden State ($\mathbf{h}_0$) Initialization:

At the first time step ($t = 1$), the GRU does not have a previous hidden state, so we initialize it as $\mathbf{h}_0$.Adjustment rule: hidden state will change every time.

- Dimensions:

$$\mathbf{h}_0 \in \mathbb{R}^{\text{BatchSize} \times h}$$

- Zero Initialization:

$$\mathbf{h}_0 = \mathbf{0} \in \mathbb{R}^{1 \times 512}.$$

- Random Initialization:

$$\mathbf{h}_0 \sim \mathcal{N}(0, 0.01).$$

At this stage, the input sequence X is fed into the GRU. To simplify the explanation, we will focus on a single batch and examine how it operates within the model. Each row in the input matrix represents a single data point from the batch, and together, they form a matrix that serves as the basis for the subsequent computations. Once the operation of a single batch is clear, it can be easily extended to multiple batches using matrix-based calculations.

The next step in the GRU is particularly important and can be a source of confusion. Many textbooks introduce the Update Gate (z_t) and Reset Gate (r_t) without having enough conceptual examples. Therefore, in this report, I will add an extra vivid example to help readers to understand the most confusing

part. Also, understanding these gates together rather than treating them as isolated operations, it's more intuitive to consider that they are calculated simultaneously. Although they share the same formula, their placement and what way they use within the GRU serve entirely different purposes. This distinction is crucial to fully grasp how the GRU balances past information with new input.

Suppose we have two small identical brushes, they are exactly the same. I can use one for cleaning my keyboard. For another one, I could use it to apply oil on food. Similarly for these two gates, they share the same formula, but we can use for different purposes to create different outcomes.

After we make initialization of $W_z, W_r, U_z, U_r, b_z, b_r$. At time t=1, these parameters are the same pairwise. However, once we reach the last step of a sequence. The parameter would update differently and change according to loss functions.

More details would be contained within introduction of candidate and formal hidden state.

3. Reset gate and Candidate hidden state (r_t & $\hat{h_t}$):

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$$

$$\hat{h}_t = \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)$$

- $r_t \in \mathbb{R}^h$: A vector in the range $[0, 1]$ that determines how much historical information to retain.

Parameter Definitions:

- $W_r x_t$: Influence of the current input on the Reset Gate.
- $U_r h_{t-1}$: Influence of the previous hidden state on the Reset Gate.
- b_r: Bias term.
- $W_h x_t$: Influence of the current input x_t on candidate hidden state.
- $r_t \odot h_{t-1}$: Reset Gate selectively filters the previous hidden state.
- U_h: Weights for the filtered historical information.
- b_h: Bias term.

- Each element $r_t[i]$ determines whether the i^{th} neuron in the previous hidden state (h_{t-1}) is included in the candidate hidden state calculation:
  - If $r_t[i] = 0$, the corresponding information is entirely discarded, and the candidate hidden state relies solely on the current input.
  - If $r_t[i] = 1$, the corresponding information is fully retained for candidate hidden state computation.
  - If $r_t[i] \in (0, 1)$, a fraction of the historical information is preserved.

The candidate hidden state ($\hat{h}_t$) represents the potential hidden state for the current time step, combining the current input with the selectively retained historical information. The final hidden state (h_t) is computed by further adjusting $\hat{h}_t$ using the Update Gate (z_t).

Numerical Example: Reset Gate and Candidate Hidden State

Given:

$$x_t = [0.1, 0.5], \quad h_{t-1} = [0.6, -0.4], \quad r_t = [0.2, 1.0]$$

Parameters:

$$W_h = \begin{bmatrix} 0.3 & -0.2 \\ 0.5 & 0.4 \end{bmatrix}, \quad U_h = \begin{bmatrix} 0.6 & -0.3 \\ 0.1 & 0.2 \end{bmatrix}, \quad b_h = [0.0, 0.1]$$

Step-by-step computation:

1. Filtered historical information via Reset Gate:

$$r_t \odot h_{t-1} = [0.2 \cdot 0.6, 1.0 \cdot (-0.4)] = [0.12, -0.4]$$

2. Compute the weighted sum of current input:

$$W_h x_t = \begin{bmatrix} 0.3 & -0.2 \\ 0.5 & 0.4 \end{bmatrix} \begin{bmatrix} 0.1 \\ 0.5 \end{bmatrix} = [0.07, 0.25]$$

3. Compute the weighted sum of filtered historical information:

$$U_h(r_t \odot h_{t-1}) = \begin{bmatrix} 0.6 & -0.3 \\ 0.1 & 0.2 \end{bmatrix} \begin{bmatrix} 0.12 \\ -0.4 \end{bmatrix} = [0.3, -0.062]$$

4. Add bias and combine terms:

$$W_h x_t + U_h(r_t \odot h_{t-1}) + b_h = [0.07, 0.25] + [0.3, -0.062] + [0.0, 0.1] = [0.37, 0.288]$$

5. Apply activation function:

$$\hat{h}_t = \tanh([0.37, 0.288]) = [0.353, 0.281]$$

The values $[0.353, 0.281]$ are the candidate hidden states ($\hat{h}_t$) in the GRU model, calculated at the current time step t. These values represent an intermediate result based on the input x_t and the filtered historical state $r_t \odot h_{t-1}$.

1. **Representation:**
   - These values are derived from a weighted combination of the current input x_t and the filtered historical state $r_t \odot h_{t-1}$, with added biases.
   - The tanh activation compresses the weighted sum into $[-1, 1]$, allowing the model to encode both excitatory (positive) and inhibitory (negative) influences.
2. **Role in Computation:**
   - The hidden states do not have direct physical units but encode abstract features of the input data. For example:
     - In house price prediction, $[0.353, 0.281]$ may correspond to abstract representations of market trends or local conditions at the current time step.
   - These values are passed to the GRU's update mechanism, where the update gate z_t determines how much of the candidate hidden state $\hat{h}_t$ contributes to the final hidden state h_t.
3. **Contextual Interpretation:**
   - For example, 0.353 might represent a positive correlation with upward market trends, while 0.281 could reflect a minor adjustment due to local factors. These values encode how the GRU integrates the influence of current and past information.

4. Update Gate and Final Hidden State (z_t & h_t)

After computing the candidate hidden state ($\hat{h}_t$), the GRU introduces the update gate (z_t) to determine the proportion of the past hidden state (h_{t-1}) and the candidate hidden state ($\hat{h}_t$) that should be retained to form the final hidden state (h_t).

The **update gate** is computed as:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z)$$

Here:

- $z_t \in \mathbb{R}^h$: A vector where each entry $z_t[i] \in [0, 1]$ controls the trade-off between retaining historical information and incorporating new information.
- $W_z x_t$: Represents the influence of the current input x_t.
- $U_z h_{t-1}$: Represents the influence of the past hidden state h_{t-1}.
- b_z: Bias term for the update gate.

Each element $z_t[i]$ decides the degree to which the i^{th} neuron retains information from the previous hidden state or updates it using the candidate hidden state.

Explanation:

- If $z_t[i]$ is close to 1, the final hidden state (h_t) heavily relies on the past hidden state (h_{t-1}).
- If $z_t[i]$ is close to 0, the final hidden state (h_t) primarily depends on the candidate hidden state ($\hat{h}_t$).
- If $z_t[i]$ is between 0 and 1, the final hidden state is a weighted combination of both the past hidden state and the candidate hidden state.

The **final hidden state** is computed as:

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

This equation can be broken down as follows:

- $z_t \odot h_{t-1}$: Retains a fraction of the past hidden state (h_{t-1}) based on the update gate (z_t).

- $(1 - z_t) \odot \hat{h}_t$: Incorporates new information from the candidate hidden state ($\hat{h}_t$) with the complementary fraction of z_t $(1 - z_t)$.

Example: Assume:

$$h_{t-1} = [0.5, 0.8, -0.3], \quad \hat{h}_t = [0.2, 0.6, -0.1], \quad z_t = [0.1, 0.5, 0.9]$$

The final hidden state (h_t) is computed as:

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

Performing element-wise operations:

$$h_t[0] = 0.1 \cdot 0.5 + 0.9 \cdot 0.2 = 0.23, \quad h_t[1] = 0.5 \cdot 0.8 + 0.5 \cdot 0.6 = 0.7, \quad h_t[2] = 0.9 \cdot (-0.3) + 0.1 \cdot (-0.1) = -0.28$$

Thus, the final hidden state is:

$$h_t = [0.23, 0.7, -0.28]$$

5. Final Output Layer

At this stage, we have the output of GRU models which is the final hidden state. However, since models are developed objected-oriented, we still need a final layer to connect with hidden state. That is output layer:

Purpose of the Output Layer:

- **Mapping to Target Space:** The output layer performs a linear transformation to convert the hidden state h_t into task-specific outputs. This is critical for tasks such as classification, where the final output must correspond to distinct categories, or regression, where it must represent continuous values.
- **Decoupling Hidden States:** By introducing an output layer, we maintain modularity and flexibility, allowing the hidden states to serve as intermediate representations, while the output layer is tailored to the specific task.

$$\hat{y_t} = W_{hy} h_t + b_{hy}$$

Here:

- $W_{hy} \in \mathbb{R}^{m \times h}$: Weight matrix connecting the hidden state to the output, where h is the hidden size and m is the output size.
- $b_{hy} \in \mathbb{R}^m$: Bias vector for the output layer.
- $\hat{y}_t \in \mathbb{R}^m$: Final output at time step t, where m is determined by the task (e.g., number of classes for classification or dimensions for regression).

Task-Specific Adjustments: Depending on the task, additional transformations might follow the output layer:

- For classification tasks, apply a softmax activation to y_t to compute probabilities:
$$\hat{y}_t = \text{softmax}(y_t)$$
- For regression tasks, y_t is directly used as the prediction.

Example: For a classification task with 5 classes and $h_t \in \mathbb{R}^{512}$:

$$W_{hy} \in \mathbb{R}^{5 \times 512}, \quad b_{hy} \in \mathbb{R}^5, \quad y_t \in \mathbb{R}^5$$

For a regression task with 3 outputs:

$$W_{hy} \in \mathbb{R}^{3 \times 512}, \quad b_{hy} \in \mathbb{R}^3, \quad y_t \in \mathbb{R}^3$$

Significance: The output layer is essential as it bridges the gap between hidden states and task-specific outputs. Without it, the hidden states cannot directly serve as the final model predictions.

6. Remaining Steps Process

The process described so far only covers a single time step x_t. In a complete sequence, the same operation is repeated for each subsequent time step, where:

$$h_t = \text{GRUCell}(x_t, h_{t-1}),$$

and the output is:

$$\hat{y}_t = W_{hy} h_t + b_{hy}.$$

This continues for all time steps $t = 1, 2, \ldots, T$, where T is the length of the sequence.

7. Calculating the Loss

Once predictions for all time steps $\{\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_T\}$ have been made, the next step is to evaluate the model's performance using a loss function. The goal is to quantify the difference between the predicted outputs $\hat{y}_t$ and the true outputs y_t.

The total loss for a sequence of length T is computed as:

$$L = \frac{1}{T} \sum_{t=1}^{T} \text{Loss}(y_t, \hat{y}_t),$$

where the loss function Loss depends on the task:

- **For classification tasks:** Cross-entropy loss is commonly used:

$$\text{Loss}(y_t, \hat{y}_t) = -\sum_{i=1}^{C} y_t[i] \log(\hat{y}_t[i]),$$

  where C is the number of classes.

- **For regression tasks:** Mean squared error (MSE) is a typical choice:

$$\text{Loss}(y_t, \hat{y}_t) = \frac{1}{m} \sum_{i=1}^{m} (y_t[i] - \hat{y}_t[i])^2,$$

  where m is the dimensionality of the output.

After calculating the total loss L, it serves as the basis for computing gradients during back-propagation. These gradients are then used to update the model parameters, as discussed in the next section.

8. Backpropagation Through Time (BPTT) and Optimizer

Backpropagation Through Time (BPTT) extends the standard backpropagation algorithm to sequential data. It involves the following steps:

- **Forward Pass**: Compute predictions $\hat{y}_t$ for each time step t using the input sequence $\{x_1, x_2, \ldots, x_T\}$ and hidden states.

- **Loss Calculation**: Aggregate the losses across all time steps:

$$L = \frac{1}{T} \sum_{t=1}^{T} \text{Loss}(y_t, \hat{y}_t).$$

- **Backward Pass**: Compute gradients for all parameters ($W_z, U_z, W_r, U_r, W_h, U_h, b_z, b_r, b_h$) by propagating errors backward through time.
- **Gradient Clipping**: Apply gradient clipping if necessary to prevent exploding gradients.

Optimizers

1. Stochastic Gradient Descent (SGD)

SGD updates the model parameters as:

$$\theta \leftarrow \theta - \eta \cdot \nabla_\theta L,$$

where:

- where θ represents model parameters
- η: learning rate (step size),
- $\nabla_\theta L$: gradient of the loss function L with respect to θ.

Key Characteristics:

- **Mini-Batch Based:** Instead of computing gradients on the entire dataset, SGD uses a small subset (mini-batch) to approximate the gradient, which is faster and memory-efficient.
- **Learning Rate Sensitivity:** The learning rate η must be carefully tuned to avoid divergence or slow convergence.
- **High Variance in Updates:** Gradients can fluctuate due to noisy mini-batches, which can also help escape local minima.

Advantages:

- Simple and easy to implement.
- Scales well with large datasets.
- Escapes shallow local minima due to noisy updates.

Disadvantages:

- Can be slow to converge near optima.

- Requires careful tuning of the learning rate.
- Does not adapt the learning rate for different parameters.

2. Adam Optimizer

Adam (short for Adaptive Moment Estimation) combines the benefits of **Momentum** and **RMSProp**, adapting learning rates dynamically for each parameter during training.

Update Rule: Adam maintains two moving averages for each parameter:

1. **First Moment Estimate (Mean):**

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla_\theta L$$

   where β_1 is the decay rate for the mean (e.g., 0.9).

2. **Second Moment Estimate (Variance):**

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla_\theta L)^2$$

   where β_2 is the decay rate for the variance (e.g., 0.999).

3. **Bias Correction:**

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

   to ensure unbiased estimates during initial steps.

4. **Parameter Update:**

$$\theta \leftarrow \theta - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

   where ϵ is a small constant to prevent division by zero.

Key Characteristics:

- Adaptive learning rates for each parameter.
- Incorporates momentum to smooth updates and speed up convergence.
- Bias correction ensures unbiased estimates early in training.

Advantages:

- Performs well with sparse gradients and high-dimensional data.
- Requires less tuning of the learning rate.
- Faster convergence in practice for many tasks.

Disadvantages:

- Can overfit and generalize poorly on some tasks.
- Consumes more memory compared to SGD.

3. Comparison of SGD and Adam

Aspect	**SGD**	**Adam**
Learning Rate	Fixed (requires tuning)	Adaptive (adjusted dynamically)
Noise Handling	High variance in updates	Reduced noise due to momentum
Convergence Speed	Slower near optima	Generally faster in practice
Memory Usage	Low (only stores gradients)	Higher (stores moments for all parameters)
Ease of Use	Requires careful tuning	Less sensitive to learning rate
Generalization	Better for simple models	Risk of overfitting

Table 1.1: Comparison of SGD and Adam Optimizers

Chapter 2

Introduction

2.1 Technological Evolution and Current Challenges

As music streaming platforms grow quickly, users face a huge amount of music every day, which makes it essential to have good recommendation systems to help them find songs they'll enjoy. Traditional recommendation systems often use collaborative filtering or content-based filtering.

Collaborative Filtering approach recommends items to users based on the preferences of other users with similar behaviors. For example, if User A and User B have overlapping music preferences, a song liked by User B but unknown to User A might be recommended to User A.

Content-Based Filtering approach uses item-specific attributes, such as song tempo, genre, or mood, to recommend similar items based on what a user has previously enjoyed. For instance, if a user listens to songs categorized as "upbeat," the system may suggest other tracks with similar attributes.

While these methods work well, they are not very flexible when it comes to adapting to changes in a user's interests. Music tastes can change based on context, mood, or trends, and these systems sometimes struggle to keep up with these changes.

To address the limitations of traditional approaches, deep learning has emerged as a powerful tool for recommendation systems, particularly in applications where user preferences evolve over time. Unlike collaborative filtering or content-

based filtering, deep learning models excel at capturing complex patterns and dynamic changes in user behavior.

This project investigates how well temporal dependencies can be modeled to reflect evolving user behaviors while maintaining computational efficiency. The following sections will delve into the data and features that form the foundation of this exploration and evaluate the effectiveness of the proposed approach.

2.2 Source of data

This dataset contains tuples of user, timestamp, artist, and song data collected from the Last.fm API using the user.getRecentTracks() method. It captures the complete listening habits of nearly 1,000 users.

The dataset comprises two files. The first, "userid-profile.tsv," includes 1,000 users and their basic information, such as gender, nationality, and registration date.

The second file, "userid-timestamp-artid-artname-traid-traname.tsv" contains six columns: user ID, timestamp, artist ID, artist name, track ID, and track name.

This second file represents the full listening history of users from user_001 to user_1000. For this project, I have chosen user_574 from Poland as the primary subject for analysis. This user's data spans from May 2008 to May 2009, with approximately nine thousand records—an ideal dataset size for model training. Most track names are in English, with only a few exceptions. These non-English tracks were removed to simplify the model's understanding of the data without significantly affecting its integrity.

2.3 Features of data

Although there are six columns in dataset, we still need extra features to fit in models. In the original dataset, we keep timestamp; track name and artist name. Additionally, by using Spotify API to search online, we could get audio features of individual songs. For each song, these features are following below:

- danceability: Represents how suitable a track is for dancing, based on tempo, rhythm stability, and beat strength. The range is from 0 to 1,

where higher values indicate a more danceable track.

- energy: Indicates the intensity or activity level of a track. The range is from 0 to 1, where higher values signify a more energetic and intense song. High-energy tracks often have a fast tempo and loudness.

- key: Represents the key in which the track is composed. The range is from 0 to 11, corresponding to the 12 musical keys ($e.g., 0 = C, 1 = C$ or Db, etc.). This is a discrete feature.

- loudness: Represents the overall loudness of a track in decibels (dB). The range typically falls between -60 and 0 dB, with higher values indicating a louder track.

- mode: Indicates the modality (scale) of a track, with 0 for minor and 1 for major. Major mode often sounds brighter, while minor mode is typically more subdued or somber.

- speechiness: Shows the presence of spoken words in a track. The range is from 0 to 1, where higher values indicate more spoken content. Tracks with high speechiness values usually have more lyrics or spoken words.

- acousticness: Measures the acoustic quality of a track. The range is from 0 to 1, with higher values indicating more acoustic sound (less electronic).

- instrumentalness: Measures the degree to which a track is instrumental (without vocals). The range is from 0 to 1, where values closer to 1 suggest a higher likelihood of the track being instrumental.

- liveness: Indicates the probability that the track was recorded live. The range is from 0 to 1, with higher values suggesting a greater likelihood of a live recording, often including audience sounds.

- valence: Describes the emotional positivity of a track. The range is from 0 to 1, where higher values indicate a more positive or cheerful mood, and lower values indicate a more negative or subdued mood.

- tempo: Represents the speed or pace of the track in beats per minute (BPM). The typical range is from 50 to 200 BPM. Faster tempos often feel more energetic, while slower tempos can create a more relaxed atmosphere.

Why These Features Were Chosen?

The features selected for this project aim to comprehensively capture the atmosphere, style, and emotional tone of each song, which are key factors in understanding user preferences. Traditional approaches often rely on categorical labels, such as genre or mood, to describe songs. While these labels can be useful, they are often too broad and fail to capture the nuanced differences between tracks. Moreover, the process of categorizing songs into discrete labels can lead to significant information loss, as many songs do not fit neatly into a single category.

To overcome this limitation, we chose to use numerical values for song attributes. Features like danceability, energy, and valence provide detailed and continuous representations of a song's characteristics. For example:

- Instead of broadly categorizing a song as "energetic," the energy feature quantifies this attribute on a scale from 0 to 1, allowing the model to understand subtle variations.
- Similarly, valence does not simply label a song as "happy" or "sad" but provides a gradient of emotional positivity, enabling more precise predictions.

By retaining these numerical features, the dataset avoids the potential loss of granularity that often occurs during data cleaning or preprocessing. This approach ensures that the recommendation system has access to as much detailed information as possible, enhancing its ability to make nuanced predictions.

Moreover, using numerical features aligns seamlessly with the requirements of time-series prediction. In sequential models, where the input consists of a series of data points over time, continuous values enable the system to identify trends and shifts in user behavior with greater precision. For instance, a user's preference for songs with increasing danceability over a specific period can be effectively captured and analyzed, providing insights that categorical labels might fail to convey.

These features were chosen not only for their relevance in describing songs but also for their ability to preserve detailed information and align with the predictive needs of time-series modeling.

2.4 Data Cleaning

After collecting the song list from the dataset, the first step involved extracting additional features for each track using the Spotify API. The API provided key audio attributes such as danceability, energy, and tempo, which are critical for building a personalized recommendation system. However, not all songs in the dataset had complete information available via the API. This necessitated an initial filtering step to identify and exclude tracks with missing features, ensuring that the dataset remained consistent and reliable.

The second step in the data cleaning process focused on removing tracks that were not encoded in English. This decision was made to simplify the data and avoid potential issues arising from multilingual or incorrectly encoded text. For example, tracks with titles in languages such as Chinese or Japanese were excluded to ensure uniformity and avoid complications during feature analysis and processing. While this reduced the overall size of the dataset, it preserved the integrity of the remaining data, which predominantly consisted of English song titles.

After these initial steps, all numerical features were standardized to ensure they were on comparable scales. Standardization was performed using z-score normalization, where each feature was transformed to have a mean of 0 and a standard deviation of 1. By standardizing globally, the dataset retains its long-term temporal relationships, enabling the model to better capture evolving user preferences over time. Month-wise standardization, while potentially useful for some tasks, could introduce inconsistencies in feature distributions between months, disrupting the temporal dependencies that are critical for sequential modeling. Furthermore, it avoids potential information leakage during testing, where future data within a specific month could inadvertently influence the computed scaling parameters.

At the end of the data cleaning process, the dataset was left with a comprehensive set of tracks, each represented by a uniform set of audio features.

Chapter 3

Experiment

3.1 GRU methods: Multi-step prediction

This method is aiming to consume **ten** data points to recommend **five** future songs.

As mentioned previously, this project draws inspiration from time-series forecasting. The primary objective is to predict which songs a user would likely listen to in the near future, using a combination of numerical and categorical features. The general approach is relatively straightforward: we utilize historical data to predict future audio-related features, and subsequently, identify the most compatible songs based on these predictions mathematically. Once identified, the recommended songs are presented to the user.

Setting Hyper parameters

Before implementing the model, it is essential to define the hyperparameters for the initial setup. The selected hyperparameters and their explanations are as follows:

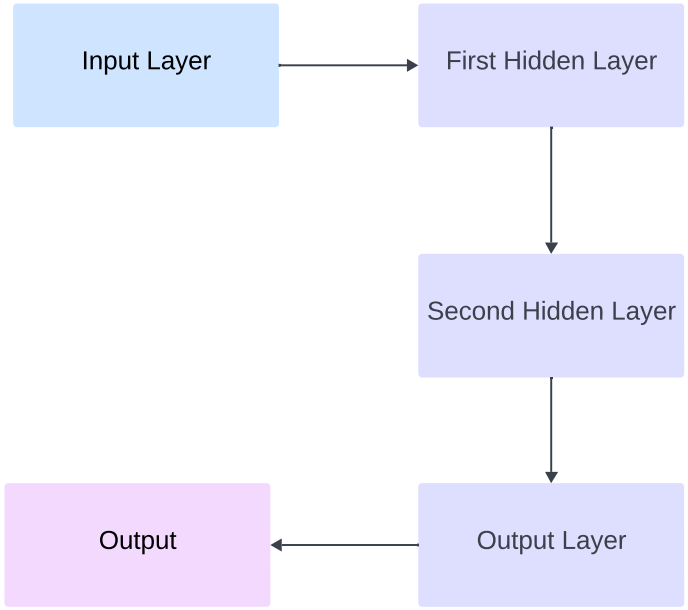
1. Hidden Layers =2
2. Each layer contains 512 neurons (Defines the size of the hidden state for each GRU layer.)

3. Input size = 11 = len(feature columns) (The size of the input vector for each time step.)
4. Output size = 11 = len(feature columns) (stick with out object, future audio features)
5. Sequence length = 10 (The number of historical time steps used as input to predict future steps.)
6. Prediction steps = 5 (The number of times the model iterates over the entire training dataset.)
7. Batch size = 1 (Defines how many sequences are fed into the model at once. Smaller values help preserve temporal order but may slow down training.)
8. Epochs = 5 (The number of times the model iterates over the entire training dataset.)
9. Criterion = nn.MSELoss() (Specifies the loss function used to calculate the error.)
10. Learning rate = 0.01 (Determines the step size for the optimizer during gradient descent.)

Layer Structure

1. **Input Layer**:
   - The data flows into the network from the input layer.
2. **First Hidden Layer**:
   - The input data is processed by the first GRU layer, using the update gate (z_t) and reset gate (r_t), to generate the hidden state h_{t1}. The result is then passed to the second hidden layer.
3. **Second Hidden Layer**:
   - The hidden state h_{t1} from the first hidden layer is processed by the second GRU layer to produce a new hidden state h_{t2}, which is then passed to the output layer.

4. **Output Layer**:
    - The hidden state h_{t2} is combined with the weight matrix to generate the final output.



Dynamic Training and Testing Dataset

- **Composition:**
    - **First Month:**
        - Only 80% of the current month's data is used as the training dataset.
    - **From the Second Month Onward:**
        - 80% of the current month's data (`current_train`).
        - 20% of the previous month's testing data, sampled at 50% (`previous_test`).
        - 80% of the previous month's training data, sampled at 20% (`previous_train`).
- These datasets are dynamically combined to form the dynamic training set (`dynamic_train`), which includes both current and historical data.

Training and Testing methods

When we train model, we feed 10 data points into the model at the same time, and we expect 5 audio features would be the outcome. The input is a matrix with shape of [10,11]. At the first time, input matrix has the expression:

$$\mathbf{X_1} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{10} \end{bmatrix}$$

where each x_t has form of :

$$x_t = [x_{t1}, x_{t2}, x_{t3}.....x_{t10}, x_{t11}]$$

Although the model takes a matrix as input and appears to process the entire matrix at once, it actually processes each vector sequentially in accordance because of the RNN structure.

In the first chapter, when we introduce one layer GRU model, we have the following:

1. Take input vector x_1
2. Generate z_1 and r_1
3. Use r_1 to calculate $\hat{h_1}$ as candidate hidden state
4. Use $\hat{h_1}$ to determine final hidden state for this step h_1

Once we have final hidden state h_t, it is our decision to make to whether we want to proceed output $\hat{y_1} = Wh_1 + b$ or we utilize h_1 in this recursive process until we reach h_{10}. In our training process, we have the following order:

1. Generate h_1 for first step and store h_1
2. Take second input vector x_2 and combine with h_1 to create final hidden state h_2
3. We do this recursively until we have h_{10} because we set our sequence of length to be ten

4. Since we want to predict five songs at one, therefore pre-defined weighted matrix $W_h y$ and hidden state will have matrix multiplication and through reshape process to make a [5,11] shape of matrix as output. Each row represents a predicted audio features.

That is for single layer hidden process. In our project, after multiple versions updated, we increase the ability of the model by making hidden layer become 2. The first layer logic remains the same as usual, once we receive the hidden state, we enter the second layer:

1. Receive hidden layer h_1 denote as h_{11} from last layer
2. Compute second layer z_1 and r_1 denote as $z_{2,1}$ and $r_{2,1}$
3. Use $r_{2,1}$ to calculate $\hat{h_{2,1}}$ as candidate hidden state
4. Use $\hat{h_{2,1}}$ to determine final hidden state for this step $h_{2,1}$
5. Repeat until we get $h_{2,10}$
6. Make prediction $\hat{Y} = W_{hy2} h_{2,10} + b$

Notice that $\hat{Y}$ has shape of [5,11]. Sometimes, due to dimension issue, $\hat{Y}$ may have shape of [1,55], using reshape method can form a desired expression.

The true value of Y and predicted value $\hat{Y}$ at the first cycle is defined as the next following five songs:

$$\mathbf{Y_1} = \begin{bmatrix} x_{11} \\ x_{12} \\ x_{13} \\ x_{14} \\ x_{15} \end{bmatrix}, \mathbf{\hat{Y}_1} = \begin{bmatrix} \hat{y_{11}} \\ \hat{y_{12}} \\ \hat{y_{13}} \\ \hat{y_{14}} \\ \hat{y_{15}} \end{bmatrix}$$

By using MSE as loss function to calculate error and update the parameters. The next input matrix and true value of Y has form of :

$$\mathbf{X_2} = \begin{bmatrix} x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{11} \end{bmatrix}, \mathbf{Y_2} = \begin{bmatrix} x_{12} \\ x_{13} \\ x_{14} \\ x_{15} \\ x_{16} \end{bmatrix}$$

Model will repeat the process until the rest data has less than ten

Testing test for prediction

Testing set would have the same procedure but fixed parameters, not changes to these weighted matrices and bais vectors

Design and Construction of the Candidate Set

The candidate set is a critical component of the recommendation system, serving as the pool of historical reference data against which the predicted results are matched to generate recommendations. The construction of the candidate set is based on a dynamic sliding window approach over the `current_test` data, with the following logic:

- The candidate set is composed of a prefix of the test data, with its size dynamically adjusted and capped by a parameter `Bar` (e.g., a maximum of 20 records).
- In the time series prediction framework, the candidate set evolves dynamically as the test data progresses:
  - When the test data is sparse at the current step (e.g., $i < \texttt{Bar} - 10$), the candidate set includes only the 10 most recent records.
  - As the test data accumulates, the candidate set expands up to the maximum window size, encompassing the 20 most recent records.
- For multi-step predictions (e.g., predicting the next 5 steps at once), the predicted features at each step are compared against all entries in the candidate set. The most similar records in the candidate set are identified and used to generate the recommendation list.
- The method of calculating distance is used as "euclidean distance"

Results and Hitting Rate

Demonstration of results is primarily focusing on Testing set. For counting the hitting rate, we consider every test as a sample, in other words, if we have 100 data in testing set with sequence of length = 10 and prediction steps = 5,

then we would generate 86 samples which contains five songs for each sample. Formula has shown below:

$$N = \text{total data - sequence length - prediction steps } +1$$

Rule: If one of songs in predicted samples has been found in the true list, then we count this sample as successful.

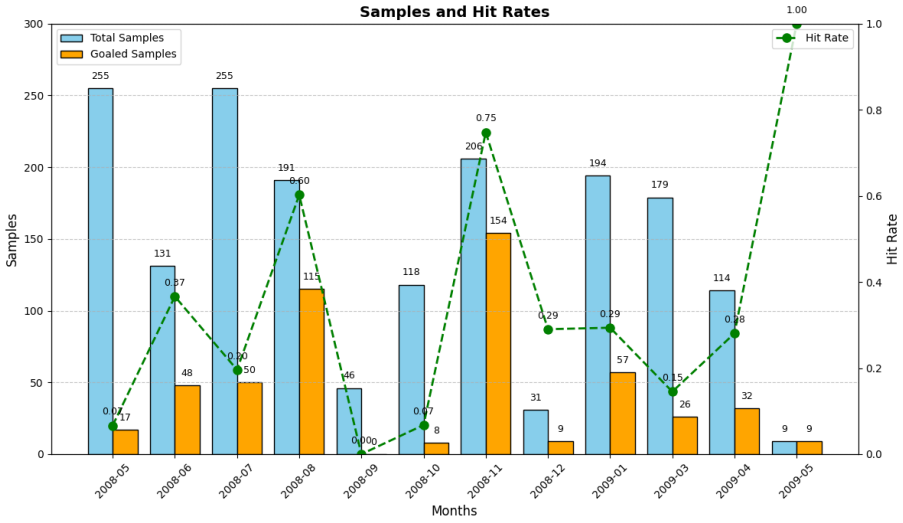
Since we consider one song could be listened multiple times. Therefore, we allow prediction list has replicates. For demonstration propose, we eliminate all replicates. An example has shown below:

```
For 2008-06 : successful samples are 22 out of 50

--- Summary of 50 sample ---
Actual songs are:
Infinity 2008 by Guru Josh Project
Rise Up by Yves Larock
Shine On by R.I.O.
Infinity 2008 by Guru Josh Project
Rise Up by Yves Larock


Predicted songs are:
Rise Up by Yves Larock
Cemeteries Of London by Coldplay
42 by Coldplay
For 2008-06 : successful samples are 23 out of 51
```

The result for each month is shown below by the chart



1. Goaled Samples (Orange Bars)

- Goaled samples represent predictions where the actual songs are successfully matched in the recommendations.
- A significant number of goaled samples can be observed in months like **2008-06**, **2008-08**, and **2008-11**(37%,60%,75%).
- Some months, like **2008-09**, show almost zero goaled samples, indicating poor model performance or a lack of matching between predicted and actual songs.

2. Hit Rates (Green Line with Dots)

- The hit rate is the proportion of goaled samples relative to the total samples.
- Months like **2009-05** show a **100% hit rate**, but the total samples for this month are exceptionally low (only 9), which skews the percentage.
- **2008-11** has a high hit rate (0.75) with a substantial number of samples (206 total, 154 goaled), suggesting strong model performance during that month.

- **2008-09** and **2008-10** have hit rates close to zero, indicating that the model struggled during these months.

In the previous multi-step prediction approach, we adopted training and testing logic: using 10 consecutive data points as input to predict the subsequent 5 points at once. This rolling-window strategy allowed us to effectively capture local temporal patterns, enabling notably high hitting rates in certain periods with distinct features, such as November 2008. However, the method proved less stable for other months, with some periods exhibiting near-zero hitting rates. This limitation demonstrates that while a range input-length, multi-step prediction can yield significant local successes, it may struggle to maintain consistent performance across the entire timeline. To address these issues, in the following method we experiment with a more flexible single-step prediction strategy on the test set, combined with improved distance measures and recommendation logic, aiming to achieve a more balanced and adaptable performance over different time segments.

3.2 GRU methods: Single-step prediction

Training and Testing methods

For the training phase, we continue to employ the same multi-step approach as before, utilizing a fixed-length input (e.g., 10 data points) to predict multiple subsequent steps (e.g., 5 data points). However, once we move into the testing phase, we depart from this multi-step prediction strategy. Instead, we adopt a single-step prediction approach, we still use a fixed length input to train model, however we are not going to generate five sequences at a time, we will focus on one single sequence. The input matrix has shown below:

$$\mathbf{X_1} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{10} \end{bmatrix}$$

where each x_t has form of :

$$x_t = [x_{t1}, x_{t2}, x_{t3}.....x_{t10}, x_{t11}]$$

The procedure is similar:

1. Take input vector x_1
2. follow the previous steps until we get $h_{2,10}$
3. Make prediction $\hat{Y} = W_{hy2}h_{2,10} + b$

Notice that $\hat{Y}$ has shape of [1,11] this time. The true value of Y and predicted value $\hat{Y}$ at the first cycle is defined as the next following **One** song:

$$\mathbf{Y_1} = \left[x_{11}\right], \mathbf{\hat{Y}_1} = \left[\hat{y_{11}}\right]$$

We updated our loss function to be **SmoothL1Loss**:

$$\text{SmoothL1Loss}(x, y) = \frac{1}{n} \sum_{i=1}^{n} z_i,$$

$$z_i = \begin{cases} \frac{1}{2}(x_i - y_i)^2 & \text{if } |x_i - y_i| < 1 \\ |x_i - y_i| - \frac{1}{2} & \text{otherwise} \end{cases}$$

x_i denotes the predicted value and y_i the ground truth. When the absolute error $|x_i - y_i|$ is small (less than 1), the loss is quadratic, behaving similarly to Mean Squared Error and ensuring a smooth gradient near zero error. When the error becomes larger, the loss grows linearly, preventing overly large gradients caused by outliers, as would occur with Mean Squared Error.Smooth L1 Loss remains applicable even after data standardization. The purpose of standardization is to scale different features and target variables into a similar range, which typically contributes to more stable model training. The definition of Smooth L1 Loss itself does not depend on the original magnitude of the data, but rather on whether the error exceeds a certain threshold (by default in package) to determine whether to use a quadratic or linear form. After standardization, the input and output of the model are generally on a comparable scale, which means that the threshold of Smooth L1 loss aligns with the magnitude of the standardized error. This helps maintain stable training and robustness against outliers.

As what we previously have done, we update our parameters for every **ten** data points

Testing test for prediction

Now, this step is complete different, for multiple steps we predict **five** songs for every **ten** songs. For single step prediction, we only generate **one** song by **one** input.

In other words: $x_1 \longrightarrow \hat{y_1}(x_2)$. Once we have predicted the next song: we trigger recommendation system: still **five** songs but all based on this single prediction.

To compute the similarity between the predicted vector $\hat{y}_1$ and the candidate vectors, we utilize a normalized dot product approach:

- **Dot Product Similarity:** The normalized vectors are compared using the dot product:

$$\text{Similarity}(\hat{y}_1, \mathbf{c}_i) = \hat{y}_1^{\text{norm}} \cdot \mathbf{c}_i^{\text{norm}}$$

  This method captures the directional alignment of the vectors, providing a robust measure of similarity between the predicted vector and each candidate vector.

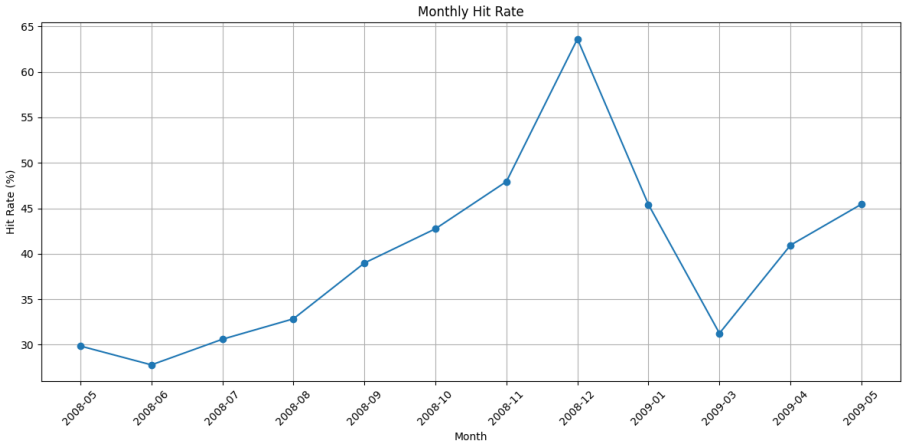
The similarity scores between $\hat{y}_1$ and all candidate vectors are ranked in descending order. To ensure diversity in recommendations, previously recommended songs are excluded. The top five candidates based on similarity are selected as the recommendation list:

$$\text{Recommendation List} = \{\mathbf{c}_{\text{top-1}}, \mathbf{c}_{\text{top-2}}, \ldots, \mathbf{c}_{\text{top-5}}\}$$

This approach ensures that the recommendation system balances granularity in single-step prediction with diversity and relevance. By iterating this step, the system builds a sequential recommendation pipeline that dynamically adapts to the predicted output at each step.

```
Sample 13:
Actual: That'S The Way It Is - Céline Dion
Recommendations:
1. Who'S Your Daddy? - Daddy Yankee
2. Can'T Stop The Rain - Cascada
3. Champion Sound - Fatboy Slim
4. That'S The Way It Is - Céline Dion
5. Trouble - Lindsey Buckingham
```

Results and Hitting Rate



The results of single-step prediction demonstrate higher stability, with hit rates maintained within the range of 30%-50% across most months. This stability indicates that the single-step prediction method can adapt well to variations in different months, providing a reliable baseline hit rate regardless of whether the data patterns are clear or not. In December 2008, the hit rate peaked at 65%, possibly because the data pattern during this month was more concentrated or the features were more distinct, allowing the model to capture short-term contextual relationships more effectively. Even in poorer-performing months, such as January 2009, the hit rate remained around 30%, avoiding extremely low performance.

This stability stems from the single-step prediction method's focus on short-term context, predicting one step at a time and using the result of each step to trigger recommendations. This approach minimizes over-reliance on long-term context. Additionally, the step-by-step prediction logic enables the recommendation system to dynamically adjust its candidate set, enhancing overall adaptability. The single-step prediction method is particularly suitable for scenarios with high data variability or unstable features, offering a robust and flexible solution.