Implementing “CatSQL: Towards Real World Natural Language to SQL Applications” And Problems Encountered

Made By:
Hossam Imam, hossamemamhosk@gmail.com
Iyad Ashraf, es-iyadashraf2025@alexu.edu.eg
Ramy Fetteha, ramyfetteha@gmail.com

Supervised By:
Professor. Hicham Al Mongui

Contents

Data Preprocessing **4**
- CAT Preprocessing 4
- Nested Queries 4
- Aliasing 5

Encoder **5**
Semantic Correction **5**
Other Papers **6**
Our Process **6**
- Loading Data 6
- Query Preprocessing 7
  - Notes 7
  - Definitions 8
- Vocabulary Generation 8
- Preparing Data For Training 9
- CAT Decoder Network 10
  - Model Initialization and Components 11
    - Initialization 11
    - Positional Encoding 12
    - Softmax Activation 12
    - Vocabulary and Embeddings 12
      - Our approach (Because It Wasn’t Clear) 12
    - BERT Encoder and Tokenizer 12
    - Transformer Decoder 13
      - Transformer Decoder Layer 13
    - Bridge Layer for Dimensionality Matching 13
    - Pointer Network 13
    - Padding and Masking 13
    - Output and Classifiers 14
      - Classifiers 14
      - Logits 14

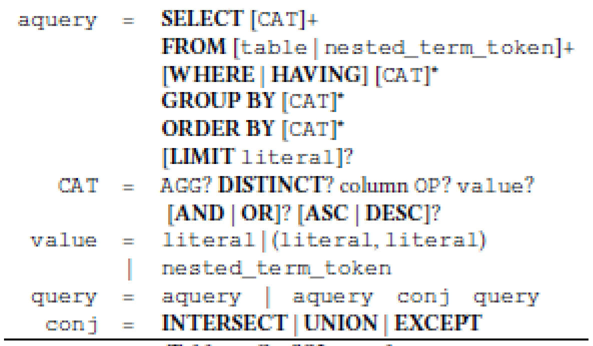
Conclusion **14**

Data Preprocessing

The only thing the paper mentions about preprocessing is for generating the **literal values embeddings and column embeddings.**
But there are a few unclear points regarding the training data itself:

CAT Preprocessing

They never specified what the ground truth is while training the CAT decoder so we had to preprocess the given queries into CATs so that we can use them for training and measuring loss.

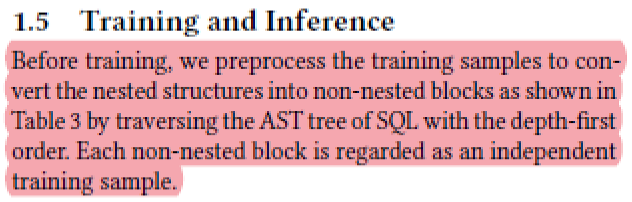


```
aquery  =  SELECT [CAT]+
           FROM [table | nested_term_token]+
           [WHERE | HAVING] [CAT]*
           GROUP BY [CAT]*
           ORDER BY [CAT]*
           [LIMIT literal]?
   CAT  =  AGG? DISTINCT? column OP? value?
            [AND | OR]? [ASC | DESC]?
 value  =  literal | (literal, literal)
        |  nested_term_token
 query  =  aquery  |  aquery  conj  query
  conj  =  INTERSECT | UNION | EXCEPT
```

Table 1: *CatSQL* template

Nested Queries

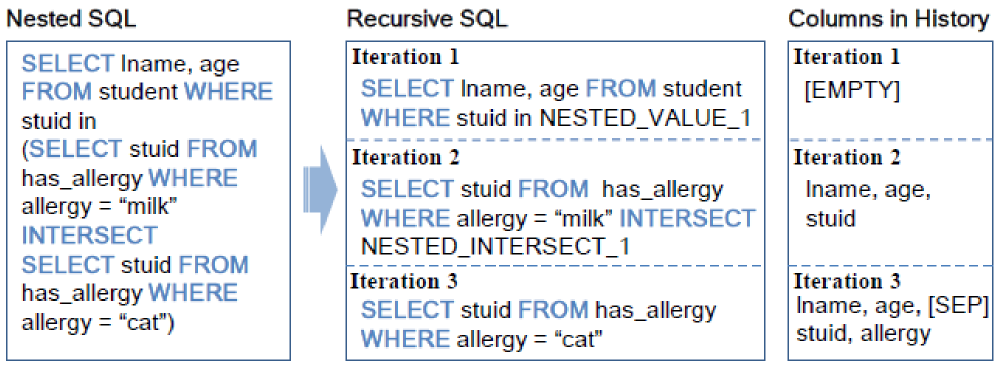
Additionally, there are no mentions in the paper itself about how nested queries are treated aside from mentioning that it is done recursively. After a lengthy amount of search, it was found in the "Model Architecture" file in their github repository which mentioned that they separated all nested queries into their own queries. So it was decided that the recursive process was to be done during the prediction process, not the training process itself.



1.5 Training and Inference

Before training, we preprocess the training samples to convert the nested structures into non-nested blocks as shown in Table 3 by traversing the AST tree of SQL with the depth-first order. Each non-nested block is regarded as an independent training sample.

They never provided the method used for doing so as well, so it was going to take us some time to preprocess such queries.



```
SELECT lname, age
FROM student WHERE
stuid in
(SELECT stuid FROM
has_allergy WHERE
allergy = “milk”
INTERSECT
SELECT stuid FROM
has_allergy WHERE
allergy = “cat”)
```

```
Iteration 1
SELECT lname, age FROM student
WHERE stuid in NESTED_VALUE_1
Iteration 2
SELECT stuid FROM  has_allergy
WHERE allergy = “milk” INTERSECT
NESTED_INTERSECT_1
Iteration 3
SELECT stuid FROM has_allergy
WHERE allergy = “cat”
```

```
Iteration 1
[EMPTY]
Iteration 2
lname, age,
stuid
Iteration 3
lname, age, [SEP]
stuid, allergy
```

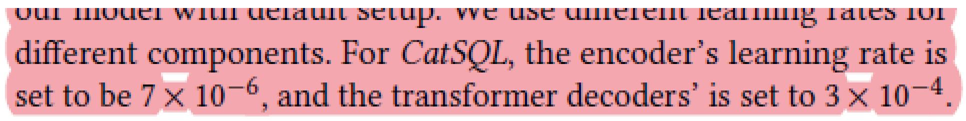
Figure 3: Converting a nested structure into a non-nested form.

Aliasing

In all the examples given in the paper, they can generate columns with aliases but they don’t mention anything about how it is treated.

Encoder

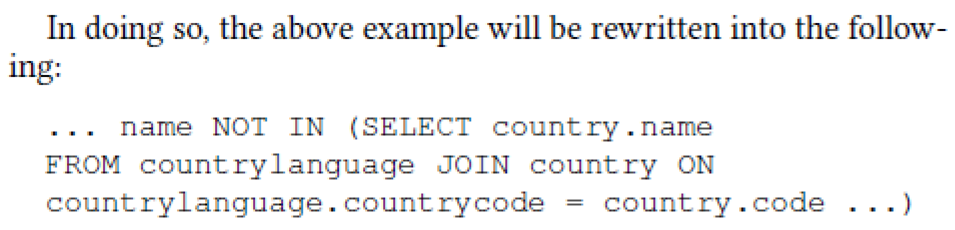
The paper mentioned that they used Grappa as their embedding encoder but were using tokens that were used in BERT. In the Model Architecture file, they were using the BERT encoder instead of Grappa. This resulted in a lot of confusion when they didn't mention anything about how they trained Grappa with the mentioned tokens, and the only mention of training Grappa was when they shared the LR of the encoder and decoder.



our model with default setup. We use different learning rates for different components. For *CatSQL*, the encoder’s learning rate is set to be 7×10^{-6}, and the transformer decoders’ is set to 3×10^{-4}.

Semantic Correction

In Join path revision, they never specify how the query is rebuilt.



In doing so, the above example will be rewritten into the following:

```
... name NOT IN (SELECT country.name
FROM countrylanguage JOIN country ON
countrylanguage.countrycode = country.code ...)
```

Other Papers

Similar papers with the same template based approach put in the necessary details while being clear about everything unlike the given paper, such as [RYANSQL: Recursively Applying Sketch-based Slot Fillings for Complex Text-to-SQL in Cross-Domain Databases]().

Our Process

Loading Data

We decided on using pytorch as it was more familiar to us than other frameworks and BERT since there were no clarifications regarding the Grappa encoder in the paper.

Initially, data is read from training .json (Spider dataset) and then we create the following dictionaries:

- database_dict_db2table: the key is the database and the values are the included tables.

```
example of database to table map of activity_1 database :
 ['activity', 'participates_in', 'faculty_participates_in', 'student', 'faculty']
```

- database_dict_db2table2cols: the key is the database and the values dictionaries containing tables as keys and columns as values.

```
example of database to cols map of activity_1 database :
{'activity': ['actid', 'activity_name'], 'participates_in': ['stuid', 'actid'],
'faculty_participates_in': ['facid', 'actid'], 'student': ['stuid', 'lname', 'fname', 'age',
'sex', 'major', 'advisor', 'city_code'], 'faculty': ['facid', 'lname', 'fname', 'rank',
'sex', 'phone', 'room', 'building']}
```

Query Preprocessing

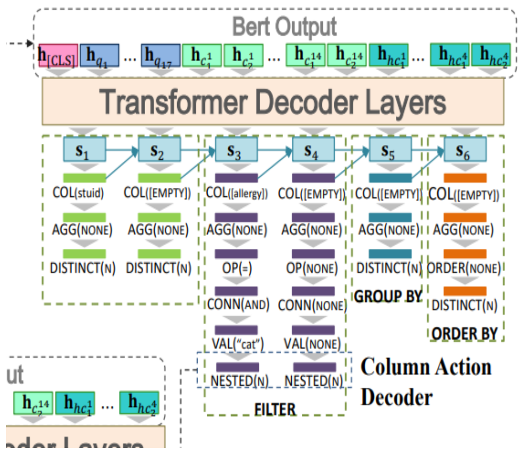
We preprocess part of the dataset (since the processing function was rather complex and some details weren't clear when it came to the CAT template they gave us) so some queries weren't processed correctly.
We specifically take the first 3400 samples as those were the ones that were correctly processed.

A query is converted from the following:

```
"SELECT count(*) FROM head WHERE age  >  56"
```

to this :

```
[['<SOS>', '<SOS>', '<SOS>', '<SOS>', '<SOS>', '<SOS>', '<SOS>'],
 ['count', 'none', '*', 'none', 'none', 'none', 'none'],
 ['EMPTY', 'EMPTY', 'EMPTY', 'EMPTY', 'EMPTY', 'EMPTY', 'EMPTY'],
 ['none', 'none', 'age', '>', '56', 'none', 'none'],
 ['EMPTY', 'EMPTY', 'EMPTY', 'EMPTY', 'EMPTY', 'EMPTY', 'EMPTY'],
 ['EMPTY', 'EMPTY', 'EMPTY', 'EMPTY', 'EMPTY', 'EMPTY', 'EMPTY'],
 ['EMPTY', 'EMPTY', 'EMPTY', 'EMPTY', 'EMPTY', 'EMPTY', 'EMPTY'],
 ['<EOS>', '<EOS>', '<EOS>', '<EOS>', '<EOS>', '<EOS>', '<EOS>'],
 ['<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>'],
 ['<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>'],
 ['<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>'],
 ['<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>'],
 ['<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>'],
 ['<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>'],
 ['<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>'],
 ['<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>'],
 ['<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>'],
 ['<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>'],
 ['<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>'],
 ['<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>', '<PAD>']]
```


Bert Output
$h_{[CLS]}$ h_{q_1} … $h_{q_{17}}$ $h_{c_1^1}$ $h_{c_2^1}$ … $h_{c_1^{14}}$ $h_{c_2^{14}}$ $h_{hc_1^1}$ … $h_{hc_1^4}$ $h_{hc_2^4}$
Transformer Decoder Layers
s_1 s_2 s_3 s_4 s_5 s_6
COL(stuid) COL([EMPTY]) COL([allergy]) COL([EMPTY]) COL([EMPTY]) COL([EMPTY])
AGG(NONE) AGG(NONE) AGG(NONE) AGG(NONE) AGG(NONE) AGG(NONE)
DISTINCT(N) DISTINCT(N) OP(=) OP(NONE) DISTINCT(N) ORDER(NONE)
CONN(AND) CONN(NONE) DISTINCT(N)
VAL("cat") VAL(NONE)
NESTED(N) NESTED(N)
FILTER
GROUP BY
ORDER BY
Column Action Decoder
ut
$h_{c_2^{14}}$ $h_{hc_1^1}$ … $h_{hc_2^4}$

And this sequence should provide the target output of the decoder .

Notes

- <SOS> token is added to the start of the CATS sequence.
- <EOS> token is added to the end of the sequence.
- <PAD> token is added to make all queries of the same length.
- <EMPTY> just indicates the end of a clause [SELECT,WHERE,GROUPBY,ORDERBY]

Definitions

1. **<SOS>** (Start of Sentence):
   a. The **<SOS>** token is used to indicate the beginning of a sentence or sequence of text.
   b. It serves as a special token that tells the model to start generating or processing text.
   c. In sequence-to-sequence models, it's often used as the initial input when generating a new sequence or during training.

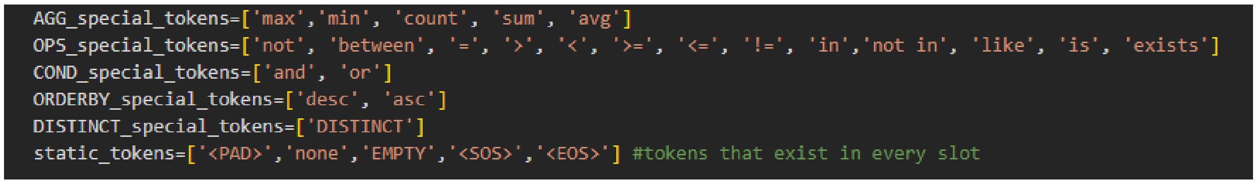
2. **<EOS>** (End of Sentence):
   a. The **<EOS>** token is used to signify the end of a sentence or sequence of text.
   b. It helps the model understand when to stop generating text.
   c. During training, it can also be used as a target token, indicating the end of the target sequence the model should produce.

3. **<PAD>** (Padding):
   a. The **<PAD>** token is used for padding sequences to ensure that they all have the same length.
   b. In many NLP tasks, it's necessary to have input sequences of consistent length for efficient batch processing.
   c. Padding is typically done with **<PAD>** tokens at the end of sequences that are shorter than the maximum sequence length in a batch.
   d. During model training, the **<PAD>** tokens are often ignored or masked so that they don't affect the loss calculation.

These tokens are especially useful when working with variable-length sequences, such as text. They help maintain consistent input sizes for neural networks and provide clear signals for the model to understand the sequence's start and end. Additionally, they aid in creating batches of data with the same sequence length, which is important for efficient training in deep learning.

Vocabulary Generation

After generating our processed queries, we start generating our vocabulary so that it can be used for encoding our input.
We do so by concatenating all the special tokens



```
AGG_special_tokens=['max','min', 'count', 'sum', 'avg']
OPS_special_tokens=['not', 'between', '=', '>', '<', '>=', '<=', '!=', 'in','not in', 'like', 'is', 'exists']
COND_special_tokens=['and', 'or']
ORDERBY_special_tokens=['desc', 'asc']
DISTINCT_special_tokens=['DISTINCT']
static_tokens=['<PAD>','none','EMPTY','<SOS>','<EOS>'] #tokens that exist in every slot
```

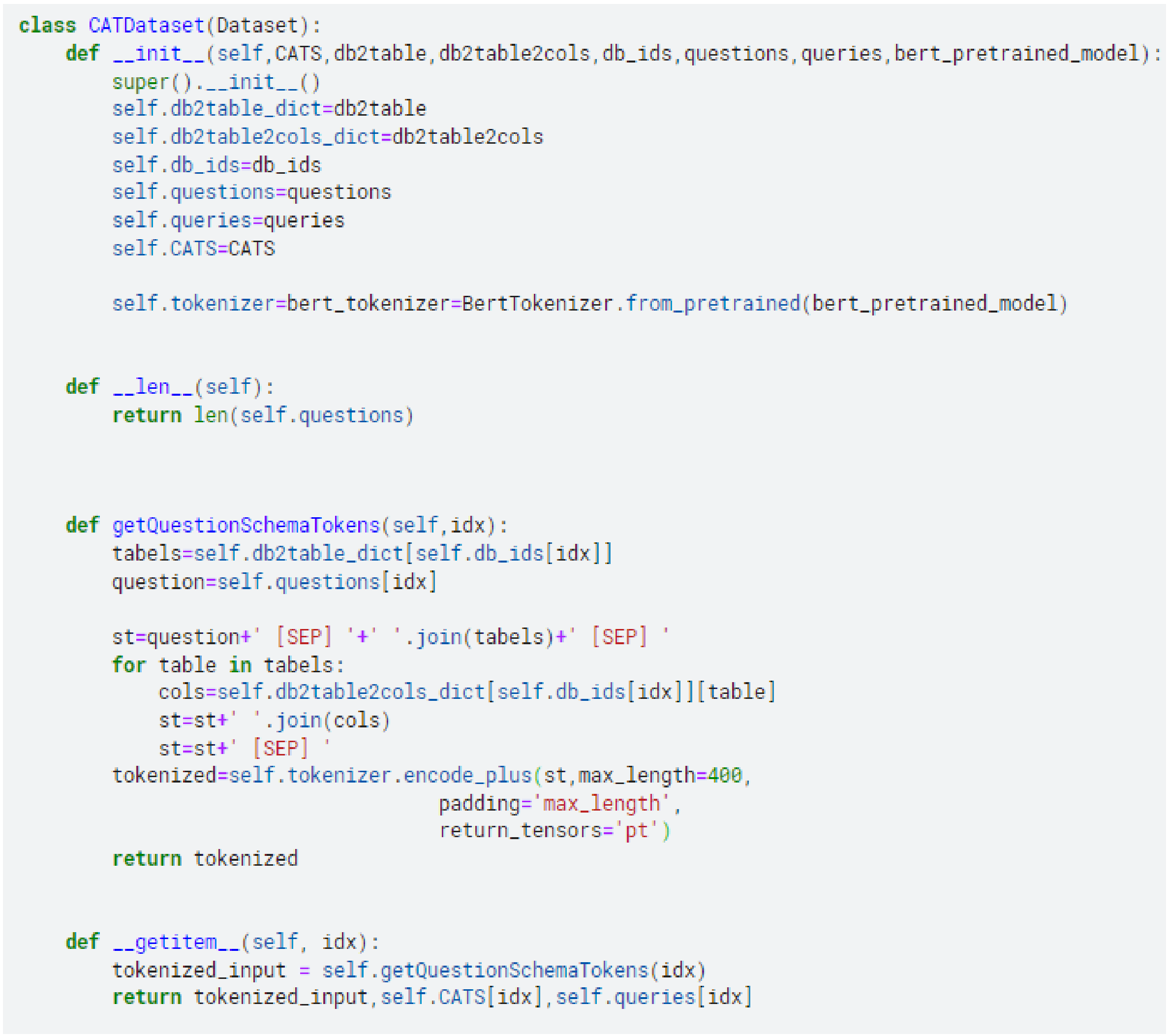
With the possible values and column (column names including aliases as well) to generate our final vocabulary.
Then we create a dictionary for each type of vocab so that it can be used later in our pointer network.

Preparing Data For Training

Since we don't have any test data available, we split our samples into:

- 2720 Training Samples.
- 340 Validation Samples.
- 340 Testing Samples.

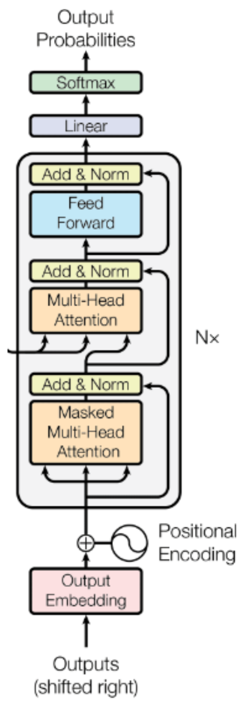
We then initialize our CAT Dataset Object which does all what was mentioned above.

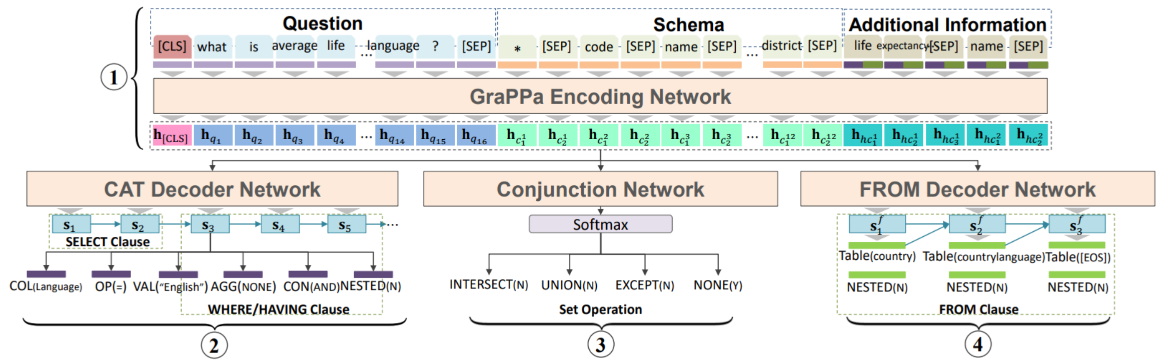


```python
class CATDataset(Dataset):
    def __init__(self,CATS,db2table,db2table2cols,db_ids,questions,queries,bert_pretrained_model):
        super().__init__()
        self.db2table_dict=db2table
        self.db2table2cols_dict=db2table2cols
        self.db_ids=db_ids
        self.questions=questions
        self.queries=queries
        self.CATS=CATS

        self.tokenizer=bert_tokenizer=BertTokenizer.from_pretrained(bert_pretrained_model)


    def __len__(self):
        return len(self.questions)



    def getQuestionSchemaTokens(self,idx):
        tabels=self.db2table_dict[self.db_ids[idx]]
        question=self.questions[idx]

        st=question+' [SEP] '+' '.join(tabels)+' [SEP] '
        for table in tabels:
            cols=self.db2table2cols_dict[self.db_ids[idx]][table]
            st=st+' '.join(cols)
            st=st+' [SEP] '
        tokenized=self.tokenizer.encode_plus(st,max_length=400,
                                            padding='max_length',
                                            return_tensors='pt')
        return tokenized


    def __getitem__(self, idx):
        tokenized_input = self.getQuestionSchemaTokens(idx)
        return tokenized_input,self.CATS[idx],self.queries[idx]
```

With the addition of tokenizing our question using BERT.

CAT Decoder Network





Transformer decoder from “Attention Is All You Need Paper”

Model Initialization and Components

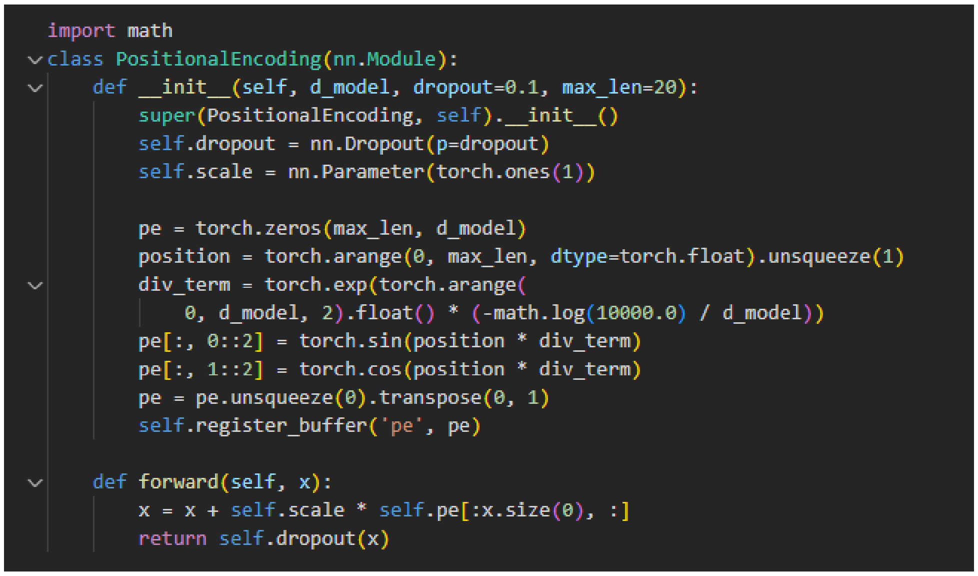
Initialization

The CATDecoder model is initialized with the pre-trained BERT model, transformer configurations, model configurations, and the target device (e.g., GPU or CPU). These parameters set the foundation for the model's operation.

```
import torch.optim as optim
from tqdm import tqdm

# Example configurations
bert_pretrained_model = 'bert-base-uncased'
transformer_config = {
    'd_model': 256,           # BERT's output hidden size
    'nhead': 8,
    'num_layers':4,
    'dim_feedforward':2048, #default
    'dropout':0.1,          #default
    'activation':'relu',    #default
    'layer_norm_eps':1e-5,  #default
    'batch_first':True,      #(batch, seq, feature), # Not default but i prefer this
    'norm_first':False ,    #default
}

model_config = {
    'vocab_size': len(total_vocab_dict),
    'AGG_vocab_size':len(indices['agg']),
    'DISTINCT_vocab_size':len(indices['distinct']),
    'COLS_vocab_size':len(indices['col']),
    'TABLE_vocab_size':len(indices['table']),
    'VALUE_vocab_size':len(indices['value']),
    'OPS_vocab_size':len(indices['ops']),
    'CONJ_vocab_size':len(indices['conj']),
    'ORDERBY_vocab_size':len(indices['orderBy']),
    'col_indices':indices['col'],
    'value_indices':indices['value'],
    'table_indices':indices['table'],
    'tgt_max_length':20
}
DEVICE = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
# Instantiate the model
model = CATDecoder(bert_pretrained_model, transformer_config,model_config,DEVICE).to(DEVICE)

EPOCHS=100
loss_fn =nn.CrossEntropyLoss().to(DEVICE)
LEARNING_RATE=3e-4
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)
```

Positional Encoding

Positional encoding is used to incorporate sequence position information into the model's input; aiding in understanding the order of CATs in a CAT sequence.



```python
import math
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, dropout=0.1, max_len=20):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)
        self.scale = nn.Parameter(torch.ones(1))

        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(
            0, d_model, 2).float() * (-math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0).transpose(0, 1)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + self.scale * self.pe[:x.size(0), :]
        return self.dropout(x)
```

Softmax Activation

The CAT Decoder utilizes the softmax activation function. This function is applied to various components to produce probability distributions.

Vocabulary and Embeddings

The CATDecoder model interacts with a vocabulary of words; these words are (column_names, values, AGG function name… etc) represented as embeddings. These embeddings are 256-dimensional and play a fundamental role in the model's comprehension of input and generation of output.

Our approach (Because It Wasn't Clear)

A CAT template:

```
AGG? DISTINCT? column OP? value? [AND | OR]? [ASC | DESC]?
```

1. Get an embedding vector of dim 256 to match with the decoder specified hidden dimension size.
2. Since one token consists of these 7 elements we would have 7 x 256-dim vectors , so averaging them would get us an 256-dim vector which should contain contextual information about all 7 of them , and should provide a good target for the decoder to learn.

BERT Encoder and Tokenizer

The BERT encoder, a component of the CATDecoder, is responsible for encoding input sequences, typically in the form of natural language questions and schema. This component utilizes a pre-trained BERT model, which has proven effective in various NLP tasks. It encodes input sequences into rich contextualized representations.

Transformer Decoder

Transformer Decoder Layer

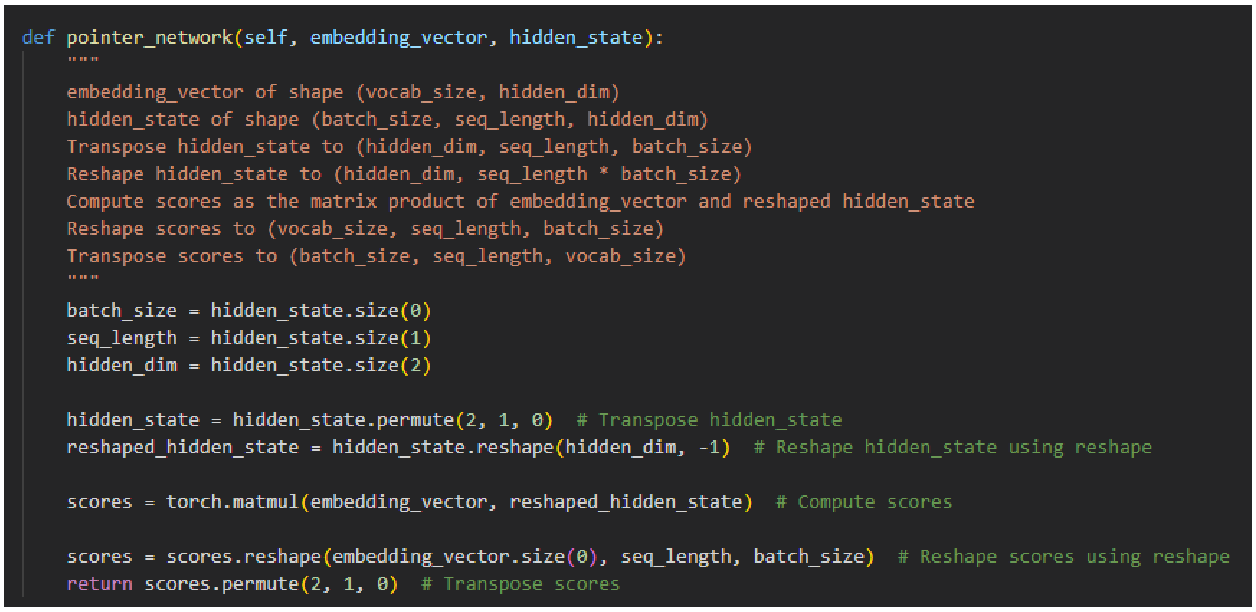
CAT Decoder Network Network consists of four transformer decoder layers. All hidden states are 256-dimensional vectors.

Bridge Layer for Dimensionality Matching

To ensure compatibility between the BERT encoder's output (768-dimensional) and the transformer decoder's expectations (specified by d_model), a linear projection layer is employed. This bridge layer maps the dimensions effectively.

Pointer Network

The CATDecoder includes a pointer network. This network computes scores by computing the inner product of the CAT hidden state vector and the column/values' embedding vector. These scores are generated (col_names,values).



```python
def pointer_network(self, embedding_vector, hidden_state):
    """
    embedding_vector of shape (vocab_size, hidden_dim)
    hidden_state of shape (batch_size, seq_length, hidden_dim)
    Transpose hidden_state to (hidden_dim, seq_length, batch_size)
    Reshape hidden_state to (hidden_dim, seq_length * batch_size)
    Compute scores as the matrix product of embedding_vector and reshaped hidden_state
    Reshape scores to (vocab_size, seq_length, batch_size)
    Transpose scores to (batch_size, seq_length, vocab_size)
    """
    batch_size = hidden_state.size(0)
    seq_length = hidden_state.size(1)
    hidden_dim = hidden_state.size(2)

    hidden_state = hidden_state.permute(2, 1, 0)  # Transpose hidden_state
    reshaped_hidden_state = hidden_state.reshape(hidden_dim, -1)  # Reshape hidden_state using reshape

    scores = torch.matmul(embedding_vector, reshaped_hidden_state)  # Compute scores

    scores = scores.reshape(embedding_vector.size(0), seq_length, batch_size)  # Reshape scores using reshape
    return scores.permute(2, 1, 0)  # Transpose scores
```

Padding and Masking

To handle sequences of varying lengths, the CATDecoder generates padding masks for source and target sequences. Additionally, target masks are applied to aid in the decoding process.

Output and Classifiers

Classifiers

The CATDecoder model includes five simple classifiers for AGG (Aggregation), DISTINCT, CONJ (Conjunction), OP (Operator), and ORDERBY (Ordering). These classifiers are used to predict specific SQL query components.

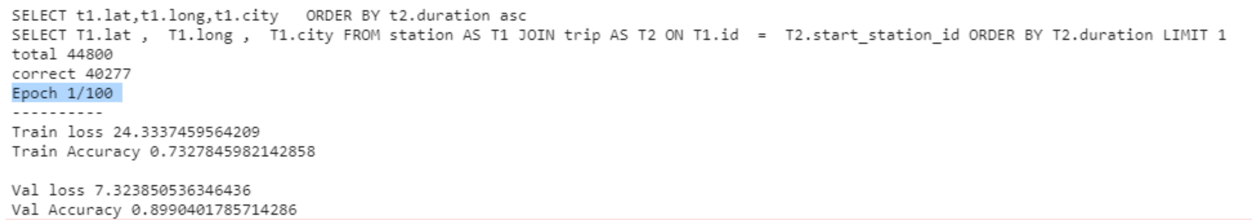
Logits

The output from these classifiers is processed using the softmax function resulting in logits. These logits represent the model's confidence in its predictions. This is later used to predict the outputs.

We implemented our CAT decoder for now and trained it for a bit.

In training the model, we assign argmax layers at the end of the model to pick the most optimal value for its corresponding slot, then we count the correctly classified and misclassified values in each epoch.

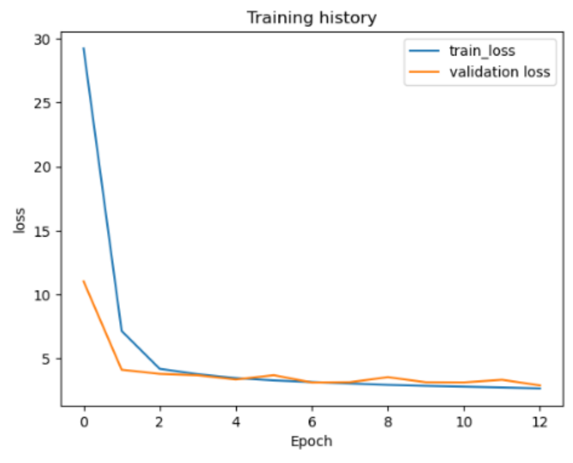
Here is an example of the training loop:

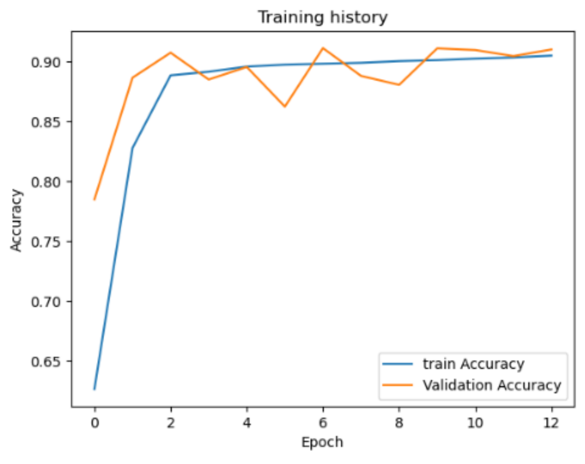


```
SELECT t1.lat,t1.long,t1.city   ORDER BY t2.duration asc
SELECT T1.lat ,  T1.long ,  T1.city FROM station AS T1 JOIN trip AS T2 ON T1.id  =  T2.start_station_id ORDER BY T2.duration LIMIT 1
total 44800
correct 40277
Epoch 1/100
----------
Train loss 24.3337459564209
Train Accuracy 0.7327845982142858

Val loss 7.323850536346436
Val Accuracy 0.8990401785714286
```

The FROM and JOINs aren't implemented yet that's why they don't appear in the predicted query (Upper query).

Evaluation & Metrics





Conclusion

Whilst the journey was full of hurdles and obstacles but we managed to produce some results and got to learn a lot of new information.
Special thanks to **Hossam** for implementing most of the model himself and for his vast knowledge and experience with pytorch and to **Iyad** who was less experienced but managed to implement a Query-To-CAT processor which helped us a lot in the data preprocessing.