```
# Please do not change this cell because some hidden tests might depend on it.
import os

# Otter grader does not handle ! commands well, so we define and use our
# own function to execute shell commands.
def shell(commands, warn=True):
    """Executes the string `commands` as a sequence of shell commands.

       Prints the result to stdout and returns the exit status.
       Provides a printed warning on non-zero exit status unless `warn`
       flag is unset.
    """
    file = os.popen(commands)
    print (file.read().rstrip('\n'))
    exit_status = file.close()
    if warn and exit_status != None:
        print(f"Completed with errors. Exit status: {exit_status}\n")
    return exit_status

shell("""
ls requirements.txt >/dev/null 2>&1
if [ ! $? = 0 ]; then
 rm -rf .tmp
 git clone https://github.com/cs187-2021/project4.git .tmp
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

```
# Initialize Otter
import otter
grader = otter.Notebook()
```

```
pip install wget
```

```
Requirement already satisfied: wget in /usr/local/lib/python3.7/dist-packages (3.2)
```

Unsupported Cell Type. Double-Click to inspect/edit the content.

CS187

Project 4: Semantic Interpretation – Question Answering

The goal of semantic parsing is to convert natural language utterances to a meaning representation such as a *logical form* expression or a *SQL query*. In the previous project segment, you built a parsing system to reconstruct parse trees from the natural-language queries in the ATIS dataset. However, that only solves an intermediary task, not the end-user task of obtaining answers to the queries.

In this final project segment, you will go further, building a semantic parsing system to convert English queries to SQL queries, so that by consulting a database you will be able to answer those questions. You will implement both a rule-based approach and an end-to-end sequence-to-sequence (seq2seq) approach. Both algorithms come with their pros and cons, and by the end of this segment you should have a basic understanding of the characteristics of the two approaches.

Goals

1. Build a semantic parsing algorithm to convert text to SQL queries based on the syntactic parse trees from the last project.
2. Build an attention-based end-to-end seq2seq system to convert text to SQL.
3. Improve the attention-based end-to-end seq2seq system with self-attention to convert text to SQL.
4. Discuss the pros and cons of the rule-based system and the end-to-end system.
5. (Optional) Use the state-of-the-art pretrained transformers for text-to-SQL conversion.

This will be an extremely challenging project, so we recommend that you start early.

Setup

```
import copy
import datetime
import math
import re
import sys
import warnings

import wget
import nltk
import sqlite3
import torch
import torch.nn as nn
```

```
import torchtext.legacy as tt

from cryptography.fernet import Fernet
from func_timeout import func_set_timeout
from torch.nn.utils.rnn import pack_padded_sequence as pack
from torch.nn.utils.rnn import pad_packed_sequence as unpack
from tqdm import tqdm
from transformers import BartTokenizer, BartForConditionalGeneration


# Set random seeds
seed = 1234
torch.manual_seed(seed)
# Set timeout for executing SQL
TIMEOUT = 3 # seconds

# GPU check: Set runtime type to use GPU where available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print (device)
```

```
cuda
```

```
## Download needed scripts and data
os.makedirs('data', exist_ok=True)
os.makedirs('scripts', exist_ok=True)
source_url = "https://raw.githubusercontent.com/nlp-course/data/master"

# Grammar to augment for this segment
if not os.path.isfile('data/grammar'):
  wget.download(f"{source_url}/ATIS/grammar_distrib4.crypt", out="data/")

  # Decrypt the grammar file
  key = b'bfksTY2BJ5VKKK9xZb1PDDLaGkdu7KCDFYfVePSEfGY='
  fernet = Fernet(key)
  with open('./data/grammar_distrib4.crypt', 'rb') as f:
    restored = Fernet(key).decrypt(f.read())
  with open('./data/grammar', 'wb') as f:
    f.write(restored)

# Download scripts and ATIS database
wget.download(f"{source_url}/scripts/trees/transform.py", out="scripts/")
wget.download(f"{source_url}/ATIS/atis_sqlite.db", out="data/")
```

```
'data//atis_sqlite.db'
```

```
# Import downloaded scripts for parsing augmented grammars
sys.path.insert(1, './scripts')
import transform as xform
```

Semantically augmented grammars

In the first part of this project segment, you'll be implementing a rule-based system for semantic interpretation of sentences. Before jumping into using such a system on the ATIS dataset – we'll get to that soon enough – let's first work with some trivial examples to get things going.

The fundamental idea of rule-based semantic interpretation is the rule of compositionality, that *the meaning of a constituent is a function of the meanings of its immediate subconstituents and the syntactic rule that combined them*. This leads to an infrastructure for specifying semantic interpretation in which each syntactic rule in a grammar (in our case, a context-free grammar) is associated with a semantic rule that applies to the meanings associated with the elements on the right-hand side of the rule.

Example: arithmetic expressions

As a first example, let's consider an augmented grammar for arithmetic expressions, familiar from lab 3-1. We again use the function `xform.parse_augmented_grammar` to parse the augmented grammar. You can read more about it in the file `scripts/transform.py`.

[ ] ↳ *30 cells hidden*

Semantics of ATIS queries

Now you're in a good position to understand and add augmentations to a more comprehensive grammar, say, one that parses ATIS queries and generates SQL queries.

In preparation for that, we need to load the ATIS data, both NL and SQL queries.

Loading and preprocessing the corpus

To simplify things a bit, we'll only consider ATIS queries whose question type (remember that from project segment 1?) is `flight_id`. We download training, development, and test splits for this subset of the ATIS corpus, including corresponding SQL queries.

```
# Acquire the datasets - training, development, and test splits of the
# ATIS queries and corresponding SQL queries
wget.download(f"{source_url}/ATIS/test_flightid.nl", out="data/")
wget.download(f"{source_url}/ATIS/test_flightid.sql", out="data/")
wget.download(f"{source_url}/ATIS/dev_flightid.nl", out="data/")
wget.download(f"{source_url}/ATIS/dev_flightid.sql", out="data/")
```

```
wget.download(f"{source_url}/ATIS/train_flightid.nl", out="data/")
wget.download(f"{source_url}/ATIS/train_flightid.sql", out="data/")
```

```
'data//train_flightid.sql'
```

Let's take a look at the data: the NL queries are in `.nl` files, and the SQL queries are in `.sql` files.

```
shell("head -1 data/dev_flightid.nl")
shell("head -1 data/dev_flightid.sql")
```

```
what flights are available tomorrow from denver to philadelphia
SELECT DISTINCT flight_1.flight_id FROM flight flight_1 , airport_service airport_servi
```

Corpus preprocessing

We'll use `torchtext` to process the data. We use two `Field`s: `SRC` for the questions, and `TGT` for the SQL queries. We'll use the tokenizer from project segment 3.

```
## Tokenizer
tokenizer = nltk.tokenize.RegexpTokenizer('\d+|st\.|[\w-]+|\$[\d\.]+|\S+')
def tokenize(string):
  return tokenizer.tokenize(string.lower())

## Demonstrating the tokenizer
## Note especially the handling of `"11pm"` and hyphenated words.
print(tokenize("Are there any first-class flights from St. Louis at 11pm for less than $3.50?
```

```
['are', 'there', 'any', 'first-class', 'flights', 'from', 'st.', 'louis', 'at', '11', 'p
```

```
SRC = tt.data.Field(include_lengths=True,         # include lengths
                    batch_first=False,            # batches will be max_len x batch_size
                    tokenize=tokenize,            # use our tokenizer
                   )
TGT = tt.data.Field(include_lengths=False,
                    batch_first=False,            # batches will be max_len x batch_size
                    tokenize=lambda x: x.split(), # use split to tokenize
                    init_token="<bos>",           # prepend <bos>
                    eos_token="<eos>")            # append <eos>
fields = [('src', SRC), ('tgt', TGT)]
```

> Note that we specified `batch_first=False` (as in lab 4-4), so that the returned batched tensors would be of size `max_length x batch_size`, which facilitates seq2seq implementation.

Now, we load the data using `torchtext`. We use the `TranslationDataset` class here because our task is essentially a translation task: "translating" questions into the corresponding SQL queries. Therefore, we also refer to the questions as the *source* side (`SRC`) and the SQL queries as the *target* side (`TGT`).

```
# Make splits for data
train_data, val_data, test_data = tt.datasets.TranslationDataset.splits(
    ('_flightid.nl', '_flightid.sql'), fields, path='./data/',
    train='train', validation='dev', test='test')

MIN_FREQ = 3
SRC.build_vocab(train_data.src, min_freq=MIN_FREQ)
TGT.build_vocab(train_data.tgt, min_freq=MIN_FREQ)

print (f"Size of English vocab: {len(SRC.vocab)}")
print (f"Most common English words: {SRC.vocab.freqs.most_common(10)}\n")

print (f"Size of SQL vocab: {len(TGT.vocab)}")
print (f"Most common SQL words: {TGT.vocab.freqs.most_common(10)}\n")

print (f"Index for start of sequence token: {TGT.vocab.stoi[TGT.init_token]}")
print (f"Index for end of sequence token: {TGT.vocab.stoi[TGT.eos_token]}")
```

```
Size of English vocab: 421
Most common English words: [('to', 3478), ('from', 3019), ('flights', 2094), ('the', 15

Size of SQL vocab: 392
Most common SQL words: [('=', 38876), ('AND', 36564), (',', 22772), ('airport_service',

Index for start of sequence token: 2
Index for end of sequence token: 3
```

Next, we batch our data to facilitate processing on a GPU. Batching is a bit tricky because the source and target will typically be of different lengths. Fortunately, `torchtext` allows us to pass in a `sort_key` function. By sorting on length, we can minimize the amount of padding on the source side, but since there is still some padding, we need to handle them with [pack](#) and [unpack](#) later on in the seq2seq part (as in lab 4-5).

```
BATCH_SIZE = 16 # batch size for training/validation
TEST_BATCH_SIZE = 1 # batch size for test, we use 1 to make beam search implementation easier

train_iter, val_iter = tt.data.BucketIterator.splits((train_data, val_data),
                                                     batch_size=BATCH_SIZE,
                                                     device=device,
                                                     repeat=False,
```

```
                                                     sort_key=lambda x: len(x.src),
                                                     sort_within_batch=True)
test_iter = tt.data.BucketIterator(test_data,
                                   batch_size=TEST_BATCH_SIZE,
                                   device=device,
                                   repeat=False,
                                   sort=False,
                                   train=False)
```

Let's look at a single batch from one of these iterators.

```
batch = next(iter(train_iter))
train_batch_text, train_batch_text_lengths = batch.src
print (f"Size of text batch: {train_batch_text.shape}")
print (f"Third sentence in batch: {train_batch_text[:, 2]}")
print (f"Length of the third sentence in batch: {train_batch_text_lengths[2]}")
print (f"Converted back to string: {' '.join([SRC.vocab.itos[i] for i in train_batch_text[:,

train_batch_sql = batch.tgt
print (f"Size of sql batch: {train_batch_sql.shape}")
print (f"Third SQL in batch: {train_batch_sql[:, 2]}")
print (f"Converted back to string: {' '.join([TGT.vocab.itos[i] for i in train_batch_sql[:, 2
```

```
Size of text batch: torch.Size([13, 16])
Third sentence in batch: tensor([ 26,  21,   4, 135,   3,  11,   2,  17,  59, 156,  82,
       device='cuda:0')
Length of the third sentence in batch: 13
Converted back to string: list all flights going from boston to atlanta before 7 am on t
Size of sql batch: torch.Size([153, 16])
Third SQL in batch: tensor([  2,  14,  31,  11,  13,  12,  16,   6,   7,  22,   6,   8,
          7,  29,   6,   8,  30,   6,  33,  40,   6,  38,  46,  15,  21,   4,
         18,   5,  19,   4,  17,   5,  20,   4,  52,   5,   9,  24,   4,  25,
          5,  26,   4,  27,   5,  28,   4,  57,   5,   9,  34,   4,  36,   5,
         37,   4,  41,   5,  44,   4,  35,   5,  43,   4, 103,   5,  42,   4,
        126,   5,  32,  72, 346,  10,  10,   3,   1,   1,   1,   1,   1,   1,
          1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,
          1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,
          1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,
          1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,
          1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1],
       device='cuda:0')
Converted back to string: <bos> SELECT DISTINCT flight_1.flight_id FROM flight flight_1
```

Alternatively, we can directly iterate over the raw examples:

```
for example in train_iter.dataset[:1]:
  train_text_1 = ' '.join(example.src) # detokenized question
  train_sql_1 = ' '.join(example.tgt)  # detokenized sql
```

```
  print (f"Question: {train_text_1}\n")
  print (f"SQL: {train_sql_1}")
```

```
Question: list all the flights that arrive at general mitchell international from variou

SQL: SELECT DISTINCT flight_1.flight_id FROM flight flight_1 , airport airport_1 , airpo
```

Establishing a SQL database for evaluating ATIS queries

The output of our systems will be SQL queries. How should we determine if the generated queries are correct? We can't merely compare against the gold SQL queries, since there are many ways to implement a SQL query that answers any given NL query.

Instead, we will execute the queries – both the predicted SQL query and the gold SQL query – on an actual database, and verify that the returned responses are the same. For that purpose, we need a SQL database server to use. We'll set one up here, using the [Python `sqlite3` module].

```
@func_set_timeout(TIMEOUT)
def execute_sql(sql):
  conn = sqlite3.connect('data/atis_sqlite.db')  # establish the DB based on the downloaded d
  c = conn.cursor()                              # build a "cursor"
  c.execute(sql)
  results = list(c.fetchall())
  c.close()
  conn.close()
  return results
```

To run a query, we use the cursor's `execute` function, and retrieve the results with `fetchall`. Let's get all the flights that arrive at General Mitchell International – the query `train_sql_1` above. There's a lot, so we'll just print out the first few.

```
predicted_ret = execute_sql(train_sql_1)

print(f"""
Executing: {train_sql_1}

Result: {len(predicted_ret)} entries starting with

{predicted_ret[:10]}
""")
```

```
Executing: SELECT DISTINCT flight_1.flight_id FROM flight flight_1 , airport airport_1 ,

Result: 534 entries starting with
```

```
[(107929,), (107930,), (107931,), (107932,), (107933,), (107934,), (107935,), (107936,),
```

For your reference, the SQL database we are using has a database schema described at https://github.com/jkkummerfeld/text2sql-data/blob/master/data/atis-schema.csv, and is consistent with the SQL queries provided in the various `.sql` files loaded above.

Rule-based parsing and interpretation of ATIS queries

First, you will implement a rule-based semantic parser using a grammar like the one you completed in the third project segment. We've placed an initial grammar in the file `data/grammar`. In addition to the helper functions defined above (`constant`, `first`, etc.), it makes use of some other simple functions. We've included those below, but you can (and almost certainly should) augment this set with others that you define as you build out the full set of augmentations.

```
def upper(term):
  return '"' + term.upper() + '"'

def weekday(day):
  return f"flight.flight_days IN (SELECT days.days_code FROM days WHERE days.day_name = '{day

def month_name(month):
  return {'JANUARY' : 1,
          'FEBRUARY' : 2,
          'MARCH' : 3,
          'APRIL' : 4,
          'MAY' : 5,
          'JUNE' : 6,
          'JULY' : 7,
          'AUGUST' : 8,
          'SEPTEMBER' : 9,
          'OCTOBER' : 10,
          'NOVEMBER' : 11,
          'DECEMBER' : 12}[month.upper()]

def airports_from_airport_name(airport_name):
  return f"(SELECT airport.airport_code FROM airport WHERE airport.airport_name = {upper(airp

def airports_from_city(city):
  return f"""
    (SELECT airport_service.airport_code FROM airport_service WHERE airport_service.city_code
      (SELECT city.city_code FROM city WHERE city.city_name = {upper(city)}))
  """
```

```
def null_condition(*args, **kwargs):
  return 1

def depart_around(time):
  return f"""
    flight.departure_time >= {add_delta(miltime(time), -15).strftime('%H%M')}
    AND flight.departure_time <= {add_delta(miltime(time), 15).strftime('%H%M')}
    """.strip()

def add_delta(tme, delta):
    # transform to a full datetime first
    return (datetime.datetime.combine(datetime.date.today(), tme) +
            datetime.timedelta(minutes=delta)).time()

def miltime(minutes):
  return datetime.time(hour=int(minutes/100), minute=(minutes % 100))

#for example "to boston"
def to_place(place):
  return f"flight.to_airport IN {place}"

#for example "from boston"
def from_place(place):
  return f"flight.from_airport IN {place}"

#for example "from boston AND to San Fran"
def conjoin(a,b):
  return f"{a} AND {b}"

#boiler select for all queries
def select(rest):
  return f"SELECT DISTINCT flight.flight_id FROM flight WHERE {rest}"

#query for specific airline
def airline_name(name):
  return f"flight.airline_code = '{name}'"

#query for a plane arriving before a time
def arrive_before(time):
  return f"flight.arrival_time < {time}"

#query for a plane arriving after a time
def depart_before(time):
  return f"flight.departure_time < {time}"
```

We can build a parser with the augmented grammar:

```
atis_grammar, atis_augmentations = xform.read_augmented_grammar('data/grammar', globals=globa
```

```
atis_grammar, atis_augmentations = xform.read_augmented_grammar('data/grammar', globals=globa
atis_parser = nltk.parse.BottomUpChartParser(atis_grammar)
```

We'll define a function to return a parse tree for a string according to the ATIS grammar (if available).

```
def parse_tree(sentence):
  """Parse a sentence and return the parse tree, or None if failure."""
  try:
    parses = list(atis_parser.parse(tokenize(sentence)))
    if len(parses) == 0:
      return None
    else:
      return parses[0]
  except:
    return None
```

We can check the overall coverage of this grammar on the training set by using the `parse_tree` function to determine if a parse is available. The grammar that we provide should get about a 40% coverage of the training set.

```
# Check coverage on training set
parsed = 0
with open("data/train_flightid.nl") as train:
  examples = train.readlines()[:]
for sentence in tqdm(examples):
  if parse_tree(sentence):
    parsed += 1
  else:
    next

print(f"\nParsed {parsed} of {len(examples)} ({parsed*100/(len(examples)):.2f}%)")
```

```
100%|██████████| 3651/3651 [00:21<00:00, 170.17it/s]
Parsed 1525 of 3651 (41.77%)
```

Goal 1: Construct SQL queries from a parse tree and evaluate the results

It's time to turn to the first major part of this project segment, implementing a rule-based semantic parsing system to answer flight-ID-type ATIS queries.

Recall that in rule-based semantic parsing, each syntactic rule is associated with a semantic composition rule. The grammar we've provided has semantic augmentations for some of the low-level phrases – cities, airports, times, airlines – but not the higher level syntactic types. You'll be adding those.

In the ATIS grammar that we provide, as with the earlier toy grammars, the augmentation for a rule with n nonterminals and m terminals on the right-hand side is assumed to be called with n positional arguments (the values for the corresponding children). The `interpret` function you've already defined should therefore work well with this grammar.

Let's run through one way that a semantic derivation might proceed, for the sample query "flights to boston":

```
sample_query = "flights to boston"
print(tokenize(sample_query))
sample_tree = parse_tree(sample_query)
sample_tree.pretty_print()
```

```
['flights', 'to', 'boston']
                S
                |
            NP_FLIGHT
                |
            NOM_FLIGHT
                |
             N_FLIGHT
      __________|_________
     |                    PP
     |                    |
     |                 PP_PLACE
     |          __________|__________
  N_FLIGHT     |                  N_PLACE
     |         |                     |
TERM_FLIGHT  P_PLACE            TERM_PLACE
     |         |                     |
  flights      to                 boston
```

Given a sentence, we first construct its parse tree using the syntactic rules, then compose the corresponding semantic rules bottom-up, until eventually we arrive at the root node with a finished SQL statement. For this query, we will go through what the possible meaning representations for the subconstituents of "flights to boston" might be. But this is just one way of doing things; other ways are possible, and you should feel free to experiment.

Working from bottom up:

1. The `TERM_PLACE` phrase "boston" uses the composition function template `constant(airports_from_city(' '.join(_RHS)))`, which will be instantiated as `constant(airports_from_city(' '.join(['boston'])))` (recall that `_RHS` is replaced by the right-hand side of the rule). The meaning of `TERM_PLACE` will be the SQL snippet

   ```
   SELECT airport_service.airport_code
   FROM airport_service
   WHERE airport_service.city_code IN
    (SELECT city.city_code
     FROM city
     WHERE city.city_name = "BOSTON")
   ```

   (This query generates a list of all of the airports in Boston.)

2. The `N_PLACE` phrase "boston" can have the same meaning as the `TERM_PLACE`.
3. The `P_PLACE` phrase "to" might be associated with a function that maps a SQL query for a list of airports to a SQL condition that holds of flights that go to one of those airports, i.e., `flight.to_airport IN (...)`.
4. The `PP_PLACE` phrase "to boston" might apply the `P_PLACE` meaning to the `TERM_PLACE` meaning, thus generating a SQL condition that holds of flights that go to one of the Boston airports:

   ```
   flight.to_airport IN
    (SELECT airport_service.airport_code
     FROM airport_service
     WHERE airport_service.city_code IN
         (SELECT city.city_code
          FROM city
          WHERE city.city_name = "BOSTON")
   ```

5. The `PP` phrase "to Boston" can again get its meaning from the `PP_PLACE`.
6. The `TERM_FLIGHT` phrase "flights" might also return a condition on flights, this time the "null condition", represented by the SQL truth value `1`. Ditto for the `N_FLIGHT` phrase "flights".
7. The `N_FLIGHT` phrase "flights to boston" can conjoin the two conditions, yielding the SQL condition

```
flight.to_airport IN
 (SELECT airport_service.airport_code
  FROM airport_service
  WHERE airport_service.city_code IN
      (SELECT city.city_code
       FROM city
       WHERE city.city_name = "BOSTON")
AND 1
```

which can be inherited by the `NOM_FLIGHT` and `NP_FLIGHT` phrases.

8. The `S` phrase "flights to boston" can use the condition provided by the `NP_FLIGHT` phrase to select all flights satisfying the condition with a SQL query like

```
SELECT DISTINCT flight.flight_id
FROM flight
WHERE flight.to_airport IN
     (SELECT airport_service.airport_code
      FROM airport_service
      WHERE airport_service.city_code IN
          (SELECT city.city_code
           FROM city
           WHERE city.city_name = "BOSTON")
   AND 1
```

Now, it's your turn to add augmentations to `data/grammar` to make this example work. The augmentations that we have provided for the grammar make use of a set of auxiliary functions that we defined above. You should feel free to add your own auxiliary functions that you make use of in the grammar.

```
#TODO: add augmentations to `data/grammar` to make this example work
atis_grammar, atis_augmentations = xform.read_augmented_grammar('data/grammar', globals=globa
atis_parser = nltk.parse.BottomUpChartParser(atis_grammar)
predicted_sql = interpret(sample_tree, atis_augmentations)
print("Predicted SQL:\n\n", predicted_sql, "\n")
```

```
Predicted SQL:

 SELECT DISTINCT flight.flight_id FROM flight WHERE 1 AND flight.to_airport IN
    (SELECT airport_service.airport_code FROM airport_service WHERE airport_service.city
```

```
    (SELECT city.city_code FROM city WHERE city.city_name = "BOSTON"))
```

Verification on some examples

With a rule-based semantic parsing system, we can generate SQL queries given questions, and then execute those queries on a SQL database to answer the given questions. To evaluate the performance of the system, we compare the returned results against the results of executing the ground truth queries.

We provide a function `verify` to compare the results from our generated SQL to the ground truth SQL. It should be useful for testing individual queries.

```
def verify(predicted_sql, gold_sql, silent=True):
  """
  Compare the correctness of the generated SQL by executing on the
  ATIS database and comparing the returned results.
  Arguments:
      predicted_sql: the predicted SQL query
      gold_sql: the reference SQL query to compare against
      silent: print outputs or not
  Returns: True if the returned results are the same, otherwise False
  """
  # Execute predicted SQL
  try:
    predicted_result = execute_sql(predicted_sql)
  except BaseException as e:
    if not silent:
      print(f"predicted sql exec failed: {e}")
    return False
  if not silent:
    print("Predicted DB result:\n\n", predicted_result[:10], "\n")

  # Execute gold SQL
  try:
    gold_result = execute_sql(gold_sql)
  except BaseException as e:
    if not silent:
      print(f"gold sql exec failed: {e}")
    return False
  if not silent:
    print("Gold DB result:\n\n", gold_result[:10], "\n")

  # Verify correctness
  if gold_result == predicted_result:
    return True
```

Let's try this methodology on a simple example: "flights from phoenix to milwaukee". we provide it along with the gold SQL query.

```
def rule_based_trial(sentence, gold_sql):
  print("Sentence: ", sentence, "\n")
  tree = parse_tree(sentence)
  print("Parse:\n\n")
  tree.pretty_print()

  predicted_sql = interpret(tree, atis_augmentations)
  print("Predicted SQL:\n\n", predicted_sql, "\n")

  if verify(predicted_sql, gold_sql, silent=False):
    print ('Correct!')
  else:
    print ('Incorrect!')
```

```
# Run this cell to reload augmentations after you make changes to `data/grammar`
atis_grammar, atis_augmentations = xform.read_augmented_grammar('data/grammar', globals=globa
atis_parser = nltk.parse.BottomUpChartParser(atis_grammar)
```

```
#TODO: add augmentations to `data/grammar` to make this example work
# Example 1
example_1 = 'flights from phoenix to milwaukee'
gold_sql_1 = """
  SELECT DISTINCT flight_1.flight_id
  FROM flight flight_1 ,
       airport_service airport_service_1 ,
       city city_1 ,
       airport_service airport_service_2 ,
       city city_2
  WHERE flight_1.from_airport = airport_service_1.airport_code
        AND airport_service_1.city_code = city_1.city_code
        AND city_1.city_name = 'PHOENIX'
        AND flight_1.to_airport = airport_service_2.airport_code
        AND airport_service_2.city_code = city_2.city_code
        AND city_2.city_name = 'MILWAUKEE'
  """

rule_based_trial(example_1, gold_sql_1)
```

```
Sentence:  flights from phoenix to milwaukee

Parse:


                                             S
                                             |
```

```
                                  NP_FLIGHT
                                      |
                                  NOM_FLIGHT
                                      |
                                   N_FLIGHT
                  ____________________|__________________
            N_FLIGHT                                     |
      _________|________                                 |
     |                  PP                               PP
     |                  |                                |
     |               PP_PLACE                         PP_PLACE
     |          ________|__________              _______|__________
  N_FLIGHT     |                 N_PLACE     |                 N_PLACE
     |         |                    |         |                    |
TERM_FLIGHT P_PLACE            TERM_PLACE P_PLACE           TERM_PLACE
     |         |                    |         |                    |
  flights     from              phoenix      to               milwaukee

Predicted SQL:

 SELECT DISTINCT flight.flight_id FROM flight WHERE 1 AND flight.from_airport IN
    (SELECT airport_service.airport_code FROM airport_service WHERE airport_service.city
      (SELECT city.city_code FROM city WHERE city.city_name = "PHOENIX"))
   AND flight.to_airport IN
    (SELECT airport_service.airport_code FROM airport_service WHERE airport_service.city
      (SELECT city.city_code FROM city WHERE city.city_name = "MILWAUKEE"))


Predicted DB result:

 [(108086,), (108087,), (301763,), (301764,), (301765,), (301766,), (302323,), (304881,)

Gold DB result:

 [(108086,), (108087,), (301763,), (301764,), (301765,), (301766,), (302323,), (304881,)

Correct!
```

To make development faster, we recommend starting with a few examples before running the full evaluation script. We've taken some examples from the ATIS dataset including the gold SQL queries that they provided. Of course, yours (and those of the project segment solution set) may differ.

```
#TODO: add augmentations to `data/grammar` to make this example work
# Example 2
example_2 = 'i would like a united flight'
gold_sql_2 = """
  SELECT DISTINCT flight_1.flight_id
  FROM flight flight_1
  WHERE flight_1.airline_code = 'UA'
  """

rule_based_trial(example_2, gold_sql_2)
```

```
rule_based_trial(example_2, gold_sql_2)
```

```
Sentence:  i would like a united flight

Parse:

                                                     S
                                         ____________|_____________________________________
                                        |
                                        |
                                    PREIGNORE
           _____________________________|____________
          |                                      PREIGNORE
          |                 _________________________|____________
          |               |                                   PREIGNORE                 ADJ
          |               |                         ____________|____________
          |               |                         |                    PREIGNORE     TER
          |               |                         |                        |
PREIGNORESYMBOL PREIGNORESYMBOL          PREIGNORESYMBOL         PREIGNORESYMBOL TERM
          |               |                         |                        |
          i             would                      like                      a

Predicted SQL:

 SELECT DISTINCT flight.flight_id FROM flight WHERE flight.airline_code = 'UA' AND 1

Predicted DB result:

 [(100094,), (100099,), (100145,), (100158,), (100164,), (100167,), (100169,), (100203,)

Gold DB result:

 [(100094,), (100099,), (100145,), (100158,), (100164,), (100167,), (100169,), (100203,)

Correct!
```

```
#TODO: add augmentations to `data/grammar` to make this example work
# Example 3
example_3 = 'i would like a flight between boston and dallas'
gold_sql_3 = """
  SELECT DISTINCT flight_1.flight_id
  FROM flight flight_1 ,
       airport_service airport_service_1 ,
       city city_1 ,
       airport_service airport_service_2 ,
       city city_2
  WHERE flight_1.from_airport = airport_service_1.airport_code
        AND airport_service_1.city_code = city_1.city_code
        AND city_1.city_name = 'BOSTON'
        AND flight_1.to_airport = airport_service_2.airport_code
        AND airport_service_2.city_code = city_2.city_code
        AND city_2.city_name = 'DALLAS'
  """
```

```
# Note that the parse tree might appear wrong: instead of
# `PP_PLACE -> 'between' N_PLACE 'and' N_PLACE`, the tree appears to be
# `PP_PLACE -> 'between' 'and' N_PLACE N_PLACE`. But it's only a visualization
# error of tree.pretty_print() and you should assume that the production is
# `PP_PLACE -> 'between' N_PLACE 'and' N_PLACE` (you can verify by printing out
# all productions).
rule_based_trial(example_3, gold_sql_3)
```

```
Sentence:  i would like a flight between boston and dallas

Parse:

                                         ____________________________________________________
                                        |
                                        |
                                        |
                                        |
                                    PREIGNORE
               _________________________|____________
              |                                  PREIGNORE
              |              _________________________|____________
              |             |                                   PREIGNORE
              |             |                     ____________|____________
              |             |                    |                      PREIGNORE      N_
              |             |                    |                          |
PREIGNORESYMBOL PREIGNORESYMBOL         PREIGNORESYMBOL        PREIGNORESYMBOL TERM
              |             |                    |                          |
              i           would                 like                        a         f

Predicted SQL:

 SELECT DISTINCT flight.flight_id FROM flight WHERE 1 AND flight.from_airport IN
    (SELECT airport_service.airport_code FROM airport_service WHERE airport_service.city
      (SELECT city.city_code FROM city WHERE city.city_name = "BOSTON"))
   AND flight.to_airport IN
    (SELECT airport_service.airport_code FROM airport_service WHERE airport_service.city
      (SELECT city.city_code FROM city WHERE city.city_name = "DALLAS"))


Predicted DB result:

 [(103171,), (103172,), (103173,), (103174,), (103175,), (103176,), (103177,), (103178,)

Gold DB result:

 [(103171,), (103172,), (103173,), (103174,), (103175,), (103176,), (103177,), (103178,)

Correct!
```

```
#TODO: add augmentations to `data/grammar` to make this example work
```

```
# Example 4
example_4 = 'show me the united flights from denver to baltimore'
gold_sql_4 = """
  SELECT DISTINCT flight_1.flight_id
  FROM flight flight_1 ,
       airport_service airport_service_1 ,
       city city_1 ,
       airport_service airport_service_2 ,
       city city_2
  WHERE flight_1.airline_code = 'UA'
        AND ( flight_1.from_airport = airport_service_1.airport_code
              AND airport_service_1.city_code = city_1.city_code
              AND city_1.city_name = 'DENVER'
              AND flight_1.to_airport = airport_service_2.airport_code
              AND airport_service_2.city_code = city_2.city_code
              AND city_2.city_name = 'BALTIMORE' )

  """

rule_based_trial(example_4, gold_sql_4)
```

```
Sentence:  show me the united flights from denver to baltimore

Parse:

                                                                             S
                        _____________________________________________________|__________
                       |                                                            NP_
                       |
                       |                                                            NOM_
                       |                                         _______________________
                       |                                        |
                       |                                        |
                       |                                        |
                       |                                        |
                       |                                        |                    N_
                       |                                        |             __________
                   PREIGNORE                                   ADJ           |
        _______________|____________                            |            |
       |                        PREIGNORE                  ADJ_AIRLINE       |
       |                ____________|____________               |            |
       |               |                     PREIGNORE     TERM_AIRLINE   N_FLIGHT
       |               |                         |              |            |
PREIGNORESYMBOL PREIGNORESYMBOL           PREIGNORESYMBOL TERM_AIRBRAND TERM_FLIGHT  P_
       |               |                         |              |            |
      show             me                       the           united      flights

Predicted SQL:

 SELECT DISTINCT flight.flight_id FROM flight WHERE flight.airline_code = 'UA' AND 1 AN
    (SELECT airport_service.airport_code FROM airport_service WHERE airport_service.cit
      (SELECT city.city_code FROM city WHERE city.city_name = "DENVER"))
   AND flight.to_airport IN
```

```
          (SELECT airport_service.airport_code FROM airport_service WHERE airport_service.city
            (SELECT city.city_code FROM city WHERE city.city_name = "BALTIMORE"))


      Predicted DB result:

       [(101231,), (101233,), (305983,)]

      Gold DB result:

       [(101231,), (101233,), (305983,)]

      Correct!
```

```
#TODO: add augmentations to `data/grammar` to make this example work
# Example 5
example_5 = 'show flights from cleveland to miami that arrive before 4pm'
gold_sql_5 = """
  SELECT DISTINCT flight_1.flight_id
  FROM flight flight_1 ,
       airport_service airport_service_1 ,
       city city_1 ,
       airport_service airport_service_2 ,
       city city_2
  WHERE flight_1.from_airport = airport_service_1.airport_code
        AND airport_service_1.city_code = city_1.city_code
        AND city_1.city_name = 'CLEVELAND'
        AND ( flight_1.to_airport = airport_service_2.airport_code
              AND airport_service_2.city_code = city_2.city_code
              AND city_2.city_name = 'MIAMI'
              AND flight_1.arrival_time < 1600 )
  """

rule_based_trial(example_5, gold_sql_5)
```

```
     Sentence:  show flights from cleveland to miami that arrive before 4pm

     Parse:


                             S
             _______________________|______________________________________________
            |                                                                 NP_FLIGHT
            |                                                                     |
            |                                                                 NOM_FLIGHT
            |                                                                     |
            |                                                                  N_FLIGHT
            |                                        _____________________________|_________
            |                                      N_FLIGHT
            |                         _________________|_________________
            |                    N_FLIGHT                                |
            |               _________|________                           |
```

```
         |            |                 PP                         PP
         |            |                 |                          |
         |            |              PP_PLACE                   PP_PLACE
         |            |          _________|__________        ________|__________
     PREIGNORE     N_FLIGHT     |               N_PLACE     |               N_PLACE
         |            |         |                  |        |                  |
  PREIGNORESYMBOL TERM_FLIGHT P_PLACE         TERM_PLACE P_PLACE         TERM_PLACE
         |            |         |                  |        |                  |
        show       flights     from            cleveland    to               miami    tha

  Predicted SQL:

   SELECT DISTINCT flight.flight_id FROM flight WHERE 1 AND flight.from_airport IN
      (SELECT airport_service.airport_code FROM airport_service WHERE airport_service.city
        (SELECT city.city_code FROM city WHERE city.city_name = "CLEVELAND"))
    AND flight.to_airport IN
     (SELECT airport_service.airport_code FROM airport_service WHERE airport_service.city
       (SELECT city.city_code FROM city WHERE city.city_name = "MIAMI"))
    AND flight.arrival_time < 1600

  Predicted DB result:

   [(107698,), (301117,)]

  Gold DB result:

   [(107698,), (301117,)]

  Correct!
```

```
#TODO: add augmentations to `data/grammar` to make this example work
# Example 6
example_6 = 'okay how about a flight on sunday from tampa to charlotte'
gold_sql_6 = """
  SELECT DISTINCT flight_1.flight_id
  FROM flight flight_1 ,
       airport_service airport_service_1 ,
       city city_1 ,
       airport_service airport_service_2 ,
       city city_2 ,
       days days_1 ,
       date_day date_day_1
  WHERE flight_1.from_airport = airport_service_1.airport_code
        AND airport_service_1.city_code = city_1.city_code
        AND city_1.city_name = 'TAMPA'
        AND ( flight_1.to_airport = airport_service_2.airport_code
              AND airport_service_2.city_code = city_2.city_code
              AND city_2.city_name = 'CHARLOTTE'
              AND flight_1.flight_days = days_1.days_code
              AND days_1.day_name = date_day_1.day_name
              AND date_day_1.year = 1991
              AND date_day_1.month_number = 8
              AND date_day_1.day_number = 27 )
```

```
              AND date_day_1.day_number = 27 )
  """

# You might notice that the gold answer above used the exact date, which is
# not easily implementable. A more implementable way (generated by the project
# segment 4 solution code) is:
gold_sql_6b = """
  SELECT DISTINCT flight.flight_id
  FROM flight
  WHERE ((((1
            AND flight.flight_days IN (SELECT days.days_code
                                       FROM days
                                       WHERE days.day_name = 'SUNDAY')
            )
           AND flight.from_airport IN (SELECT airport_service.airport_code
                                       FROM airport_service
                                       WHERE airport_service.city_code IN (SELECT city.city_c
                                                                            FROM city
                                                                            WHERE city.city_na
          AND flight.to_airport IN (SELECT airport_service.airport_code
                                    FROM airport_service
                                    WHERE airport_service.city_code IN (SELECT city.city_code
                                                                         FROM city
                                                                         WHERE city.city_name
  """

rule_based_trial(example_6, gold_sql_6b)
```

```
     Sentence:  okay how about a flight on sunday from tampa to charlotte

     Parse:

                                          ___________________________________________________
                                         |
                                         |
                                         |
                                         |
                                         |
                                         |
                                         |
                                         |
                                     PREIGNORE
             ____________________________|____________
            |                                     PREIGNORE
            |                ____________________________|____________
            |               |                                  PREIGNORE
            |               |                          ____________|____________
            |               |                         |                     PREIGNORE      N_
            |               |                         |                         |
     PREIGNORESYMBOL PREIGNORESYMBOL           PREIGNORESYMBOL           PREIGNORESYMBOL TER
            |               |                         |                         |
           okay            how                      about                       a
```

```
Predicted SQL:

 SELECT DISTINCT flight.flight_id FROM flight WHERE 1 AND flight.flight_days IN (SELECT
    (SELECT airport_service.airport_code FROM airport_service WHERE airport_service.city
      (SELECT city.city_code FROM city WHERE city.city_name = "TAMPA"))
   AND flight.to_airport IN
    (SELECT airport_service.airport_code FROM airport_service WHERE airport_service.city
      (SELECT city.city_code FROM city WHERE city.city_name = "CHARLOTTE"))


Predicted DB result:

 [(101860,), (101861,), (101862,), (101863,), (101864,), (101865,), (305231,)]

Gold DB result:

 [(101860,), (101861,), (101862,), (101863,), (101864,), (101865,), (305231,)]

Correct!
```

```
#TODO: add augmentations to `data/grammar` to make this example work
# Example 7
example_7 = 'list all flights going from boston to atlanta that leaves before 7 am on thursda
gold_sql_7 = """
  SELECT DISTINCT flight_1.flight_id
  FROM flight flight_1 ,
       airport_service airport_service_1 ,
       city city_1 ,
       airport_service airport_service_2 ,
       city city_2 ,
       days days_1 ,
       date_day date_day_1
  WHERE flight_1.from_airport = airport_service_1.airport_code
        AND airport_service_1.city_code = city_1.city_code
        AND city_1.city_name = 'BOSTON'
        AND ( flight_1.to_airport = airport_service_2.airport_code
              AND airport_service_2.city_code = city_2.city_code
              AND city_2.city_name = 'ATLANTA'
              AND ( flight_1.flight_days = days_1.days_code
                    AND days_1.day_name = date_day_1.day_name
                    AND date_day_1.year = 1991
                    AND date_day_1.month_number = 5
                    AND date_day_1.day_number = 24
                    AND flight_1.departure_time < 700 ) )
  """

# Again, the gold answer above used the exact date, as opposed to the
# following approach:
gold_sql_7b = """
  SELECT DISTINCT flight.flight_id
  FROM flight
```

```
    WHERE ((1
            AND ((((1
                    AND flight.from_airport IN (SELECT airport_service.airport_code
                                                FROM airport_service
                                                WHERE airport_service.city_code IN (SELECT city
                                                                                    FROM city
                                                                                    WHERE city.
                   AND flight.to_airport IN (SELECT airport_service.airport_code
                                             FROM airport_service
                                             WHERE airport_service.city_code IN (SELECT city.ci
                                                                                 FROM city
                                                                                 WHERE city.cit
                  AND flight.departure_time <= 0700)
                 AND flight.flight_days IN (SELECT days.days_code
                                            FROM days
                                            WHERE days.day_name = 'THURSDAY'))))
    """
```

```
rule_based_trial(example_7, gold_sql_7b)
```

```
Sentence:  list all flights going from boston to atlanta that leaves before 7 am on thur

Parse:

                                   S
       ____________________________|______________________________________________________
      |                                                                                  NP_F
      |                                                                                   
      |         ____________________________________________________________________________
      |        |
      |        |
      |        |
      |        |
      |        |                                                                          N_F
      |        |                                 ____________________________________________
      |        |                               N_FLIGHT
      |        |                       ________|______________________________
      |        |                N_FLIGHT                                      |
      |        |          ______________|________                             |
      |        |         |                       PP                           PP
      |        |         |                       |                            |
      |        |         |                    PP_PLACE                     PP_PLACE
      |        |         |               ________|__________         ________|___________
   PREIGNORE   |     N_FLIGHT           |                 N_PLACE   |                  N_F
      |        |         |              |                    |      |
PREIGNORESYMBOL DET TERM_FLIGHT      P_PLACE            TERM_PLACE P_PLACE             TERM
      |        |         |         ______|________           |      |
     list     all     flights   going           from      boston    to                 atl

Predicted SQL:

 SELECT DISTINCT flight.flight_id FROM flight WHERE 1 AND flight.from_airport IN
    (SELECT airport_service.airport_code FROM airport_service WHERE airport_service.city
      (SELECT city.city_code FROM city WHERE city.city_name = "BOSTON"))
```

```
    AND flight.to_airport IN
     (SELECT airport_service.airport_code FROM airport_service WHERE airport_service.city
       (SELECT city.city_code FROM city WHERE city.city_name = "ATLANTA"))
    AND flight.departure_time < 700 AND flight.flight_days IN (SELECT days.days_code FROM

Predicted DB result:

 [(100014,)]

Gold DB result:

 [(100014,)]

Correct!
```

```
#TODO: add augmentations to `data/grammar` to make this example work
# Example 8
example_8 = 'list the flights from dallas to san francisco on american airlines'
gold_sql_8 = """
  SELECT DISTINCT flight_1.flight_id
  FROM flight flight_1 ,
       airport_service airport_service_1 ,
       city city_1 ,
       airport_service airport_service_2 ,
       city city_2
  WHERE flight_1.airline_code = 'AA'
        AND ( flight_1.from_airport = airport_service_1.airport_code
              AND airport_service_1.city_code = city_1.city_code
              AND city_1.city_name = 'DALLAS'
              AND flight_1.to_airport = airport_service_2.airport_code
              AND airport_service_2.city_code = city_2.city_code
              AND city_2.city_name = 'SAN FRANCISCO' )
  """

rule_based_trial(example_8, gold_sql_8)
```

```
Sentence:  list the flights from dallas to san francisco on american airlines

Parse:

                                                                                S
                        ________________________________________________________________|_
                       |
                       |
                       |
                       |
                       |
                       |                                                               __________
                       |                                                       N_FLIGHT
                       |                                     _________________|__________
                       |                             N_FLIGHT
```

```
                     |                                 _________|________
                     |                                |                  PP
                     |                                |                  |
                 PREIGNORE                            |               PP_PLACE
         ____________|____________                    |         ________|__________        _
        |                         PREIGNORE      N_FLIGHT      |                  N_PLACE      |
        |                            |               |         |                     |         |
PREIGNORESYMBOL              PREIGNORESYMBOL TERM_FLIGHT P_PLACE            TERM_PLACE P_PL/
        |                            |               |         |                     |         |
        |                            |               |         |                     |         |
       list                         the           flights     from                 dallas      tc

Predicted SQL:

 SELECT DISTINCT flight.flight_id FROM flight WHERE 1 AND flight.from_airport IN
    (SELECT airport_service.airport_code FROM airport_service WHERE airport_service.city
      (SELECT city.city_code FROM city WHERE city.city_name = "DALLAS"))
   AND flight.to_airport IN
    (SELECT airport_service.airport_code FROM airport_service WHERE airport_service.city
      (SELECT city.city_code FROM city WHERE city.city_name = "SAN FRANCISCO"))
   AND flight.airline_code = 'AA'

Predicted DB result:

 [(108452,), (108454,), (108456,), (111083,), (111085,), (111086,), (111090,), (111091,)

Gold DB result:

 [(108452,), (108454,), (108456,), (111083,), (111085,), (111086,), (111090,), (111091,)

Correct!
```

Systematic evaluation on a test set

We can perform a more systematic evaluation by checking the accuracy of the queries on an entire test set for which we have gold queries. The `evaluate` function below does just this, calculating precision, recall, and F1 metrics for the test set. It takes as argument a "predictor" function, which maps token sequences to predicted SQL queries. We've provided a predictor function for the rule-based model in the next cell (and a predictor for the seq2seq system below when we get to that system).

The rule-based system does not generate predictions for all queries; many queries won't parse. The precision and recall metrics take this into account in measuring the efficacy of the method. The recall metric captures what proportion of *all of the test examples* for which the system generates a correct query. The precision metric captures what proportion of *all of the test examples for which a prediction is generated* for which the system generates a correct query. (Recall that F1 is just the geometric mean of precision and recall.)

Once you've made some progress on adding augmentations to the grammar, you can evaluate your progress by seeing if the precision and recall have improved. For reference, the solution code achieves precision of about 71% and recall of about 27% for an F1 of 40%.

```
def evaluate(predictor, dataset, num_examples=0, silent=True):
  """Evaluate accuracy of `predictor` by executing predictions on a
  SQL database and comparing returned results against those of gold queries.

  Arguments:
      predictor:    a function that maps a token sequence (provided by torchtext)
                    to a predicted SQL query string
      dataset:      the dataset of token sequences and gold SQL queries
      num_examples: number of examples from `dataset` to use; all of
                    them if 0
      silent: if set to False, will print out logs
  Returns: precision, recall, and F1 score
  """
  # Prepare to count results
  if num_examples <= 0:
    num_examples = len(dataset)
  example_count = 0
  predicted_count = 0
  correct = 0
  incorrect = 0

  # Process the examples from the dataset
  for example in tqdm(dataset[:num_examples]):
    example_count += 1
    # obtain query SQL
    predicted_sql = predictor(example.src)
    if predicted_sql == None:
      continue
    predicted_count += 1
    # obtain gold SQL
    gold_sql = ' '.join(example.tgt)

    # check that they're compatible
    if verify(predicted_sql, gold_sql):
      correct += 1
    else:
      incorrect += 1

  # Compute and return precision, recall, F1
  precision = correct / predicted_count if predicted_count > 0 else 0
  recall = correct / example_count
  f1 = (2 * precision * recall) / (precision + recall) if precision + recall > 0 else 0
  return precision, recall, f1
```

```
def rule_based_predictor(tokens):
  query = ' '.join(tokens)    # detokenized query
  tree = parse_tree(query)
  if tree is None:
    return None
  try:
    predicted_sql = interpret(tree, atis_augmentations)
  except Exception as err:
    return None
  return predicted_sql
```

```
precision, recall, f1 = evaluate(rule_based_predictor, test_iter.dataset, num_examples=0)
print(f"precision: {precision:3.2f}")
print(f"recall:    {recall:3.2f}")
print(f"F1:        {f1:3.2f}")
```

```
100%|██████████| 332/332 [00:03<00:00, 100.74it/s]precision: 0.65
recall:    0.25
F1:        0.36
```

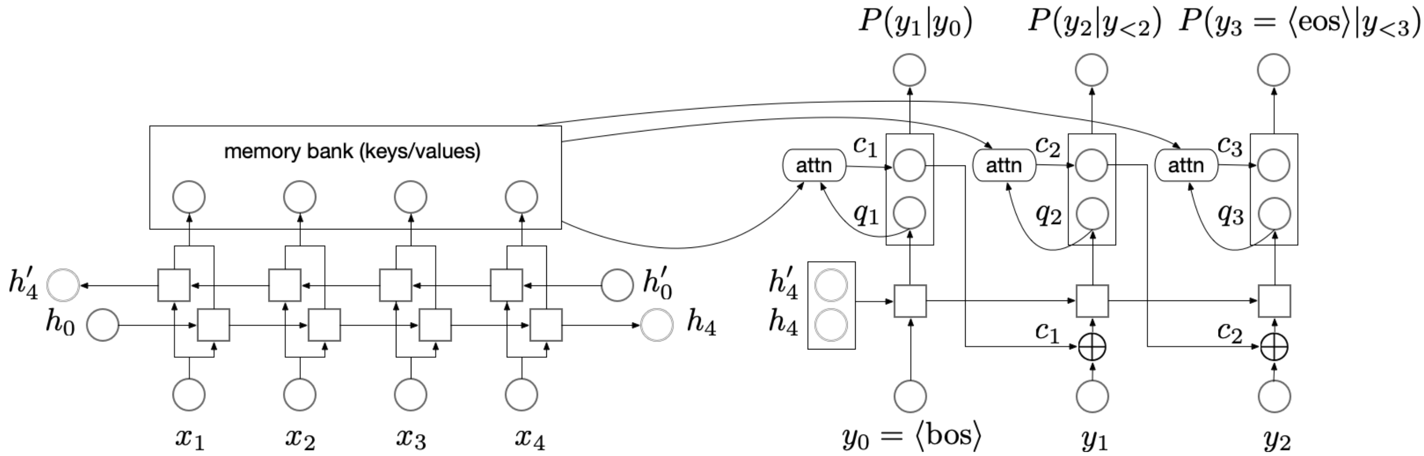
End-to-End Seq2Seq Model

In this part, you will implement a seq2seq model **with attention mechanism** to directly learn the translation from NL query to SQL. You might find labs 4-4 and 4-5 particularly helpful, as the primary difference here is that we are using a different dataset.

Note: We recommend using GPUs to train the model in this part (one way to get GPUs is to use [Google Colab](https://colab.research.google.com) and clicking Menu -> Runtime -> Change runtime type -> GPU), as we need to use a very large model to solve the task well. For development we recommend starting with a smaller model and training for only 1 epoch.

Goal 2: Implement a seq2seq model (with attention)

In lab 4-5, you implemented a neural encoder-decoder model with attention. That model was used to convert English number phrases to numbers, but one of the biggest advantages of neural models is that we can easily apply them to different tasks (such as machine translation and document summarization) by using different training datasets.



Implement the class `AttnEncoderDecoder` to convert natural language queries into SQL statements. You may find that you can reuse most of the code you wrote for lab 4-5. A reasonable way to proceed is to implement the following methods:

- **Model**
  1. `__init__`: an initializer where you create network modules.
  2. `forward`: given source word ids of size `(max_src_len, batch_size)`, source lengths of size `(batch_size)` and decoder input target word ids `(max_tgt_len, batch_size)`, returns logits `(max_tgt_len, batch_size, V_tgt)`. For better modularity you might want to implement it by implementing two functions `forward_encoder` and `forward_decoder`.
- **Optimization**
  3. `train_all`: compute loss on training data, compute gradients, and update model parameters to minimize the loss.
  4. `evaluate_ppl`: evaluate the current model's perplexity on a given dataset iterator, we use the perplexity value on the validation set to select the best model.
- **Decoding**
  5. `predict`: Generates the target sequence given a list of source tokens using beam search decoding. Note that here you can assume the batch size to be 1 for simplicity.

```
import torch.nn as nn


def attention(batched_Q, batched_K, batched_V, mask=None):
  """
  Performs the attention operation and returns the attention matrix
  `batched_A` and the context matrix `batched_C` using queries
  `batched_Q`, keys `batched_K`, and values `batched_V`.
```

```
    Arguments:
        batched_Q: (q_len, bsz, D)
        batched_K: (k_len, bsz, D)
        batched_V: (k_len, bsz, D)
        mask: (bsz, q_len, k_len). An optional boolean mask *disallowing*
              attentions where the mask value is *`False`*.
    Returns:
        batched_A: the normalized attention scores (bsz, q_len, k_ken)
        batched_C: a tensor of size (q_len, bsz, D).
    """
    # Check sizes
    D = batched_Q.size(-1)
    bsz = batched_Q.size(1)
    q_len = batched_Q.size(0)
    k_len = batched_K.size(0)
    assert batched_K.size(-1) == D and batched_V.size(-1) == D
    assert batched_K.size(1) == bsz and batched_V.size(1) == bsz
    assert batched_V.size(0) == k_len
    if mask is not None:
      assert mask.size() == torch.Size([bsz, q_len, k_len])
    batched_Q = torch.transpose(batched_Q, dim0=0, dim1=1).to(device)
    batched_K = torch.transpose(batched_K, dim0=0, dim1=1).to(device)
    batched_V = torch.transpose(batched_V, dim0=0, dim1=1).to(device)

    if mask is not None:
      batched_A = torch.bmm(batched_Q, torch.transpose(batched_K, dim0=1, dim1=2)).masked_fill(
    else:
      batched_A = torch.bmm(batched_Q, torch.transpose(batched_K, dim0=1, dim1=2)).to(device)
    batched_A = torch.softmax(batched_A, dim=-1).to(device)
    batched_C = torch.transpose(torch.bmm(batched_A, batched_V), 0, 1).to(device)
    # Verify that things sum up to one properly.
    assert torch.all(torch.isclose(batched_A.sum(-1),
                                   torch.ones(bsz, q_len).to(device)))
    return batched_A, batched_C



  class Beam():
    """
    Helper class for storing a hypothesis, its score and its decoder hidden state.
    """
    def __init__(self, decoder_state, tokens, score):
      self.decoder_state = decoder_state
      self.tokens = tokens
      self.score = score

  class BeamSearcher():
    """
    Main class for beam search.
    """
    def __init__(self, model):
```

```
    self.model = model
    self.bos_id = model.bos_id
    self.eos_id = model.eos_id
    self.padding_id_src = model.padding_id_src
    self.V = model.V_tgt


  def beam_search(self, src, src_lengths, K, max_T):
    """
    Performs beam search decoding.
    Arguments:
        src: src batch of size (max_src_len, 1)
        src_lengths: src lengths of size (1)
        K: beam size
        max_T: max possible target length considered
    Returns:
        a list of token ids and a list of attentions
    """
    finished = []
    all_attns = []
    # Initialize the beam
    self.model.eval()
    memory_bank, encoder_final_state = self.model.forward_encoder(src, src_lengths)
    init_beam = Beam(encoder_final_state, [self.bos_id], 0)
    beams = [init_beam]

    with torch.no_grad():
      for t in range(max_T): # main body of search over time steps

        # Expand each beam by all possible tokens y_{t+1}
        all_total_scores = []
        for beam in beams:
          y_1_to_t, score, decoder_state = beam.tokens, beam.score, beam.decoder_state
          y_t = y_1_to_t[-1]
          src_mask = src.ne(self.padding_id_src)
          logits, decoder_state, attn = self.model.forward_decoder_incrementally(decoder_stat
          total_scores = score + logits
          all_total_scores.append(total_scores)
          all_attns.append(attn) # keep attentions for visualization
          beam.decoder_state = decoder_state # update decoder state in the beam
        all_total_scores = torch.stack(all_total_scores) # (K, V) when t>0, (1, V) when t=0

        # Find K best next beams
        # The code below has the same functionality as line 6-12, but is more efficient
        all_scores_flattened = all_total_scores.view(-1) # K*V when t>0, 1*V when t=0
        topk_scores, topk_ids = all_scores_flattened.topk(K, 0)
        beam_ids = topk_ids.div(self.V, rounding_mode='floor')
        next_tokens = topk_ids - beam_ids * self.V
        new_beams = []
        for k in range(K):
          beam_id = beam_ids[k]       # which beam it comes from
```

```
        y_t_plus_1 = next_tokens[k] # which y_{t+1}
        score = topk_scores[k]
        beam = beams[beam_id]
        decoder_state = beam.decoder_state
        y_1_to_t = beam.tokens

        new_beam = Beam(decoder_state, y_1_to_t + [y_t_plus_1], score)
        new_beams.append(new_beam)
      beams = new_beams

      # Set aside completed beams
      # TODO - move completed beams to `finished` (and remove them from `beams`)
      keep_beams = []
      for beam in beams:
          if beam.tokens[-1] == self.eos_id:
              finished.append(beam)
          else:
              keep_beams.append(beam)
      beams = keep_beams

      # Break the loop if everything is completed
      if len(beams) == 0:
          break

    # Return the best hypothesis
    if len(finished) > 0:
      finished = sorted(finished, key=lambda beam: -beam.score)
      return finished[0].tokens, all_attns
    else: # when nothing is finished, return an unfinished hypothesis
      return beams[0].tokens, all_attns


#TODO - implement the `AttnEncoderDecoder` class.
class AttnEncoderDecoder(nn.Module):
  def __init__(self, src_field, tgt_field, hidden_size=64, layers=3):
    """
    Initializer. Creates network modules and loss function.
    Arguments:
        src_field: src field
        tgt_field: tgt field
        hidden_size: hidden layer size of both encoder and decoder
        layers: number of layers of both encoder and decoder
    """
    super().__init__()
    self.src_field = src_field
    self.tgt_field = tgt_field

    # Keep the vocabulary sizes available
    self.V_src = len(src_field.vocab.itos)
    self.V_tgt = len(tgt_field.vocab.itos)
```

```
    # Get special word ids
    self.padding_id_src = src_field.vocab.stoi[src_field.pad_token]
    self.padding_id_tgt = tgt_field.vocab.stoi[tgt_field.pad_token]
    self.bos_id = tgt_field.vocab.stoi[tgt_field.init_token]
    self.eos_id = tgt_field.vocab.stoi[tgt_field.eos_token]

    # Keep hyper-parameters available
    self.embedding_size = hidden_size
    self.hidden_size = hidden_size
    self.layers = layers

    # Create essential modules
    self.word_embeddings_src = nn.Embedding(self.V_src, self.embedding_size)
    self.word_embeddings_tgt = nn.Embedding(self.V_tgt, self.embedding_size)

    # RNN cells
    self.encoder_rnn = nn.LSTM(
      input_size    = self.embedding_size,
      hidden_size   = hidden_size // 2, # to match decoder hidden size
      num_layers    = layers,
      bidirectional = True              # bidirectional encoder
    )
    self.decoder_rnn = nn.LSTM(
      input_size    = self.embedding_size,
      hidden_size   = hidden_size,
      num_layers    = layers,
      bidirectional = False             # unidirectional decoder
    )

    # Final projection layer
    self.hidden2output = nn.Linear(2*hidden_size, self.V_tgt) # project the concatenation to

    # Create loss function
    self.loss_function = nn.CrossEntropyLoss(reduction='sum',
                                             ignore_index=self.padding_id_tgt)

  def forward_encoder(self, src, src_lengths):
    """
    Encodes source words `src`.
    Arguments:
        src: src batch of size (max_src_len, bsz)
        src_lengths: src lengths of size (bsz)
    Returns:
        memory_bank: a tensor of size (src_len, bsz, hidden_size)
        (final_state, context): `final_state` is a tuple (h, c) where h/c is of size
                                (layers, bsz, hidden_size), and `context` is `None`.
    """
    src_embeddings = self.word_embeddings_src(src)
    padded = pack(src_embeddings, src_lengths.cpu()) # batch_first=False (default) b/c seqLen
    memory_bank, (h, c) = self.encoder_rnn(padded)
    memory_bank, _ = unpack(memory_bank)
```

```
    h_reshape = h.reshape(int(h.shape[0]/2), 2, h.shape[1], h.shape[2])
    c_reshape = c.reshape(int(c.shape[0]/2), 2, c.shape[1], c.shape[2])
    l2r_h = h_reshape[:, 0]
    r2l_h = h_reshape[:, 1]
    l2r_c = c_reshape[:, 0]
    r2l_c = c_reshape[:, 1]
    join_h = torch.cat([l2r_h, r2l_h], dim=-1)
    join_c = torch.cat([l2r_c, r2l_c], dim=-1)
    final_state = (join_h, join_c)
    context = None
    return memory_bank, (final_state, context)

  def forward_decoder(self, encoder_final_state, tgt_in, memory_bank, src_mask):
    """
    Decodes based on encoder final state, memory bank, src_mask, and ground truth
    target words.
    Arguments:
        encoder_final_state: (final_state, None) where final_state is the encoder
                             final state used to initialize decoder. None is the
                             initial context (there's no previous context at the
                             first step).
        tgt_in: a tensor of size (tgt_len, bsz)
        memory_bank: a tensor of size (src_len, bsz, hidden_size), encoder outputs
                     at every position
        src_mask: a tensor of size (src_len, bsz): a boolean tensor, `False` where
                  src is padding (we disallow decoder to attend to those places).
    Returns:
        Logits of size (tgt_len, bsz, V_tgt) (before the softmax operation)
    """
    max_tgt_length = tgt_in.size(0)

    # Initialize decoder state, note that it's a tuple (state, context) here
    decoder_states = encoder_final_state

    all_logits = []
    for i in range(max_tgt_length):
      logits, decoder_states, attn = \
        self.forward_decoder_incrementally(decoder_states,
                                           tgt_in[i],
                                           memory_bank,
                                           src_mask,
                                           normalize=False)
      all_logits.append(logits)             # list of bsz, vocab_tgt
    all_logits = torch.stack(all_logits, 0) # tgt_len, bsz, vocab_tgt
    return all_logits

  def forward(self, src, src_lengths, tgt_in):
    """
    Performs forward computation, returns logits.
    Arguments:
```

```
            src: src batch of size (max_src_len, bsz)
            src_lengths: src lengths of size (bsz)
            tgt_in:  a tensor of size (tgt_len, bsz)
        """
        src_mask = src.ne(self.padding_id_src) # max_src_len, bsz
        # Forward encoder
        memory_bank, encoder_final_state = self.forward_encoder(src, src_lengths)
        # Forward decoder
        logits = self.forward_decoder(encoder_final_state, tgt_in, memory_bank, src_mask)
        return logits

    def forward_decoder_incrementally(self, prev_decoder_states, tgt_in_onestep,
                                      memory_bank, src_mask,
                                      normalize=True):
        """
        Forward the decoder for a single step with token `tgt_in_onestep`.
        This function will be used both in `forward_decoder` and in beam search.
        Note that bsz can be greater than 1.
        Arguments:
            prev_decoder_states: a tuple (prev_decoder_state, prev_context). `prev_context`
                                 is `None` for the first step
            tgt_in_onestep: a tensor of size (bsz), tokens at one step
            memory_bank: a tensor of size (src_len, bsz, hidden_size), encoder outputs
                         at every position
            src_mask: a tensor of size (src_len, bsz): a boolean tensor, `False` where
                      src is padding (we disallow decoder to attend to those places).
            normalize: use log_softmax to normalize or not. Beam search needs to normalize,
                       while `forward_decoder` does not
        Returns:
            logits: log probabilities for `tgt_in_token` of size (bsz, V_tgt)
            decoder_states: (`decoder_state`, `context`) which will be used for the
                            next incremental update
            attn: normalized attention scores at this step (bsz, src_len)
        """
        prev_decoder_state, prev_context = prev_decoder_states

        tgt_embeddings = self.word_embeddings_tgt(tgt_in_onestep.to(device)).to(device)

        if prev_context is not None:
            decoder_inp = tgt_embeddings + prev_context
        else:
            decoder_inp = tgt_embeddings
            decoder_inp = decoder_inp.unsqueeze(0)

        decoder_outs, decoder_state = self.decoder_rnn(decoder_inp, prev_decoder_state)

        src_mask = torch.transpose(src_mask, 0, 1).unsqueeze(1)

        attn, context = attention(decoder_outs, memory_bank, memory_bank, mask=src_mask)

        attn = attn.squeeze(1)
```

```
    concat_out = torch.cat((decoder_outs, context), dim=2)

    decoder_states = (decoder_state, context)

    logits = self.hidden2output(concat_out).squeeze(0)
    if normalize:
      logits = torch.log_softmax(logits, dim=-1)
    return logits, decoder_states, attn

  def evaluate_ppl(self, iterator):
    """Returns the model's perplexity on a given dataset `iterator`."""
    # Switch to eval mode
    self.eval()
    total_loss = 0
    total_words = 0
    for batch in iterator:
      # Input and target
      src, src_lengths = batch.src
      tgt = batch.tgt # max_length_sql, bsz
      tgt_in = tgt[:-1] # remove <eos> for decode input (y_0=<bos>, y_1, y_2)
      tgt_out = tgt[1:] # remove <bos> as target        (y_1, y_2, y_3=<eos>)
      # Forward to get logits
      logits = self.forward(src, src_lengths, tgt_in)
      # Compute cross entropy loss
      loss = self.loss_function(logits.view(-1, self.V_tgt), tgt_out.view(-1))
      total_loss += loss.item()
      total_words += tgt_out.ne(self.padding_id_tgt).float().sum().item()
    return math.exp(total_loss/total_words)

  def train_all(self, train_iter, val_iter, epochs=10, learning_rate=0.001):
    """Train the model."""
    # Switch the module to training mode
    self.train()
    # Use Adam to optimize the parameters
    optim = torch.optim.Adam(self.parameters(), lr=learning_rate)
    best_validation_ppl = float('inf')
    best_model = None
    # Run the optimization for multiple epochs
    for epoch in range(epochs):
      total_words = 0
      total_loss = 0.0
      for batch in tqdm(train_iter):
        # Zero the parameter gradients
        self.zero_grad()
        # Input and target
        src, src_lengths = batch.src # text: max_src_length, bsz
        tgt = batch.tgt # max_tgt_length, bsz
        tgt_in = tgt[:-1] # Remove <eos> for decode input (y_0=<bos>, y_1, y_2)
        tgt_out = tgt[1:] # Remove <bos> as target        (y_1, y_2, y_3=<eos>)
        bsz = tgt.size(1)
```

```
        # Run forward pass and compute loss along the way.
        logits = self.forward(src, src_lengths, tgt_in)
        loss = self.loss_function(logits.view(-1, self.V_tgt), tgt_out.view(-1))
        # Training stats
        num_tgt_words = tgt_out.ne(self.padding_id_tgt).float().sum().item()
        total_words += num_tgt_words
        total_loss += loss.item()
        # Perform backpropagation
        loss.div(bsz).backward()
        optim.step()

      # Evaluate and track improvements on the validation dataset
      validation_ppl = self.evaluate_ppl(val_iter)
      self.train()
      if validation_ppl < best_validation_ppl:
        best_validation_ppl = validation_ppl
        self.best_model = copy.deepcopy(self.state_dict())
      epoch_loss = total_loss / total_words
      print (f'Epoch: {epoch} Training Perplexity: {math.exp(epoch_loss):.4f} '
             f'Validation Perplexity: {validation_ppl:.4f}')

  def predict(self, tokens, K, max_T):
    #uses beam search with the tokens passed in
    beam_searcher = BeamSearcher(model)
    src, src_lengths = self.src_field.process([tokens])
    src = src.to(device)
    prediction, _ = beam_searcher.beam_search(src, src_lengths, K, max_T)

    #makes output "pretty" and executable
    prediction = ' '.join([TGT.vocab.itos[token] for token in prediction])
    prediction = prediction.lstrip('<bos>').rstrip('<eos>').strip()

    return prediction
```

We provide the recommended hyperparameters for the final model in the script below, but you are free to tune the hyperparameters or change any part of the provided code.

> For quick debugging, we recommend starting with smaller models (by using a very small `hidden_size`), and only a single epoch. If the model runs smoothly, then you can train the full model on GPUs.

```
EPOCHS = 20 # epochs; we recommend starting with a smaller number like 1
LEARNING_RATE = 1e-4 # learning rate

# Instantiate and train classifier
model = AttnEncoderDecoder(SRC, TGT,
  hidden_size    = 1024,
```

```
  layers        = 1,
).to(device)

model.train_all(train_iter, val_iter, epochs=EPOCHS, learning_rate=LEARNING_RATE)
model.load_state_dict(model.best_model)

# Evaluate model performance, the expected value should be < 1.2
print (f'Validation perplexity: {model.evaluate_ppl(val_iter):.3f}')
```

```
100%|██████████| 229/229 [03:09<00:00,  1.21it/s]
Epoch: 0 Training Perplexity: 4.3638 Validation Perplexity: 1.7769
100%|██████████| 229/229 [03:08<00:00,  1.21it/s]
Epoch: 1 Training Perplexity: 1.5196 Validation Perplexity: 1.4457
100%|██████████| 229/229 [03:06<00:00,  1.23it/s]
Epoch: 2 Training Perplexity: 1.3199 Validation Perplexity: 1.3012
100%|██████████| 229/229 [03:06<00:00,  1.23it/s]
Epoch: 3 Training Perplexity: 1.2367 Validation Perplexity: 1.2472
100%|██████████| 229/229 [03:04<00:00,  1.24it/s]
Epoch: 4 Training Perplexity: 1.1851 Validation Perplexity: 1.2109
100%|██████████| 229/229 [03:08<00:00,  1.21it/s]
Epoch: 5 Training Perplexity: 1.1531 Validation Perplexity: 1.1920
100%|██████████| 229/229 [03:06<00:00,  1.23it/s]
Epoch: 6 Training Perplexity: 1.1249 Validation Perplexity: 1.1590
100%|██████████| 229/229 [03:05<00:00,  1.23it/s]
Epoch: 7 Training Perplexity: 1.1007 Validation Perplexity: 1.1420
100%|██████████| 229/229 [03:08<00:00,  1.21it/s]
Epoch: 8 Training Perplexity: 1.0856 Validation Perplexity: 1.1354
100%|██████████| 229/229 [03:04<00:00,  1.24it/s]
Epoch: 9 Training Perplexity: 1.0724 Validation Perplexity: 1.1218
100%|██████████| 229/229 [03:07<00:00,  1.22it/s]
Epoch: 10 Training Perplexity: 1.0612 Validation Perplexity: 1.1229
100%|██████████| 229/229 [03:04<00:00,  1.24it/s]
Epoch: 11 Training Perplexity: 1.0530 Validation Perplexity: 1.1094
100%|██████████| 229/229 [03:05<00:00,  1.23it/s]
Epoch: 12 Training Perplexity: 1.0454 Validation Perplexity: 1.1108
100%|██████████| 229/229 [03:07<00:00,  1.22it/s]
Epoch: 13 Training Perplexity: 1.0418 Validation Perplexity: 1.1128
100%|██████████| 229/229 [03:06<00:00,  1.23it/s]
Epoch: 14 Training Perplexity: 1.0381 Validation Perplexity: 1.1045
100%|██████████| 229/229 [03:07<00:00,  1.22it/s]
Epoch: 15 Training Perplexity: 1.0302 Validation Perplexity: 1.1045
100%|██████████| 229/229 [03:06<00:00,  1.23it/s]
Epoch: 16 Training Perplexity: 1.0267 Validation Perplexity: 1.0982
100%|██████████| 229/229 [03:03<00:00,  1.25it/s]
Epoch: 17 Training Perplexity: 1.0226 Validation Perplexity: 1.0995
100%|██████████| 229/229 [03:06<00:00,  1.23it/s]
Epoch: 18 Training Perplexity: 1.0208 Validation Perplexity: 1.0978
100%|██████████| 229/229 [03:04<00:00,  1.24it/s]
Epoch: 19 Training Perplexity: 1.0198 Validation Perplexity: 1.0989
Validation perplexity: 1.098
```

With a trained model, we can convert questions to SQL statements. We recommend making sure that the model can generate at least reasonable results on the examples from before, before evaluating on the full test set.

```
def seq2seq_trial(sentence, gold_sql):
  print("Sentence: ", sentence, "\n")
  tokens = tokenize(sentence)

  predicted_sql = model.predict(tokens, K=1, max_T=400)
  print("Predicted SQL:\n\n", predicted_sql, "\n")

  if verify(predicted_sql, gold_sql, silent=False):
    print ('Correct!')
  else:
    print ('Incorrect!')
```

```
seq2seq_trial(example_1, gold_sql_1)
```

```
Sentence:  flights from phoenix to milwaukee

Predicted SQL:

 SELECT DISTINCT flight_1.flight_id FROM flight flight_1 , airport_service airport_servi

Predicted DB result:

 [(108086,), (108087,), (301763,), (301764,), (301765,), (301766,), (302323,), (304881,)

Gold DB result:

 [(108086,), (108087,), (301763,), (301764,), (301765,), (301766,), (302323,), (304881,)

Correct!
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:54: UserWarning: To copy cc
```

```
seq2seq_trial(example_2, gold_sql_2)
```

```
Sentence:  i would like a united flight

Predicted SQL:

 SELECT DISTINCT flight_1.flight_id FROM flight flight_1 , airport_service airport_servi

Predicted DB result:

 []

Gold DB result:

 [(100094,), (100099,), (100145,), (100158,), (100164,), (100167,), (100169,), (100203,)

Incorrect!
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:54: UserWarning: To copy cc
```

```
seq2seq_trial(example_3, gold_sql_3)
```

```
Sentence:  i would like a flight between boston and dallas

Predicted SQL:

 SELECT DISTINCT flight_1.flight_id FROM flight flight_1 , airport_service airport_servi

Predicted DB result:

 [(103171,), (103172,), (103173,), (103174,), (103175,), (103176,), (103177,), (103178,)

Gold DB result:

 [(103171,), (103172,), (103173,), (103174,), (103175,), (103176,), (103177,), (103178,)

Correct!
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:54: UserWarning: To copy co
```

```
seq2seq_trial(example_4, gold_sql_4)
```

```
Sentence:  show me the united flights from denver to baltimore

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:54: UserWarning: To copy co
Predicted SQL:

 SELECT DISTINCT flight_1.flight_id FROM flight flight_1 , airport_service airport_servi

Predicted DB result:

 [(101231,), (101233,), (305983,)]

Gold DB result:

 [(101231,), (101233,), (305983,)]

Correct!
```

```
seq2seq_trial(example_5, gold_sql_5)
```

```
Sentence:  show flights from cleveland to miami that arrive before 4pm

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:54: UserWarning: To copy co
Predicted SQL:

 SELECT DISTINCT flight_1.flight_id FROM flight flight_1 , airport_service airport_servi

Predicted DB result:

 [(107698,), (301117,)]
```

```
Gold DB result:

 [(107698,), (301117,)]

Correct!
```

```
seq2seq_trial(example_6, gold_sql_6b)
```

```
Sentence:  okay how about a flight on sunday from tampa to charlotte

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:54: UserWarning: To copy cc
Predicted SQL:

 SELECT DISTINCT flight_1.flight_id FROM flight flight_1 , airport_service airport_servi

Predicted DB result:

 [(101860,), (101861,), (101862,), (101863,), (101864,), (101865,), (305231,)]

Gold DB result:

 [(101860,), (101861,), (101862,), (101863,), (101864,), (101865,), (305231,)]

Correct!
```

```
seq2seq_trial(example_7, gold_sql_7b)
```

```
Sentence:  list all flights going from boston to atlanta that leaves before 7 am on thur

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:54: UserWarning: To copy cc
Predicted SQL:

 SELECT DISTINCT flight_1.flight_id FROM flight flight_1 , airport_service airport_servi

Predicted DB result:

 [(100014,), (100015,), (100016,), (304692,), (307328,)]

Gold DB result:

 [(100014,)]

Incorrect!
```

```
seq2seq_trial(example_8, gold_sql_8)
```

```
Sentence:  list the flights from dallas to san francisco on american airlines

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:54: UserWarning: To copy cc
Predicted SQL:
```

```
 SELECT DISTINCT flight_1.flight_id FROM flight flight_1 , airport_service airport_servi

Predicted DB result:

 [(108452,), (108454,), (108456,), (111083,), (111085,), (111086,), (111090,), (111091,)

Gold DB result:

 [(108452,), (108454,), (108456,), (111083,), (111085,), (111086,), (111090,), (111091,)

Correct!
```

Evaluation

Now we are ready to run the full evaluation. A proper implementation should reach more than 35% precision/recall/F1.

```
def seq2seq_predictor(tokens):
  prediction = model.predict(tokens, K=1, max_T=400)
  return prediction
```

```
precision, recall, f1 = evaluate(seq2seq_predictor, test_iter.dataset, num_examples=0)
print(f"precision: {precision:3.2f}")
print(f"recall:    {recall:3.2f}")
print(f"F1:        {f1:3.2f}")
```

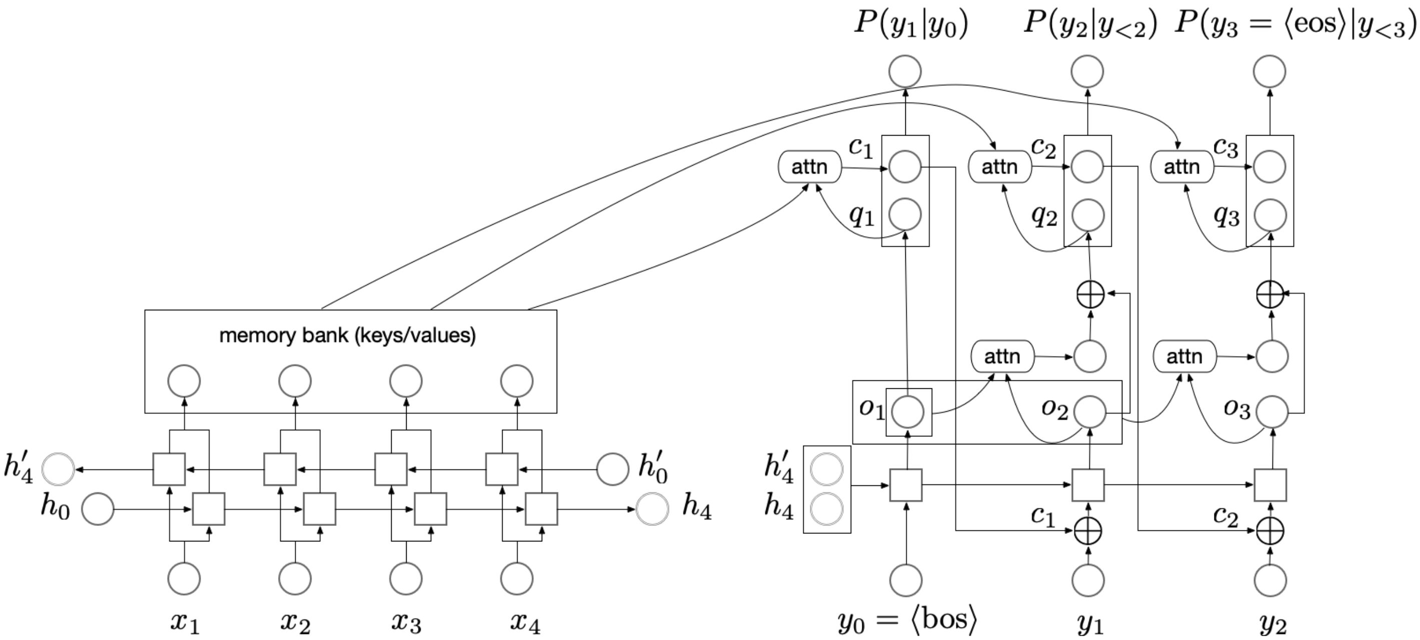
```
  0%|          | 0/332 [00:00<?, ?it/s]/usr/local/lib/python3.7/dist-packages/ipykernel_
100%|██████████| 332/332 [01:15<00:00,  4.41it/s]precision: 0.39
recall:    0.39
F1:        0.39
```

Goal 3: Implement a seq2seq model (with cross attention and self attention)

In the previous section, you have implemented a seq2seq model with attention. The attention mechanism used in that section is usually referred to as "cross-attention", as at each decoding step, the decoder attends to encoder outputs, enabling a dynamic view on the encoder side as decoding proceeds.

Similarly, we can have a dynamic view on the decoder side as well as decoding proceeds, i.e., the decoder attends to decoder outputs at previous steps. This is called "self attention", and has been found very useful in modern neural architectures such as transformers.

Augment the seq2seq model you implemented before with a decoder self-attention mechanism as class `AttnEncoderDecoder2`. A model diagram can be found below:



At each decoding step, the decoder LSTM first produces an output state o_t, then it attends to all previous output states $o_1, \ldots, o_{t-1}$ (decoder self-attention). You need to special case the first decoding step to not perform self-attention, as there are no previous decoder states. The attention result is added to o_t itself and the sum is used as q_t to attend to the encoder side (encoder-decoder cross-attention). The rest of the model is the same as encoder-decoder with attention.

```
#TODO - implement the `AttnEncoderDecoder2` class.
class AttnEncoderDecoder2(nn.Module):
  def __init__(self, src_field, tgt_field, hidden_size=64, layers=3):
    """
    Initializer. Creates network modules and loss function.
    Arguments:
        src_field: src field
        tgt_field: tgt field
        hidden_size: hidden layer size of both encoder and decoder
        layers: number of layers of both encoder and decoder
    """
    super().__init__()
    self.src_field = src_field
    self.tgt_field = tgt_field

    # Keep the vocabulary sizes available
    self.V_src = len(src_field.vocab.itos)
    self.V_tgt = len(tgt_field.vocab.itos)

    # Get special word ids
    self.padding_id_src = src_field.vocab.stoi[src_field.pad_token]
    self.padding_id_tgt = tgt_field.vocab.stoi[tgt_field.pad_token]
```

```
    self.bos_id = tgt_field.vocab.stoi[tgt_field.init_token]
    self.eos_id = tgt_field.vocab.stoi[tgt_field.eos_token]

    # Keep hyper-parameters available
    self.embedding_size = hidden_size
    self.hidden_size = hidden_size
    self.layers = layers

    # Create essential modules
    self.word_embeddings_src = nn.Embedding(self.V_src, self.embedding_size)
    self.word_embeddings_tgt = nn.Embedding(self.V_tgt, self.embedding_size)

    # RNN cells
    self.encoder_rnn = nn.LSTM(
      input_size    = self.embedding_size,
      hidden_size   = hidden_size // 2, # to match decoder hidden size
      num_layers    = layers,
      bidirectional = True              # bidirectional encoder
    )
    self.decoder_rnn = nn.LSTM(
      input_size    = self.embedding_size,
      hidden_size   = hidden_size,
      num_layers    = layers,
      bidirectional = False             # unidirectional decoder
    )

    # Final projection layer
    self.hidden2output = nn.Linear(2*hidden_size, self.V_tgt) # project the concatenation to

    # Create loss function
    self.loss_function = nn.CrossEntropyLoss(reduction='sum',
                                             ignore_index=self.padding_id_tgt)

  def forward_encoder(self, src, src_lengths):
    """
    Encodes source words `src`.
    Arguments:
        src: src batch of size (max_src_len, bsz)
        src_lengths: src lengths of size (bsz)
    Returns:
        memory_bank: a tensor of size (src_len, bsz, hidden_size)
        (final_state, context): `final_state` is a tuple (h, c) where h/c is of size
                                (layers, bsz, hidden_size), and `context` is `None`.
    """

    src_embeddings = self.word_embeddings_src(src)
    padded = pack(src_embeddings, src_lengths.cpu()) # batch_first=False (default) b/c seqLen
    memory_bank, (h, c) = self.encoder_rnn(padded)
    memory_bank, _ = unpack(memory_bank)

    h_reshape = h.reshape(int(h.shape[0]/2), 2, h.shape[1], h.shape[2])
```

```
    c_reshape = c.reshape(int(c.shape[0]/2), 2, c.shape[1], c.shape[2])
    l2r_h = h_reshape[:, 0]
    r2l_h = h_reshape[:, 1]
    l2r_c = c_reshape[:, 0]
    r2l_c = c_reshape[:, 1]
    join_h = torch.cat([l2r_h, r2l_h], dim=-1)
    join_c = torch.cat([l2r_c, r2l_c], dim=-1)
    final_state = (join_h, join_c)
    context = None
    prev_decoder_outs = None #new variable for self-attention
    return memory_bank, (final_state, context,prev_decoder_outs)

  def forward_decoder(self, encoder_final_state, tgt_in, memory_bank, src_mask):
    """
    Decodes based on encoder final state, memory bank, src_mask, and ground truth
    target words.
    Arguments:
        encoder_final_state: (final_state, None) where final_state is the encoder
                             final state used to initialize decoder. None is the
                             initial context (there's no previous context at the
                             first step).
        tgt_in: a tensor of size (tgt_len, bsz)
        memory_bank: a tensor of size (src_len, bsz, hidden_size), encoder outputs
                     at every position
        src_mask: a tensor of size (src_len, bsz): a boolean tensor, `False` where
                  src is padding (we disallow decoder to attend to those places).
    Returns:
        Logits of size (tgt_len, bsz, V_tgt) (before the softmax operation)
    """
    max_tgt_length = tgt_in.size(0)

    # Initialize decoder state, note that it's a tuple (state, context) here
    decoder_states = encoder_final_state

    all_logits = []
    for i in range(max_tgt_length):
      logits, decoder_states, attn = \
        self.forward_decoder_incrementally(decoder_states,
                                           tgt_in[i],
                                           memory_bank,
                                           src_mask,
                                           normalize=False)
      all_logits.append(logits)             # list of bsz, vocab_tgt
    all_logits = torch.stack(all_logits, 0) # tgt_len, bsz, vocab_tgt
    return all_logits

  def forward(self, src, src_lengths, tgt_in):
    """
    Performs forward computation, returns logits.
    Arguments:
        src: src batch of size (max_src_len, bsz)
```

```
            src_lengths: src lengths of size (bsz)
            tgt_in:  a tensor of size (tgt_len, bsz)
        """
        src_mask = src.ne(self.padding_id_src) # max_src_len, bsz
        # Forward encoder
        memory_bank, encoder_final_state = self.forward_encoder(src, src_lengths)
        # Forward decoder
        logits = self.forward_decoder(encoder_final_state, tgt_in, memory_bank, src_mask)
        return logits

    def forward_decoder_incrementally(self, prev_decoder_states, tgt_in_onestep,
                                      memory_bank, src_mask,
                                      normalize=True):
        """
        Forward the decoder for a single step with token `tgt_in_onestep`.
        This function will be used both in `forward_decoder` and in beam search.
        Note that bsz can be greater than 1.
        Arguments:
            prev_decoder_states: a tuple (prev_decoder_state, prev_context). `prev_context`
                                 is `None` for the first step
            tgt_in_onestep: a tensor of size (bsz), tokens at one step
            memory_bank: a tensor of size (src_len, bsz, hidden_size), encoder outputs
                         at every position
            src_mask: a tensor of size (src_len, bsz): a boolean tensor, `False` where
                      src is padding (we disallow decoder to attend to those places).
            normalize: use log_softmax to normalize or not. Beam search needs to normalize,
                       while `forward_decoder` does not
        Returns:
            logits: log probabilities for `tgt_in_token` of size (bsz, V_tgt)
            decoder_states: (`decoder_state`, `context`) which will be used for the
                            next incremental update
            attn: normalized attention scores at this step (bsz, src_len)
        """

        prev_decoder_state, prev_context, prev_decoder_outs = prev_decoder_states

        tgt_embeddings = self.word_embeddings_tgt(tgt_in_onestep.to(device)).to(device)

        if prev_context is not None:
            decoder_inp = tgt_embeddings + prev_context
            decoder_inp = decoder_inp.unsqueeze(0)
        else:
            decoder_inp = tgt_embeddings
            decoder_inp = decoder_inp.unsqueeze(0)

        decoder_outs, decoder_state = self.decoder_rnn(decoder_inp, prev_decoder_state)

        src_mask = torch.transpose(src_mask, 0, 1).unsqueeze(1)

        #check to see if we have previously run decoder
        if prev_decoder_outs is not None:
```

```
      pre_attn, context = attention(decoder_outs, prev_decoder_outs, prev_decoder_outs)
      #update decoder outs with the context from the previous
      decoder_outs_updated = decoder_outs + context
      prev_decoder_outs = torch.cat((prev_decoder_outs,decoder_outs), dim =0)
    else:
      decoder_outs_updated = decoder_outs
      prev_decoder_outs = decoder_outs

    #run attention (again in some cases) which will be used for the next step
    attn, context = attention(decoder_outs_updated, memory_bank, memory_bank, mask=src_mask)

    decoder_outs_updated = decoder_outs_updated.squeeze(0)
    attn = attn.squeeze(1)
    context = context.squeeze(0)
    concat_out = torch.cat((decoder_outs_updated, context), dim=1)

    #make sure to update decoder states for a tuple of 3
    decoder_states = (decoder_state, context, prev_decoder_outs)

    logits = self.hidden2output(concat_out).squeeze(0)
    if normalize:
      logits = torch.log_softmax(logits, dim=-1)
    return logits, decoder_states, attn

  def evaluate_ppl(self, iterator):
    """Returns the model's perplexity on a given dataset `iterator`."""
    # Switch to eval mode
    self.eval()
    total_loss = 0
    total_words = 0
    for batch in iterator:
      # Input and target
      src, src_lengths = batch.src
      tgt = batch.tgt # max_length_sql, bsz
      tgt_in = tgt[:-1] # remove <eos> for decode input (y_0=<bos>, y_1, y_2)
      tgt_out = tgt[1:] # remove <bos> as target        (y_1, y_2, y_3=<eos>)
      # Forward to get logits
      logits = self.forward(src, src_lengths, tgt_in)
      # Compute cross entropy loss
      loss = self.loss_function(logits.view(-1, self.V_tgt), tgt_out.view(-1))
      total_loss += loss.item()
      total_words += tgt_out.ne(self.padding_id_tgt).float().sum().item()
    return math.exp(total_loss/total_words)

  def train_all(self, train_iter, val_iter, epochs=10, learning_rate=0.001):
    """Train the model."""
    # Switch the module to training mode
    self.train()
    # Use Adam to optimize the parameters
    optim = torch.optim.Adam(self.parameters(), lr=learning_rate)
    best_validation_ppl = float('inf')
```

```
    best_model = None
    # Run the optimization for multiple epochs
    for epoch in range(epochs):
      total_words = 0
      total_loss = 0.0
      for batch in tqdm(train_iter):
        # Zero the parameter gradients
        self.zero_grad()
        # Input and target
        src, src_lengths = batch.src # text: max_src_length, bsz
        tgt = batch.tgt # max_tgt_length, bsz
        tgt_in = tgt[:-1] # Remove <eos> for decode input (y_0=<bos>, y_1, y_2)
        tgt_out = tgt[1:] # Remove <bos> as target        (y_1, y_2, y_3=<eos>)
        bsz = tgt.size(1)
        # Run forward pass and compute loss along the way.
        logits = self.forward(src, src_lengths, tgt_in)
        loss = self.loss_function(logits.view(-1, self.V_tgt), tgt_out.view(-1))
        # Training stats
        num_tgt_words = tgt_out.ne(self.padding_id_tgt).float().sum().item()
        total_words += num_tgt_words
        total_loss += loss.item()
        # Perform backpropagation
        loss.div(bsz).backward()
        optim.step()

      # Evaluate and track improvements on the validation dataset
      validation_ppl = self.evaluate_ppl(val_iter)
      self.train()
      if validation_ppl < best_validation_ppl:
        best_validation_ppl = validation_ppl
        self.best_model = copy.deepcopy(self.state_dict())
      epoch_loss = total_loss / total_words
      print (f'Epoch: {epoch} Training Perplexity: {math.exp(epoch_loss):.4f} '
             f'Validation Perplexity: {validation_ppl:.4f}')

  def predict(self, tokens, K, max_T):

    #uses beam search with the tokens passed in
    beam_searcher = BeamSearcher(model)
    src, src_lengths = self.src_field.process([tokens])
    src = src.to(device)
    prediction, _ = beam_searcher.beam_search(src, src_lengths, K, max_T)

    #makes the outputs more readable and actual queries
    prediction = ' '.join([TGT.vocab.itos[token] for token in prediction])
    prediction = prediction.lstrip('<bos>').rstrip('<eos>').strip()

    return prediction

EPOCHS = 30 # epochs, we recommend starting with a smaller number like 1
LEARNING_RATE = 1e-4 # learning rate
```

```
# Instantiate and train classifier
model2 = AttnEncoderDecoder2(SRC, TGT,
  hidden_size    = 1024,
  layers         = 1,
).to(device)

model2.train_all(train_iter, val_iter, epochs=EPOCHS, learning_rate=LEARNING_RATE)
model2.load_state_dict(model2.best_model)

# Evaluate model performance, the expected value should be < 1.2
print (f'Validation perplexity: {model2.evaluate_ppl(val_iter):.3f}')
```

```
100%|██████████| 229/229 [04:11<00:00,  1.10s/it]
Epoch: 0 Training Perplexity: 3.9621 Validation Perplexity: 1.8483
100%|██████████| 229/229 [04:10<00:00,  1.09s/it]
Epoch: 1 Training Perplexity: 1.5532 Validation Perplexity: 1.4992
100%|██████████| 229/229 [04:09<00:00,  1.09s/it]
Epoch: 2 Training Perplexity: 1.3617 Validation Perplexity: 1.3751
100%|██████████| 229/229 [04:06<00:00,  1.08s/it]
Epoch: 3 Training Perplexity: 1.2718 Validation Perplexity: 1.2775
100%|██████████| 229/229 [04:08<00:00,  1.08s/it]
Epoch: 4 Training Perplexity: 1.2148 Validation Perplexity: 1.2406
100%|██████████| 229/229 [04:10<00:00,  1.09s/it]
Epoch: 5 Training Perplexity: 1.1794 Validation Perplexity: 1.2165
100%|██████████| 229/229 [04:04<00:00,  1.07s/it]
Epoch: 6 Training Perplexity: 1.1474 Validation Perplexity: 1.2018
100%|██████████| 229/229 [04:07<00:00,  1.08s/it]
Epoch: 7 Training Perplexity: 1.1280 Validation Perplexity: 1.1739
100%|██████████| 229/229 [04:10<00:00,  1.09s/it]
Epoch: 8 Training Perplexity: 1.1065 Validation Perplexity: 1.1526
100%|██████████| 229/229 [04:07<00:00,  1.08s/it]
Epoch: 9 Training Perplexity: 1.0919 Validation Perplexity: 1.1521
100%|██████████| 229/229 [04:06<00:00,  1.08s/it]
Epoch: 10 Training Perplexity: 1.0840 Validation Perplexity: 1.1406
100%|██████████| 229/229 [04:08<00:00,  1.09s/it]
Epoch: 11 Training Perplexity: 1.0715 Validation Perplexity: 1.1237
100%|██████████| 229/229 [04:09<00:00,  1.09s/it]
Epoch: 12 Training Perplexity: 1.0601 Validation Perplexity: 1.1251
100%|██████████| 229/229 [04:05<00:00,  1.07s/it]
Epoch: 13 Training Perplexity: 1.0577 Validation Perplexity: 1.1291
100%|██████████| 229/229 [04:04<00:00,  1.07s/it]
Epoch: 14 Training Perplexity: 1.0514 Validation Perplexity: 1.1176
100%|██████████| 229/229 [04:06<00:00,  1.08s/it]
Epoch: 15 Training Perplexity: 1.0475 Validation Perplexity: 1.1158
100%|██████████| 229/229 [04:09<00:00,  1.09s/it]
Epoch: 16 Training Perplexity: 1.0440 Validation Perplexity: 1.1128
100%|██████████| 229/229 [04:06<00:00,  1.08s/it]
Epoch: 17 Training Perplexity: 1.0367 Validation Perplexity: 1.1137
100%|██████████| 229/229 [04:05<00:00,  1.07s/it]
Epoch: 18 Training Perplexity: 1.0337 Validation Perplexity: 1.1119
100%|██████████| 229/229 [04:07<00:00,  1.08s/it]
Epoch: 19 Training Perplexity: 1.0272 Validation Perplexity: 1.1186
100%|██████████| 229/229 [04:05<00:00,  1.07s/it]
Epoch: 20 Training Perplexity: 1.0275 Validation Perplexity: 1.1105
```

```
100%|██████████| 229/229 [04:08<00:00,  1.09s/it]
Epoch: 21 Training Perplexity: 1.0244 Validation Perplexity: 1.1068
100%|██████████| 229/229 [04:06<00:00,  1.07s/it]
Epoch: 22 Training Perplexity: 1.0223 Validation Perplexity: 1.1141
100%|██████████| 229/229 [04:07<00:00,  1.08s/it]
Epoch: 23 Training Perplexity: 1.0270 Validation Perplexity: 1.1110
100%|██████████| 229/229 [04:08<00:00,  1.09s/it]
Epoch: 24 Training Perplexity: 1.0224 Validation Perplexity: 1.1054
100%|██████████| 229/229 [04:04<00:00,  1.07s/it]
Epoch: 25 Training Perplexity: 1.0190 Validation Perplexity: 1.1036
100%|██████████| 229/229 [04:04<00:00,  1.07s/it]
Epoch: 26 Training Perplexity: 1.0152 Validation Perplexity: 1.1012
100%|██████████| 229/229 [04:10<00:00,  1.09s/it]
Epoch: 27 Training Perplexity: 1.0159 Validation Perplexity: 1.1031
100%|██████████| 229/229 [04:02<00:00,  1.06s/it]
Epoch: 28 Training Perplexity: 1.0127 Validation Perplexity: 1.1038
```

Evaluation

Now we are ready to run the full evaluation. A proper implementation should reach more than 35% precision/recall/F1.

```
def seq2seq_predictor2(tokens):
  prediction = model2.predict(tokens, K=1, max_T=400)
  return prediction
```

```
precision, recall, f1 = evaluate(seq2seq_predictor2, test_iter.dataset, num_examples=0)
print(f"precision: {precision:3.2f}")
print(f"recall:    {recall:3.2f}")
print(f"F1:        {f1:3.2f}")
```

```
  0%|          | 0/332 [00:00<?, ?it/s]/usr/local/lib/python3.7/dist-packages/ipykernel_
100%|██████████| 332/332 [01:20<00:00,  4.12it/s]precision: 0.39
recall:    0.39
F1:        0.39
```

Discussion

Goal 4: Compare the pros and cons of rule-based and neural approaches.

Compare the pros and cons of the rule-based approach and the neural approaches with relevant examples from your experiments above. Concerning the accuracy, which approach would you choose to be used in a product? Explain.

First, let's consider the pros and cons of each system in the abstract. The pros of a rule-based system are that you can directly see how it is working and "connect the dots" should there be an error. You can then assess that error and hopefully catch more cases that may come in the future. One could also make the argument that the rule-based approach is more natural, as it is based off of natural language. However, its biggest pros could also be its biggest cons. If a query comes out incorrect, then someone must manually program the rule to fix it. In general, the con of this approach is that everything must be done by hand. Compare this to the neural approaches. Here, a con is that we can't really see in detail what's going on in the neural system – that is to say that, if a query returns the wrong results, we cannot "trace back" to find out why, we can only give it more data. That being said, a neural model can continually improve over time as more training and testing data become available, meaning that it can learn form it's mistakes with much less human interaction than the rule-based approach. In addition, it is easy to convert scenarios for neural approaches, which we saw in this very project. For example, we took much of the code from lab 4-5, which had nothing to do with ATIS, however it is much more of a transformation system which can be adapted very easily to other purposes.

We can see these play out in our examples. With our rule-based approach, we were building the system around the examples specifically – this allowed us to get 100% on the queries we knew it would ask, while also making it easy to trace back when debugging (I found that feature particularly useful as I was testing different augmentations). Furthermore, if people ask for things in a natural way (which of course we can't guarantee), we can be surer that our grammar will encompass their needs (for precision in my augmentation, I ended up getting around 65% overall). Compare this to our neural models, which not only took *much* longer to run (30 epics took around two hours), but also only gave a precision of about 39% (I got 43% in my best test scenarios though).

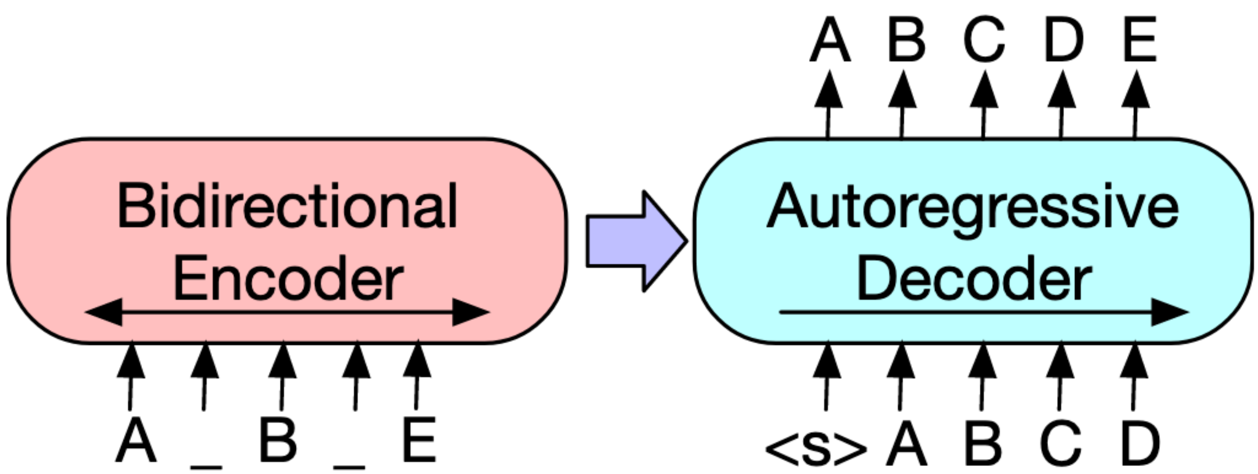
I would generalize to say that rule-based is easier to decipher, faster to run, but requires much more human attention throughout its use, while the neural approach can be much more hands off and adaptable but can also yield less accurate results with little to do but give it more data. For these reasons, I would choose the rule-based approach in a product, as it is much easier to debug and (at least from what we've seen) is both faster and more accurate.

(Optional) Goal 5: Use state-of-the-art pretrained transformers

The most recent breakthrough in natural-language processing stems from the use of pretrained transformer models. For example, you might have heard of pretrained transformers such as [GPT-3] and [BERT]. (BERT is already used in [Google search].) These models are usually trained on vast amounts of text data using variants of language modeling objectives, and researchers have found that finetuning them on downstream tasks usually results in better performance as compared to training a model from scratch.

In the previous part, you implemented an LSTM-based sequence-to-sequence approach. To "upgrade" the model to be a state-of-the-art pretrained transformer only requires minor modifications.

The pretrained model that we will use is [BART](), which uses a bidirectional transformer encoder and a unidirectional transformer decoder, as illustrated in the below diagram (image courtesy [https://arxiv.org/pdf/1910.13461]()):



We can see that this model is strikingly similar to the LSTM-based encoder-decoder model we've been using. The only difference is that they use transformers instead of LSTMs. Therefore, we only need to change the modeling parts of the code, as we will see later.

[ ] ↳ *26 cells hidden*

Debrief

Question: We're interested in any thoughts you have about this project segment so that we can improve it for later years, and to inform later segments for this year. Please list any issues that arose or comments you have to improve the project segment. Useful things to comment on might include the following:

- Was the project segment clear or unclear? Which portions?
- Were the readings appropriate background for the project segment?
- Are there additions or changes you think would make the project segment better?

but you should comment on whatever aspects you found especially positive or negative.

I think that, while pretty difficult, the labs and readings did a fine job of preparing me for this project. I thought that the segments were clear enough - with maybe the exception being Goal 3 (the post on Ed was incredibly helpful and should maybe be included in the future). The biggest problems were with Colab - it was honestly pretty bad. Having to copy and create a new drive (had to use three drives for this) every so often was pretty awful - in general you may want to find a way to make the course a bit more PC friendly, as I used Colab all semester and it wasn't great. Especially for something like this, where I had to be connected to the internet at all times, it was basically impossible to do in the background, and it could freeze and need to be restarted at any moment. Overall though the project was satisfying as always to complete.

Instructions for submission of the project segment

This project segment should be submitted to Gradescope at http://go.cs187.info/project4-submit-code and http://go.cs187.info/project4-submit-pdf, which will be made available some time before the due date.

Project segment notebooks are manually graded, not autograded using otter as labs are. (Otter is used within project segment notebooks to synchronize distribution and solution code however.) **We will not run your notebook before grading it.** Instead, we ask that you submit the already freshly run notebook. The best method is to "restart kernel and run all cells", allowing time for all cells to be run to completion. You should submit your code to Gradescope at the code submission assignment at http://go.cs187.info/project4-submit-code. Make sure that you are also submitting your `data/grammar` file as part of your solution code as well.

We also request that you **submit a PDF of the freshly run notebook**. The simplest method is to use "Export notebook to PDF", which will render the notebook to PDF via LaTeX. If that doesn't work, the method that seems to be most reliable is to export the notebook as HTML (if you are using Jupyter Notebook, you can do so using `File -> Print Preview`), open the HTML in a browser, and print it to a file. Then make sure to add the file to your git commit. Please name the file the same name as this notebook, but with a `.pdf` extension. (Conveniently, the methods just described will use that name by default.) You can then perform a git commit and push and submit the commit to Gradescope at http://go.cs187.info/project4-submit-pdf.

End of project segment 4 {-}

0s completed at 11:14 PM