Implementation

In this section, you will test different types of recurrent neural networks in order to implement the task of speech recognition. Collection

The data of this exercise, which contains a number of audio files along with the text corresponding to each one, can be accessed in this link.

First, do the necessary pre-processing to prepare the data set as input for the model.

In the next step, consider three models: LSTM, RNN and GRU. The architecture of these models and the number of layers of each one is arbitrary, but note that the structure of the models should be in such a way that they can be compared with each other. including that the activity function, loss function and optimizer are the same in the three models.

In order to evaluate the models, use loss, accuracy and f1-score criteria and draw the learning graph of each model in terms of accuracy. In the last step, after evaluating the models by analyzing the results and mentioning the reason, announce the best model and report some examples of the best network outputs that include the predicted texts for each audio file in the evaluation data.

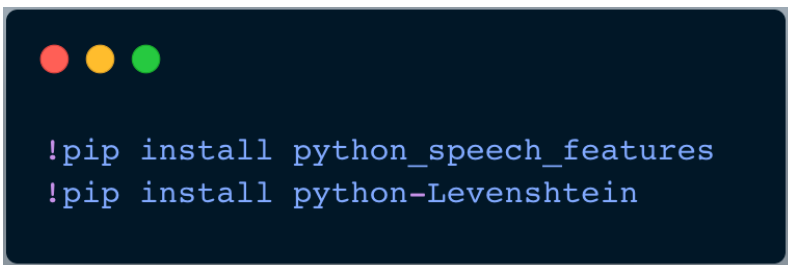
Introduction

In this section, we will test different types of recurrent neural networks in order to implement the problem of speech recognition. Speech recognition is one of the important issues in the field of speech processing, which involves converting the audio signal into text. For this purpose, to implement this problem, we will use the data set that contains a number of audio files along with the text corresponding to each one.

First, in order to prepare the dataset, we will perform the necessary pre-processing. In the next step, we will consider three models: LSTM, RNN and GRU.

In order to evaluate the models, we will use loss, accuracy and f1-score criteria and two other criteria, word error rate and sentence error rate, and we will draw the corresponding graphs. In the last step, after evaluating the models by analyzing the results and mentioning the reason, we will report the best model.

Installing packages and requirements



The ``python_speech_features'' package is a Python library that provides functionality for working with speech signal processing, such as extracting features from audio signals.

The ``python-Levenshtein'' package provides the Levenshtein distance algorithm. This algorithm is used to calculate the difference between two strings and can be useful for tasks such as spelling correction or fuzzy string matching.

```
import pandas as pd  # library for data manipulation and analysis
import os  # module for interacting with the operating system
import shutil  # module for high-level file operations
import numpy as np  # library for numerical computing
import scipy.io.wavfile as wav  # module for reading and writing WAV files
import matplotlib.pyplot as plt  # library for plotting data
from tqdm import tqdm_notebook as tqdm  # module for creating progress bars

# deep learning libraries
import torch
from torch import nn
from torch.utils.data import Dataset, DataLoader
from torch.nn.utils.rnn import pad_sequence
from torch.utils.tensorboard import SummaryWriter
from tensorflow.keras.layers import LSTM, Dense, Masking

# machine learning libraries
from sklearn.model_selection import train_test_split  # module for splitting data into train and test
sets
from sklearn.metrics import f1_score  # function for calculating F1 score

# audio signal processing libraries
from python_speech_features import mfcc  # module for computing Mel Frequency Cepstral Coefficients
(MFCCs)
from IPython.display import Audio, display  # module for playing audio files
```

Libraries are divided into five main groups:

Data manipulation and analysis: Pandas library is used for data manipulation and analysis.

Operating system and file operations: The os and shutil modules are used to interact with the operating system and perform high-level file operations.

Libraries for numerical calculation and plotting: numpy and matplotlib.pyplot are used for numerical calculations and data plotting.

Deep learning libraries: torch and tensorflow.keras.layers are used to build and train deep learning models.

Audio signal processing libraries: python_speech_features is used to calculate MFCC frequency coefficients and IPython.display to play audio files.

```
df = pd.read_excel('/content/drive/MyDrive/Deep_HW4/Persian-Speech-To-Text-Maps.xlsx')

file_name = '/content/drive/MyDrive/Deep_HW4/'
file_name += os.listdir(file_name)[0]

(rate,sig) = wav.read(file_name)
```

This code first receives the xlsx file that contains labels and audio text. Next, it uses the "scipy.io.wavfile.read()" function to read audio data from the file specified by "file_name". This function returns two values: the audio sampling rate ("rate") and the audio signal itself ("signal")

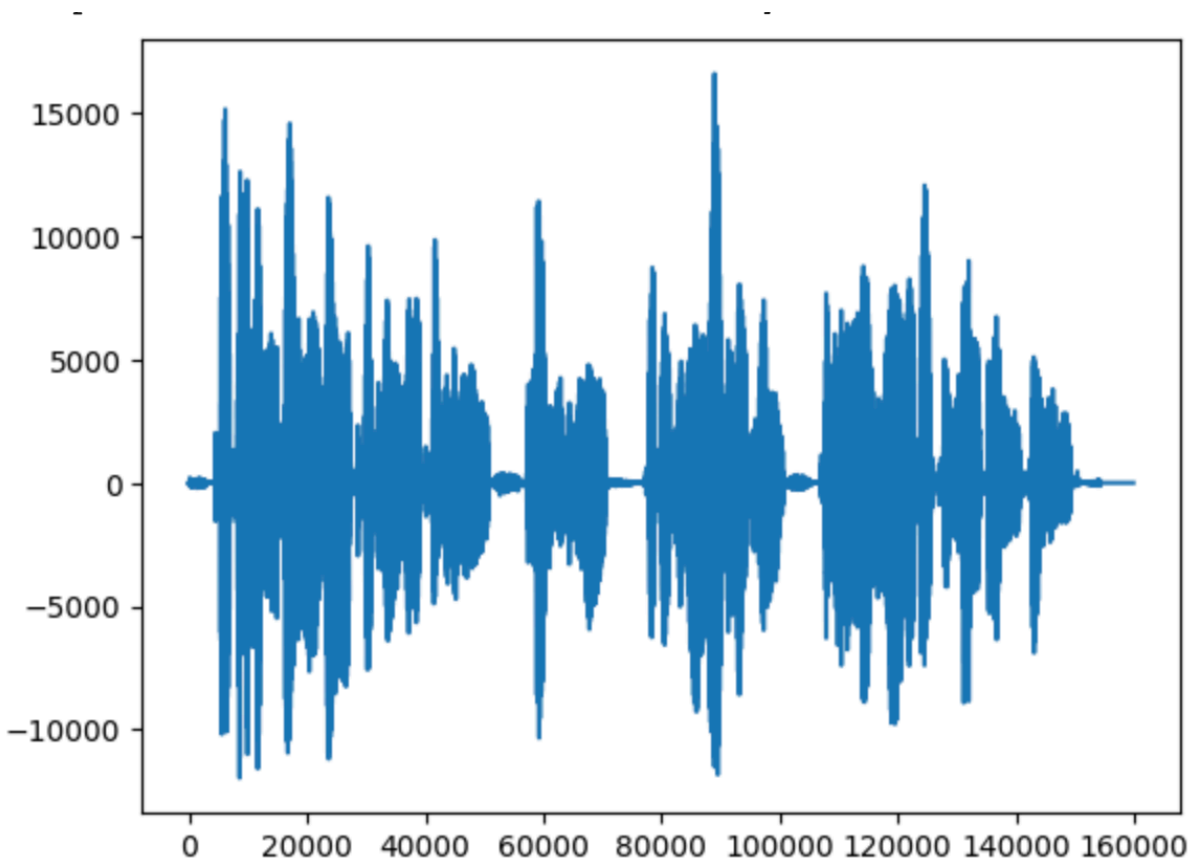
These variables store the audio signal data and sample rate respectively from a WAV file.

An audio signal is a continuous analog signal that is captured and stored as a sequence of digital values. In a WAV file, each of these digital values is represented by a 16-bit or 24-bit integer. The `sig' variable contains these digital values in the form of a one-dimensional numpy array.

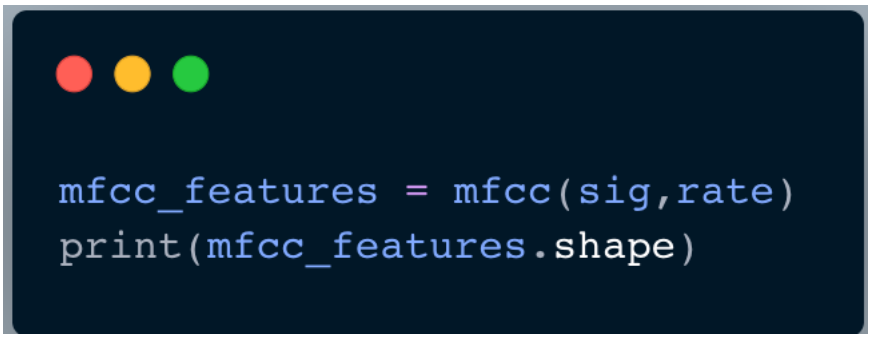
The sampling rate, "rate", is also the number of samples per second taken from the analog audio signal during the recording process and stored in the WAV file. It is usually measured in Hertz (Hz) and is typically in the range of 8000 to 48000 Hz for most audio applications. Sampling rate is an important parameter for audio signal processing because it determines the frequency range that can be accurately represented in the digital domain.

In short, "sig" and "rate" are two important pieces of information that are required to display and process digital audio signals stored in a WAV file.

Next, we draw the audio signal.



Next, we need to convert the audio data into features that can be injected into the network.



This code uses the function "python_speech_features.mfcc()" to calculate the MFCC of the audio signal "sig". MFCCs are commonly used to analyze and model audio signals in speech and music processing.

MFCC stands for Mel Frequency Cepstral Coefficients and is one of the branches of audio signal processing in the field of machine learning and speech processing. MFCC is a type of audio signal features used to identify and distinguish between words or other speech units.

MFCC consists of three stages of audio signal processing:

.1Signal preprocessing: In this step, the audio signal is divided into several short time frames to extract the temporal features in the signal. Each frame is inverted with a Hamming-like window and then converted to the frequency domain using Discrete Fourier Transform (DFT.(

.2Mel-scale conversion: In this step, audio signal frequencies are converted to Mel scale. The Mel scale is based on the sensory response of the human ear to different frequencies. By converting frequencies to the Mel scale, features such as the frequency bands of the human ear can be simulated.

.3Calculation of cepstral coefficients: In this step, through discrete cosine transformation, cepstral coefficients are calculated. These coefficients represent the second transformation of the logarithm of the audio signal energy and show the spectral characteristics of the signal.

MFCC is used for speech signal processing in many speech processing and speech recognition applications. For example, in speech recognition applications, audio signal recognition, audio wave analysis, and many other speech processing applications, MFCC is used as a feature.

The "mfcc" function takes two arguments: the audio signal "sig" and the sampling rate "rate". MFCCs are calculated by first dividing the audio signal into short overlapping frames, typically 20–30 ms in duration, and then applying a series of signal processing

steps to each frame. These steps usually include windowing, Fourier transform and Mel frequency filtering to extract the spectral features of the signal. Finally, the brain coefficients are obtained by applying the discrete cosine transformation to the logarithm of Mel filter bank energies.

MFCCs are returned as a trivial array of the form "(num_frames, num_cepstral_coeffs)", where "num_cepstral_coeffs" is the number of frames in the signal and "num_cepstral_coeffs" is the number of cepstral coefficients calculated for each frame. The shape of the MFCC array is printed using the "print()" function on the next line.

Briefly, this code calculates the MFCCs of the audio signal "sig" using the function "python_speech_features.mfcc()" and stores the result in the variable "mfcc_features"

.

```
for i in range(500):
    text = df.loc[i, 'audio']
    loc = text.replace("myaudio/", "")
    file_name = '/content/drive/MyDrive/Deep_HW4/' + loc
    (rate,sig) = wav.read(file_name)
    mfcc_features = mfcc(sig,rate)
    df.loc[i, 'MFCC features'] = [mfcc_features]
```

This code does the following for each row:

1. Extracts the value of the 'audio' column of the current row using the '.loc()' method of the DataFrame and stores it in the 'text' variable.
2. Removes the prefix "myaudio/" from the "text" value using the "replace()" method and stores the result in the "loc" variable.
3. Creates the full path of the WAV file corresponding to the current row by concatenating the string '/content/drive/MyDrive/Deep_HW4/' with the value 'loc'. The obtained file path is stored in the "file_name" variable.
4. Reads the audio data from the WAV file specified by "file_name" using the "wav.read()" function of the "scipy.io.wavfile" module. The sampling rate and the audio signal are stored in the variables 'rate' and 'sig' respectively.
5. Computes the MFCC features of the audio signal using the “python_speech_features.mfcc()” function from the “python_speech_features” module. The resulting MFCC features are stored in the “mfcc_features” variable.

6. Stores the calculated MFCC features in the "MFCC Features" column of the current row of the DataFrame "df" using the ".loc()" method.

The end result of this code is to calculate and store the MFCC features of the specified audio signals in the first 500 rows of DataFrame 'df'. These features can be used for further analysis or as input to a machine learning model for tasks such as speech recognition or voice classification.

```
unique_chars = pd.unique(list(df['text'].sum()))
unique_chars.sort()
```

This code first creates a list of all the unique characters that appear in the "text" column of the DataFrame. The code then sorts the unique characters alphabetically using the ``.sort()'' method so that they can be printed in a consistent order.

This type of code can be useful for exploring features of a text dataset, such as the set of characters that appear in the text and their frequency. It can also be used to preprocess text data by mapping each unique character to a unique integer index, which is often required to feed text data to a machine learning model.

```
# Define a dictionary of hard-to-read characters and their replacements
char_dict = {'ى' :'ئ' ,'ا' :'أ' ,'ا' :'آ', unique_chars[30]: 'ن'}

# Convert the keys and values in char_dict to Unicode code points
char_codepoints = {ord(k): ord(v) for k, v in char_dict.items()}

# Apply the character mapping to the 'text' column of the DataFrame
def translate_text(text):
    """Translates a string by replacing hard-to-read characters."""
    return text.translate(char_codepoints)

df['text'] = df['text'].apply(translate_text)
```

This code defines a dictionary called "char_dict" that maps hard-to-read characters to their substitutes. For example, the key "a" maps to the value "a", the key "a" maps to the value "a", and so on.

The code then converts the "char_dict" keys and values into Unicode code points using dictionary understanding. The resulting dictionary, called "char_codepoints", maps the

Unicode code point of each key character to the Unicode code point of its corresponding value character.

Finally, the code defines a function called "translate_text" that takes a string as input and uses the "translate()" method to replace each hard-to-read character in the string. The translate() method uses the char_codepoints dictionary to replace each key character with its corresponding value character.

The ``apply()'' method is then used to apply the ``translate_text'' function to the ``text'' column of the ``df'' DataFrame, effectively replacing each hard-read character with their replacements. The resulting modified "text" column is again stored in the DataFrame "df."

This type of code can be useful when dealing with text data that contains characters that are difficult to process or display correctly, such as non-ASCII characters in languages with non-Latin scripts. By defining a mapping from hard-to-read characters to their replacements and applying this mapping to textual data, we can make the work and understanding of the data easier.

The output of the data frame after applying pre-processing is as follows.

	audio	text	MFCC features
0	myaudio/12440123.wav	... ابنبات های تومک را گرفت گفت ممنون پسرم نه من	[[[11.53494370737191, -3.370294705789073, -4.2...
1	myaudio/12440124.wav	... قبلا هم به تو گفتم هیچ وقتم استراحت ندارم فقط	[[[10.105730436984143, -2.4352324174037996, -3...
2	myaudio/12440126.wav	...می شد او را یک فیلسوف به حساب اورد او عاشق چیز	[[[9.91634332069268, -2.5599461851402547, -4.9...
3	myaudio/12440127.wav	...اگر تومک فراموش می کرد برایش ابنبات بیاورد درس	[[[11.225249404021213, -3.406866944024388, -2....
4	myaudio/12440128.wav	... نان های ادویه دار قلبی شکل را بیشتر دوست داشت	[[[8.637159353566885, -2.7571319924894624, -5....
...	...	...	...
495	myaudio/12560439.wav	...و زیبای شهر تهران به حساب می اید این پارک در ح	[[[15.512500755155884, -34.664955861022364, -1...
496	myaudio/12560440.wav	...باغ پرندگان تهران در شمال شرق تهران و در دل جن	[[[13.953515756973522, -2.395566524010967, -14...
497	myaudio/12560443.wav	...و مخصوص پرندگانی ایست که قابلیت پرواز دارند پو	[[[12.28639023707803, -27.30243460225483, -21....
498	myaudio/12560444.wav	...جلوگیری می کند احداث این باغ سه سال به طول انج	[[[13.19515533481687, -30.72443819788632, -10....
499	myaudio/12560446.wav	...پردیس سینمایی ملت از بزرگترین مراکز سینمایی ته	[[[13.229105799634326, -8.476017093719932, -16...

500 rows × 3 columns

```
# Define a list of unique characters for the target language
target_chars = unique_chars.tolist()

# Add special characters for blank, padding, start, and end
target_chars = ['-', '<PAD>', '<S>', '<E>'] + target_chars

# Create dictionaries to map characters to integers and vice versa
char_to_int = {char: index for index, char in enumerate(target_chars)}
int_to_char = {index: char for index, char in enumerate(target_chars)}
```

This code defines a mapping between characters and integers for a target language. It is assumed that a list of unique characters for the target language has already been calculated and stored in the ``chars_unique'' variable.

The first step of the code converts the "unique_chars" list to a NumPy array using the ".tolist()" method. The resulting list is stored in the "target_chars" variable.

The second step of the code adds special characters for blank, padding, start and end to the "target_chars" list. These special characters are commonly used when processing textual data for machine learning tasks. The '-' character is often used as a blank character to indicate a space between words. '<PAD>' is used to add sequences of fixed length, '<S>' and '<E>' are used to mark the start and end of a sequence respectively.

The third step of the code creates two dictionaries named "char_to_int" and "int_to_char" to map characters to integers and vice versa. The "char_to_int" dictionary maps each character in "charget_chars" to its corresponding index in the list, while the "int_to_char" dictionary maps each index to its corresponding character in the list.

These dictionaries can be used to encode text data as a sequence of integers, which is often required to feed text data to a machine learning model. By mapping each character to a unique integer index, we can represent text data as a sequence of integers that can be easily processed by the model. Inverse mapping from integers to characters is also useful for decoding model output into human-readable text.

```
# Define a function to tokenize text using a pre-defined character-to-integer mapping
def tokenize_text(text):
    """Converts a string of text to a sequence of integers."""

    # Initialize the sequence with a start token
    sequence = [2]

    # Map each character in the text to its integer representation
    for char in text:
        char_int = char_to_int[char]
        sequence.append(char_int)

    # Add an end token to the sequence
    sequence.append(3)

    # Convert the sequence to a NumPy array and return it
    return [np.array(sequence)]
```

This code defines a function called "tokenize_text" that takes a string of text as input and converts it to a sequence of integers using a predefined character-to-integer mapping.

This function first initializes a "sequence" list with a starting token, represented by the integer value 2. This start token is added to the beginning of the sequence to indicate the start of a new text sequence.

This function then maps each character in the input text to the corresponding integer representation using the "char_to_int" dictionary. The resulting integer values are added to the "sequence" list.

After all characters have been mapped to integers, this function adds an end token to the "sequence" list. This terminator is represented by the integer value 3 and is added to the end of the sequence to indicate the end of the text sequence.

Finally, the ``sequence'' list is converted to a NumPy array using the ``np.array'' function and returned as a list containing one element.

This type of code is often used when preparing textual data for machine learning models. By mapping each character in the text to a unique integer index, we can represent text data as a sequence of integers that can be easily processed by the model. The start and end markers are used to indicate the start and end of each sequence, which is important when working with sequential data such as text.

Finally, the obtained frame is as follows.

	audio	text	MFCC features	text tokenized
0	myaudio/12440123.wav	... ابنبات های تومک را گرفت گفت ممنون پسرم نه من	[[[11.53494370737191, -3.370294705789073, -4.2...	[[2, 8, 9, 31, 9, 8, 10, 4, 32, 8, 40, 4, 10, ...
1	myaudio/12440124.wav	... قبلا هم به تو گفتم هیچ وقتم استراحت ندارم فقط	[[[10.105730436984143, -2.4352324174037996, -3...	[[2, 28, 9, 29, 8, 4, 32, 30, 4, 9, 32, 4, 10,...
2	myaudio/12440126.wav	...می شد او را یک فیلسوف به حساب اورد او عاشق چیز	[[[9.91634332069268, -2.5599461851402547, -4.9...	[[2, 30, 40, 4, 20, 15, 4, 8, 33, 4, 17, 8, 4,...
3	myaudio/12440127.wav	...اگر تومک فراموش می کرد برایش ابنبات بیاورد درس	[[[11.225249404021213, -3.406866944024388, -2....	[[2, 8, 39, 17, 4, 10, 33, 30, 38, 4, 27, 17, ...
4	myaudio/12440128.wav	... نان های ادویه دار قلبی شکل را بیشتر دوست داشت	[[[8.637159353566885, -2.7571319924894624, -5....	[[2, 31, 8, 31, 4, 32, 8, 40, 4, 8, 15, 33, 40...
...	...	...	...	...
495	myaudio/12560439.wav	...و زیبای شهر تهران به حساب می اید این پارک در ح	[[[15.512500755155884, -34.664955861022364, -1...	[[2, 33, 4, 18, 40, 9, 8, 40, 4, 20, 32, 17, 4...
496	myaudio/12560440.wav	...باغ پرندگان تهران در شمال شرق تهران و در دل جن	[[[13.953515756973522, -2.395566524010967, -14...	[[2, 9, 8, 26, 4, 35, 17, 31, 15, 39, 8, 31, 4...
497	myaudio/12560443.wav	...و مخصوص پرندگانی ایست که قابلیت پرواز دارند پو	[[[12.28639023707803, -27.30243460225483, -21....	[[2, 33, 4, 30, 14, 21, 33, 21, 4, 35, 17, 31,...
498	myaudio/12560444.wav	...جلوگیری می کند احداث این باغ سه سال به طول انج	[[[13.195155334816 87, -30.72443819788632, -10....	[[2, 12, 29, 33, 39, 40, 17, 40, 4, 30, 40, 4,...
499	myaudio/12560446.wav	...پردیس سینمایی ملت از بزرگترین مراکز سینمایی ته	[[[13.229105799634326, -8.476017093719932, -16...	[[2, 35, 17, 15, 40, 19, 4, 19, 40, 31, 30, 8,...

500 rows × 4 columns

"Audio" column contains file paths of audio recordings. The "text" column contains the textual transcription of the audio recordings. The "MFCC Features" column contains the Mel Cepstral Frequency Coefficients (MFCCs) extracted from the audio recordings. MFCCs are commonly used in speech recognition tasks to represent the spectral characteristics of speech signals.

The "marked text" column contains marked versions of text transcripts. The text is converted from a sequence of characters to a sequence of integers using a predefined character-to-integer mapping. Start and end markers are added to the beginning and end of each sequence, respectively.

This DataFrame is used as input to a machine learning model designed to perform speech recognition or speech-to-text tasks. MFCC features can be used as audio input to the model, while marked text can be used as output targets.

```
class SpeechRecognitionDataset(Dataset):
    def __init__(self, df):
        super().__init__()

        self.df = df.reindex()

    def __len__(self):
        return self.df.shape[0]

    def __getitem__(self, i):
        row = self.df.iloc[i]

        return row['MFCC features'][0], row['text tokenized'][0], row['text']
```

This code defines a custom PyTorch dataset named "SpeechRecognitionDataset" that is used to load and preprocess data for a speech recognition model. The dataset takes as input a DataFrame 'df', which is assumed to contain audio recordings and their corresponding text transcriptions.

The ``__init__()'' method initializes the dataset and calls the ``reindex()'' method on the input DataFrame to ensure that the data is in the correct order. The __len__() method returns the number of instances in the dataset, which is equal to the number of rows in the DataFrame.

The ``__getitem__()'' method is used to retrieve an instance of the dataset. It takes an index "i" as input and returns the MFCC features, text markup, and original text for the corresponding audio recording and transcription. MFCC features and text markup are returned as tensors, while the original text is returned as a string.

The MFCC features and text tokenization of the input DataFrame are accessed by indexing the 'MFCC features' and 'text tokenized' columns, respectively. Since each of these columns contains a list of arrays (one array per audio segment), only the first array is returned. This is because the dataset assumes that each audio segment and its corresponding transcription are independent samples and are processed separately by the model.

This dataset can be used to load and preprocess data for a speech recognition model implemented in PyTorch. The dataset ensures that the data is loaded in the correct format and can be easily fed to the model during training and inference.

```
# Split the DataFrame into train and test sets
train, test_dataset = train_test_split(df, test_size=0.2, random_state=42)
train_dataset, valid_dataset = train_test_split(train, test_size=0.2, random_state=42)

# Print the sizes of the train and test sets
print("Train set size:", len(train_dataset))
print("Test set size:", len(test_dataset))
print("Validation set size:", len(valid_dataset))
```

This code splits a DataFrame 'df' into three separate datasets for training, testing and validation using the 'train_test_split' function from the Scikit-learn library.

The "train_test_split" function is called twice to split the data twice. The first call splits the original DataFrame into a training set "train" and a test set "test_dataset" with a test size of 20% of the original data and a randomness mode of 42 to ensure reproducibility. The second call splits the training set "train" into a new training set "train_dataset" and a validation set "valid_dataset" with a validation size of 20% of the training data and a randomness mode of 42.

The size of each obtained data set is printed to the console using the "len()" function.

By splitting the original DataFrame into separate training, validation, and test sets, we can evaluate the performance of a machine learning model on unseen data. The training set is used to train the model, the validation set is used to adjust the hyperparameters and monitor the model's performance during training, and the test set is used to evaluate the final performance of the model after training is completed.

```
train_dataset = SpeechRecognitionDataset(train_dataset)
test_dataset = SpeechRecognitionDataset(test_dataset)
valid_dataset = SpeechRecognitionDataset(valid_dataset)
```

This code creates PyTorch datasets for training, testing, and validation sets using the "SpeechRecognitionDataset" class defined earlier. The SpeechRecognitionDataset class takes a DataFrame as input and preprocesses the data so that it can be used as input to a speech recognition model.

Each of the datasets ("train_dataset", "test_dataset" and "valid_dataset") is instantiated as an instance of the "SpeechRecognitionDataset" class, meaning that the data in each dataset is defined using the same methods. are loaded and preprocessed. SpeechRecognitionDataset class. This ensures that the data in each

dataset is properly formatted and can be used as input to the speech recognition model.

By creating separate datasets for each of the training, test, and validation sets, we can easily import the data into the PyTorch model during training and evaluation. The dataset can be used to create PyTorch data loaders, which can efficiently load a batch of data during training and testing and can help improve model performance.

A sample of its data is as follows.

```
(array([[ 13.62568575,  -3.3175881 , -16.11102065, ...,   0.88697396,
          3.06039665,  -2.28761419],
       [ 13.54501303,  -3.5967809 , -14.88764173, ...,  -4.49186266,
          0.84562289,  -4.01774741],
       [ 13.44333561,  -4.10979872, -15.48195416, ...,  -9.73361525,
         -2.7058801 ,  -1.51978709],
       ...,
       [ 10.25747896, -23.33675272, -11.93382051, ..., -12.24532404,
          0.48061785,  15.21665637],
       [  9.58522594, -25.51941706, -10.10262258, ..., -18.12538203,
         -4.58360393,  14.01372521],
       [  8.78055547, -25.58931087,  -3.43031976, ...,  -4.32293298,
         -3.65340841,   9.78029154]]),
array([ 2, 30, 12, 30, 33, 25, 32,  4, 15, 17, 40,  8, 36, 32,  4, 40,  4,
        8, 32, 31, 38,  4, 15, 17,  4, 35, 31, 12,  4, 38, 40, 29, 33, 30,
       10, 17, 40,  4, 12, 31, 33,  9,  4, 17, 33, 19, 10,  8, 40,  4, 19,
       40, 30, 40, 31,  4, 15, 20, 10,  4,  8, 18,  4, 10, 33,  8,  9, 25,
        4, 27, 40, 17, 33, 18, 38, 33, 32,  4, 28, 17,  8, 17,  4, 15,  8,
       17, 15,  3]),
'مجموعه دریاچه ی اهنک در پنج کیلومتری جنوب روستای سیمین دشت از توابع فیروزکوه قرار دارد')
```

The first element is a dim array of the form "(n_samples, n_features)". This array contains the MFCC feature extracted from the audio file.

The second element is a dim array of the form "(n_tokens,)". The array contains the tokenized version of the text transcription that corresponds to the audio recording. The text is converted from a sequence of characters to a sequence of integers using a predefined character-to-integer mapping.

The third element is a string that includes the transcription of the original text into Persian script. The transcription of the text corresponds to the audio recording from which the MFCC features are extracted.

This result shows that the code successfully extracts MFCC features and corresponding text transcripts from audio recordings and can convert the text transcripts into a format that can be used as the target of the model. A transcription of the original text is also retained for reference purposes.

```
def padded_features(b):
    (x, y, t) = zip(*b)
    xl = np.array([len(a) for a in x])
    yl = np.array([len(a) for a in y])

    xp = np.zeros((len(x), max(xl), x[0].shape[1]))
    yp = np.zeros((len(y), max(yl)))
    for i in range(len(x)):
        xp[i, :xl[i], :] = x[i]
        yp[i, :yl[i]] = y[i]

    return torch.from_numpy(xp), torch.from_numpy(yp), torch.from_numpy(xl), torch.from_numpy(yl), t
```

This code defines a function called "padded_features" that is used to add a sequence of MFCC features and transcribe text so that they can be used as input to a PyTorch model. This function takes as input a set of samples (`b`), where each sample is a tuple containing MFCC features, text markup, and transcription of the original text for the audio recording.

This function first unpacks the MFCC features, text markup, and original text transcription using the ``zip()'' function from the input handle. It then calculates the length of each MFCC feature sequence and the text-marked sequence using the len() function and stores them in the xl and yl arrays, respectively.

This function then creates zero-padded numpy arrays "xp" and "yp" of dimensions "(batch_size, max(xl), n_mfcc)" and "(batch_size, max(yl))", where Batch_size is Is. The number of samples in the batch and "n_mfcc" is the number of MFCC features per frame. These arrays are used to store padded sequences of MFCC features and text tokenization, respectively.

The function then loops over each instance in the batch and appends the corresponding MFCC features and text-tagged sequences using the arrays "xp" and "yp" with zeros. The MFCC feature sequence is padded with zeros along the time axis (i.e., the second dimension of the "xp" array), while the text-marked sequence is padded with zeros along the sequence axis (i.e., the second dimension of the "yp" array). The padded sequences are then stored in the corresponding rows of the "xp" and "yp" arrays.

This function then creates PyTorch tensors from padded numpy arrays using the ``torch.from_numpy()'' function and returns them along with the length of the original MFCC feature and tokenized sequences of text (`xl' and 'yl', respectively). Overwrite the original text (`t`). These tensors can be used as input to the PyTorch model during training and evaluation.

```
train_datalader = DataLoader(train_dataset, batch_size=8, shuffle=True, collate_fn=padded_features, num_workers=2)
valid_datalader = DataLoader(valid_dataset, batch_size=8, shuffle=True, collate_fn=padded_features, num_workers=2)
test_datalader = DataLoader(test_dataset, batch_size=8, shuffle=True, collate_fn=padded_features, num_workers=2)
```

This code creates PyTorch "DataLoader" objects for the training, validation, and testing datasets. A "DataLoader" is a PyTorch class that provides an iterable over a dataset that can be used to efficiently load data during training and evaluation.

Each "DataLoader" is created with the corresponding dataset ("train_dataset", "valid_dataset" or "test_dataset") as input. The "child_size" argument is set to 8, which means that the "DataLoader" will load the data in batches of 8 instances. The "shuffle" argument is set to "True", which means that the data is shuffled before each epoch during training.

The "collate_fn" argument is set to "padded_features", which is the function used to add the MFCC feature and text tokenized sequences for each set of samples. This function is called on each batch of instances before it is returned by the 'DataLoader'. The "num_workers" argument is set to 2, which means that two worker processes are used to load the data in parallel.

By creating "DataLoader" objects for each dataset, we can efficiently load batches of MFCC features and text tokenized sequences during training and evaluation, which can help improve model performance.

Designing models

Using ``baseModel'' class, it is possible to implement several different models for speech recognition. For example, we can implement three models ``LSTM'', ``GRU'', and ``RNN'' by taking the `baseModel' class as parent. These models are derived using ``LSTM'', ``GRU'', and ``RNN'' layers, respectively, with their own parameters. To use these models, we can create each one as an object and give it the necessary inputs and get the output.

```
class baseModel(nn.Module):
    def __init__(self):
        super(baseModel, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv1d(13, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Conv1d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Conv1d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Dropout(0.2)
        )

        self.specificModel = None

        self.fc = nn.Linear(128, 36)

    def forward(self, x):
        # CNN input must be (batch, features, seq)
        x = x.transpose(1, 2)
        x = self.conv(x)

        # RNN input must not be batch first (seq, batch, feature)
        x = x.permute([2, 0, 1])
        x, _ = self.specificModel(x)

        x = self.fc(x)

        return x
```

This code defines a PyTorch "nn.Module" class named "baseModel". This class is used as a base to implement a model for speech recognition so that the comparison can be done well.

The "__init__()" method defines the structure of the model. This model consists of a one-dimensional convolutional neural network (CNN) followed by a specific model, which can be a recurrent neural network (RNN) or another type of neural network. CNN takes a sequence of MFCC features as input and applies a set of convolutional layers with ReLU activation functions and elimination layers. The CNN output is then sent to a special model that further processes the sequence. Finally, a fully connected layer with 36 output nodes is applied to generate the model output.

The "forward()" method also defines the forward movement of the model. In general, this ``baseModel'' class provides a flexible structure for implementing a speech recognition model that can accommodate both CNN and RNN components.

```
class LSTM(baseModel):
    def __init__(self):
        super(LSTM, self).__init__()

        self.specificModel = nn.LSTM(256, hidden_size=128, num_layers=3, dropout=0.2)
```

```
class GRU(baseModel):
    def __init__(self):
        super(GRU, self).__init__()

        self.specificModel = nn.GRU(256, hidden_size=128, num_layers=3, dropout=0.2)
```

```
class RNN(baseModel):
    def __init__(self):
        super(RNN, self).__init__()

        self.specificModel = nn.RNN(256, hidden_size=128, num_layers=3, dropout=0.2)
```

This code defines three PyTorch "nn.Module" classes, named "RNN", "LSTM", and "GRU", which are used to implement Recurrent Neural Network (RNN)-based models for speech recognition. These classes inherit from the ``baseModel'' class, which provides the basic structure of the models.

Each of the RNN, LSTM, and GRU classes overrides the __init__() method of the baseModel class to specify the specific model architecture. For example, in the class "RNN", a layer "nn.RNN" is defined with 256 input features, 128 hidden units, 3 layers and a dropout rate of 0.2. Similarly, an nn.LSTM layer with the same parameters is

defined in the `LSTM' class, and an nn.GRU layer is defined in the `GRU' class with the same parameters.

By defining separate classes for each RNN model type, the code makes it easy to switch between different RNN types during model testing and development. Each class can be instantiated as an object that can be used to train and evaluate the corresponding RNN model for speech recognition.

Model training

```
def greedy_CTC_decoder(emission, labels):

    indices = torch.argmax(emission, dim=-1)
    indices = torch.unique_consecutive(indices, dim=-1)
    indices = [i.item() for i in indices if i > 3]
    joined = "".join([labels[i] for i in indices])
    return joined.strip()
```

I This code defines a function named ``greedy_CTC_decoder'' that implements a greedy decoding algorithm for a speech recognition task using the connection temporal classification (CTC) loss function. The inputs to the function are ``emission'', which is a tensor containing the emission probabilities for each frame of the input sequence, and ``labels'', which is a list of character labels.

This function first calculates the index of the most likely label for each frame of the input sequence using "torch.argmax(emission, dim=-1)". The resulting tensor has the form ``[num_seq,]'', where each element is an integer representing the most likely label for the corresponding frame.

The function then uses torch.unique_consecutive(indexes, dim=-1) to find unique consecutive elements in the resulting tensor. This operation returns a tensor containing the unique consecutive elements in the input tensor. The resulting tensor is then converted to a Python list using list comprehensions and filtered to remove empty labels (indicated by indices less than 3.(

The resulting list of indices is used to extract the corresponding character tags from the "tags" list and join them to a string using ".join()". Finally, the resulting string is stripped of any leading or trailing whitespace using the ``strip()'' method and returned as the output of the function.

In general, this function performs a simple decoding algorithm that selects the most probable label for each frame of the input sequence and removes any consecutive

duplicate labels and empty labels to produce the final transcription of the input speech signal.

```
def compute_wer(ref, hyp, t="v"):
    if(t=="v"):
      ref_words = ref.split()
      hyp_words = hyp.split()
    else:
      ref_words = ref
      hyp_words = hyp

    ref_len = len(ref_words)
    hyp_len = len(hyp_words)
    wer = Lev.distance(ref_words, hyp_words) / ref_len

    return wer
```

This code defines a function called "compute_wer" that calculates the word error rate (WER) between a reference string ("ref") and a hypothesis string ("hyp"). The function takes an optional argument 't' which can be used to specify the format of input strings.

This function calculates the length of reference and hypothesis list using "len()". The WER is calculated as the Lowenstein distance between the reference lists and the hypothesis divided by the length of the reference list. The Levenshtein distance is the minimum number of insertions, deletions and substitutions required to convert one sequence into another and is calculated using the 'distance()' method of the 'Lev' module.

The function returns the WER value as output. Optionally, the function can also calculate the positional error rate (PER) by dividing the Lowenstein distance by the length of the hypothesis list instead of the reference list. However, this functionality is already described in the code.

In general, the "compute_wer" function provides a simple and efficient implementation of the WER metric for evaluating the performance of speech recognition systems.

```
def compute_ser(ref, hyp, t="v"):

    if(t=="v"):
      ref_sents = ref.split('.')
      hyp_sents = hyp.split('.')
    else:
      ref_sents = ref
      hyp_sents = hyp

    ref_len = len(ref_sents)
    hyp_len = len(hyp_sents)
    ser = sum([1 for i in range(ref_len) if i >= hyp_len or ref_sents[i] != hyp_sents[i]]) / ref_len
    return ser
```

This code defines a function called "compute_ser" that calculates the sentence error rate (SER) between a reference string ("ref") and a hypothesis string ("hyp"). The function takes an optional argument 't' which can be used to specify the format of input strings.

This function calculates the length of reference and hypothesis list using "len()". SER is calculated as the fraction of sentences in the reference list that are not correctly identified by the system. Specifically, the function loops over each sentence in the reference list and increments a counter if the corresponding sentence in the hypothesis list does not match the reference sentence. If the hypothesis list is shorter than the reference list, the function assumes that the remaining sentences in the reference list are not recognized by the system.

The function returns the SER value as output.

Overall, the "compute_ser" function provides a simple and efficient implementation of the SER metric for evaluating the performance of speech recognition systems.

```python
def pad_texts(text1, text2, pad_char=' '):
    max_len = max(len(text1), len(text2))
    return list(text1.ljust(max_len, pad_char)), list(text2.ljust(max_len, pad_char))

def train(model, epoch, dataloaders, criterion, optimizer, device, training_name):


    train_history = {'loss': [], 'accuracy': [], 'f1-score': [], 'ser':[], 'wer':[]}
    valid_history = {'loss': [], 'accuracy': [], 'f1-score': [], 'ser':[], 'wer':[]}

    for e in tqdm(range(epoch)):
        # training phase
        model.train()
        train_loss = 0
        train_accuracy = 0
        train_f1 = 0
        total_count = 0
        wert =0
        sert = 0
        for sample in dataloaders['train']:
            optimizer.zero_grad()

            model_input = sample[0].float().to(device)
            target = sample[1].float().to(device)
            input_lens = sample[2].to(device)
            target_lens = sample[3].to(device)

            prediction = model(model_input)
            loss = criterion(prediction.log_softmax(dim=-1), target, input_lens, target_lens)

            loss.backward()
            optimizer.step()

            train_loss += loss.item() * prediction.size(0)
            total_count += prediction.size(0)

            for i in range(prediction.size(1)):
                predicted_text = greedy_CTC_decoder(prediction[:, i], int_to_char)

                if predicted_text == sample[4][i]:
                    train_accuracy += 1

                # Pad texts to the same length
                padded_target, padded_predicted = pad_texts(sample[4][i], predicted_text)

                # Calculate precision, recall, and F1-score
                precision, recall, f1, _ = precision_recall_fscore_support(padded_target, padded_predicted, average=None,
zero_division=0)
                train_f1 += np.mean(f1)

                sample_wer = compute_wer(target, predicted_text, 't')
                # print("sample_wer: ", sample_wer)
                wert += sample_wer
                # pert += sample_per

                sample_ser = compute_ser(target, predicted_text, 't')
                sert += sample_ser



        train_loss /= total_count
        train_accuracy /= total_count
        train_f1 /= total_count

        train_history['loss'].append(train_loss)
        train_history['accuracy'].append(train_accuracy)
        train_history['f1-score'].append(train_f1)
        train_history['wer'].append(wert/total_count)
        train_history['ser'].append(sert/total_count)

        # eval phase
        model.eval()

        total_loss = 0
        correct_count = 0
        total_count = 0
        wer = 0
        ser = 0

        precisions = []
        recalls = []
        f1_scores = []

        for sample in dataloaders['valid']:
            model_input = sample[0].float().to(device)
            target = sample[1].float().to(device)
            input_lens = sample[2].to(device)
            target_lens = sample[3].to(device)
            target_texts = sample[4]

            with torch.no_grad():
                prediction = model(model_input)

                loss = criterion(prediction.log_softmax(dim=-1), target, input_lens, target_lens)
                total_loss += loss.item() * prediction.size(0)

                for i in range(prediction.size(1)):
                    predicted_text = greedy_CTC_decoder(prediction[:, i], int_to_char)

                    if predicted_text == target_texts[i]:
                        correct_count += 1
                    total_count += 1

                    sample_wer = compute_wer(target_texts[i], predicted_text)
                    wer += sample_wer

                    sample_ser = compute_ser(target_texts[i], predicted_text)
                    ser += sample_ser

                    # Pad texts to the same length
                    padded_target, padded_predicted = pad_texts(target_texts[i], predicted_text)

                    # Calculate precision, recall, and F1-score
                    precision, recall, f1, _ = precision_recall_fscore_support(padded_target, padded_predicted, average=None,
zero_division=0)
                    precisions.extend(precision)
                    recalls.extend(recall)
                    f1_scores.extend(f1)

        valid_history['loss'].append(total_loss / total_count)
        valid_history['accuracy'].append(correct_count / total_count)
        valid_history['f1-score'].append(np.mean(f1_scores))
        valid_history['wer'].append(wer / total_count)
        valid_history['ser'].append(ser/ total_count)

    return model, train_history, valid_history
```

This code defines a function called "Train" that trains a speech recognition model using a dataset of loaders and a given number of epochs. This function takes several arguments, including model, optimizer, loss metric, device, and training name.

This function first initializes empty dictionaries to store training and validation performance metrics such as loss, accuracy, F1 score, SER, and WER.

The function then loops through the specified number of cycles and performs the following steps:

- Using `model.train()` sets the model to training mode
- Initializes variables to store training metrics such as loss, accuracy, F1 score, SER, and WER.
- loops through the training Dataloader and performs the following steps for each instance:
- Clears gradients using `optimizer.zero_grad`()
- Converts the input and target sequences to the appropriate device using "to (device."(
- passes the input sequences through the model to obtain the predicted output sequences
- Calculates the loss between the predicted sequence and the target using the given loss metric
- Using "backward()" and "step()" it propagates the losses backward and updates the model parameters.
- Calculates various criteria such as accuracy, F1 score, SER and WER and adds them to the relevant variables.
- Calculates the average training metrics across all samples and adds them to the training history dictionary
- Then the function goes to the validation stage:
- Using `model.eval()` puts the model in evaluation mode
- Initializes variables to store validation metrics such as loss, precision, F1 score, SER, and WER.
- Loops through the Validation Dataloader and performs the following steps for each instance:
- Converts the input and target sequences to the appropriate device using "to (device."(
- passes the input sequences through the model to obtain the predicted output sequences
- Calculates the loss between the predicted sequence and the target using the given loss metric
- Calculates various criteria such as accuracy, F1 score, SER and WER and adds them to the relevant variables.

- Calculates the average validation criteria across all samples and adds them to the validation date dictionary

Finally, the trained model and dictionary function returns the training and validation history, which can be used to analyze the model's performance during training.

```
dataloaders = {'train': train_datalader, 'valid': valid_datalader}

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

This code initializes two variables "dataloader" and "device."

Generally, these two lines of code are used to set the data loading and device settings for the training process. The "dataload" dictionary is used for convenient access to training and validation data during training, while the "device" variable specifies where the model and data are loaded during training.

RNN model training

```
model_rnn = RNN()
criterion = nn.CTCLoss()
optimizer = torch.optim.Adam(model_rnn.parameters(), lr=0.001)
model_rnn = model_rnn.to(device)

model_rnn, train_rnn, valid_rnn = train(model_rnn, 250, dataloaders, criterion, optimizer, device, 'RNN training')
```

This code trains a Recurrent Neural Network (RNN) model for speech recognition using the "Train" function defined elsewhere in the code.

The first line of code creates an instance of the RNN class that represents the RNN model.

The second line defines the loss metric as the connection time classification (CTC) loss using the "nn.CTCLoss()" function.

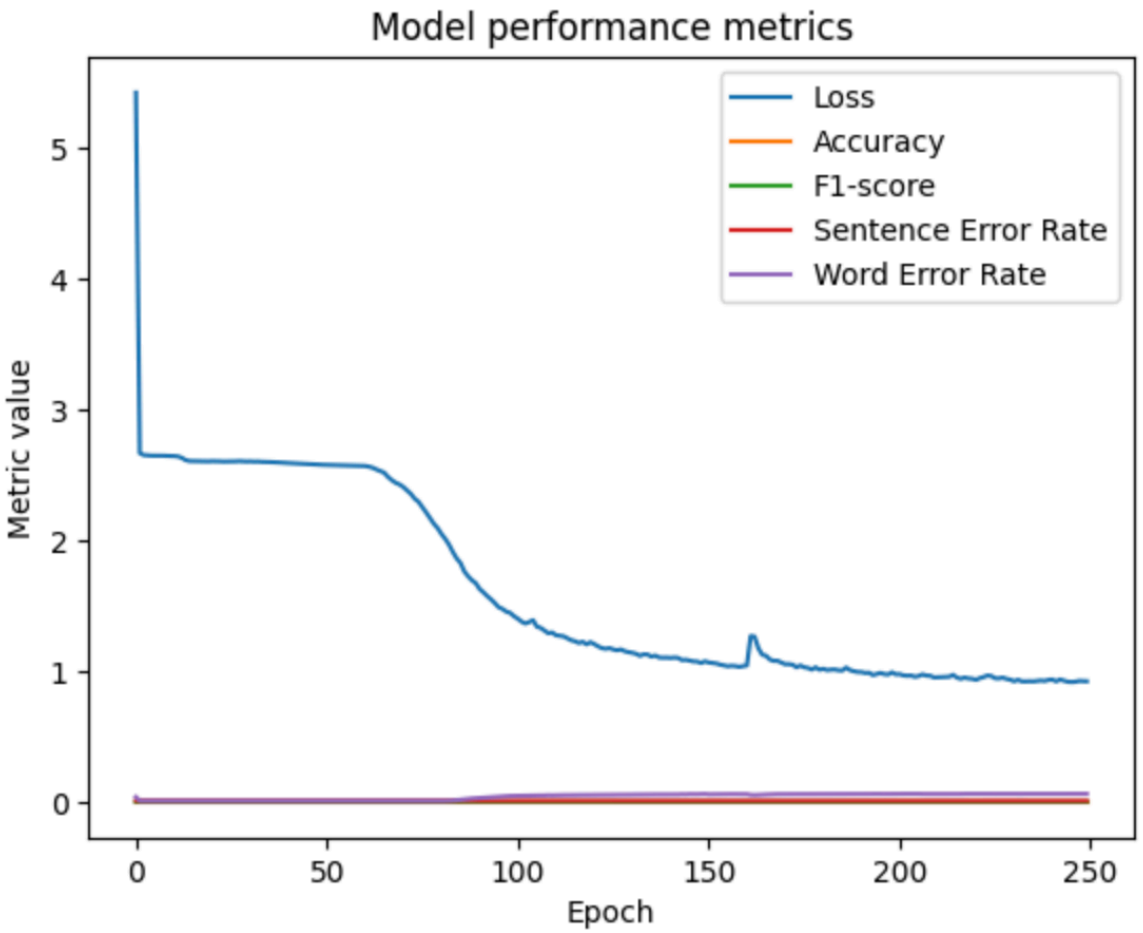
The third line initializes an Adam optimizer with a learning rate of 0.001 to optimize the parameters of the RNN model. The `model_rnn.parameters()` argument specifies which parameters in the model should be optimized.

The fourth line transfers the RNN model to the device specified by the previously set "device" variable.

The final line calls the "train" function to train the RNN model using the specified hyperparameters and dataloaders. This function returns the trained model as well as dictionaries containing the training and validation history of the RNN model. These

dictionaries can be used to visualize the training progress and evaluate the performance of the RNN model in the training and validation sets. The "training_name" argument is set to "RNN training" to identify the specific training course for later use.

We display the obtained results.



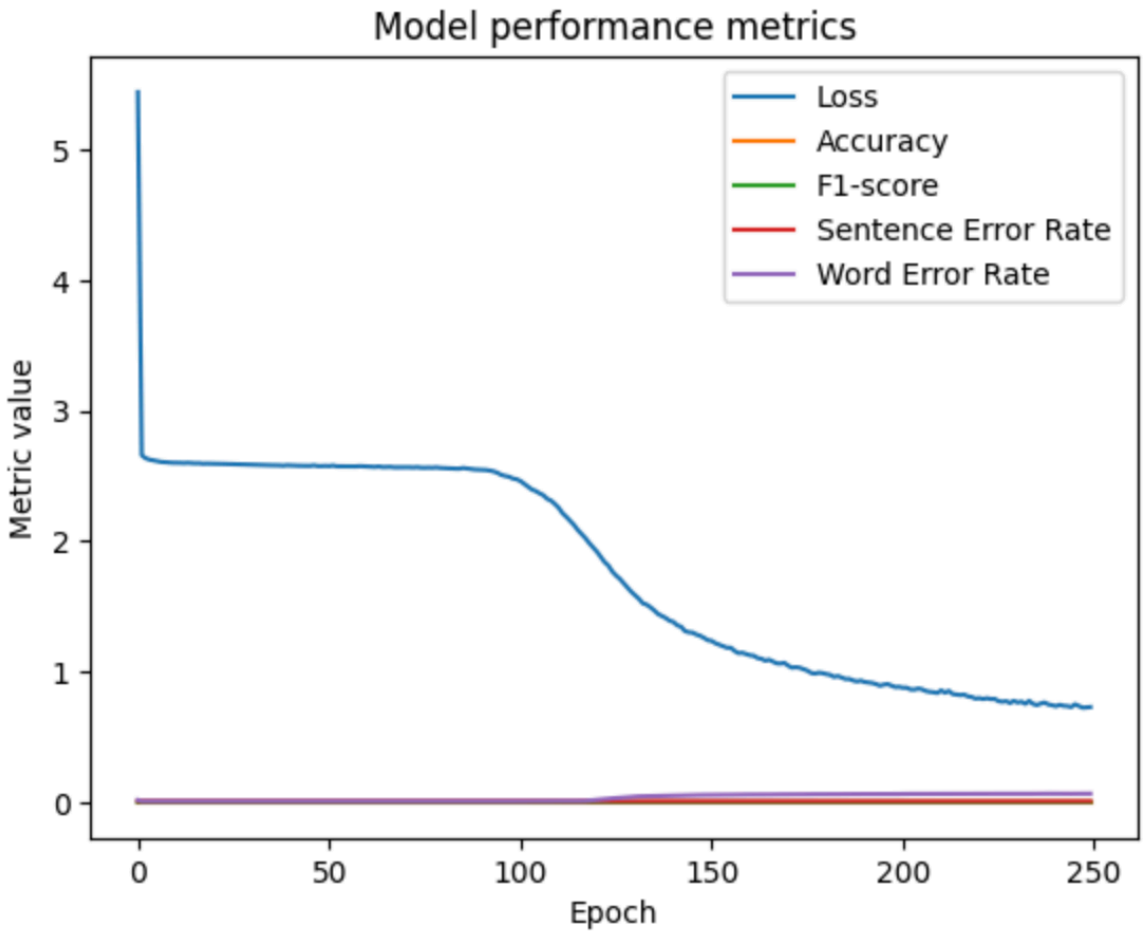
All the criteria together for the RNN model are as above. But further, we will compare the criteria of the models in more detail to find the best model.

GRU model training

The GRU model is trained in the same way as the RNN model.

```
model_gru = GRU()
criterion = nn.CTCLoss()
optimizer = torch.optim.Adam(model_gru.parameters(), lr=0.001)
model_gru = model_gru.to(device)

model_gru, train_gru, valid_gru = train(model_gru, 250, dataloaders, criterion, optimizer, device, 'GRU training')
```

The results are as follows.

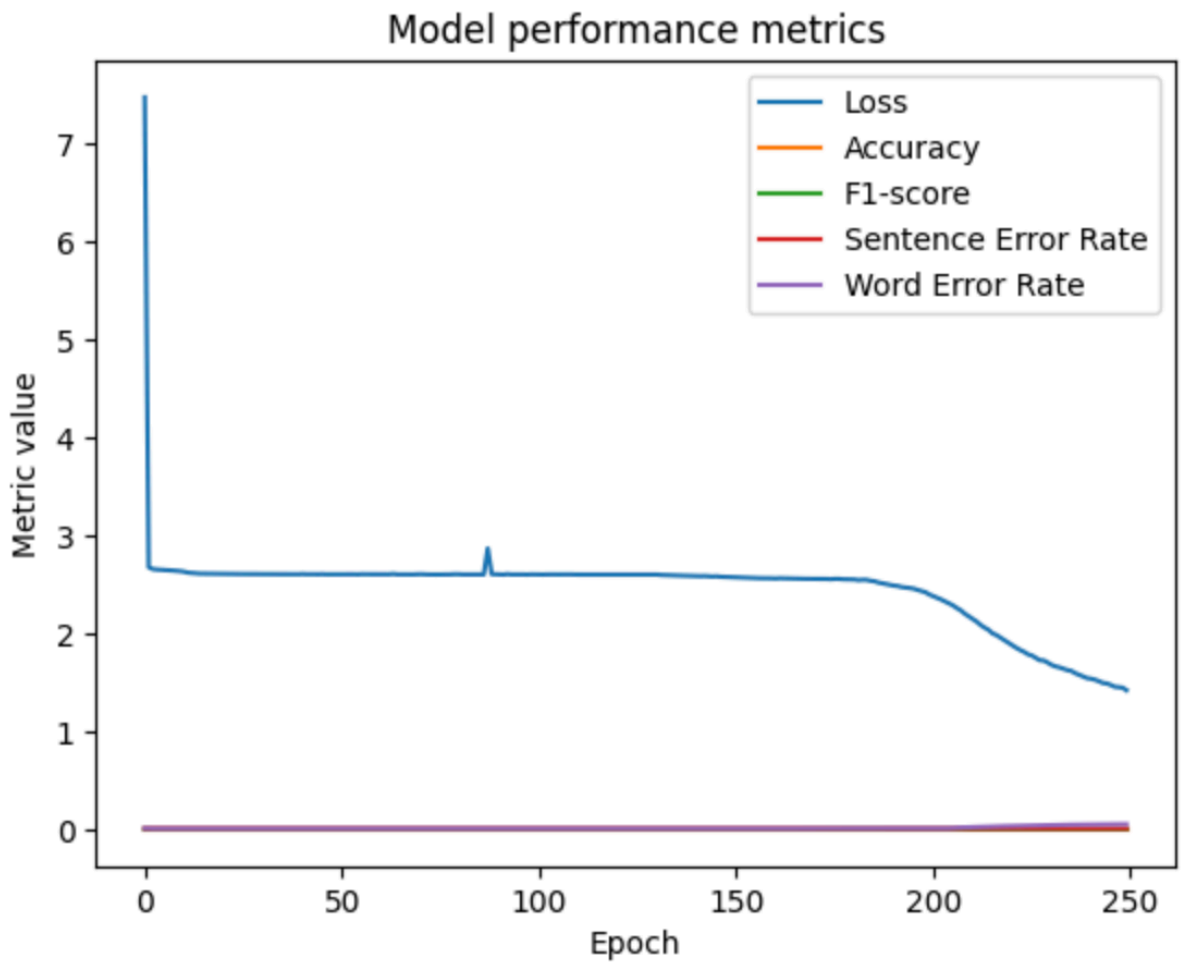


LSTM model training

LSTM model training is done like the previous models.

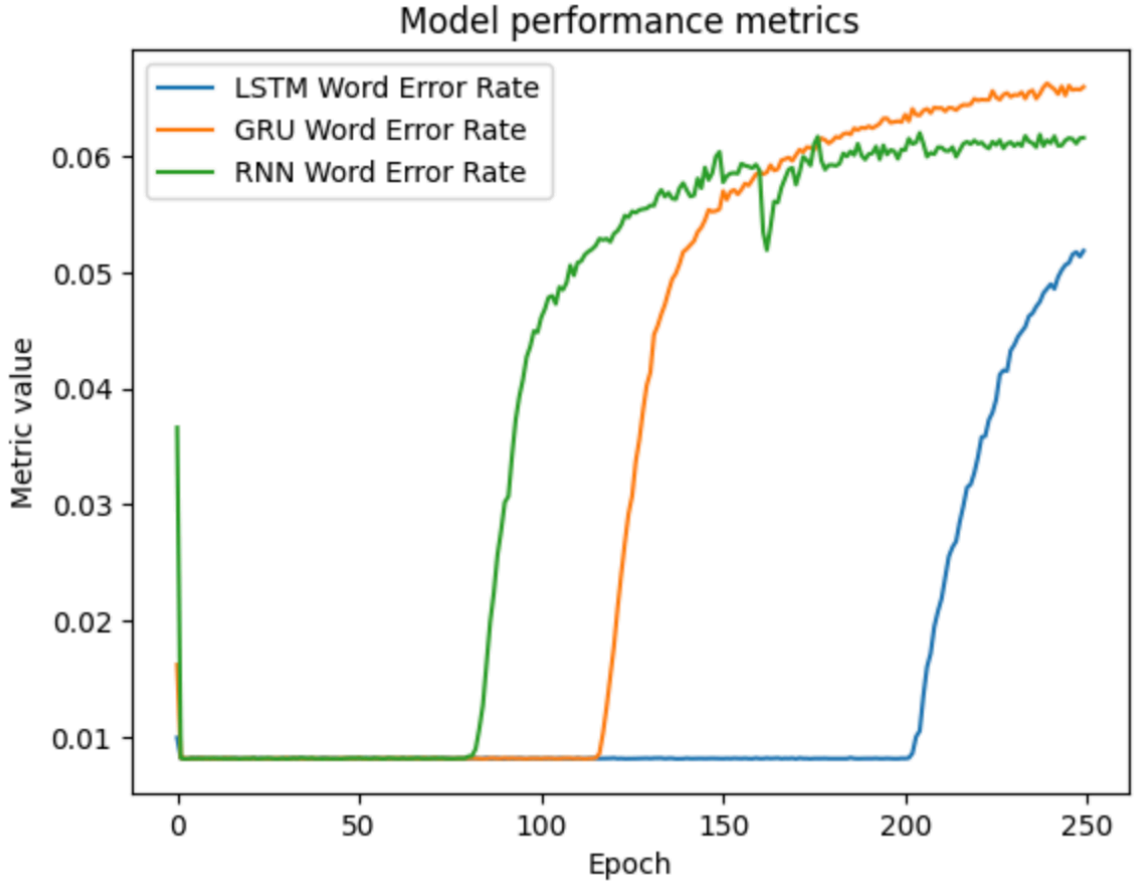
```
model_lstm = LSTM()
criterion = nn.CTCLoss()
optimizer = torch.optim.Adam(model_lstm.parameters(), lr=0.001)
model_lstm = model_lstm.to(device)

model_lstm, train_lstm, valid_lstm = train(model_lstm, 250, dataloaders, criterion, optimizer, device, 'LSTM training')
```

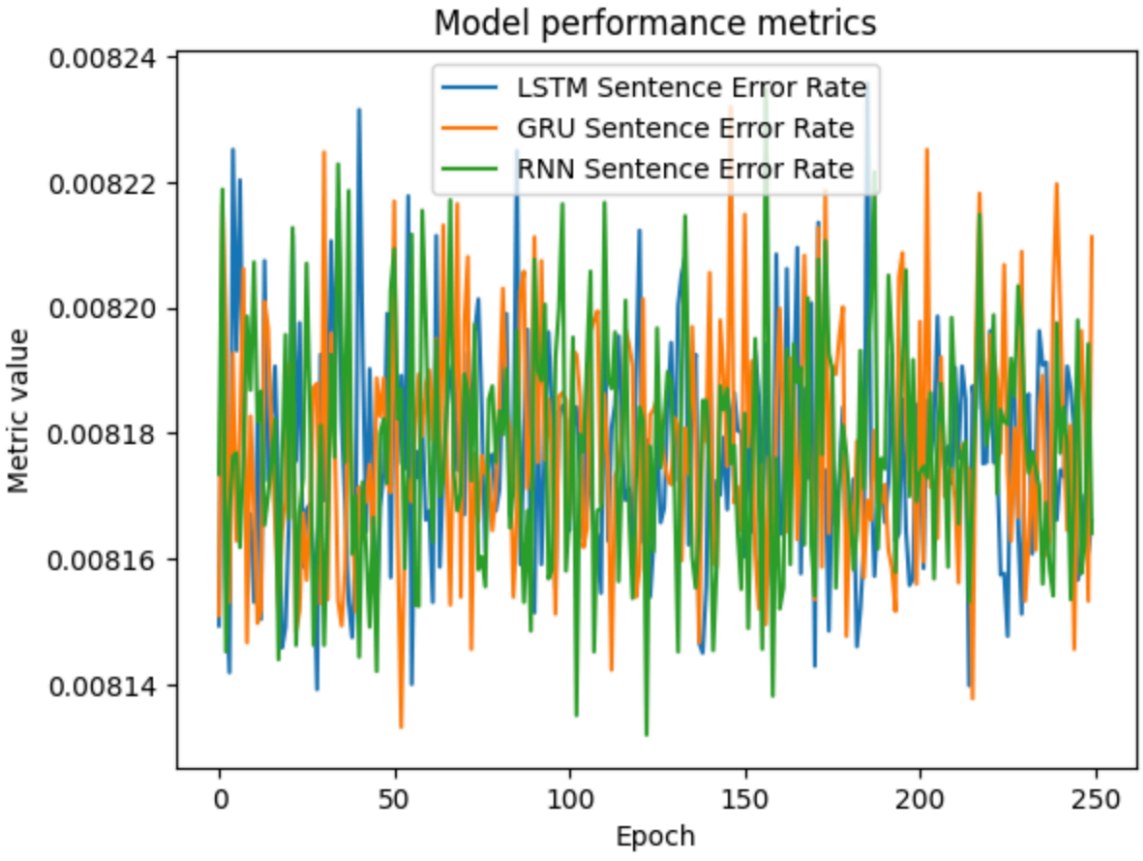
The results are as follows.

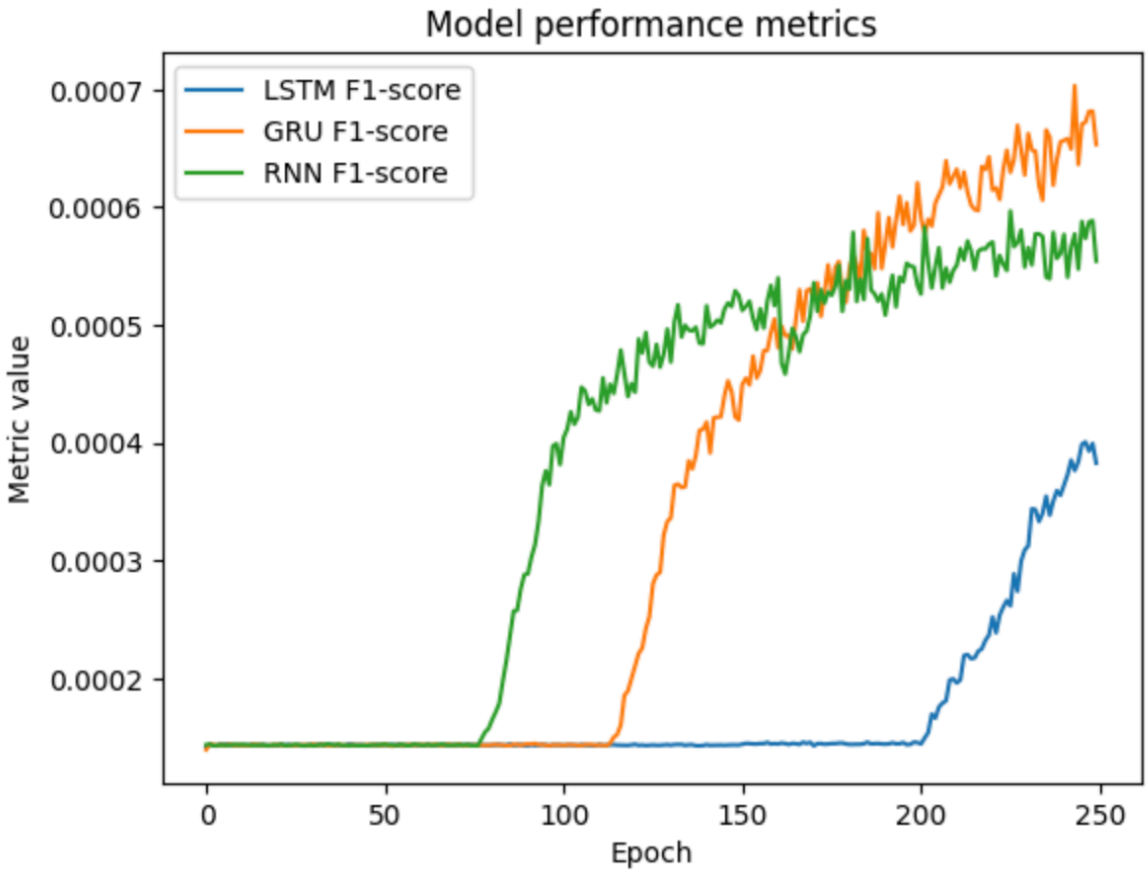


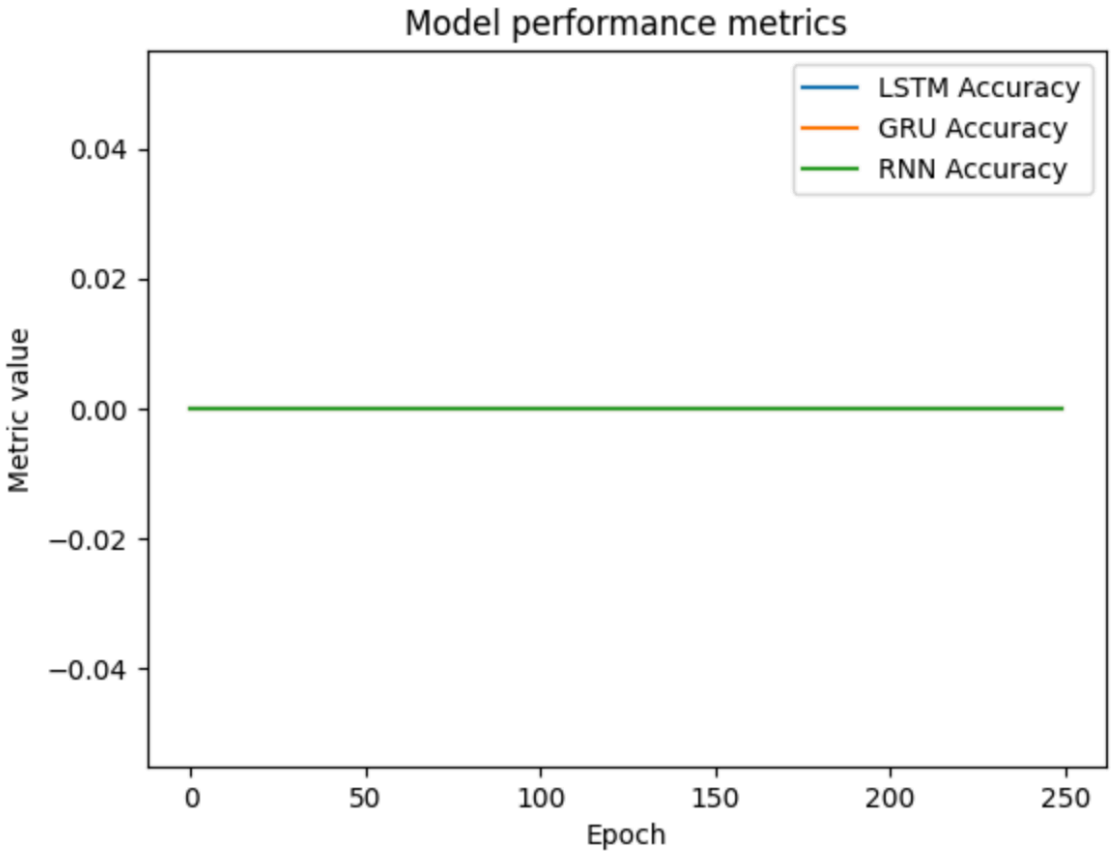
Comparison of models

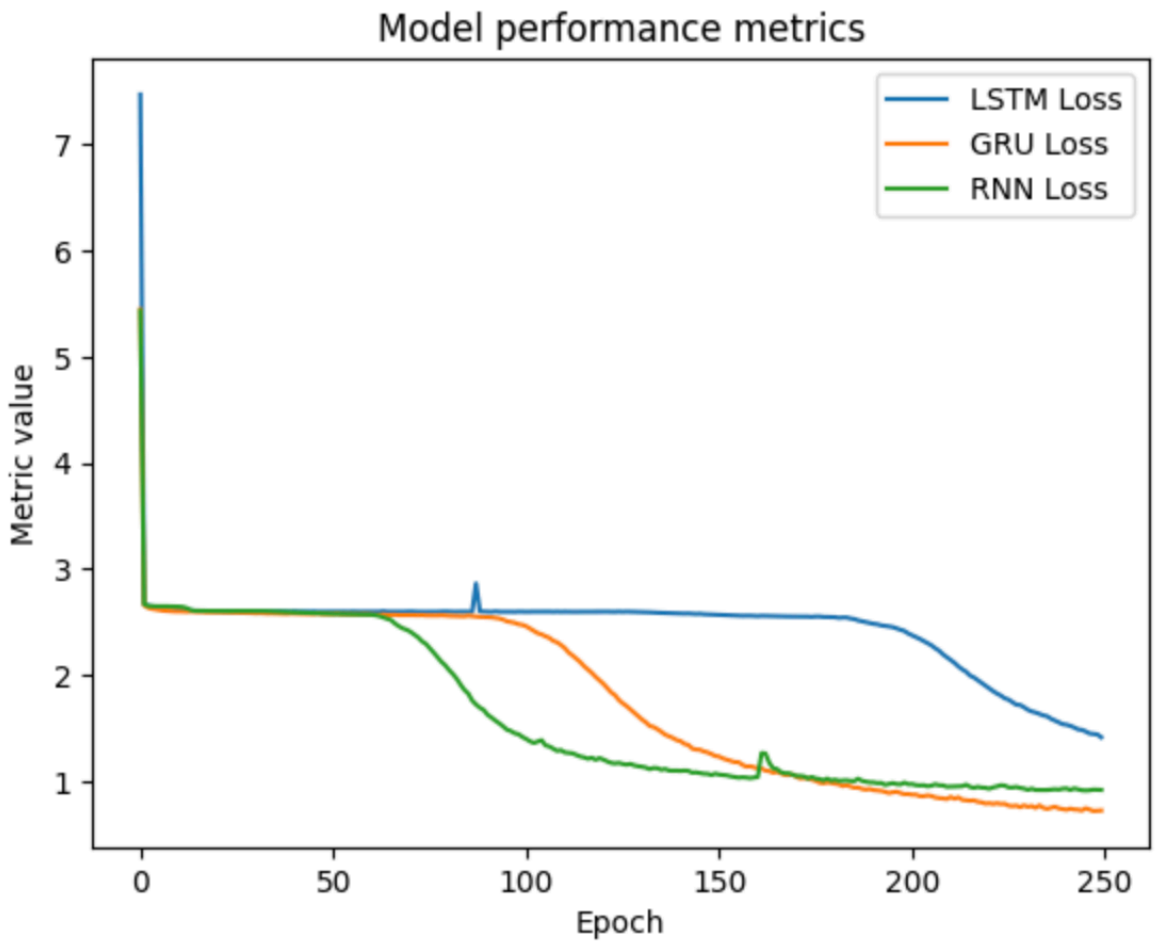
We compare the different criteria of the models side by side.


Model performance metrics
LSTM Word Error Rate
GRU Word Error Rate
RNN Word Error Rate
Metric value
0.06
0.05
0.04
0.03
0.02
0.01
0
50
100
150
200
250
Epoch


Model performance metrics
LSTM Sentence Error Rate
GRU Sentence Error Rate
RNN Sentence Error Rate
Metric value
0.00824
0.00822
0.00820
0.00818
0.00816
0.00814
0
50
100
150
200
250
Epoch


Model performance metrics
LSTM F1-score
GRU F1-score
RNN F1-score
Metric value
0.0007
0.0006
0.0005
0.0004
0.0003
0.0002
0
50
100
150
200
250
Epoch


Model performance metrics
LSTM Accuracy
GRU Accuracy
RNN Accuracy
Metric value
0.04
0.02
0.00
−0.02
−0.04
0
50
100
150
200
250
Epoch



Accuracy, loss, f1-score, SER, and WER variables are very important in machine learning training problems and are used as models' performance evaluation criteria.

- The accuracy measure shows how well the model has performed correctly in recognizing the input data. This measure is equal to the number of correct predictions divided by the total number of predictions. A higher number is better for accuracy. As shown in the graphs and results. The accuracy criterion is not a good criterion for evaluating this speech recognition task. One of the reasons for the low accuracy and low scores in this task is the small number of data.
- The loss criterion indicates how much the model has been able to reduce prediction errors. As the prediction accuracy of the model increases, this criterion decreases and vice versa. In general, the goal is to reduce loss.
- The f1-score criterion is a comprehensive measure of precision and recall. In short, this measure is used to evaluate the performance of the model on experimental data. A higher number is better for f1-score.
- The Sentence Error Rate measure shows how much the model was able to identify a sentence correctly. This criterion is used as an evaluation criterion for speech recognition systems.

- The Word Error Rate measure shows how well the model has been able to correctly identify the words in a sentence. For example, if the WER was equal to 0.1, it means that the model did not recognize 10% of the words correctly.

In general, to evaluate the performance of the model, we should consider all these criteria together. In general, the models that perform well in all these criteria have the best performance and can be used for real applications.

And according to the graphs above, the GRU model achieves better results.

The obtained samples are as follows.

```
----------------------------------------
Predicted:
سرز اب رودخانه  لار سر دلار به رود هراز م رزد ه در استان مازندران است
Target:
سرریز اب رودخانه ی لار از سد لار به رود هراز می ریزد که در استان مازندران است
----------------------------------------

    ----------------------------------------
    Predicted:
    رودخانه  ن ا رودخانه  ستلان رود به تول سوستلو متر است ه از ررته وه توشال سشمه رفت
    Target:
    رودخانه ی کن یا رودخانه ی سولقان رودی به طول سی سه کیلومتر است که از رشته کوه توچال سرچشمه گرفته
    ----------------------------------------
    Predicted:
    منطش تار ممنوه اب د در اامن جنوب رشته ه البرز
    Target:
    منطقه ی شکار ممنوع کاوه ده در دامنه ی جنوبی رشته کوه البرز
    ----------------------------------------
```