FINAL PROJECT
ADVANCED DEEP LEARNING
NATURAL LANGUAGE To SQL QUERY

Team Members	UBIT	UBID
LAKSHANA KUNDAN	lkundan	50537728
YASHWANTH CHENNU	ychennu	50537845
AMANULLA SHAIK	ashaik5	50538642

DATASET:

Nature of Dataset:

- WikiSQL is designed to facilitate the training and evaluation of models that can interpret natural language questions and generate corresponding SQL queries.
- It consists of pairs of natural language questions and SQL queries, along with the associated tables on which the queries are performed.

Data Types:

- The natural language questions are in plain text format.
- SQL queries are represented as structured query statements.
- Both are of categorical data.

Variety of Queries:

- The SQL queries often involve a variety of SQL operations such as SELECT, WHERE, and AGGREGATE functions.
- Queries vary in complexity from simple single condition statements to more complex queries involving multiple conditions.

```
metaData
```

```
DatasetDict({
    test: Dataset({
        features: ['phase', 'question', 'table', 'sql'],
        num_rows: 15878
    })
    validation: Dataset({
        features: ['phase', 'question', 'table', 'sql'],
        num_rows: 8421
    })
    train: Dataset({
        features: ['phase', 'question', 'table', 'sql'],
        num_rows: 56355
    })
})
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 56355 entries, 0 to 56354
Data columns (total 2 columns):
 #   Column    Non-Null Count  Dtype 
---  ------    --------------  ----- 
 0   question  56355 non-null  object
 1   sql       56355 non-null  object
dtypes: object(2)
memory usage: 880.7+ KB
```

```
df.isnull().sum()
```

```
question    0
sql         0
dtype: int64
```

```
trainDataFrame.head()
```

	question	sql
0	Tell me what the notes are for South Australia	SELECT Notes FROM table WHERE Current slogan =...
1	What is the current series where the new serie...	SELECT Current series FROM table WHERE Notes =...
2	What is the format for South Australia?	SELECT Format FROM table WHERE State/territory...
3	Name the background colour for the Australian ...	SELECT Text/background colour FROM table WHERE...
4	how many times is the fuel propulsion is cng?	SELECT COUNT Fleet Series (Quantity) FROM tabl...

```
validationDataFrame
```

	question	sql
0	What position does the player who played for b...	SELECT Position FROM table WHERE School/Club T...
1	How many schools did player number 3 play at?	SELECT COUNT School/Club Team FROM table WHERE...
2	What school did player number 21 play for?	SELECT School/Club Team FROM table WHERE No. = 21
3	Who is the player that wears number 42?	SELECT Player FROM table WHERE No. = 42
4	What player played guard for toronto in 1996-97?	SELECT Player FROM table WHERE Position = Guar...
...	...	...
8416	Which loss has an attendance greater than 49,6...	SELECT Loss FROM table WHERE Attendance > 49,6...
8417	What is the largest attendance that has tigers...	SELECT MAX Attendance FROM table WHERE Opponen...
8418	Which party has Peter A. Quinn as a representa...	SELECT Party FROM table WHERE Representative =...
8419	Which state does Jimmy Quillen represent?	SELECT State FROM table WHERE Representative =...
8420	What is the lifespan of the democratic party i...	SELECT Lifespan FROM table WHERE Party = democ...

8421 rows × 2 columns

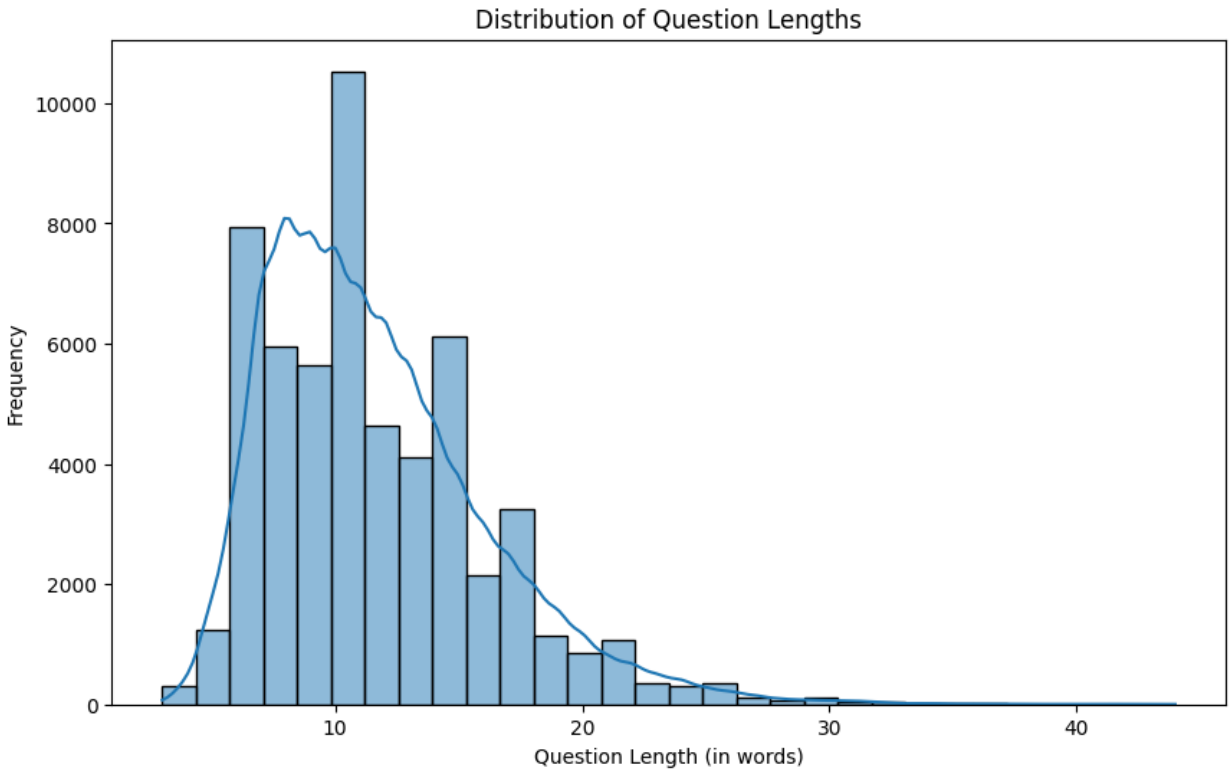
```
testDataFrame
```

	question	sql
0	What is terrence ross' nationality	SELECT Nationality FROM table WHERE Player = T...
1	What clu was in toronto 1995-96	SELECT School/Club Team FROM table WHERE Years...
2	which club was in toronto 2003-06	SELECT School/Club Team FROM table WHERE Years...
3	how many schools or teams had jalen rose	SELECT COUNT School/Club Team FROM table WHERE...
4	Where was Assen held?	SELECT Round FROM table WHERE Circuit = Assen
...	...	...
15873	After 1972, how many points did Marlboro Team ...	SELECT Points FROM table WHERE Year > 1972 AND...
15874	What chassis had 39 points?	SELECT Chassis FROM table WHERE Points = 39
15875	How many points did the Ford V8 with a Tyrrell...	SELECT Points FROM table WHERE Engine = ford v...
15876	Before 1976 and with 12 points, what chassis d...	SELECT Chassis FROM table WHERE Engine = ford ...
15877	What year did Elf Team Tyrrell have 39 points ...	SELECT SUM Year FROM table WHERE Entrant = elf...

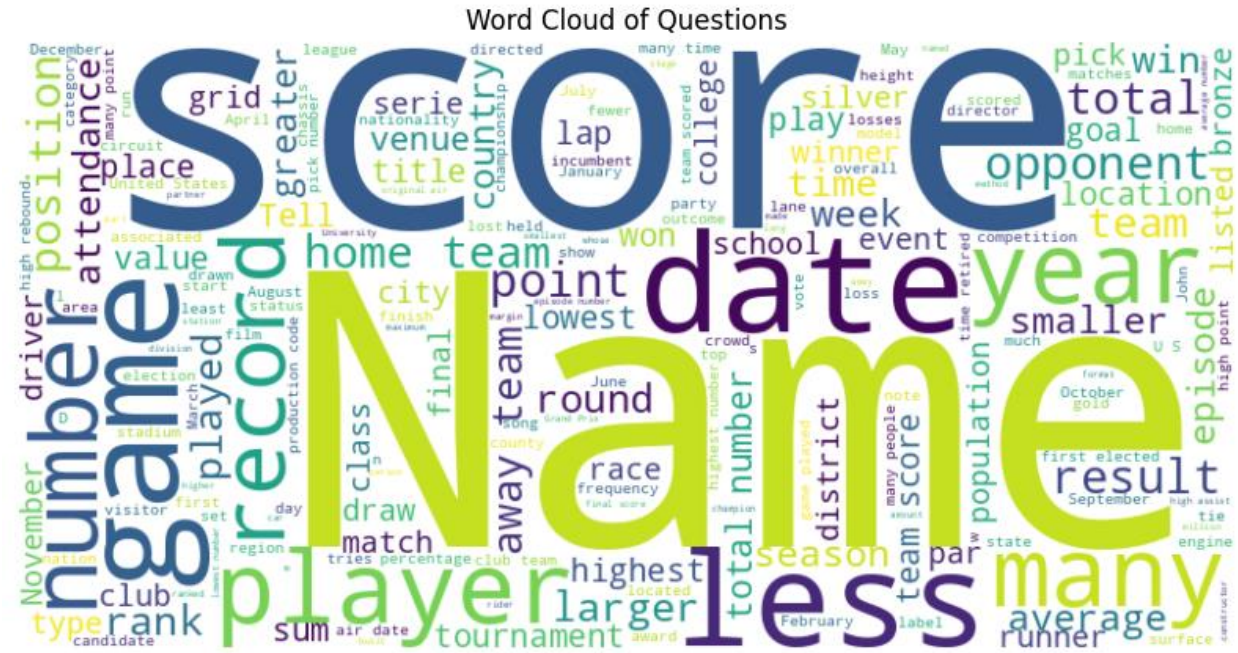
15878 rows × 2 columns

DATA PREPROCESSING:

EDA (DATA VISULIZATION):

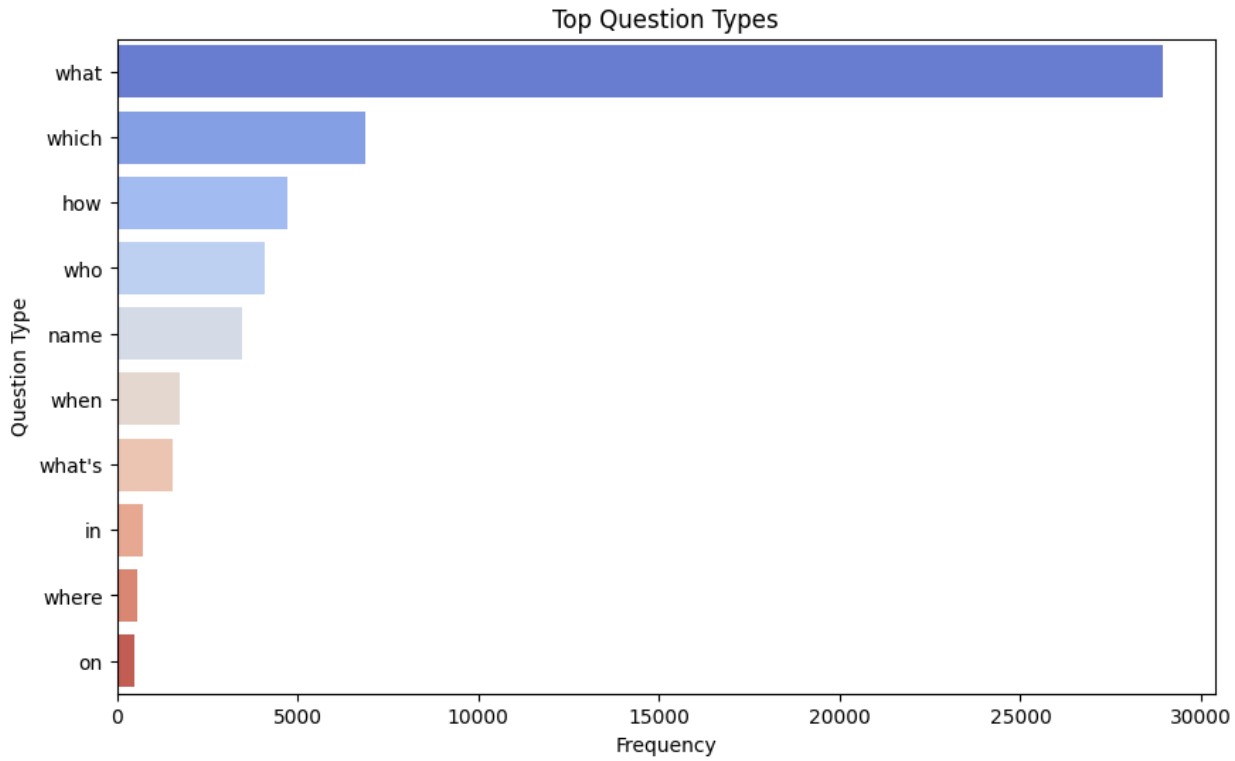


The distribution of question lengths can be observed in the graph. The word length of the question is plotted on the x-axis, while the frequency is plotted on the y-axis. Remarkably, most of the questions are between 5 and 10 words long, with an average of 8 words. Question lengths longer than 20 words result in a dramatic decline in frequency. This distribution is depicted in a bar graph, and the trend is more smoothly visualized with an overlaying blue line graph. Overall, the evidence points to a higher frequency of short questions and a lower frequency of longer, more complicated ones. In conclusion, the graph emphasizes the frequency of shorter questions in the dataset and shows the relationship between question length and frequency.

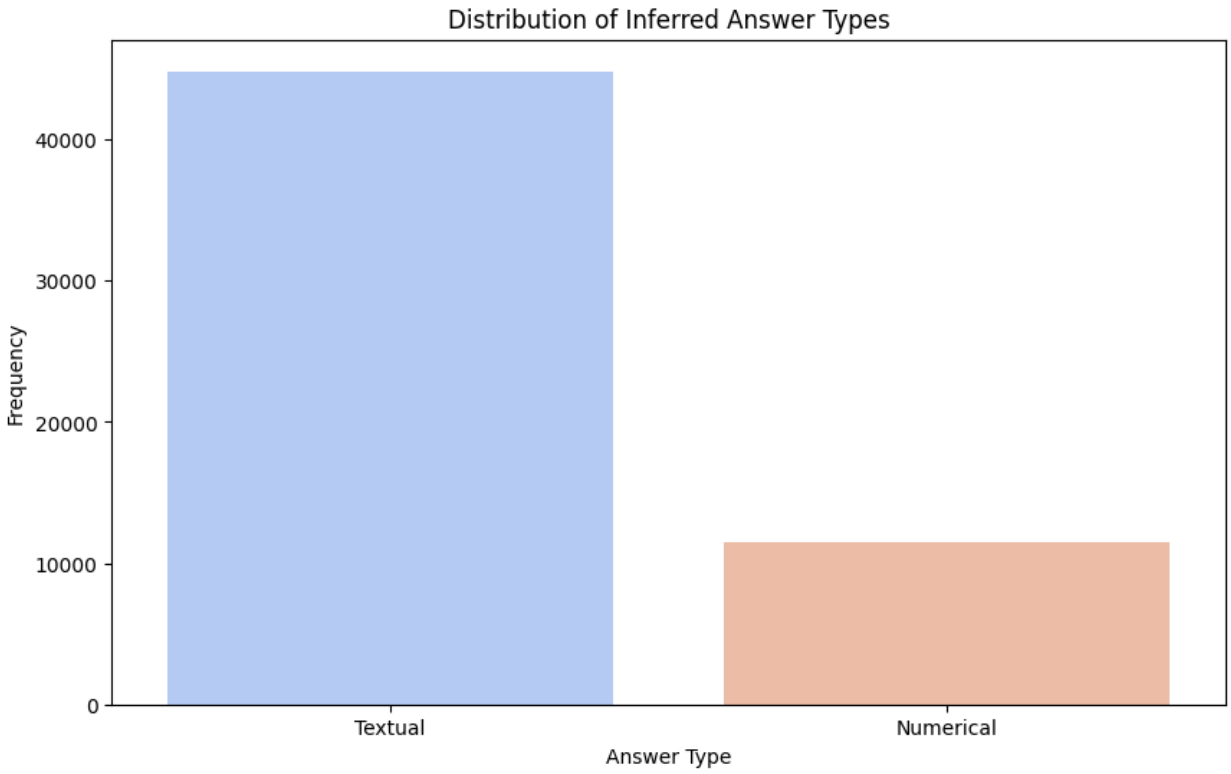


The graphic displays a colorful word cloud with a variety of terms most likely related to queries, games, sports, and general knowledge. The words are shown in a variety of colors and sizes to indicate their importance or frequency. Important terms like "score," "game," "name," and "date" stand out and imply that they are frequently used in inquiries. This broad range of terms provides information on a variety of subjects and questions.

The graphic displays a colorful word cloud with a variety of terms most likely related to queries, games, sports, and general knowledge. The words are shown in a variety of colors and sizes to indicate their importance or frequency. Important terms like "score," "game," "name," and "date" stand out and imply that they are frequently used in inquiries. This broad range of terms provides information on a variety of subjects and questions.

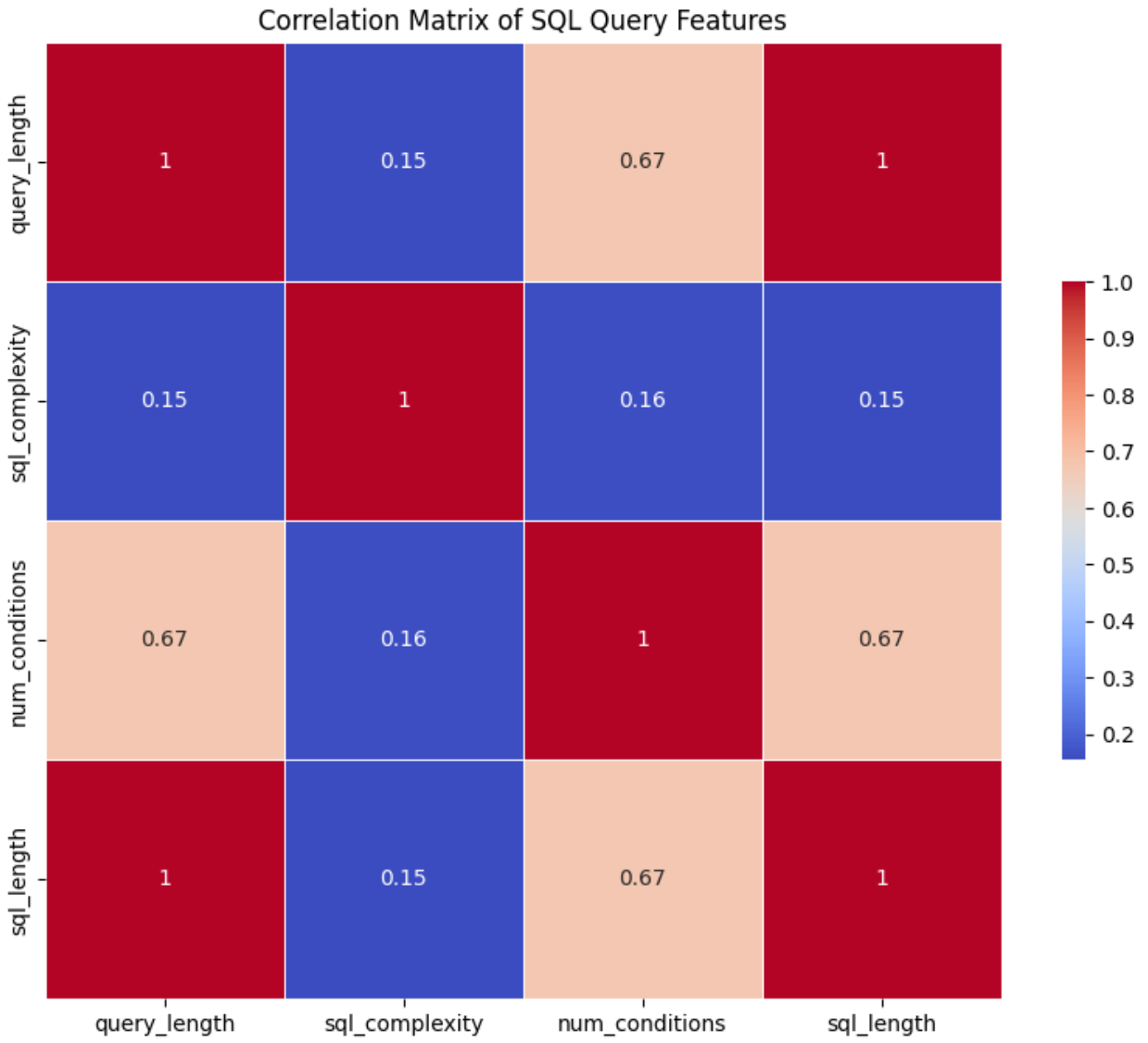


The "Top Question Types" graph provides information about how frequently various question kinds are asked. The type "what" is the most prevalent among them, appearing over 25,000 times, and is indicated by the longest blue bar. The frequencies of other inquiry categories, such "which," "how," and "who," are noticeably lower. Less common inquiry categories are also indicated with a lighter color or red, such as "name," "when," and "what's." All things considered, this graph clarifies the distribution of question kinds within the dataset.



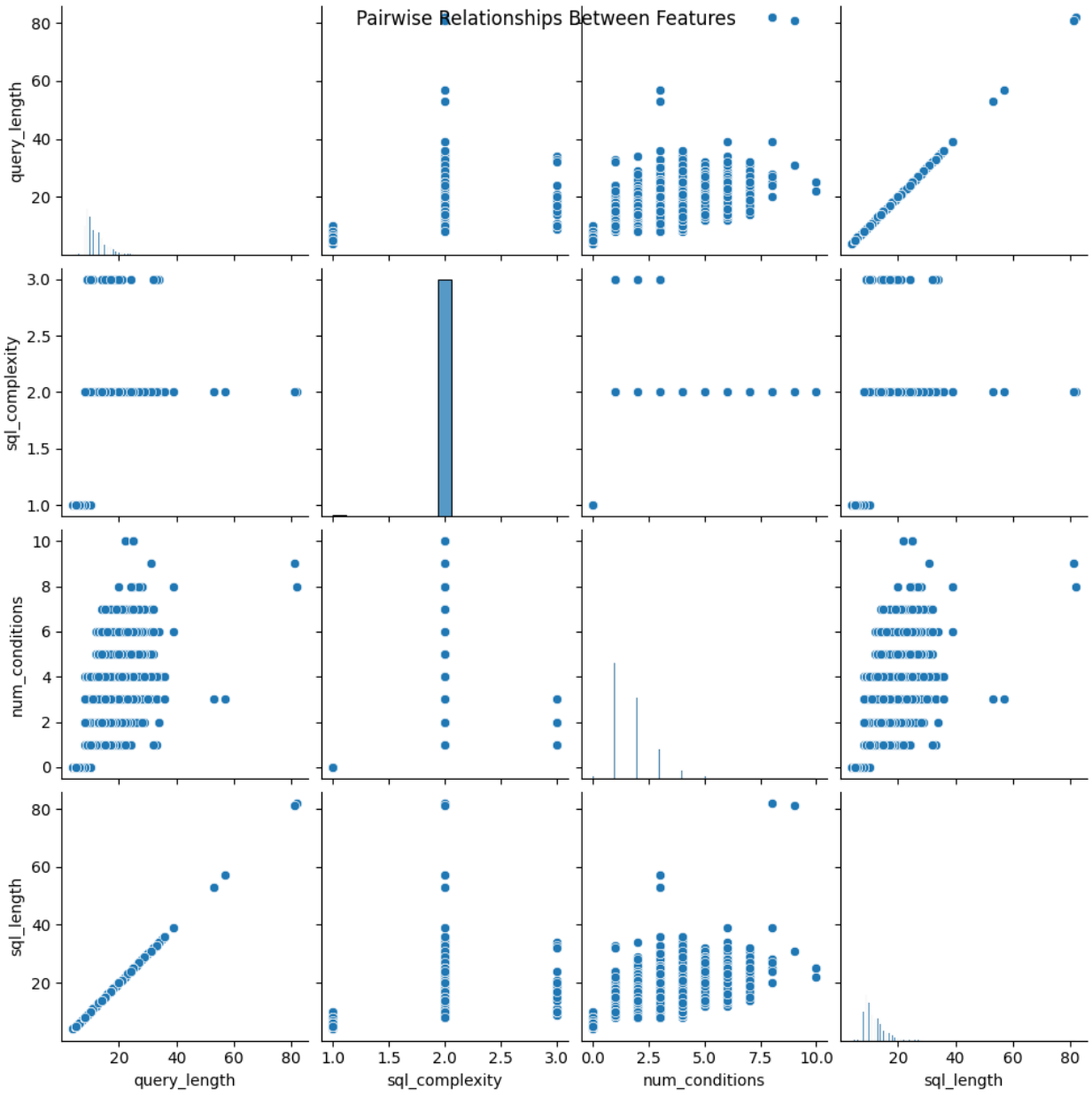
The bar graph, titled "Distribution of Inferred Answer Types," illustrates the categorization of answers derived from a particular dataset, highlighting the frequency of each type. It shows two main categories of inferred answers: textual and numerical. The most notable category, represented by a blue bar, is the textual answers, reaching approximately 40,000 on the frequency scale, making it the predominant type within the dataset.

- This suggests that the majority of the answers derived from the dataset are in text form. On the other hand, numerical answers, depicted by a peach-colored bar, ascend to about 10,000 on the same axis. This indicates that numerical answers, while significant, are considerably less common compared to their textual counterparts.



The links between different SQL query features, such as Query Length, SQL Complexity, Number of Conditions, and SQL Length, are visualized in the supplied correlation matrix. The correlation coefficient between these features is displayed in each cell of the matrix; values range from 0.15, which denotes poor association, to 1, which denotes significant correlation. Color coding is used in the matrix; blue indicates weak or no relationships while red indicates significant positive correlations.

- Notably, the correlation between query length and SQL complexity is just 0.15, indicating a significantly weaker association, whereas the correlation between query length and SQL complexity is 1, suggesting a perfect positive correlation. Using this tool will help you better understand how various SQL query characteristics work together, which will help you develop query optimization techniques.



The provided scatterplot matrix shows the pairwise relationships between various SQL query features such as Query Length, SQL Complexity, Number of Conditions, and SQL Length. Each plot reveals the distribution and correlation of two features, where the diagonal histograms show the distribution of a single feature. There appears to be a particularly strong positive correlation between Query Length and SQL Length, as indicated by the upward trend in their scatterplot. The plots with SQL Complexity show very discrete, low-variability clusters, suggesting limited distinct values for this feature. This type of visual analysis is key for understanding feature behavior and can guide database optimization efforts.

Model Architecture:

Model 1: Transformers

```
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Embedding, Dense, LayerNormalization, Dropout, MultiHeadAttention

def transformer_encoder(inputs, head_size, num_heads, ff_dim, dropout_rate):
    attention = MultiHeadAttention(num_heads=num_heads, key_dim=head_size)(inputs, inputs)
    attention = Dropout(dropout_rate)(attention)
    attention = LayerNormalization(epsilon=1e-6)(attention + inputs)

    ff_output = Dense(ff_dim, activation='relu')(attention)
    ff_output = Dense(inputs.shape[-1])(ff_output)
    ff_output = Dropout(dropout_rate)(ff_output)
    encoder_output = LayerNormalization(epsilon=1e-6)(ff_output + attention)
    return encoder_output

def transformer_decoder(inputs, enc_output, head_size, num_heads, ff_dim, dropout_rate):
    self_attention = MultiHeadAttention(num_heads=num_heads, key_dim=head_size)(inputs, inputs)
    self_attention = Dropout(dropout_rate)(self_attention)
    self_attention = LayerNormalization(epsilon=1e-6)(self_attention + inputs)

    attention = MultiHeadAttention(num_heads=num_heads, key_dim=head_size)(self_attention, enc_output)
    attention = Dropout(dropout_rate)(attention)
    attention = LayerNormalization(epsilon=1e-6)(attention + self_attention)

    ff_output = Dense(ff_dim, activation='relu')(attention)
    ff_output = Dense(inputs.shape[-1])(ff_output)
    ff_output = Dropout(dropout_rate)(ff_output)
    decoder_output = LayerNormalization(epsilon=1e-6)(ff_output + attention)
    return decoder_output


embedding_dim = 256
vocab_size = 32128
head_size = 64
num_heads = 8
ff_dim = 512
dropout_rate = 0.1
max_length = 512

inputs_enc = Input(shape=(None,))
inputs_dec = Input(shape=(None,))

enc_emb = Embedding(input_dim=vocab_size, output_dim=embedding_dim)(inputs_enc)
dec_emb = Embedding(input_dim=vocab_size, output_dim=embedding_dim)(inputs_dec)

enc_output = transformer_encoder(enc_emb, head_size, num_heads, ff_dim, dropout_rate)
dec_output = transformer_decoder(dec_emb, enc_output, head_size, num_heads, ff_dim, dropout_rate)

final_output = Dense(vocab_size, activation='softmax')(dec_output)

model = Model(inputs=[inputs_enc, inputs_dec], outputs=final_output)
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')

model.summary()
```

```
model.summary()
```

Layer (type)	Output Shape	Param #	Connected to
multi_head_attention_1 (MultiHeadAttention)	(None, None, 256)	526080	['embedding_1[0][0]', 'embedding_1[0][0]']
dense_1 (Dense)	(None, None, 256)	131328	['dense[0][0]']
dropout_2 (Dropout)	(None, None, 256)	0	['multi_head_attention_1[0][0]']
dropout_1 (Dropout)	(None, None, 256)	0	['dense_1[0][0]']
tf.__operators__.add_2 (TFOpLambda)	(None, None, 256)	0	['dropout_2[0][0]', 'embedding_1[0][0]']
tf.__operators__.add_1 (TFOpLambda)	(None, None, 256)	0	['dropout_1[0][0]', 'layer_normalization[0][0]']
layer_normalization_2 (LayerNormalization)	(None, None, 256)	512	['tf.__operators__.add_2[0][0]']
layer_normalization_1 (LayerNormalization)	(None, None, 256)	512	['tf.__operators__.add_1[0][0]']
multi_head_attention_2 (MultiHeadAttention)	(None, None, 256)	526080	['layer_normalization_2[0][0]', 'layer_normalization_1[0][0]']
dropout_3 (Dropout)	(None, None, 256)	0	['multi_head_attention_2[0][0]']
tf.__operators__.add_3 (TFOpLambda)	(None, None, 256)	0	['dropout_3[0][0]', 'layer_normalization_2[0][0]']
layer_normalization_3 (LayerNormalization)	(None, None, 256)	512	['tf.__operators__.add_3[0][0]']
dense_2 (Dense)	(None, None, 512)	131584	['layer_normalization_3[0][0]']
dense_3 (Dense)	(None, None, 256)	131328	['dense_2[0][0]']
dropout_4 (Dropout)	(None, None, 256)	0	['dense_3[0][0]']
tf.__operators__.add_4 (TFOpLambda)	(None, None, 256)	0	['dropout_4[0][0]', 'layer_normalization_3[0][0]']
layer_normalization_4 (LayerNormalization)	(None, None, 256)	512	['tf.__operators__.add_4[0][0]']
dense_4 (Dense)	(None, None, 32128)	8256896	['layer_normalization_4[0][0]']

```
Total params: 26813056 (102.28 MB)
Trainable params: 26813056 (102.28 MB)
Non-trainable params: 0 (0.00 Byte)
```

The model described is a deep learning architecture with multiple layers, including embeddings, multi-head attention, dense, and dropout layers, primarily designed for sequence processing tasks. It utilizes an input of two different sequences, processed through separate embedding layers, each producing a 256-dimensional output. The architecture features several normalization and dropout steps to improve training stability and prevent overfitting. The final output is produced by a dense layer with 32,128 units, indicating a large vocabulary size, typical for tasks like language modeling or machine translation.

```
input_sentence = "what is the age of the girl named lucky"
predicted_sql_query = predict_sql(input_sentence, tokenizer, model, max_length=20)
predicted_sql_query
```

```
1/1 [==============================] - 0s 134ms/step
1/1 [==============================] - 0s 82ms/step
1/1 [==============================] - 0s 62ms/step
1/1 [==============================] - 0s 78ms/step
1/1 [==============================] - 0s 94ms/step
1/1 [==============================] - 0s 73ms/step
1/1 [==============================] - 0s 68ms/step
1/1 [==============================] - 0s 86ms/step
1/1 [==============================] - 0s 89ms/step
1/1 [==============================] - 0s 151ms/step
1/1 [==============================] - 0s 65ms/step
1/1 [==============================] - 0s 60ms/step
1/1 [==============================] - 0s 66ms/step
1/1 [==============================] - 0s 73ms/step
1/1 [==============================] - 0s 64ms/step
1/1 [==============================] - 0s 61ms/step
1/1 [==============================] - 0s 60ms/step
1/1 [==============================] - 0s 66ms/step
1/1 [==============================] - 0s 59ms/step
'select select age age age from table lucky = name '
```

Optimization:

Earlystopping:

Early stopping is configured to monitor the validation loss, waiting for 10 epochs before halting training if no improvement is seen, and it restores the weights to those of the best performing epoch. The learning rate reduction strategy is set to reduce the learning rate by a factor of 0.2 if there's no improvement in validation loss after 5 epochs, with a minimum learning rate set to 0.0001. The model uses the RMSprop optimizer with a starting learning rate of 0.001, which is suitable for a variety of tasks, and is compiled with a loss function tailored for classifications tasks involving large output spaces, such as 'sparse_categorical_crossentropy'.

```
input_sentence = "what is the age of the girl named lucky"
predicted_sql_query = predict_sql(input_sentence, tokenizer, model, max_length=20)
predicted_sql_query
```

```
1/1 [==============================] - 0s 65ms/step
1/1 [==============================] - 0s 71ms/step
1/1 [==============================] - 0s 65ms/step
1/1 [==============================] - 0s 66ms/step
1/1 [==============================] - 0s 70ms/step
1/1 [==============================] - 0s 63ms/step
1/1 [==============================] - 0s 80ms/step
1/1 [==============================] - 0s 68ms/step
1/1 [==============================] - 0s 73ms/step
1/1 [==============================] - 0s 60ms/step
1/1 [==============================] - 0s 63ms/step
1/1 [==============================] - 0s 66ms/step
1/1 [==============================] - 0s 71ms/step
1/1 [==============================] - 0s 69ms/step
1/1 [==============================] - 0s 85ms/step
1/1 [==============================] - 0s 63ms/step
1/1 [==============================] - 0s 71ms/step
1/1 [==============================] - 0s 76ms/step
1/1 [==============================] - 0s 70ms/step
'select from table table tablet table table table'
```

Learning rate:

In the second optimization approach, used similar callbacks but with a focus on testing higher initial learning rates. The RMSprop optimizer is set with a learning rate of 0.5 to investigate the model's behavior with faster initial learning phases. EarlyStopping is again utilized to monitor validation loss, stopping the training after ten epochs without improvement to safeguard against overfitting. ReduceLROnPlateau is configured to reduce the learning rate when the validation loss does not improve, ensuring fine-grained control over the learning process. This setup aims to balance rapid convergence with the stability needed to achieve reliable model performance.

```
input_sentence = "what is the age of the girl named lucky"
predicted_sql_query = predict_sql(input_sentence, tokenizer, model, max_length=20)
predicted_sql_query
```

```
1/1 [==============================] - 0s 70ms/step
1/1 [==============================] - 0s 99ms/step
1/1 [==============================] - 0s 88ms/step
1/1 [==============================] - 0s 78ms/step
1/1 [==============================] - 0s 77ms/step
1/1 [==============================] - 0s 111ms/step
1/1 [==============================] - 0s 62ms/step
1/1 [==============================] - 0s 65ms/step
1/1 [==============================] - 0s 60ms/step
1/1 [==============================] - 0s 69ms/step
1/1 [==============================] - 0s 81ms/step
1/1 [==============================] - 0s 64ms/step
1/1 [==============================] - 0s 63ms/step
1/1 [==============================] - 0s 63ms/step
1/1 [==============================] - 0s 40ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 52ms/step
1/1 [==============================] - 0s 59ms/step
1/1 [==============================] - 0s 64ms/step
'the age of the player lucky lucky lucky what is the age of the player lucky what is the age of the player lucky what is the age of the player lucky what is th
lucky what is the age of the player lucky what is the age of the player lucky what is'
```

Model 2: Encoder, Decoder, LSTM with Attention layers

```
from transformers import T5Tokenizer


tokenizer = T5Tokenizer.from_pretrained('t5-small')
def tokenize_texts(text_list, max_length=512):
    return tokenizer(
        text_list,
        max_length=max_length,
        padding='max_length',
        truncation=True,
        return_tensors='np'
    )


X_tokenized = tokenize_texts(X)
y_tokenized = tokenize_texts(y)
```

```
Special tokens have been added in the vocabulary, make sure the associated word embeddings are fine-tuned or trained.
```

1. Embedding Layer: This layer is fundamental as it transforms the discrete tokens (words from natural language queries and SQL commands) into dense vectors. This representation is crucial as it embeds semantic meaning into each token, making the model's subsequent processing more effective. The embeddings being shared between the encoder and decoder ensure uniformity and efficiency, allowing both parts of the model to operate in a cohesive manner.

2. Masking Layer: Applied separately to both the encoder and decoder embeddings, this layer ensures that padding added to equalize sequence lengths does not affect the model's learning. By ignoring these padded values, the model focuses solely on meaningful data, enhancing the accuracy and efficiency of translation.

3. LSTM Layers: Employed in both the encoder and decoder, these layers are key for managing sequence data. The encoder LSTM processes the input natural language .

```
vocab_size = tokenizer.vocab_size
embedding_dim = 64 # Embedding size
latent_dim = 512     # LSTM units

embedding_layer = Embedding(input_dim=vocab_size, output_dim=embedding_dim)


encoder_inputs = Input(shape=(None,), dtype='int32', name='encoder_inputs')
encoder_embedded = embedding_layer(encoder_inputs)
encoder_masking = Masking(mask_value=tokenizer.pad_token_id)(encoder_embedded)
encoder_lstm = LSTM(latent_dim, return_state=True, name='encoder_lstm')
encoder_outputs, state_h, state_c = encoder_lstm(encoder_masking)
encoder_states = [state_h, state_c]


decoder_inputs = Input(shape=(None,), dtype='int32', name='decoder_inputs')
decoder_embedded = embedding_layer(decoder_inputs)
decoder_masking = Masking(mask_value=tokenizer.pad_token_id)(decoder_embedded)
decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True, name='decoder_lstm')
decoder_outputs, _, _ = decoder_lstm(decoder_masking, initial_state=encoder_states)


attention_layer = Attention(name='attention_layer')
attention_result = attention_layer([decoder_outputs, encoder_outputs])


decoder_concat_input = Concatenate(axis=-1, name='concat_layer')([decoder_outputs, attention_result])


decoder_dense = Dense(vocab_size, activation='softmax', name='output_layer')
decoder_outputs = decoder_dense(decoder_concat_input)


model = Model([encoder_inputs, decoder_inputs], decoder_outputs)
model.compile(optimizer=RMSprop(learning_rate=0.001), loss='sparse_categorical_crossentropy')


model.summary()
```

The model summary featuring an encoder-decoder structure with LSTM layers and an attention mechanism. It employs shared embeddings (Embedding layer with 2,048,000 parameters) for both the encoder and decoder inputs, facilitating the efficient handling of input sequences. The encoder LSTM compresses the input sequence into a context state, while the decoder LSTM uses this context along with its inputs to generate predictions step-by-step, enhanced by the attention mechanism that dynamically focuses on different parts of the input sequence. The final output is generated by a dense layer with a substantial number of parameters (32,000 classes multiplied by the 1024 outputs from the concatenated layer), making this model particularly suited for tasks with large output vocabularies, such as machine translation or text generation.

```
Model: "model_5"
__________________________________________________________________________________________________
 Layer (type)                Output Shape                 Param #   Connected to
==================================================================================================
 decoder_inputs (InputLayer  [(None, None)]               0         []
 )

 encoder_inputs (InputLayer  [(None, None)]               0         []
 )

 embedding_5 (Embedding)     (None, None, 64)             2048000   ['encoder_inputs[0][0]',
                                                                     'decoder_inputs[0][0]']

 masking_10 (Masking)        (None, None, 64)             0         ['embedding_5[0][0]']

 masking_11 (Masking)        (None, None, 64)             0         ['embedding_5[1][0]']

 encoder_lstm (LSTM)         [(None, 512),                1181696   ['masking_10[0][0]']
                              (None, 512),
                              (None, 512)]

 decoder_lstm (LSTM)         [(None, None, 512),          1181696   ['masking_11[0][0]',
                              (None, 512),                           'encoder_lstm[0][1]',
                              (None, 512)]                           'encoder_lstm[0][2]']

 attention_layer (Attention  (None, None, 512)            0         ['decoder_lstm[0][0]',
 )                                                                   'encoder_lstm[0][0]']

 concat_layer (Concatenate)  (None, None, 1024)           0         ['decoder_lstm[0][0]',
                                                                     'attention_layer[0][0]']

 output_layer (Dense)        (None, None, 32000)          3280000   ['concat_layer[0][0]']
                                                          0

==================================================================================================
Total params: 37211392 (141.95 MB)
Trainable params: 37211392 (141.95 MB)
Non-trainable params: 0 (0.00 Byte)
__________________________________________________________________________________________________
```

```
input_text = "what is the age of the girl lucky"
predicted_sql = predict_sql_query(input_text, tokenizer, model)
print(predicted_sql)
```

```
1/1 [==============================] - 2s 2s/step
1/1 [==============================] - 3s 3s/step
1/1 [==============================] - 0s 35ms/step
1/1 [==============================] - 0s 35ms/step
1/1 [==============================] - 0s 35ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 37ms/step
1/1 [==============================] - 0s 36ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 46ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 39ms/step
1/1 [==============================] - 0s 36ms/step
1/1 [==============================] - 0s 39ms/step
1/1 [==============================] - 0s 39ms/step
1/1 [==============================] - 0s 42ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 23ms/step
select from table where      <pad><pad><pad><pad><pad><pad><pad><pad><pad><pad>
```

Optimization:

Early stopping:

In this strategy, we employed the EarlyStopping and ReduceLROnPlateau callbacks to enhance training efficiency. EarlyStopping monitors validation loss and halts training if no improvement occurs over five epochs, effectively preventing overfitting by restoring the best model weights. Alongside, ReduceLROnPlateau adjusts the learning rate by a factor of 0.2 if no improvement in validation loss is observed for two epochs, with a floor set at 0.0001. This dynamic adjustment helps in refining the learning process, allowing for finer steps towards optimal loss minimization. These methods collectively ensure that the model achieves high accuracy without wasting computational resources.

```
early_stopping = EarlyStopping(monitor='val_loss', patience=10, verbose=1, mode='min', restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=0.0001, verbose=1, mode='min')
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)
optimizer = RMSprop(learning_rate=0.001, rho=0.9, epsilon=3e-03)

model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy')


history = model.fit(
    [X_train, decoder_input_data_train],
    decoder_output_data_train,
    batch_size=8,
    epochs=10,
    validation_data=(
        [X_val, decoder_input_data_val],
        decoder_output_data_val
    ),
     callbacks=[reduce_lr, early_stopping]
)
```

```
input_text = "what is the age of the girl lucky"
predicted_sql = predict_sql_query(input_text, tokenizer, model)
print(predicted_sql)
```

```
1/1 [==============================] - 7s 7s/step
1/1 [==============================] - 7s 7s/step
1/1 [==============================] - 0s 105ms/step
1/1 [==============================] - 0s 76ms/step
1/1 [==============================] - 0s 86ms/step
1/1 [==============================] - 0s 56ms/step
1/1 [==============================] - 0s 126ms/step
1/1 [==============================] - 0s 64ms/step
1/1 [==============================] - 0s 132ms/step
1/1 [==============================] - 0s 155ms/step
1/1 [==============================] - 0s 118ms/step
1/1 [==============================] - 0s 113ms/step
1/1 [==============================] - 0s 121ms/step
1/1 [==============================] - 0s 127ms/step
1/1 [==============================] - 0s 96ms/step
1/1 [==============================] - 0s 64ms/step
1/1 [==============================] - 0s 110ms/step
1/1 [==============================] - 0s 71ms/step
1/1 [==============================] - 0s 86ms/step
1/1 [==============================] - 0s 83ms/step
select select from table table where =<pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad>
```

Learning rate:

In the second optimization approach, used similar callbacks but with a focus on testing higher initial learning rates. The RMSprop optimizer is set with a learning rate of 0.5 to investigate the model's behavior with faster initial learning phases. EarlyStopping is again utilized to monitor validation loss, stopping the training after ten epochs without improvement to safeguard against overfitting. ReduceLROnPlateau is configured to reduce the learning rate when the validation loss does not improve, ensuring fine-grained control over the learning process. This setup aims to balance rapid convergence with the stability needed to achieve reliable model performance.

```
early_stopping = EarlyStopping(monitor='val_loss', patience=10, verbose=1, mode='min', restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=0.0001, verbose=1, mode='min')
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)
optimizer = RMSprop(learning_rate=5e-01, rho=0.8, epsilon=5e-01)

model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy')


# Training the model
history = model.fit(
    [X_train, decoder_input_data_train],
    decoder_output_data_train,
    batch_size=8,
    epochs=10,
    validation_data=(
        [X_val, decoder_input_data_val],
        decoder_output_data_val
    ),
     callbacks=[reduce_lr, early_stopping]
)
```

```
input_text = "what is the age of the girl lucky"
predicted_sql = predict_sql_query(input_text, tokenizer, model)
print(predicted_sql)
```

```
1/1 [==============================] - 3s 3s/step
1/1 [==============================] - 3s 3s/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 22ms/step
select from table where = =<pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad>
```

Model 3: Encoder, Decoder, GRU with Attention layers

```python
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, GRU, Dense, Embedding, Concatenate, Attention, Masking
from tensorflow.keras.optimizers import RMSprop


vocab_size = tokenizer.vocab_size
embedding_dim = 256
latent_dim = 512


embedding_layer = Embedding(input_dim=vocab_size, output_dim=embedding_dim)


encoder_inputs = Input(shape=(None,), dtype='int32', name='encoder_inputs')
encoder_embedded = embedding_layer(encoder_inputs)
encoder_gru = GRU(latent_dim, return_sequences=True, return_state=True, name='encoder_gru')
encoder_outputs, state_h = encoder_gru(encoder_embedded)


decoder_inputs = Input(shape=(None,), dtype='int32', name='decoder_inputs')
decoder_embedded = embedding_layer(decoder_inputs)
decoder_gru = GRU(latent_dim, return_sequences=True, return_state=True, name='decoder_gru')
decoder_outputs, _ = decoder_gru(decoder_embedded, initial_state=state_h)


attention_layer = Attention(name='attention_layer')
attention_result = attention_layer([decoder_outputs, encoder_outputs])


decoder_concat_input = Concatenate(axis=-1, name='concat_layer')([decoder_outputs, attention_result])


decoder_dense = Dense(vocab_size, activation='softmax', name='output_layer')
decoder_outputs = decoder_dense(decoder_concat_input)


model = Model([encoder_inputs, decoder_inputs], decoder_outputs)


model.compile(optimizer=RMSprop(learning_rate=0.001), loss='sparse_categorical_crossentropy')
model.summary()
```

```
Model: "model_3"
__________________________________________________________________________________________________
 Layer (type)                Output Shape              Param #   Connected to
==================================================================================================
 decoder_inputs (InputLayer  [(None, None)]            0         []
 )

 encoder_inputs (InputLayer  [(None, None)]            0         []
 )

 embedding_1 (Embedding)     (None, None, 256)         8192000   ['encoder_inputs[0][0]',
                                                                  'decoder_inputs[0][0]']

 encoder_gru (GRU)           [(None, None, 512),       1182720   ['embedding_1[0][0]']
                              (None, 512)]

 decoder_gru (GRU)           [(None, None, 512),       1182720   ['embedding_1[1][0]',
                              (None, 512)]                        'encoder_gru[0][1]']

 attention_layer (Attention  (None, None, 512)         0         ['decoder_gru[0][0]',
 )                                                                'encoder_gru[0][0]']

 concat_layer (Concatenate)  (None, None, 1024)        0         ['decoder_gru[0][0]',
                                                                  'attention_layer[0][0]']

 output_layer (Dense)        (None, None, 32000)       3280000   ['concat_layer[0][0]']
                                                       0

==================================================================================================
Total params: 43357440 (165.40 MB)
Trainable params: 43357440 (165.40 MB)
Non-trainable params: 0 (0.00 Byte)
__________________________________________________________________________________________________
```

The GRU model consists of an encoder and a decoder, both utilizing GRU (Gated Recurrent Unit) layers, with the encoder transforming the input sequence into a context vector and the decoder generating the output sequence. An attention mechanism is employed to enhance the decoder's performance by focusing on specific parts of the input sequence during each step of the decoding process. The model is substantial in size, with over 43 million parameters, indicating its capacity to handle complex language patterns and large vocabulary sizes.

```
input_sentence = "select score from table."
predicted_sql_query = predict_sql(input_sentence, tokenizer, model, max_length=20)
```

```
1/1 [==============================] - 0s 70ms/step
1/1 [==============================] - 0s 66ms/step
1/1 [==============================] - 0s 75ms/step
1/1 [==============================] - 0s 64ms/step
1/1 [==============================] - 0s 59ms/step
1/1 [==============================] - 0s 86ms/step
1/1 [==============================] - 0s 52ms/step
1/1 [==============================] - 0s 67ms/step
1/1 [==============================] - 0s 33ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 50ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 38ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 68ms/step
```

```
print("Predicted SQL Query:", predicted_sql_query)
```

```
Predicted SQL Query: select min by ( ())))))))))))))
```

Optimization:

Early stopping:

Early stopping is configured to monitor the validation loss, waiting for 10 epochs before halting training if no improvement is seen, and it restores the weights to those of the best performing epoch. The learning rate reduction strategy is set to reduce the learning rate by a factor of 0.2 if there's no improvement in validation loss after 5 epochs, with a minimum learning rate set to 0.0001. The model uses the RMSprop optimizer with a starting learning rate of 0.001, which is suitable for a variety of tasks, and is compiled with a loss function tailored for classifications tasks involving large output spaces, such as 'sparse_categorical_crossentropy'.

```
input_sentence = "select score from table."
predicted_sql_query = predict_sql(input_sentence, tokenizer, model, max_length=20)
```

```
1/1 [==============================] - 0s 70ms/step
1/1 [==============================] - 0s 66ms/step
1/1 [==============================] - 0s 75ms/step
1/1 [==============================] - 0s 64ms/step
1/1 [==============================] - 0s 59ms/step
1/1 [==============================] - 0s 86ms/step
1/1 [==============================] - 0s 52ms/step
1/1 [==============================] - 0s 67ms/step
1/1 [==============================] - 0s 33ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 50ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 38ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 68ms/step
```

```
print("Predicted SQL Query:", predicted_sql_query)
```

```
Predicted SQL Query: select min by ( ()))))))))))))))
```

Learning rate:

In the second optimization approach, used similar callbacks but with a focus on testing higher initial learning rates. The RMSprop optimizer is set with a learning rate of 0.5 to investigate the model's behavior with faster initial learning phases. EarlyStopping is again utilized to monitor validation loss, stopping the training after ten epochs without improvement to safeguard against overfitting. ReduceLROnPlateau is configured to reduce the learning rate when the validation loss does not improve, ensuring fine-grained control over the learning process. This setup aims to balance rapid convergence with the stability needed to achieve reliable model performance.

```
input_text = "what is the age of the girl lucky"
predicted_sql = predict_sql_query(input_text, tokenizer, model)
print(predicted_sql)
```

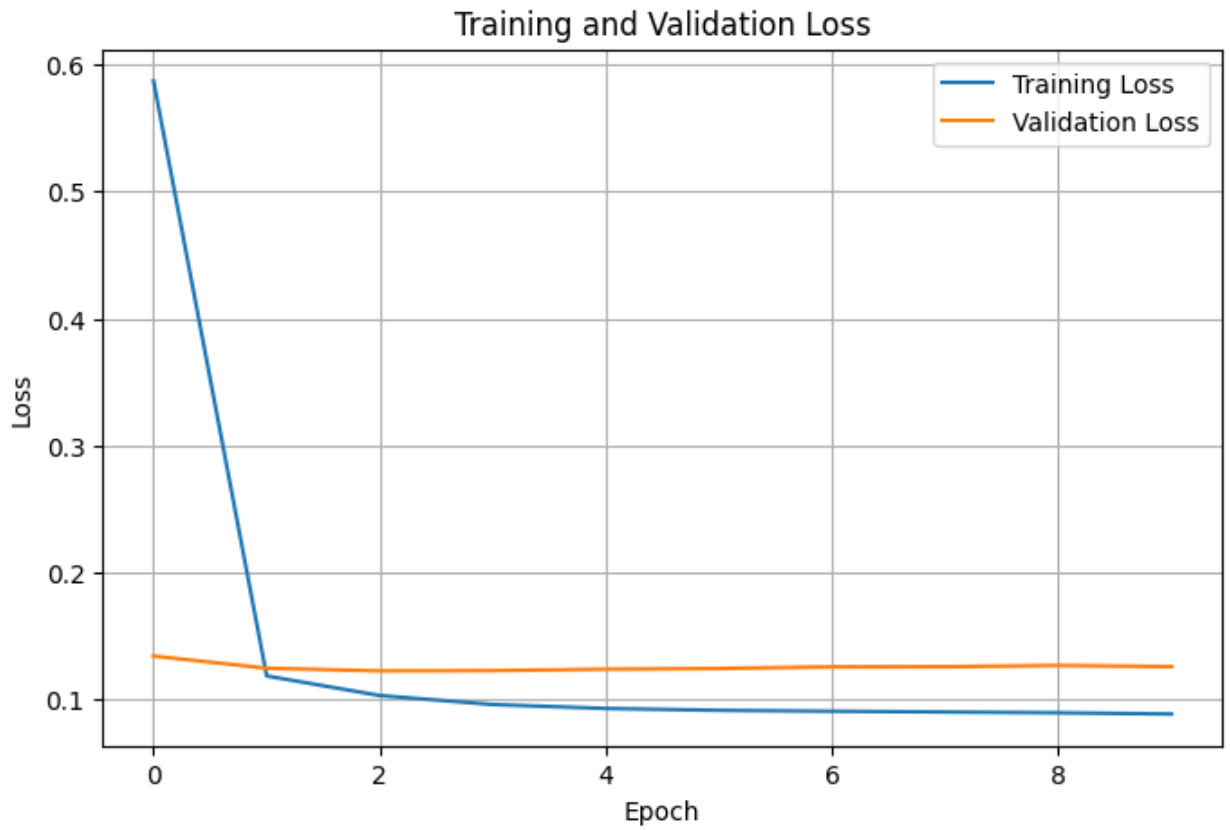
```
1/1 [==============================] - 1s 1s/step
1/1 [==============================] - 1s 1s/step
1/1 [==============================] - 0s 57ms/step
1/1 [==============================] - 0s 59ms/step
1/1 [==============================] - 0s 65ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 76ms/step
1/1 [==============================] - 0s 37ms/step
1/1 [==============================] - 0s 42ms/step
1/1 [==============================] - 0s 51ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 36ms/step
1/1 [==============================] - 0s 60ms/step
1/1 [==============================] - 0s 57ms/step
1/1 [==============================] - 0s 80ms/step
select select age age age age age good  from bad from from fromfrom from from from from from from from
```

Results:

Model 1: Transformers

```
]
   history = model.fit(
       [X_train, y_train_shifted],
       y_train,
       validation_data=([X_val, y_val_shifted], y_val),
       batch_size=256,
       epochs=10
   )
```

```
Epoch 1/10
250/250 [==============================] - 64s 224ms/step - loss: 0.5872 - val_loss: 0.1339
Epoch 2/10
250/250 [==============================] - 42s 167ms/step - loss: 0.1182 - val_loss: 0.1243
Epoch 3/10
250/250 [==============================] - 41s 165ms/step - loss: 0.1028 - val_loss: 0.1222
Epoch 4/10
250/250 [==============================] - 41s 164ms/step - loss: 0.0957 - val_loss: 0.1224
Epoch 5/10
250/250 [==============================] - 40s 158ms/step - loss: 0.0926 - val_loss: 0.1235
Epoch 6/10
250/250 [==============================] - 41s 163ms/step - loss: 0.0911 - val_loss: 0.1241
Epoch 7/10
250/250 [==============================] - 40s 158ms/step - loss: 0.0904 - val_loss: 0.1254
Epoch 8/10
250/250 [==============================] - 40s 161ms/step - loss: 0.0898 - val_loss: 0.1255
Epoch 9/10
250/250 [==============================] - 41s 163ms/step - loss: 0.0892 - val_loss: 0.1265
Epoch 10/10
250/250 [==============================] - 40s 161ms/step - loss: 0.0882 - val_loss: 0.1256
```



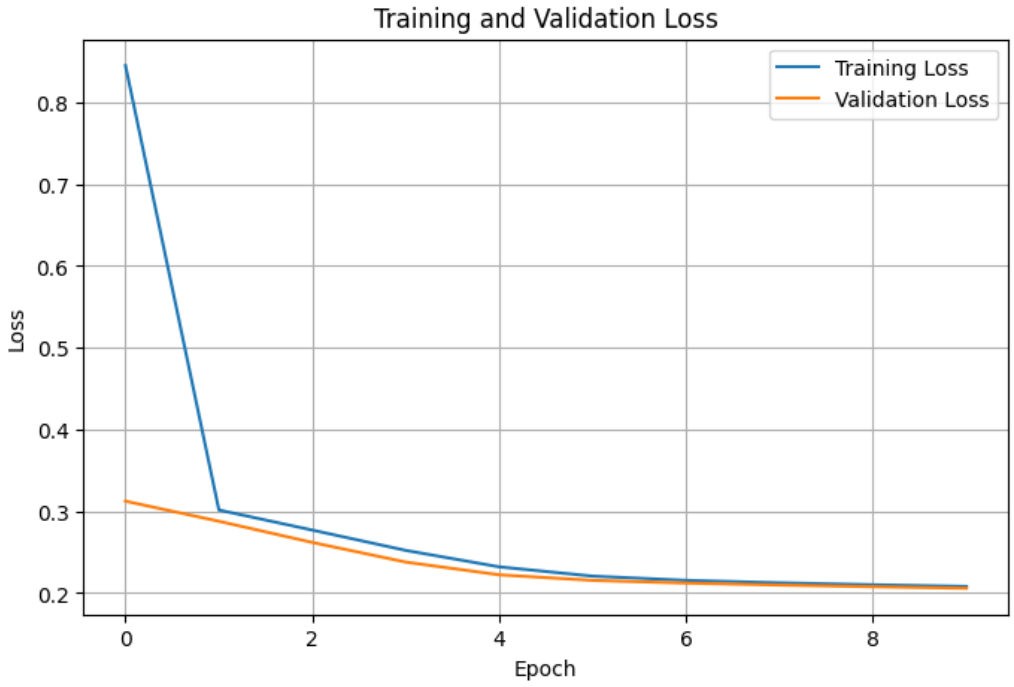
The graph shows the training and validation loss for a model over 8 epochs. Initially, the training loss drops sharply from around 0.5 to just below 0.2 within the first epoch and then continues to decrease slowly, stabilizing at approximately 0.1. The validation loss, on the other hand, starts lower at about 0.3 and converges closely to the training loss, ending slightly higher at around 0.1 after a few epochs. This pattern suggests the model is training effectively without significant overfitting, as indicated by the parallel and low divergence between training and validation loss curves.

Optimization:

EarlyStopping:

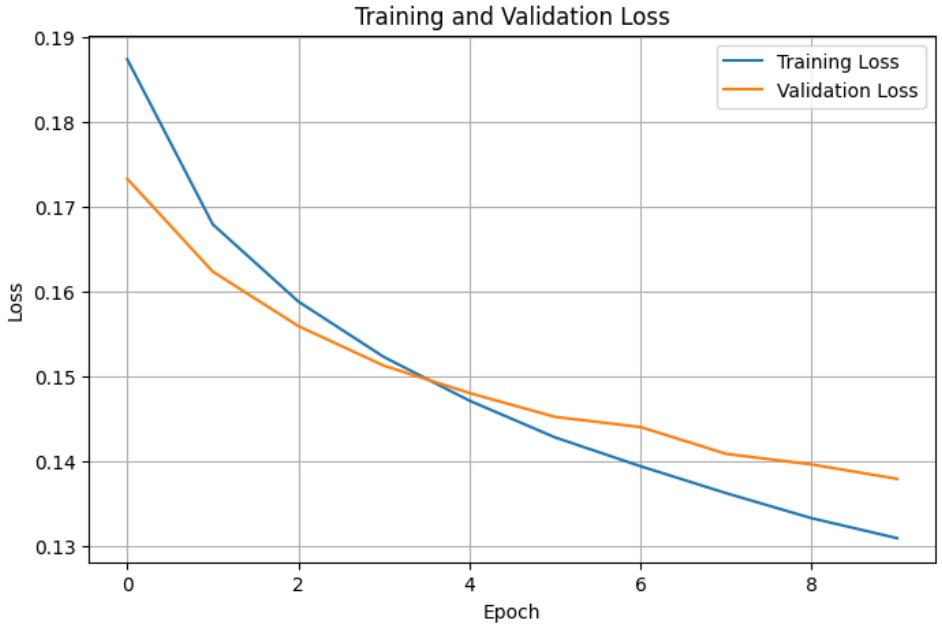
```
Epoch 1/10
250/250 [==============================] - 52s 172ms/step - loss: 0.8453 - val_loss: 0.3124 - lr: 0.0010
Epoch 2/10
250/250 [==============================] - 37s 150ms/step - loss: 0.3016 - val_loss: 0.2878 - lr: 0.0010
Epoch 3/10
250/250 [==============================] - 37s 150ms/step - loss: 0.2770 - val_loss: 0.2619 - lr: 0.0010
Epoch 4/10
250/250 [==============================] - 38s 152ms/step - loss: 0.2522 - val_loss: 0.2378 - lr: 0.0010
Epoch 5/10
250/250 [==============================] - 38s 154ms/step - loss: 0.2320 - val_loss: 0.2223 - lr: 0.0010
Epoch 6/10
250/250 [==============================] - 40s 159ms/step - loss: 0.2207 - val_loss: 0.2155 - lr: 0.0010
Epoch 7/10
250/250 [==============================] - 41s 163ms/step - loss: 0.2155 - val_loss: 0.2122 - lr: 0.0010
Epoch 8/10
250/250 [==============================] - 41s 162ms/step - loss: 0.2125 - val_loss: 0.2098 - lr: 0.0010
Epoch 9/10
250/250 [==============================] - 40s 162ms/step - loss: 0.2101 - val_loss: 0.2077 - lr: 0.0010
Epoch 10/10
250/250 [==============================] - 40s 160ms/step - loss: 0.2081 - val_loss: 0.2058 - lr: 0.0010
```

```
250/250 [==============================] - 5s 19ms/step - loss: 0.2480
Test Loss: 0.2479938268661499
```



Learning Rate:

```
Epoch 1/10
250/250 [==============================] - 55s 179ms/step - loss: 0.1874 - val_loss: 0.1733 - lr: 0.5000
Epoch 2/10
250/250 [==============================] - 37s 150ms/step - loss: 0.1679 - val_loss: 0.1623 - lr: 0.5000
Epoch 3/10
250/250 [==============================] - 38s 152ms/step - loss: 0.1588 - val_loss: 0.1559 - lr: 0.5000
Epoch 4/10
250/250 [==============================] - 38s 152ms/step - loss: 0.1523 - val_loss: 0.1512 - lr: 0.5000
Epoch 5/10
250/250 [==============================] - 38s 153ms/step - loss: 0.1471 - val_loss: 0.1480 - lr: 0.5000
Epoch 6/10
250/250 [==============================] - 39s 157ms/step - loss: 0.1428 - val_loss: 0.1452 - lr: 0.5000
Epoch 7/10
250/250 [==============================] - 39s 156ms/step - loss: 0.1394 - val_loss: 0.1440 - lr: 0.5000
Epoch 8/10
250/250 [==============================] - 40s 159ms/step - loss: 0.1362 - val_loss: 0.1408 - lr: 0.5000
Epoch 9/10
250/250 [==============================] - 40s 158ms/step - loss: 0.1333 - val_loss: 0.1396 - lr: 0.5000
Epoch 10/10
250/250 [==============================] - 39s 156ms/step - loss: 0.1309 - val_loss: 0.1379 - lr: 0.5000
```



Comparison:

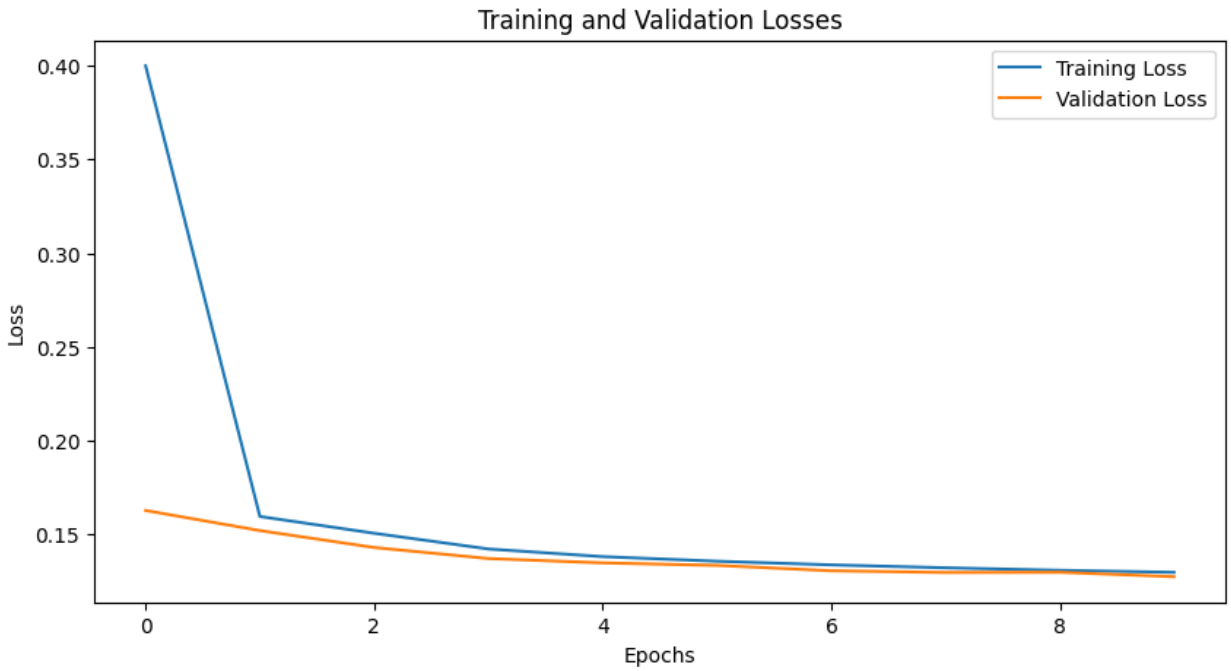
The learning rate method graph indicates the most controlled learning process with minimal variance between training and validation, suggesting effective use of techniques like early stopping and learning rate adjustments. The base-model graph shows robust general learning capabilities with both losses converging smoothly to a low value. The method with early-stopping graph, while showing good convergence, starts with higher errors, which might indicate a less efficient initial learning phase compared to the other two.

Model 2: Encoder, Decoder, LSTM with Attention layers

```python
history = model.fit(
    [X_train, decoder_input_data_train],
    decoder_output_data_train,
    batch_size=64,
    epochs=10,
    validation_data=(
        [X_val, decoder_input_data_val],
        decoder_output_data_val
    )
)
```

```
Epoch 1/10
250/250 [==============================] - 642s 3s/step - loss: 0.3999 - val_loss: 0.1629
Epoch 2/10
250/250 [==============================] - 632s 3s/step - loss: 0.1597 - val_loss: 0.1521
Epoch 3/10
250/250 [==============================] - 640s 3s/step - loss: 0.1507 - val_loss: 0.1431
Epoch 4/10
250/250 [==============================] - 656s 3s/step - loss: 0.1423 - val_loss: 0.1372
Epoch 5/10
250/250 [==============================] - 644s 3s/step - loss: 0.1382 - val_loss: 0.1349
Epoch 6/10
250/250 [==============================] - 658s 3s/step - loss: 0.1358 - val_loss: 0.1336
Epoch 7/10
250/250 [==============================] - 648s 3s/step - loss: 0.1339 - val_loss: 0.1307
Epoch 8/10
250/250 [==============================] - 642s 3s/step - loss: 0.1323 - val_loss: 0.1298
Epoch 9/10
250/250 [==============================] - 638s 3s/step - loss: 0.1310 - val_loss: 0.1299
Epoch 10/10
250/250 [==============================] - 639s 3s/step - loss: 0.1299 - val_loss: 0.1276
```

```
63/63 [==============================] - 27s 429ms/step - loss: 0.1279
Test Loss: 0.1279466152191162
```

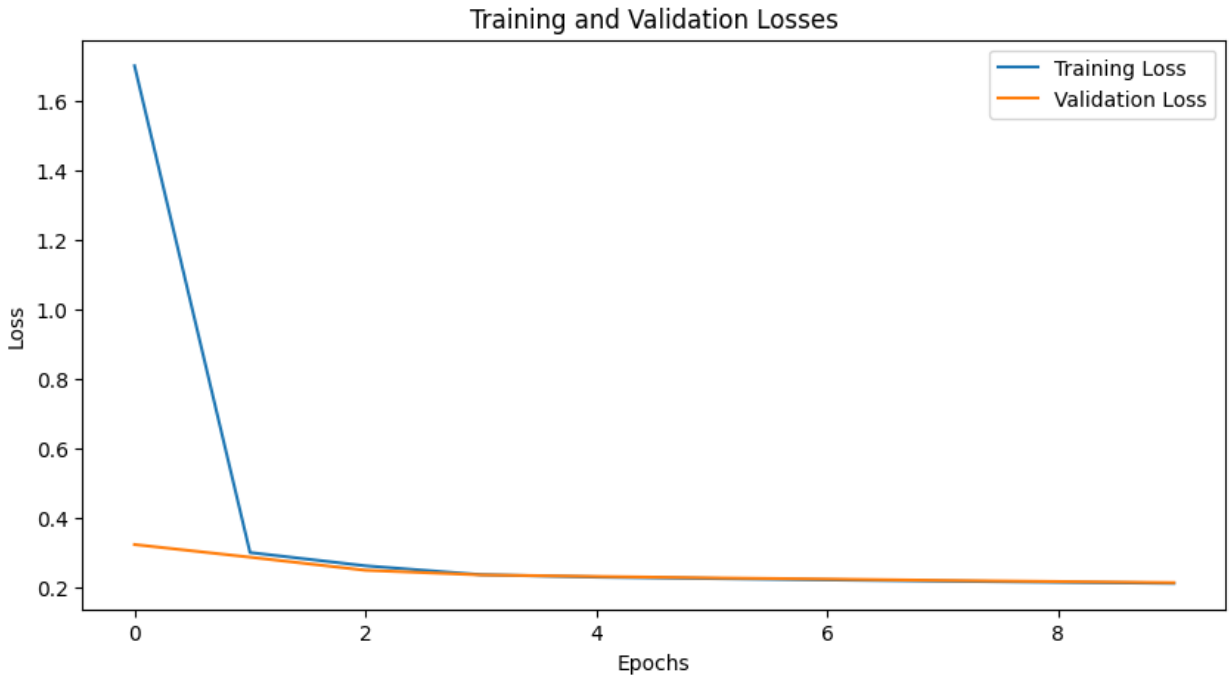


The graph displays the training and validation losses over several epochs for a deep learning model. Initially, the training loss starts at a high value of approximately 0.40, but it quickly drops and converges to around 0.15 by the second epoch. In contrast, the validation loss begins at a lower value, around 0.35, and also decreases rapidly, converging close to the training loss at approximately 0.15 by the second epoch as well. Throughout the remaining epochs, both losses remain stable with minimal fluctuation, suggesting that the model achieves a good generalization on the validation set without signs of overfitting.

Earlystopping:

```
Epoch 1/10
500/500 [==============================] - 198s 373ms/step - loss: 1.6990 - val_loss: 0.3235 - lr: 0.0010
Epoch 2/10
500/500 [==============================] - 180s 361ms/step - loss: 0.3005 - val_loss: 0.2871 - lr: 0.0010
Epoch 3/10
500/500 [==============================] - 178s 356ms/step - loss: 0.2624 - val_loss: 0.2495 - lr: 0.0010
Epoch 4/10
500/500 [==============================] - 178s 357ms/step - loss: 0.2369 - val_loss: 0.2363 - lr: 0.0010
Epoch 5/10
500/500 [==============================] - 178s 357ms/step - loss: 0.2298 - val_loss: 0.2317 - lr: 0.0010
Epoch 6/10
500/500 [==============================] - 178s 356ms/step - loss: 0.2258 - val_loss: 0.2276 - lr: 0.0010
Epoch 7/10
500/500 [==============================] - 178s 357ms/step - loss: 0.2219 - val_loss: 0.2242 - lr: 0.0010
Epoch 8/10
500/500 [==============================] - 178s 356ms/step - loss: 0.2184 - val_loss: 0.2203 - lr: 0.0010
Epoch 9/10
500/500 [==============================] - 178s 357ms/step - loss: 0.2151 - val_loss: 0.2171 - lr: 0.0010
Epoch 10/10
500/500 [==============================] - 178s 356ms/step - loss: 0.2120 - val_loss: 0.2139 - lr: 0.0010
```

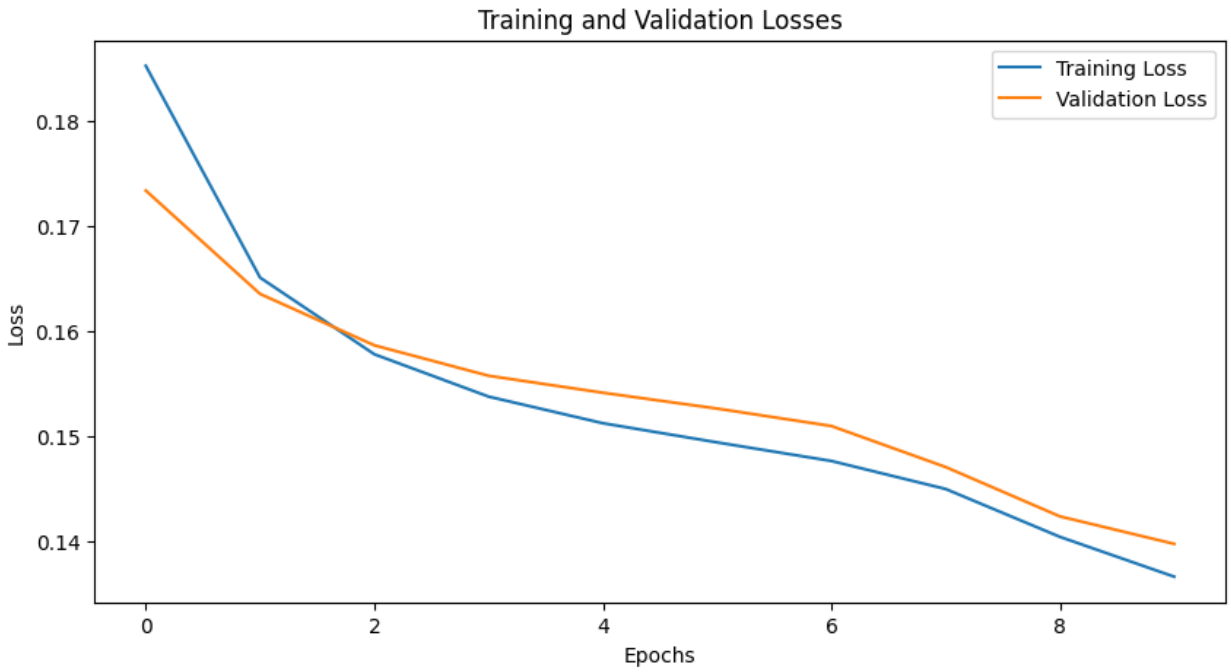
```
16/16 [==============================] - 7s 409ms/step - loss: 0.2158
Test Loss: 0.21578489243984222
```



Learning rate:

```
Epoch 1/10
500/500 [==============================] - 191s 371ms/step - loss: 0.1853 - val_loss: 0.1734 - lr: 0.5000
Epoch 2/10
500/500 [==============================] - 178s 356ms/step - loss: 0.1651 - val_loss: 0.1635 - lr: 0.5000
Epoch 3/10
500/500 [==============================] - 177s 354ms/step - loss: 0.1578 - val_loss: 0.1586 - lr: 0.5000
Epoch 4/10
500/500 [==============================] - 178s 357ms/step - loss: 0.1537 - val_loss: 0.1557 - lr: 0.5000
Epoch 5/10
500/500 [==============================] - 177s 355ms/step - loss: 0.1512 - val_loss: 0.1541 - lr: 0.5000
Epoch 6/10
500/500 [==============================] - 177s 355ms/step - loss: 0.1494 - val_loss: 0.1526 - lr: 0.5000
Epoch 7/10
500/500 [==============================] - 177s 355ms/step - loss: 0.1476 - val_loss: 0.1509 - lr: 0.5000
Epoch 8/10
500/500 [==============================] - 178s 355ms/step - loss: 0.1449 - val_loss: 0.1470 - lr: 0.5000
Epoch 9/10
500/500 [==============================] - 178s 355ms/step - loss: 0.1404 - val_loss: 0.1423 - lr: 0.5000
Epoch 10/10
500/500 [==============================] - 179s 357ms/step - loss: 0.1366 - val_loss: 0.1397 - lr: 0.5000
```

```
16/16 [==============================] - 7s 408ms/step - loss: 0.1401
Test Loss: 0.14013977348804474
```



Comparison:

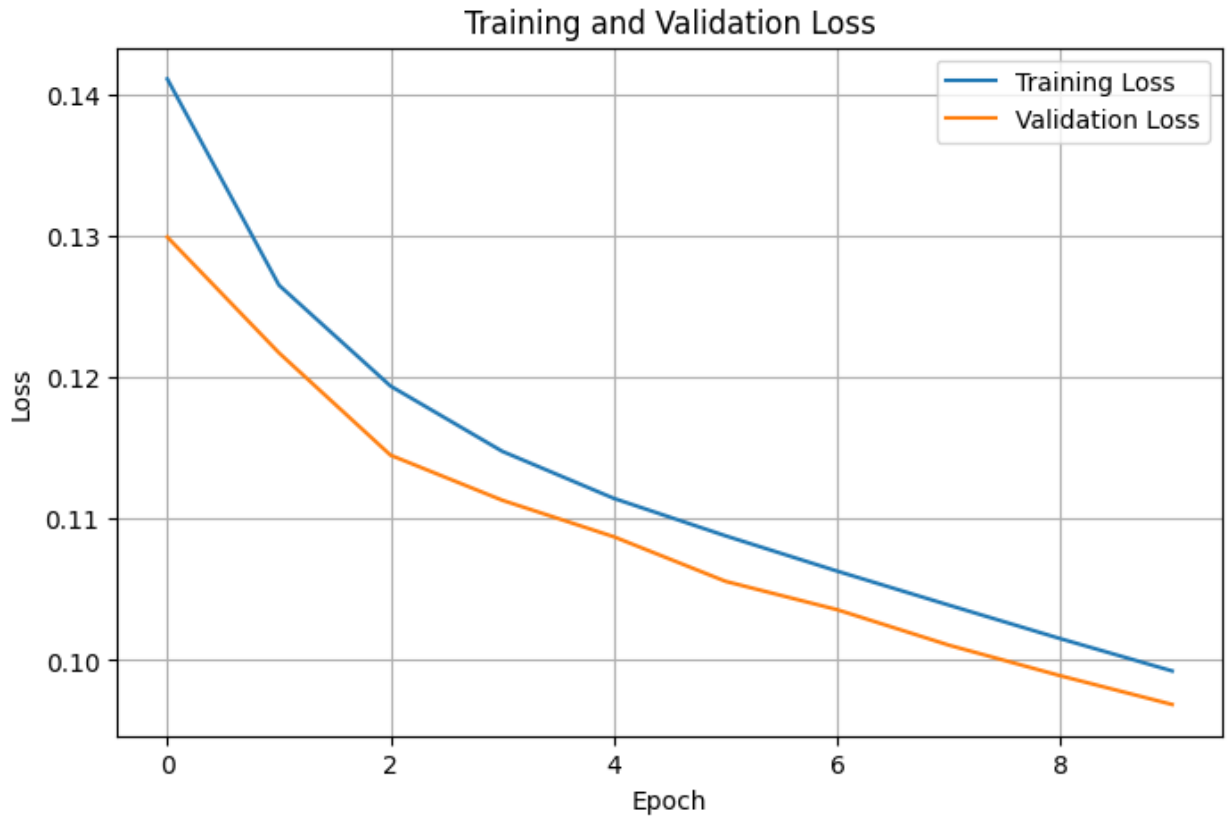
Overall, the Base Model appears to be performing the best, with rapid convergence to a low loss level and excellent balance between training and validation, indicating effective learning and generalization capabilities. The EarlyStopping Model also performs well, particularly in preventing overfitting, while the Learning Rate Adjustment Model, despite its slow start, shows promising convergence characteristics.

Model 3: Encoder, Decoder, GRU with Attention layers

```
history = model.fit(
    [X_train, y_train],
    y_train,
    batch_size=64,
    epochs=10,
    validation_data=([X_val, y_val], y_val)
)
```

```
Epoch 1/10
250/250 [==============================] - 603s 2s/step - loss: 0.1411 - val_loss: 0.1299
Epoch 2/10
250/250 [==============================] - 592s 2s/step - loss: 0.1265 - val_loss: 0.1217
Epoch 3/10
250/250 [==============================] - 596s 2s/step - loss: 0.1194 - val_loss: 0.1145
Epoch 4/10
250/250 [==============================] - 599s 2s/step - loss: 0.1148 - val_loss: 0.1113
Epoch 5/10
250/250 [==============================] - 602s 2s/step - loss: 0.1114 - val_loss: 0.1087
Epoch 6/10
250/250 [==============================] - 603s 2s/step - loss: 0.1088 - val_loss: 0.1056
Epoch 7/10
250/250 [==============================] - 617s 2s/step - loss: 0.1063 - val_loss: 0.1036
Epoch 8/10
250/250 [==============================] - 603s 2s/step - loss: 0.1039 - val_loss: 0.1011
Epoch 9/10
250/250 [==============================] - 614s 2s/step - loss: 0.1015 - val_loss: 0.0989
Epoch 10/10
250/250 [==============================] - 599s 2s/step - loss: 0.0992 - val_loss: 0.0969
```

```
63/63 [==============================] - 25s 401ms/step - loss: 0.0971
Test Loss: 0.09712319076061249
```

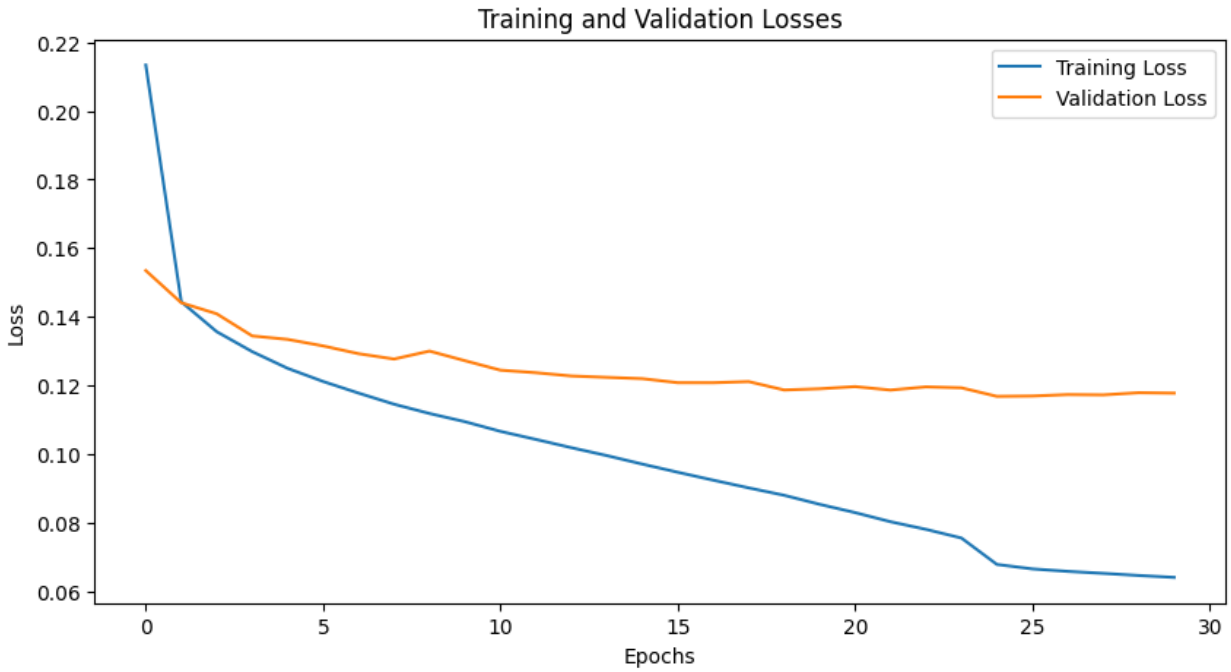


Earlystopping:

```
Epoch 21/30
800/800 [==============================] - 59s 74ms/step - loss: 0.0831 - val_loss: 0.1197 - lr: 0.0100
Epoch 22/30
800/800 [==============================] - 60s 75ms/step - loss: 0.0804 - val_loss: 0.1187 - lr: 0.0100
Epoch 23/30
800/800 [==============================] - 60s 75ms/step - loss: 0.0782 - val_loss: 0.1196 - lr: 0.0100
Epoch 24/30
800/800 [==============================] - ETA: 0s - loss: 0.0756
Epoch 24: ReduceLROnPlateau reducing learning rate to 0.0019999999552965165.
800/800 [==============================] - 60s 75ms/step - loss: 0.0756 - val_loss: 0.1194 - lr: 0.0100
Epoch 25/30
800/800 [==============================] - 60s 75ms/step - loss: 0.0680 - val_loss: 0.1169 - lr: 0.0020
Epoch 26/30
800/800 [==============================] - 60s 75ms/step - loss: 0.0667 - val_loss: 0.1170 - lr: 0.0020
Epoch 27/30
800/800 [==============================] - 59s 74ms/step - loss: 0.0659 - val_loss: 0.1174 - lr: 0.0020
Epoch 28/30
800/800 [==============================] - 60s 75ms/step - loss: 0.0654 - val_loss: 0.1173 - lr: 0.0020
Epoch 29/30
800/800 [==============================] - 60s 74ms/step - loss: 0.0647 - val_loss: 0.1180 - lr: 0.0020
Epoch 30/30
800/800 [==============================] - ETA: 0s - loss: 0.0642
Epoch 30: ReduceLROnPlateau reducing learning rate to 0.001.
800/800 [==============================] - 60s 75ms/step - loss: 0.0642 - val_loss: 0.1178 - lr: 0.0020
```

```
print("Test Loss:", test_loss)
```

```
4/4 [==============================] - 2s 290ms/step - loss: 0.1168
Test Loss: 0.1167786568403244
```

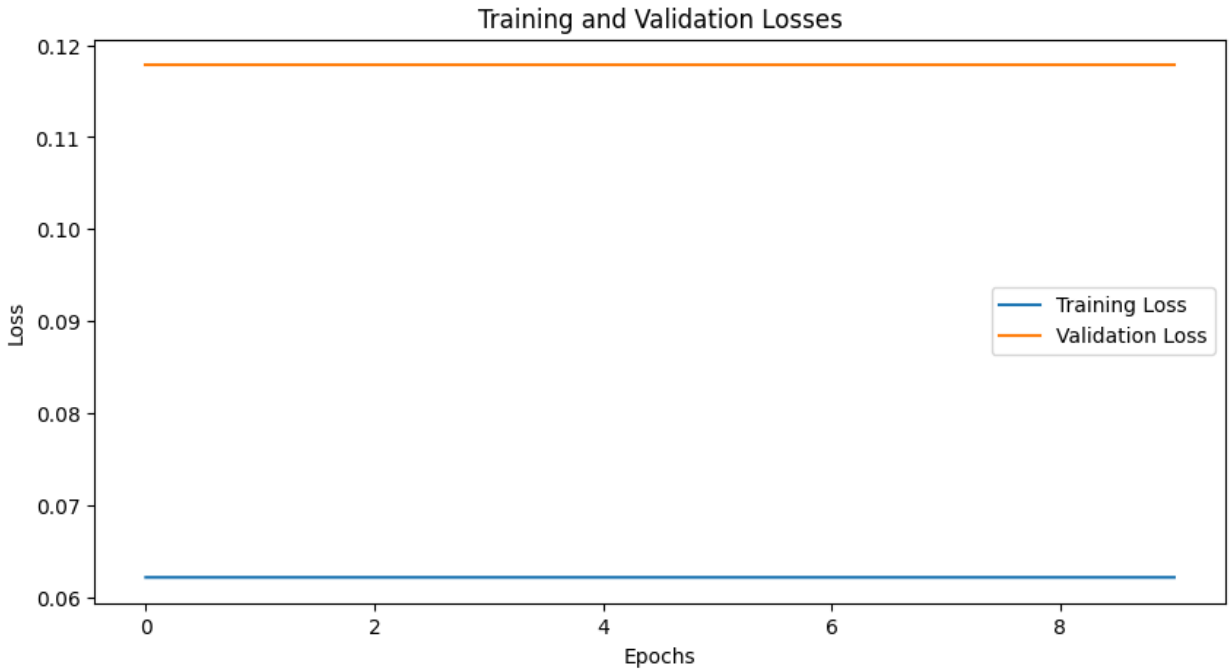


Learning-rate:

```
Epoch 1/10
200/200 [==============================] - 44s 205ms/step - loss: 0.0621 - val_loss: 0.1178 - lr: 1.0000e-08
Epoch 2/10
200/200 [==============================] - 39s 196ms/step - loss: 0.0621 - val_loss: 0.1178 - lr: 1.0000e-08
Epoch 3/10
200/200 [==============================] - 39s 196ms/step - loss: 0.0621 - val_loss: 0.1178 - lr: 1.0000e-08
Epoch 4/10
200/200 [==============================] - 39s 196ms/step - loss: 0.0621 - val_loss: 0.1178 - lr: 1.0000e-08
Epoch 5/10
200/200 [==============================] - 38s 192ms/step - loss: 0.0621 - val_loss: 0.1178 - lr: 1.0000e-08
Epoch 6/10
200/200 [==============================] - 39s 196ms/step - loss: 0.0621 - val_loss: 0.1178 - lr: 1.0000e-08
Epoch 7/10
200/200 [==============================] - 38s 191ms/step - loss: 0.0621 - val_loss: 0.1178 - lr: 1.0000e-08
Epoch 8/10
200/200 [==============================] - 38s 191ms/step - loss: 0.0621 - val_loss: 0.1178 - lr: 1.0000e-08
Epoch 9/10
200/200 [==============================] - 39s 196ms/step - loss: 0.0621 - val_loss: 0.1178 - lr: 1.0000e-08
Epoch 10/10
200/200 [==============================] - 39s 196ms/step - loss: 0.0621 - val_loss: 0.1178 - lr: 1.0000e-08
```

```
print("Test Loss:", test_loss)
```

```
4/4 [==============================] - 3s 316ms/step - loss: 0.1168
Test Loss: 0.1167786717414856
```



Comparison of model3:
The baseline model demonstrates a steady decrease in both training and validation losses, hinting at good learning capabilities but also potential overfitting as training progresses. The early stopping model shows a rapid decrease in losses with the validation loss leveling off quickly, indicating that this method effectively prevents overfitting by stopping the training when the model starts to lose generalization. In contrast, the modified learning rate model exhibits very small gaps between training and validation losses which plateau early, suggesting a well-maintained balance but possibly at the cost of not learning enough. Overall, the early stopping model appears to be the most effective in balancing learning depth and generalization across unseen data.

Best Performing Model: Model 3 appears to be the best performing model. It shows effective learning and generalization capabilities over a longer period, indicated by the sustained reduction in both training and validation losses and their close convergence. This model balances learning and fitting to the data without the early stopping seen in the other models, which might prevent achieving the lowest possible loss.

Real-World Deep Learning Application

Problem Description:

Our project targets the translation of natural language queries into SQL commands. This technology is pivotal for allowing non-technical users to interact seamlessly with databases, thereby democratizing access to data and enabling more straightforward data queries without requiring SQL expertise.

Real-World Relevance:

In practical applications, our NL to SQL models can revolutionize business operations by empowering all employees, regardless of their technical background, to perform data queries. This capability significantly speeds up decision-making and enhances accessibility to data-driven insights.

Data Utilization:

For training, we utilized the WikiSQL dataset, which consists of real-world questions paired with SQL queries and their corresponding database tables. This dataset is crucial for ensuring our models are adept at interpreting and processing a wide range of user queries accurately.

Novelty

Distinct Model Approaches:

We have explored three different neural network architectures-Transformers, LSTMs, and GRUs, independently to determine which model best suits the complex nature of translating NL to SQL. Each model was selected for its specific strengths in handling sequence data and contextual relationships.

Model-Specific Innovations:

- Transformers: We leveraged the Transformer's parallel processing and attention mechanisms to enhance the understanding of longer and more complex queries.

- LSTMs: We utilized LSTMs for their ability to remember long-term dependencies, making them ideal for maintaining context in longer queries.

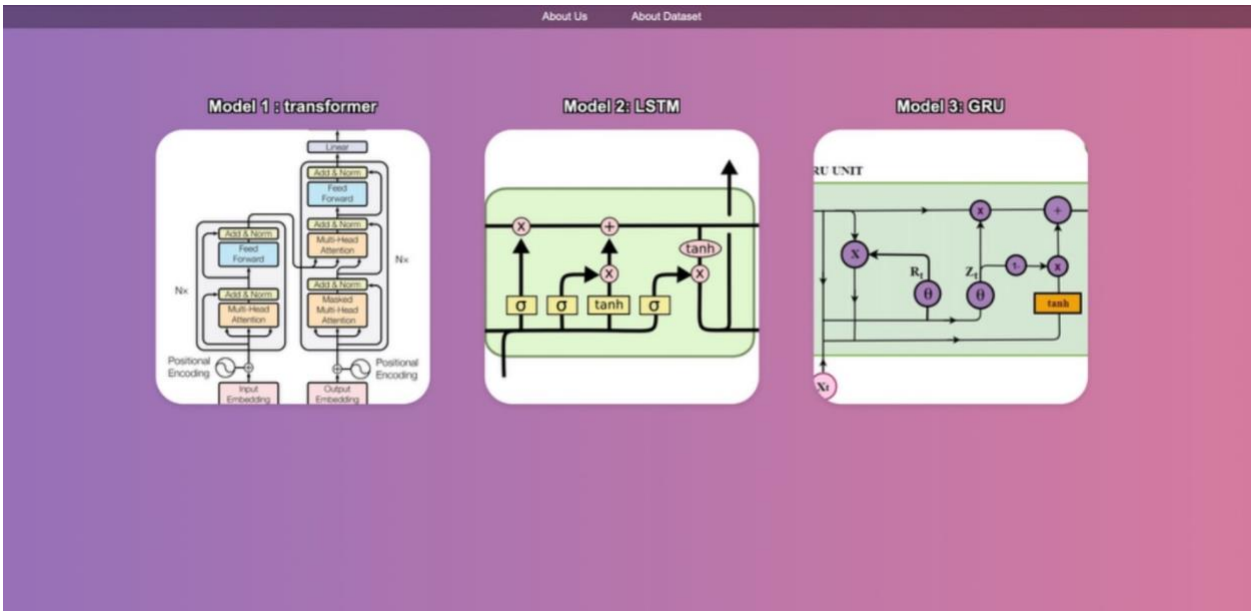
- GRUs: GRUs were employed for their efficiency and performance in smaller datasets or less complex queries compared to LSTMs.

Performance Improvement:

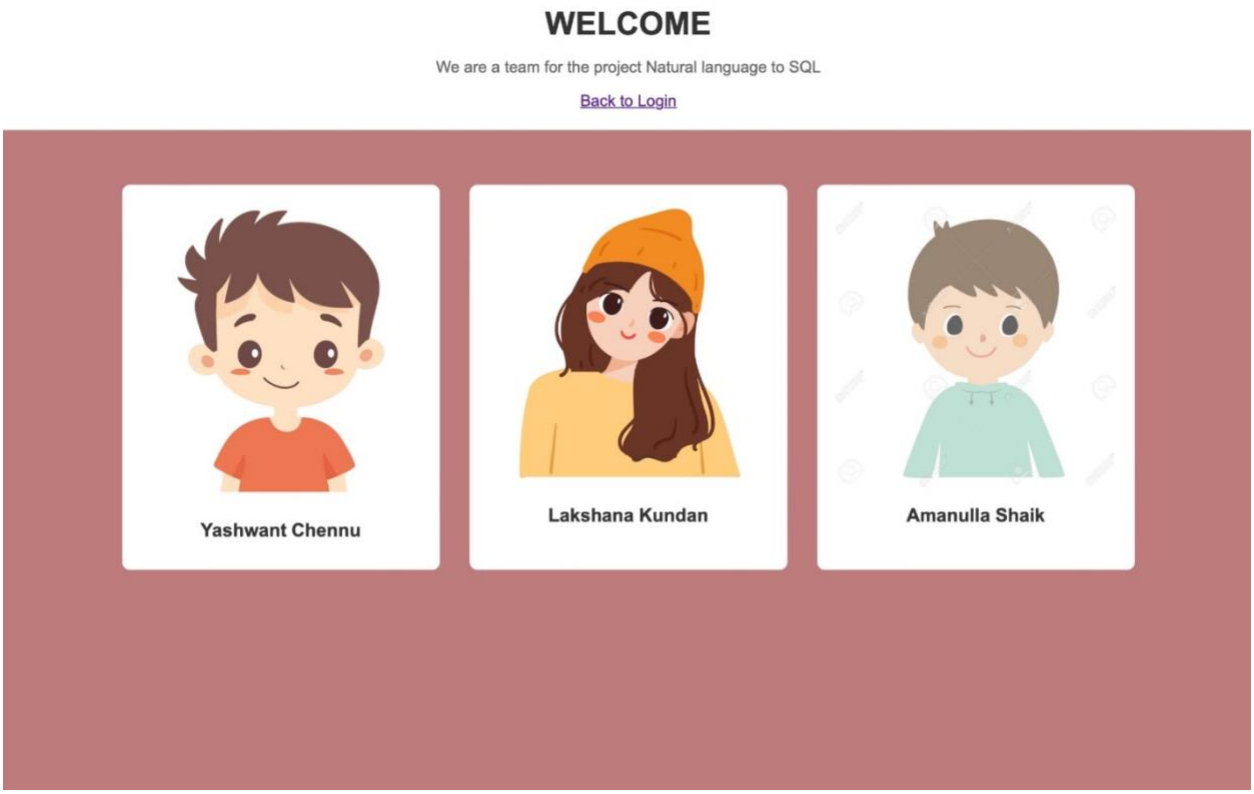
Each model was benchmarked against existing methods to identify which architecture provides the best accuracy, processing time, and scalability for different types of queries. Our findings indicate distinct scenarios where one model may outperform the others, providing valuable insights into tailored application deployments.

Deployment:

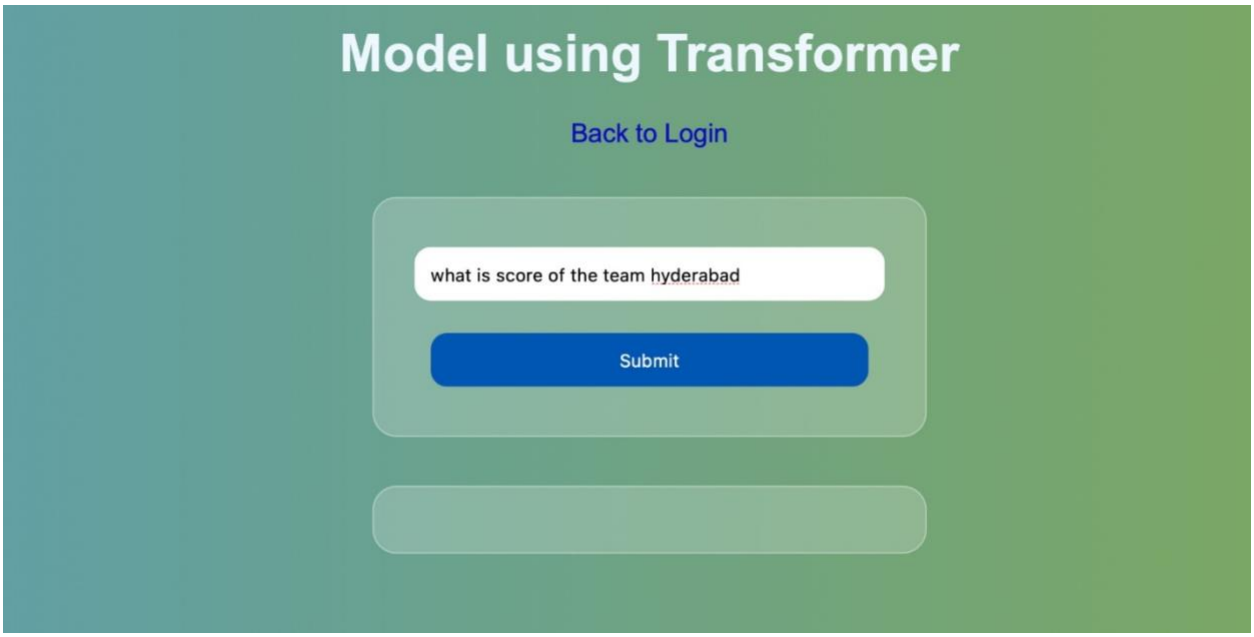
We have developed a interface (deployed locally). The following are the snippets of the website designed which has home page, about us page etc.



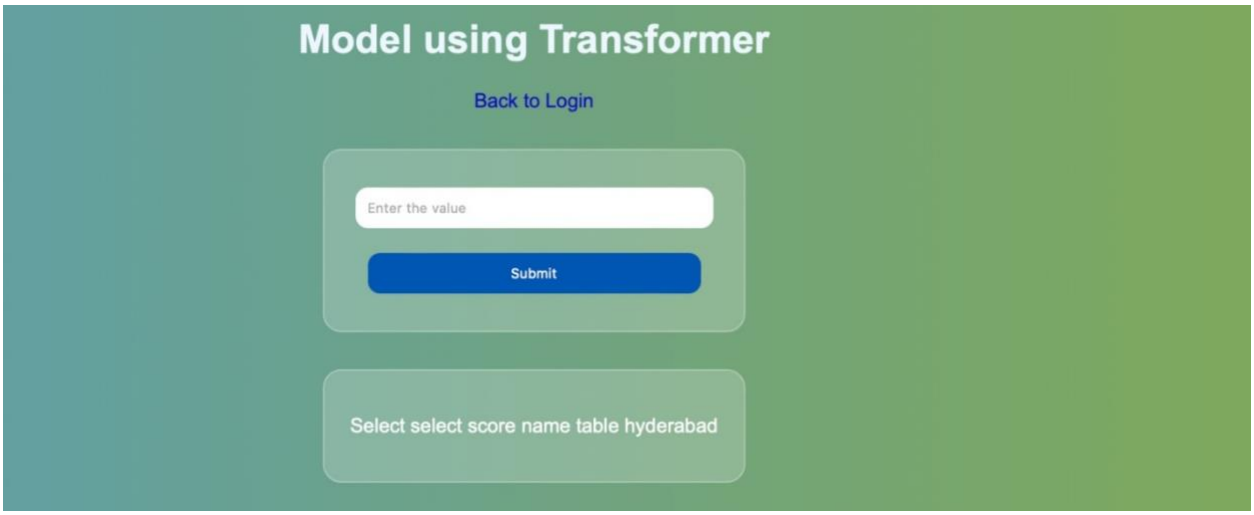
- The above image is for home page where the user can select a model among transformers, LSTM and GRU. After selecting a model it will redirect to the new page to upload the input.
- Already each model is trained with corresponding model weights. So, the user just need to enter the input.



- The above image is for about us page displaying the details of our team.



- Enter the NL and submit the request, the corresponding model performers translation and result the output SQL query as shown below.



Contribution table:

Team Members	Project Part	Contribution
Lakshana Kundan	Preprocessing, Model2, Report, Website	33.3%
Yashwanth Chennu	Preprocessing, Model1, Report, Website	33.3%
Amanulla Shaik	Preprocessing, Model3, Report, Website	33.3%