In [67]:

```
# 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/project2.git .tmp
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

In [68]:

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

%%latex \newcommand{\vect}[1]{\mathbf{#1}} \newcommand{\cnt}[1]{\sharp(#1)} \newcommand{\argmax}[1]{\underset{#1}{\operatorname{argmax}}} \newcommand{\softmax}{\operatorname{softmax}} \newcommand{\Prob}{\Pr} \newcommand{\given}{\,|\,}

CS187

Project 2: Sequence labeling – The slot filling task

Introduction

The second segment of the project involves a sequence labeling task, in which the goal is to label the tokens in a text. Many NLP tasks have this general form. Most famously is the task of *part-of-speech labeling* as you explored in lab 2-4, where the tokens in a text are to be labeled with their part of speech (noun, verb, preposition, etc.). In this project segment, however, you'll use sequence labeling to implement a system for filling the slots in a template that is intended to describe the meaning of an ATIS query. For instance, the sentence

```
What's the earliest arriving flight between Boston and Washington
DC?
```

might be associated with the following slot-filled template:

```
flight_id
    fromloc.cityname: boston
    toloc.cityname: washington
    toloc.state: dc
    flight_mod: earliest arriving
```

You may wonder how this task is a sequence labeling task. We label each word in the source sentence with a tag taken from a set of tags that correspond to the slot-labels. For each slot-label, say `flight_mod`, there are two tags: `B-flight_mod` and `I-flight_mod`. These are used to mark the beginning (B) or interior (I) of a phrase that fills the given slot. In addition, there is a tag for other (O) words that are not used to fill any slot. (This technique is thus known as IOB encoding.) Thus the sample sentence would be labeled as follows:

Token	Label
`BOS`	`O`
`what's`	`O`
`the`	`O`
`earliest`	`B-flight_mod`
`arriving`	`I-flight_mod`
`flight`	`O`
`between`	`O`
`boston`	`B-fromloc.city_name`
`and`	`O`
`washington`	`B-toloc.city_name`
`dc`	`B-toloc.state_code`
`EOS`	`O`

> See below for information about the `BOS` and `EOS` tokens.

The template itself is associated with the question type for the sentence, perhaps as recovered from the sentence in the last project segment.

In this segment, you'll implement three methods for sequence labeling: a hidden Markov model (HMM) and two recurrent neural networks, a simple RNN and a long short-term memory network (LSTM). By the end of this homework, you should have grasped the pros and cons of the statistical and neural approaches.

Goals

1. Implement an HMM-based approach to sequence labeling.
2. Implement an RNN-based approach to sequence labeling.
3. Implement an LSTM-based approach to sequence labeling.
4. (Optional) Compare the performances of HMM and RNN/LSTM with different amounts of training data. Discuss the pros and cons of the HMM approach and the neural approach.

Setup

In [69]:

```
import copy
import math
import matplotlib.pyplot as plt
import random

import wget
import torch
import torch.nn as nn
import torchtext.legacy as tt

from tqdm.auto import tqdm
```

In [70]:

```
# Set random seeds
seed = 1234
random.seed(seed)
torch.manual_seed(seed)

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

```
cpu
```

Loading data

We download the ATIS dataset, already presplit into training, validation (dev), and test sets.

In [71]:

```
# Prepare to download needed data
def download_if_needed(filename, source='./', dest='./'):
    os.makedirs(data_path, exist_ok=True) # ensure destination
    os.path.exists(f"./{dest}{filename}") or wget.download(source + filename, ou

source_path = "https://raw.githubusercontent.com/nlp-course/data/master/ATIS/"
data_path = "data/"

# Download files
for filename in ["atis.train.txt",
                 "atis.dev.txt",
                 "atis.test.txt"
```

```
                ]:
    download_if_needed(filename, source_path, data_path)
```

Data preprocessing

We again use `torchtext` to load data and convert words to indices in the vocabulary. We use one field `TEXT` for processing the question, and another field `TAG` for processing the sequence labels.

We treat words occurring fewer than three times in the training data as *unknown words*. They'll be replaced by the unknown word type `<unk>` .

In [72]:

```
MIN_FREQ = 3

TEXT = tt.data.Field(init_token="<bos>", batch_first=False) # batches are of siz
TAG = tt.data.Field(init_token="<bos>", batch_first=False)  # ditto
fields = (('text', TEXT), ('tag', TAG))

train, val, test = tt.datasets.SequenceTaggingDataset.splits(
  fields=fields,
  path='./data/',
  train='atis.train.txt',
  validation='atis.dev.txt',
  test='atis.test.txt'
)

TEXT.build_vocab(train.text, min_freq=MIN_FREQ)
TAG.build_vocab(train.tag)
```

We can get some sense of the datasets by looking at the size and some elements of the text and tag vocabularies.

In [73]:

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

print(f"Number of tags: {len(TAG.vocab)}")
print(f"Most common tags: {TAG.vocab.freqs.most_common(10)}")
```

```
Size of English vocabulary: 518
Most common English words: [('BOS', 4274), ('EOS', 4274), ('to', 3682), ('from',
3203), ('flights', 2075), ('the', 1745), ('on', 1343), ('flight', 1035), ('me',
1005), ('what', 985)]

Number of tags: 104
Most common tags: [('O', 38967), ('B-toloc.city_name', 3751), ('B-fromloc.city_n
ame', 3726), ('I-toloc.city_name', 1039), ('B-depart_date.day_name', 835), ('I-f
romloc.city_name', 636), ('B-airline_name', 610), ('B-depart_time.period_of_da
y', 555), ('I-airline_name', 374), ('B-depart_date.day_number', 351)]
```

Special tokens and tags

You'll have already noticed the `BOS` and `EOS` , special tokens that the dataset developers used to indicate the beginning and end of the sentence; we'll leave them in the data.

We've also passed in `init_token="<bos>"` for both torchtext fields. Torchtext will prepend these to the sequence of words and tags. This relieves us from estimating the initial distribution of tags and tokens in HMMs, since we always start with a token `<bos>` whose tag is also `<bos>`. We'll be able to refer to these tags as exemplified here:

In [74]:
```
print(f"""
Initial tag string: {TAG.init_token}
Initial tag id:     {TAG.vocab.stoi[TAG.init_token]}
""")
```

```
Initial tag string: <bos>
Initial tag id:     2
```

Finally, since `torchtext` will be providing the sentences in the training corpus in "batches", `torchtext` will force the sentences within a batch to be the same length by padding them with a special token. Again, we can access that token as shown here:

In [75]:
```
print(f"""
Pad tag string: {TAG.pad_token}
Pad tag id:     {TAG.vocab.stoi[TAG.pad_token]}
""")
```

```
Pad tag string: <pad>
Pad tag id:     1
```

Now, we can iterate over the dataset using `torchtext`'s iterator. We'll use a non-trivial batch size to gain the benefit of training on multiple sentences at a shot. You'll need to be careful about the shapes of the various tensors that are being manipulated.

In [76]:
```
BATCH_SIZE = 20

train_iter, val_iter, test_iter = tt.data.BucketIterator.splits(
    (train, val, test),
    batch_size=BATCH_SIZE,
    repeat=False,
    device=device)
```

Each batch will be a tensor of size `max_length x batch_size`. Let's examine a batch.

In [77]:
```
# Get the first batch
batch = next(iter(train_iter))

# What's its shape? Should be max_length x batch_size.
print(f'Shape of batch text tensor: {batch.text.shape}\n')

# Extract the first sentence in the batch, both text and tags
first_sentence = batch.text[:, 0]
first_tags = batch.tag[:, 0]

# Print out the first sentence, as token ids and as text
print("First sentence in batch")
print(f"{first_sentence}")
print(f"{' '.join([TEXT.vocab.itos[i] for i in first_sentence])}\n")
```

```
print("First tags in batch")
print(f"{first_tags}")
print(f"{[TAG.vocab.itos[i] for i in first_tags]}")
```

```
Shape of batch text tensor: torch.Size([22, 20])

First sentence in batch
tensor([ 2,  3, 21, 45, 88, 44,  7, 39, 28, 20, 54, 18, 22,  4,  1,  1,  1,  1,
         1,  1,  1,  1])
<bos> BOS i need information for flights leaving baltimore and arriving in atlan
ta EOS <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad>

First tags in batch
tensor([2, 3, 3, 3, 3, 3, 3, 3, 5, 3, 3, 3, 4, 3, 1, 1, 1, 1, 1, 1, 1, 1])
['<bos>', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'B-fromloc.city_name', 'O', 'O',
'O', 'B-toloc.city_name', 'O', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pa
d>', '<pad>', '<pad>']
```

The goal of this project is to predict the sequence of tags `batch.tag` given a sequence of words `batch.text`.

Majority class labeling

As usual, we can get a sense of the difficulty of the task by looking at a simple baseline, tagging every token with the majority tag. Here's a table of tag frequencies for the most frequent tags:

In [78]:

```
def count_tags(iterator):
  tag_counts = torch.zeros(len(TAG.vocab.itos), device=device)

  for batch in iterator:
    tags = batch.tag.view(-1)
    tag_counts.scatter_add_(0, tags, torch.ones(tags.shape).to(device))

  ## Alternative untensorized implementation for reference
  # for batch in iterator:                    # for each batch
  #   for sent_id in range(len(batch)):       # ... each sentence in the batch
  #     for tag in batch.tag[:, sent_id]:     # ... each tag in the sentence
  #       tag_counts[tag] += 1                # bump the tag count

  # Ignore paddings
  tag_counts[TAG.vocab.stoi[TAG.pad_token]] = 0
  return tag_counts

tag_counts = count_tags(train_iter)

for tag_id in range(len(TAG.vocab.itos)):
  print(f'{tag_id:3}  {TAG.vocab.itos[tag_id]:30}{tag_counts[tag_id].item():3.0f
```

```
  0  <unk>                            0
  1  <pad>                            0
  2  <bos>                         4274
  3  O                            38967
  4  B-toloc.city_name             3751
  5  B-fromloc.city_name           3726
  6  I-toloc.city_name             1039
  7  B-depart_date.day_name         835
```

```
 8  I-fromloc.city_name            636
 9  B-airline_name                 610
10  B-depart_time.period_of_day    555
11  I-airline_name                 374
12  B-depart_date.day_number       351
13  B-depart_date.month_name       340
14  B-depart_time.time             321
15  B-round_trip                   311
16  I-round_trip                   303
17  B-depart_time.time_relative    290
18  B-cost_relative                281
19  B-flight_mod                   264
20  I-depart_time.time             258
21  B-stoploc.city_name            202
22  B-city_name                    191
23  B-arrive_time.time             182
24  B-class_type                   181
25  B-arrive_time.time_relative    162
26  I-class_type                   148
27  I-arrive_time.time             142
28  B-flight_stop                  141
29  B-airline_code                 109
30  I-depart_date.day_number       105
31  I-fromloc.airport_name         103
32  B-toloc.state_name              84
33  B-toloc.state_code              81
34  B-arrive_date.day_name          78
35  B-fromloc.airport_name          75
36  B-depart_date.date_relative     72
37  B-flight_number                 72
38  B-depart_date.today_relative    70
39  I-airport_name                  61
40  I-city_name                     53
41  B-arrive_time.period_of_day     51
42  B-fare_basis_code               51
43  B-flight_time                   51
44  B-fromloc.state_code            51
45  B-or                            49
46  B-aircraft_code                 48
47  B-meal_description              48
48  B-meal                          47
49  I-cost_relative                 45
50  I-stoploc.city_name             45
51  B-airport_name                  44
52  B-transport_type                43
53  B-fromloc.state_name            42
54  B-arrive_date.day_number        40
55  B-arrive_date.month_name        40
56  B-depart_time.period_mod        39
57  B-flight_days                   37
58  B-connect                       36
59  I-toloc.airport_name            35
60  B-fare_amount                   34
61  I-fare_amount                   33
62  B-economy                       32
63  B-toloc.airport_name            28
64  B-mod                           24
65  I-flight_time                   24
66  B-airport_code                  22
67  B-depart_date.year              20
68  B-toloc.airport_code            19
```

```
69   B-arrive_time.start_time       18
70   B-depart_time.end_time         18
71   B-depart_time.start_time       18
72   I-transport_type               18
73   B-arrive_time.end_time         17
74   I-arrive_time.end_time         16
75   B-fromloc.airport_code         14
76   B-restriction_code             14
77   I-depart_time.end_time         13
78   I-flight_mod                   12
79   I-flight_stop                  12
80   B-arrive_date.date_relative    10
81   I-toloc.state_name             10
82   I-restriction_code              9
83   B-return_date.date_relative     8
84   I-depart_time.start_time        8
85   I-economy                       8
86   B-state_code                    7
87   I-arrive_time.start_time        7
88   I-fromloc.state_name            7
89   B-state_name                    6
90   I-depart_date.today_relative    6
91   I-depart_time.period_of_day     5
92   B-period_of_day                 4
93   I-arrive_date.day_number        4
94   B-day_name                      3
95   B-meal_code                     3
96   B-stoploc.state_code            3
97   B-arrive_time.period_mod        2
98   B-toloc.country_name            2
99   I-arrive_time.time_relative     2
100  I-meal_code                     2
101  I-return_date.date_relative     2
102  B-return_date.day_number        1
103  B-return_date.month_name        1
```

It looks like the `'O'` (other) tag is, unsurprisingly, the most frequent tag (except for the padding tag). The proportion of tokens labeled with that tag (ignoring the padding tag) gives us a good baseline accuracy for this sequence labeling task. To verify that intuition, we can calculate the accuracy of the majority tag on the test set:

In [79]:

```
tag_counts_test = count_tags(test_iter)
majority_baseline_accuracy = (
  tag_counts_test[TAG.vocab.stoi['O']]
  / tag_counts_test.sum()
)
print(f'Baseline accuracy: {majority_baseline_accuracy:.3f}')
```

```
Baseline accuracy: 0.634
```

HMM for sequence labeling

Having established the baseline to beat, we turn to implementing an HMM model.

Notation

First, let's start with some notation. We use $\mathcal{V} = \langle \mathcal{V}_1, \mathcal{V}_2, \ldots \mathcal{V}_V \rangle$ to denote the vocabulary of word types and $Q = \langle Q_1, Q_2, \ldots, Q_N \rangle$ to denote the possible tags, which is the state space of the HMM. Thus V is the number of word types in the vocabulary and N is the number of states (tags).

We use $\mathbf{w} = w_1 \cdots w_T \in \mathcal{V}^T$ to denote the string of words at "time steps" t (where t varies from 1 to T). Similarly, $\mathbf{q} = q_1 \cdots q_T \in Q^T$ denotes the corresponding sequence of states (tags).

Training an HMM by counting

Recall that an HMM is defined via a transition matrix A, which stores the probability of moving from one state Q_i to another Q_j, that is,

$$A_{ij} = \Pr(q_{t+1} = Q_j \,|\, q_t = Q_i)$$

and an emission matrix B, which stores the probability of generating word $\mathcal{V}_j$ given state Q_i, that is,

$$B_{ij} = \Pr(w_t = \mathcal{V}_j \,|\, q_t = Q_i)$$

> As is typical in notating probabilities, we'll use abbreviations
>
> $$\Pr(q_{t+1} \,|\, q_t) \equiv \Pr(q_{t+1} = Q_j \,|\, q_t = Q_i) \tag{1}$$
> $$\Pr(w_t \,|\, q_t) \equiv \Pr(w_t = \mathcal{V}_j \,|\, q_t = Q_i) \tag{2}$$
>
> where the i and j are clear from context.

In our case, since the labels are observed in the training data, we can directly use counting to determine (maximum likelihood) estimates of A and B.

Goal 1(a): Find the transition matrix

The matrix A contains the transition probabilities: A_{ij} is the probability of moving from state Q_i to state Q_j in the training data, so that $\sum_{j=1}^{N} A_{ij} = 1$ for all i.

We find these probabilities by counting the number of times state Q_j appears right after state Q_i, as a proportion of all of the transitions from Q_i.

$$A_{ij} = \frac{\sharp(Q_i, Q_j) + \delta}{\sum_k \left(\sharp(Q_i, Q_k) + \delta\right)}$$

(In the above formula, we also used add-δ smoothing.)

Using the above definition, implement the method `train_A` in the `HMM` class below, which calculates and returns the A matrix as a tensor of size $N \times N$.

> You'll want to go ahead and implement this part now, and test it below, before moving on to the next goal.

> Remember that the training data is being delivered to you batched.

Goal 1(b): Find the emission matrix B

Similar to the transition matrix, the emission matrix contains the emission probabilities such that B_{ij} is probability of word $w_t = \mathcal{V}_j$ conditioned on state $q_t = Q_i$.

We can find this by counting as well.

$$B_{ij} = \frac{\sharp(Q_i, \mathcal{V}_j) + \delta}{\sum_k \left(\sharp(Q_i, \mathcal{V}_k) + \delta\right)} = \frac{\sharp(Q_i, \mathcal{V}_j) + \delta}{\sharp(Q_i) + \delta V}$$

Using the above definitions, implement the `train_B` method in the `HMM` class below, which calculates and returns the B matrix as a tensor of size $N \times V$.

> You'll want to go ahead and implement this part now, and test it below, before moving on to the next goal.

Sequence labeling with a trained HMM

Now that you're able to train an HMM by estimating the transition matrix A and the emission matrix B, you can apply it to the task of labeling a sequence of words $\mathbf{w} = w_1 \cdots w_T$. Our goal is to find the most probable sequence of tags $\hat{\mathbf{q}} \in Q^T$ given a sequence of words $\mathbf{w} \in \mathcal{V}^T$.

$$\begin{aligned}
\hat{\mathbf{q}} &= \underset{\mathbf{q} \in Q^T}{\operatorname{argmax}}(\Pr(\mathbf{q} \mid \mathbf{w})) \\
&= \underset{\mathbf{q} \in Q^T}{\operatorname{argmax}}(\Pr(\mathbf{q}, \mathbf{w})) \\
&= \underset{\mathbf{q} \in Q^T}{\operatorname{argmax}}\left(\Pi_{t=1}^T \Pr(w_t \mid q_t) \Pr(q_t \mid q_{t-1})\right)
\end{aligned}$$

where $\Pr(w_t = \mathcal{V}_j \mid q_t = Q_i) = B_{ij}$, $\Pr(q_t = Q_j \mid q_{t-1} = Q_i) = A_{ij}$, and q_0 is the predefined initial tag `TAG.vocab.stoi[TAG.init_token]`.

Goal 1(c): Viterbi algorithm

Implement the `predict` method, which should use the Viterbi algorithm to find the most likely sequence of tags for a sequence of `words`.

> Warning: It may take up to 30 minutes to tag the entire test set depending on your implementation. (A fully tensorized implementation can be much faster though.) We highly recommend that you begin by experimenting with your code using a *very small subset* of the dataset, say two or three sentences, ramping up from there.
>
> Hint: Consider how to use vectorized computations where possible for speed.

Evaluation

We've provided you with the `evaluate` function, which takes a dataset iterator and uses `predict` on each sentence in each batch, comparing against the gold tags, to determine the accuracy of the model on the test set.

In [84]:

```
import numpy as np
class HMMTagger():
  def __init__ (self, text, tag):
    self.text = text
    self.tag = tag
    self.V = len(text.vocab.itos)     # vocabulary size
    self.N = len(tag.vocab.itos)      # state space size
    self.initial_state_id = tag.vocab.stoi[tag.init_token]
    self.pad_state_id = tag.vocab.stoi[tag.pad_token]
    self.pad_word_id = text.vocab.stoi[text.pad_token]

  def train_A(self, iterator, delta):
    """Returns A for training dataset `iterator` using add-`delta` smoothing."""
    # Create A table
    A = torch.zeros(self.N, self.N, device=device)
    #TODO: Add your solution from Goal 1(a) here.
    #      The returned value should be a tensor for the A matrix
    #      of size N x N.
    for batch in iterator:
        #print(batch.batch_size)
        #print(len(batch.tag))
        #print(batch.tag)
        for i in range(batch.batch_size):
            tags = batch.tag[:,i]
            #print(tags)
            indices = zip(tags[:-1], tags[1:])
            for first, second in indices:
                if first != self.pad_state_id:
                    A[first][second] += 1
    for row in A:
        denominator = torch.sum(row) + (delta * self.N)
        if denominator != 0:
            for i in range(self.N):
                row[i] = (row[i] + delta) / (denominator)
    #print(A)
    return A

  def train_B(self, iterator, delta):
    """Returns B for training dataset `iterator` using add-`delta` smoothing."""
    # Create B
    B = torch.zeros(self.N, self.V, device=device)

    #TODO: Add your solution from Goal 1 (b) here.
    #      The returned value should be a tensor for the $B$ matrix
    #      of size N x V.
    for batch in iterator:
        #print(len(batch.tag))
        #print(len(batch.text))
        tags = batch.tag.T
        texts = batch.text.T
        for i in range(len(tags)):
            indices = zip(tags[i], texts[i])
```

```
            for first, second in indices:
                B[first][second] += 1
    for row in B:
        denominator = torch.sum(row) + (delta * self.V)
        if denominator != 0:
            for i in range(self.V):
                row[i] = (row[i] + delta) / (denominator)
    #print(B)
    return B

  def train_all(self, iterator, delta=0.01):
    """Stores A and B (actually, their logs) for training dataset `iterator`."""
    self.log_A = self.train_A(iterator, delta).log()
    self.log_B = self.train_B(iterator, delta).log()

  def predict(self, words):
      """Returns the most likely sequence of tags for a sequence of `words`.
      Arguments:
        words: a tensor of size (seq_len,)
      Returns:
        a list of tag ids
      """
      #TODO: Add your solution from Goal 1 (c) here.
      #       The returned value should be a list of tag ids.
      seq_len = len(words)
      backpointerArr = torch.zeros(self.N, seq_len, device=device)

      tagSequence = torch.zeros(self.N, seq_len, device=device)

      for row in range(self.N):
          tagSequence[row][0] = -math.inf
      tagSequence[self.initial_state_id][0] = 1

      for col in range(1, seq_len):
          for row in range(self.N):
              prev = self.log_A[: , row] + tagSequence[: , col-1]
              tagSequence[row][col] = self.log_B[row][words[col]] + torch.max(
              backpointerArr[row][col] = torch.argmax(prev)

      bestpath = torch.zeros(len(words), device=device)

      lastCol = tagSequence[: , seq_len-1]

      maxTag = torch.argmax(lastCol)

      columnCount = seq_len - 1

      while columnCount >= 0:
          bestpath[columnCount] = maxTag
          maxTag = int(backpointerArr[maxTag][columnCount])
          columnCount -= 1

      return bestpath

  def evaluate(self, iterator):
    """Returns the model's token accuracy on a given dataset `iterator`."""
    correct = 0
    total = 0
    for batch in tqdm(iterator, leave=False):
      for sent_id in range(len(batch)):
        words = batch.text[:, sent_id]
```

```
        words = words[words.ne(self.pad_word_id)] # remove paddings
        tags_gold = batch.tag[:, sent_id]
        tags_pred = self.predict(words)
        for tag_gold, tag_pred in zip(tags_gold, tags_pred):
          if tag_gold == self.pad_state_id:  # stop once we hit padding
            break
          else:
            total += 1
            if tag_pred == tag_gold:
              correct += 1
    return correct/total
```

Putting everything together, you should now be able to train and evaluate the HMM. A correct implementation can be expected to reach above **90% test set accuracy** after running the following cell.

In [85]:

```
# Instantiate and train classifier
hmm_tagger = HMMTagger(TEXT, TAG)
hmm_tagger.train_all(train_iter)

# Evaluate model performance
print(f'Training accuracy: {hmm_tagger.evaluate(train_iter):.3f}\n'
      f'Test accuracy:     {hmm_tagger.evaluate(test_iter):.3f}')
```

```
Training accuracy: 0.915
Test accuracy:     0.906
```

RNN for Sequence Labeling

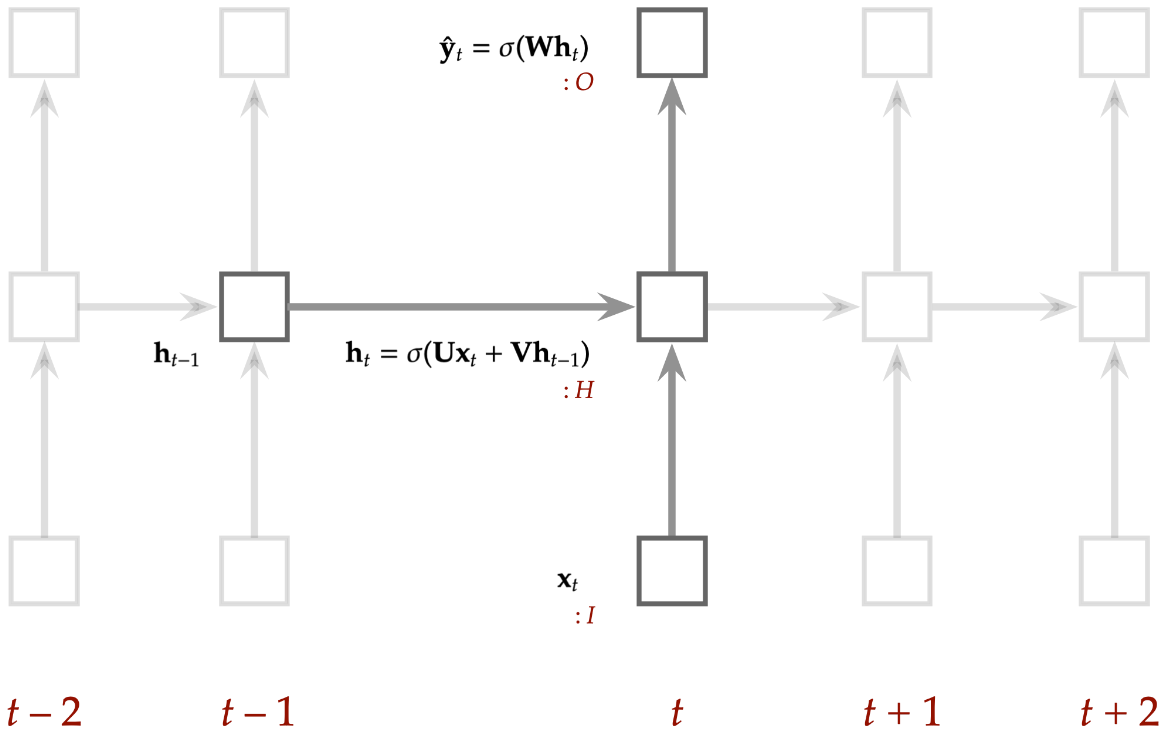
HMMs work quite well for this sequence labeling task. Now let's take an alternative (and more trendy) approach: RNN/LSTM-based sequence labeling. Similar to the HMM part of this project, you will also need to train a model on the training data, and then use the trained model to decode and evaluate some testing data.

After unfolding an RNN, the cell at time t generates the observed output $\mathbf{y}_t$ based on the input $\mathbf{x}_t$ and the hidden state of the previous cell $\mathbf{h}_{t-1}$, according to the following equations.

$$\mathbf{h}_t = \sigma(\mathbf{U}\mathbf{x}_t + \mathbf{V}\mathbf{h}_{t-1})$$
$$\hat{\mathbf{y}}_t = \operatorname{softmax}(\mathbf{W}\mathbf{h}_t)$$

The parameters here are the elements of the matrices $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{W}$. Similar to the last project segment, we will perform the forward computation, calculate the loss, and then perform the backward computation to compute the gradients with respect to these model parameters. Finally, we will adjust the parameters opposite the direction of the gradients to minimize the loss, repeating until convergence.

You've seen these kinds of neural network models before, for language modeling in lab 2-3 and sequence labeling in lab 2-5. The code there should be very helpful in implementing an `RNNTagger` class below. Consequently, we've provided very little guidance on the implementation. We do recommend you follow the steps below however.



Goal 2(a): RNN training

Implement the forward pass of the RNN tagger and the loss function. A reasonable way to proceed is to implement the following methods:

1. `forward(self, text_batch)` : Performs the RNN forward computation over a whole `text_batch` (`batch.text` in the above data loading example). The `text_batch` will be of shape `max_length x batch_size` . You might run it through the following layers: an embedding layer, which maps each token index to an embedding of size `embedding_size` (so that the size of the mapped batch becomes `max_length x batch_size x embedding_size`); then an RNN, which maps each token embedding to a vector of `hidden_size` (the size of all outputs is `max_length x batch_size x hidden_size`); then a linear layer, which maps each RNN output element to a vector of size N (which is commonly referred to as "logits", recall that $N = |Q|$, the size of the tag set).

This function is expected to return `logits` , which provides a logit for each tag of each word of each sentence in the batch (structured as a tensor of size `max_length x batch_size x N`).

> You might find the following functions useful:
>
> - `nn.Embedding`
> - `nn.Linear`
> - `nn.RNN`

1. `compute_loss(self, logits, tags)` : Computes the loss for a batch by comparing `logits` of a batch returned by `forward` to `tags` , which stores the true tag ids for the

batch. Thus `logits` is a tensor of size `max_length x batch_size x N`, and `tags` is a tensor of size `max_length x batch_size`. Note that the criterion functions in `torch` expect outputs of a certain shape, so you might need to perform some shape conversions.

> You might find `nn.CrossEntropyLoss` from the last project segment useful. Note that if you use `nn.CrossEntropyLoss` then you should not use a softmax layer at the end since that's already absorbed into the loss function. Alternatively, you can use `nn.LogSoftmax` as the final sublayer in the forward pass, but then you need to use `nn.NLLLoss`, which does not contain its own softmax. We recommend the former, since working in log space is usually more numerically stable.
>
> Be careful about the shapes/dimensions of tensors. You might find `torch.Tensor.view` useful for reshaping tensors.

1. `train_all(self, train_iter, val_iter, epochs=10, learning_rate=0.001)`: Trains the model on training data generated by the iterator `train_iter` and validation data `val_iter`.The `epochs` and `learning_rate` variables are the number of epochs (number of times to run through the training data) to run for and the learning rate for the optimizer, respectively. You can use the validation data to determine which model was the best one as the epocks go by. Notice that our code below assumes that during training the best model is stored so that `rnn_tagger.load_state_dict(rnn_tagger.best_model)` restores the parameters of the best model.

Goal 2(b) RNN decoding

Implement a method to predict the tag sequence associated with a sequence of words:

1. `predict(self, text_batch)`: Returns the batched predicted tag sequences associated with a batch of sentences.
2. `def evaluate(self, iterator)`: Returns the accuracy of the trained tagger on a dataset provided by `iterator`.

In [86]:

```
class RNNTagger(nn.Module):
    def __init__(self, text, tag, embedding_size, hidden_size):
        super().__init__()
        self.text = text
        self.tag = tag
        self.N = len(tag.vocab.itos)
        self.V = len(text.vocab.itos)
        self.embedding_size = embedding_size
        self.hidden_size = hidden_size

        self.word_embeddings = nn.Embedding(self.V, embedding_size)
        self.rnn = nn.RNN(input_size=embedding_size,
                          hidden_size=hidden_size,
                          bidirectional=True)
        self.hidden2output = nn.Linear(hidden_size*2, self.N)
```

```
    pad_id = self.tag.vocab.stoi[self.tag.pad_token]
    self.loss_function = nn.CrossEntropyLoss(reduction='sum', ignore_index=p

    self.init_parameters()

def init_parameters(self, init_low=-0.15, init_high=0.15):
    """Initialize parameters. We usually use larger initial values for small
    See http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf for a more
    in-depth discussion.
    """
    for p in self.parameters():
        p.data.uniform_(init_low, init_high)

def forward(self, text_batch):
    hidden = None
    logits = self.hidden2output(self.rnn(self.word_embeddings(text_batch),hi
    return logits

def compute_loss(self, logits, tags):
    return self.loss_function(logits.view(-1, self.N), tags.view(-1))

def train_all(self, train_iter, val_iter, epochs = 10, learning_rate = 0.001
    self.train()
    optim = torch.optim.Adam(self.parameters(), lr=learning_rate)
    best_validation_accuracy = -float('inf')
    best_model = None
    for epoch in range(epochs):
      total = 0
      running_loss = 0.0
      for batch in tqdm(train_iter):
        self.zero_grad()
        words = batch.text
        tags = batch.tag
        logits = self.forward(words)
        loss = self.compute_loss(logits, tags)
        (loss/words.size(1)).backward()
        optim.step()
        total += 1
        running_loss += loss.item()


      validation_accuracy = self.evaluate(val_iter)
      if validation_accuracy > best_validation_accuracy:
        best_validation_accuracy = validation_accuracy
        self.best_model = copy.deepcopy(self.state_dict())
      epoch_loss = running_loss / total
      print (f'Epoch: {epoch} Loss: {epoch_loss:.4f} '
             f'Validation accuracy: {validation_accuracy:.4f}')
def predict(self, text_batch):
    """Returns the most likely sequence of tags for a sequence of words in `

    Arguments:
      text_batch: a tensor containing word ids of size (seq_len, 1)
    Returns:
      tag_batch: a tensor containing tag ids of size (seq_len, 1)
    """
    #TODO: your code below
    logits = self.forward(text_batch)
    tag_batch = torch.argmax(logits, dim = 2)
    return tag_batch
```

```
    def evaluate(self, iterator):
        """Returns the model's performance on a given dataset `iterator`.

        Arguments:
          iterator
        Returns:
          overall accuracy, and precision, recall, and F1 for comma
        """
        correct = 0
        total = 0
        pad_id = TAG.vocab.stoi[TAG.pad_token]
        for batch in tqdm(iterator):
          words = batch.text
          tags = batch.tag
          tags_pred = self.predict(words)
          mask = tags.ne(pad_id)
          cor = (tags == tags_pred)[mask]
          correct += cor.float().sum().item()
          total += mask.float().sum().item()

        return correct/total
```

Now train your tagger on the training and validation set. Run the cell below to train an RNN, and evaluate it. A proper implementation should reach about **95%+ accuracy**.

In [87]:

```
# Instantiate and train classifier
rnn_tagger = RNNTagger(TEXT, TAG, embedding_size=36, hidden_size=36).to(device)
rnn_tagger.train_all(train_iter, val_iter, epochs=10, learning_rate=0.001)
rnn_tagger.load_state_dict(rnn_tagger.best_model)

# Evaluate model performance
print(f'Training accuracy: {rnn_tagger.evaluate(train_iter):.3f}\n'
      f'Test accuracy:     {rnn_tagger.evaluate(test_iter):.3f}')
```

```
Epoch: 0 Loss: 447.2533 Validation accuracy: 0.8167

Epoch: 1 Loss: 163.7799 Validation accuracy: 0.9200

Epoch: 2 Loss: 85.5635 Validation accuracy: 0.9441

Epoch: 3 Loss: 57.4101 Validation accuracy: 0.9543

Epoch: 4 Loss: 42.9645 Validation accuracy: 0.9621

Epoch: 5 Loss: 33.8012 Validation accuracy: 0.9685

Epoch: 6 Loss: 27.4417 Validation accuracy: 0.9733

Epoch: 7 Loss: 22.5431 Validation accuracy: 0.9779

Epoch: 8 Loss: 18.9634 Validation accuracy: 0.9794

Epoch: 9 Loss: 16.1390 Validation accuracy: 0.9806
```

```
Training accuracy: 0.990
Test accuracy:     0.979
```

LSTM for slot filling

Did your RNN perform better than HMM? How much better was it? Was that expected?

RNNs tend to exhibit the vanishing gradient problem. To remedy this, the Long-Short Term Memory (LSTM) model was introduced. In PyTorch, we can simply use `nn.LSTM` .

In this section, you'll implement an LSTM model for slot filling. If you've got the RNN model well implemented, this should be extremely straightforward. Just copy and paste your solution, change the call to `nn.RNN` to a call to `nn.LSTM` , and make any other minor adjustments that are necessary. In particular, LSTMs have *two* recurrent parts, `h` and `c` . You'll thus need to initialize both of these when performing forward computations.

In [55]:

```
class LSTMTagger(nn.Module):
    def __init__(self, text, tag, embedding_size, hidden_size):
        super().__init__()
        self.text = text
        self.tag = tag
        self.N = len(tag.vocab.itos)
        self.V = len(text.vocab.itos)
        self.embedding_size = embedding_size
        self.hidden_size = hidden_size

        self.word_embeddings = nn.Embedding(self.V, embedding_size)
        self.lstm = nn.LSTM(input_size=embedding_size,
                            hidden_size=hidden_size,
                            bidirectional=True)
        self.hidden2output = nn.Linear(hidden_size*2, self.N)

        pad_id = self.tag.vocab.stoi[self.tag.pad_token]
        self.loss_function = nn.CrossEntropyLoss(reduction='sum', ignore_index=p

        self.init_parameters()
    def init_parameters(self, init_low=-0.15, init_high=0.15):
        """Initialize parameters. We usually use larger initial values for small
        See http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf for a more
        in-depth discussion.
        """
        for p in self.parameters():
            p.data.uniform_(init_low, init_high)

    def forward(self, text_batch):
        hidden = None
        logits = self.hidden2output(self.lstm(self.word_embeddings(text_batch),h
        return logits

    def compute_loss(self, logits, tags):
        return self.loss_function(logits.view(-1, self.N), tags.view(-1))

    def train_all(self, train_iter, val_iter, epochs = 10, learning_rate = 0.001
        self.train()
        optim = torch.optim.Adam(self.parameters(), lr=learning_rate)
```

```
    best_validation_accuracy = -float('inf')
    best_model = None
    for epoch in range(epochs):
      total = 0
      running_loss = 0.0
      for batch in tqdm(train_iter):
        self.zero_grad()
        words = batch.text
        tags = batch.tag
        logits = self.forward(words)
        loss = self.compute_loss(logits, tags)
        (loss/words.size(1)).backward()
        optim.step()
        total += 1
        running_loss += loss.item()

      validation_accuracy = self.evaluate(val_iter)
      if validation_accuracy > best_validation_accuracy:
        best_validation_accuracy = validation_accuracy
        self.best_model = copy.deepcopy(self.state_dict())
      epoch_loss = running_loss / total
      print (f'Epoch: {epoch} Loss: {epoch_loss:.4f} '
             f'Validation accuracy: {validation_accuracy:.4f}')
  def predict(self, text_batch):
    """Returns the most likely sequence of tags for a sequence of words in `

    Arguments:
      text_batch: a tensor containing word ids of size (seq_len, 1)
    Returns:
      tag_batch: a tensor containing tag ids of size (seq_len, 1)
    """
    #TODO: your code below
    logits = self.forward(text_batch)
    tag_batch = torch.argmax(logits, dim = 2)
    return tag_batch

  def evaluate(self, iterator):
    """Returns the model's performance on a given dataset `iterator`.

    Arguments:
      iterator
    Returns:
      overall accuracy, and precision, recall, and F1 for comma
    """
    correct = 0
    total = 0
    pad_id = TAG.vocab.stoi[TAG.pad_token]
    for batch in tqdm(iterator):
      words = batch.text
      tags = batch.tag
      tags_pred = self.predict(words)
      mask = tags.ne(pad_id)
      cor = (tags == tags_pred)[mask]
      correct += cor.float().sum().item()
      total += mask.float().sum().item()

    return correct/total
```

Run the cell below to train an LSTM, and evaluate it. A proper implementation should reach about **95%+ accuracy**.

In [56]:
```
# Instantiate and train classifier
lstm_tagger = LSTMTagger(TEXT, TAG, embedding_size=36, hidden_size=36).to(device
lstm_tagger.train_all(train_iter, val_iter, epochs=10, learning_rate=0.001)
lstm_tagger.load_state_dict(lstm_tagger.best_model)

# Evaluate model performance
print(f'Training accuracy: {lstm_tagger.evaluate(train_iter):.3f}\n'
      f'Test accuracy:     {lstm_tagger.evaluate(test_iter):.3f}')
```

```
Epoch: 0 Loss: 557.4358 Validation accuracy: 0.7796

Epoch: 1 Loss: 236.0250 Validation accuracy: 0.8488

Epoch: 2 Loss: 156.1624 Validation accuracy: 0.9021

Epoch: 3 Loss: 107.9332 Validation accuracy: 0.9296

Epoch: 4 Loss: 78.9057 Validation accuracy: 0.9388

Epoch: 5 Loss: 60.8047 Validation accuracy: 0.9496

Epoch: 6 Loss: 48.3766 Validation accuracy: 0.9569

Epoch: 7 Loss: 39.5688 Validation accuracy: 0.9638

Epoch: 8 Loss: 33.0261 Validation accuracy: 0.9686

Epoch: 9 Loss: 28.2274 Validation accuracy: 0.9716

Training accuracy: 0.982
Test accuracy:     0.972
```

(Optional) Goal 4: Compare HMM to RNN/LSTM with different amounts of training data

Vary the amount of training data and compare the performance of HMM to RNN or LSTM (Since RNN is similar to LSTM, picking one of them is enough.) Discuss the pros and cons of HMM and RNN/LSTM based on your experiments.

> This part is more open-ended. We're looking for thoughtful experiments and analysis of the results, not any particular result or conclusion.

The code below shows how to subsample the training set with downsample ratio `ratio` . To speedup evaluation we only use 50 test samples.

In [ ]:
```
ratio = 0.1
test_size = 50
```

```
# Set random seeds to make sure subsampling is the same for HMM and RNN
random.seed(seed)
torch.manual_seed(seed)

train, val, test = tt.datasets.SequenceTaggingDataset.splits(
            fields=fields,
            path='./data/',
            train='atis.train.txt',
            validation='atis.dev.txt',
            test='atis.test.txt')

# Subsample
random.shuffle(train.examples)
train.examples = train.examples[:int(math.floor(len(train.examples)*ratio))]
random.shuffle(test.examples)
test.examples = test.examples[:test_size]

# Rebuild vocabulary
TEXT.build_vocab(train.text, min_freq=MIN_FREQ)
TAG.build_vocab(train.tag)
```

In [ ]:

```
...
```

In [ ]:

```
...
```

In [ ]:

```
...
```

Type your answer here, replacing this text.

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 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?

I think the project segments were very clear this time around. I didn't have too many questions on the implementations to use. The readings and lab segments were very helpful for this project segment. I was able to get a lot out of both parts and apply it to this project segment. I would say this project segment was overall well taught and accessible and wouldn't recommend any additions or changes to make the segment better.

Instructions for submission of the project

segment

This project segment should be submitted to Gradescope at http://go.cs187.info/project2-submit-code and http://go.cs187.info/project2-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/project2-submit-code.

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/project2-submit-pdf.