CS236299 Project Segment 2: Sequence labeling – The slot filling task

May 31, 2022

[1]:

```
# 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/cs236299-2022-spring/project2.git .tmp
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

[2]:

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

1 236299 - Introduction to Natural Language Processing

1.1 Project 2: Sequence labeling – The slot filling task

2 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.

2.1 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. 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.

2.2 Setup

[3]:
```
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
```

[4]:
```
# 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
```

2.3 Loading data

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

[5]:
```
# 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,␣
 ↪out=dest)

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)
```

2.4 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>`.

[6]:
```
MIN_FREQ = 3

TEXT = tt.data.Field(init_token="<bos>", batch_first=False) # batches are of␣
 ↪size max_len x bsz
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.

[7]:
```
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_name', 3726), ('I-toloc.city_name', 1039),
('B-depart_date.day_name', 835), ('I-fromloc.city_name', 636),
('B-airline_name', 610), ('B-depart_time.period_of_day', 555),
('I-airline_name', 374), ('B-depart_date.day_number', 351)]
```

2.5 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:

[8]:
```python
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:

[9]:
```python
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.

[10]:
```python
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.

[11]:
```python
# 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
atlanta 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>',
'<pad>', '<pad>', '<pad>']
```

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

3 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:

[12]:
```
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:

[13]:
```
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
```

4 HMM for sequence labeling

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

4.1 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).

4.2 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,

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

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

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

As is typical in notating probabilities, we'll use abbreviations

$$\Pr(q_{t+1} \mid q_t) \equiv \Pr(q_{t+1} = Q_j \mid q_t = Q_i) \quad (1)$$
$$\Pr(w_t \mid q_t) \equiv \Pr(w_t = \mathcal{V}_j \mid q_t = Q_i) \quad (2)$$

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.

4.2.1 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.

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

(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.

4.2.2 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.

4.3 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 $\mathbf{w} \in \mathcal{V}^T$.

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

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]`.

4.3.1 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.

4.4 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.

[14]:
```
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.
    batchs = iter(iterator)
    while True:
        try:
            batch = next(batchs)
            for i in range(len(batch.tag[0])):
                sentence_tags = batch.tag[:,i]
                for j in range(len(sentence_tags)-1):
                    q_prev = sentence_tags[j].item()
                    q_t = sentence_tags[j+1].item()
                    A[q_prev][q_t] += 1
        except StopIteration:
            break

    for i in range(self.N):
        A[i] = (A[i] + delta) / (A[i].sum().item() + delta * self.V)

    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.

    batchs = iter(iterator)
    while True:
        try:
            batch = next(batchs)
            for i in range(len(batch.tag[0])):
                sentence_tags = batch.tag[:,i]
                sentence_text = batch.text[:,i]
                for j in range(len(sentence_tags)):
                    q_t = sentence_tags[j].item()
                    w_t = sentence_text[j].item()
                    B[q_t][w_t] += 1
        except StopIteration:
            break

    for i in range(self.N):
        B[i] = (B[i]+ delta) / (B[i].sum().item() + delta * self.V)

    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 = words.size(0)
    path_probs = torch.zeros(self.N, seq_len, device=device)
    path_pointers = torch.zeros(self.N, seq_len, device=device)

    path_probs[self.initial_state_id][0] = 1
    path_probs[:,0] = path_probs[:,0].log()

    for t in range(1, seq_len):
        prev_probs = path_probs[:,t-1]
        w_t = words[t].item()
        b_t = self.log_B[:,w_t]
```

```
        for tag in range(self.N):
          a_tag = self.log_A[:,tag]
          probs_for_tag = prev_probs + a_tag + b_t[tag].item()
          path_probs[tag][t] = probs_for_tag.max().item()
          path_pointers[tag][t] = probs_for_tag.argmax().item()
    best_last_pointer = path_probs[:,seq_len-1].argmax().item()
    bestpath = [best_last_pointer]
    for t in reversed(range(1,seq_len)):
      best_last_pointer = int(path_pointers[best_last_pointer][t].item())
      bestpath.insert(0, best_last_pointer)
    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
    if total == 0:
      return 0
    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.

[15]:
```
# 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}')
```

```
0%|          | 0/214 [00:00<?, ?it/s]

0%|          | 0/30 [00:00<?, ?it/s]
```

```
Training accuracy: 0.916
Test accuracy:     0.907
```

5 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.

$$
\begin{aligned}
\mathbf{h}_t &= \sigma(\mathbf{U}\mathbf{x}_t + \mathbf{V}\mathbf{h}_{t-1}) \\
\hat{\mathbf{y}}_t &= \mathrm{softmax}(\mathbf{W}\mathbf{h}_t)
\end{aligned}
$$

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.

5.1 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

2. `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.

3. `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.

5.2 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`.

[16]:
```python
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.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]
        self.embedding_size = embedding_size
        self.hidden_size = hidden_size

        # Create essential modules
```

```
        self.word_embeddings = nn.Embedding(self.V, embedding_size) # Lookup layer
        self.rnn = nn.RNN(input_size=embedding_size, hidden_size=hidden_size)
        self.hidden2output = nn.Linear(hidden_size, self.N)

        # Create loss function
        pad_id = self.tag.vocab.stoi[self.tag.pad_token]
        self.loss_function = nn.CrossEntropyLoss(reduction='sum', ignore_index=pad_id)

        # initialize parameters randomly
        torch.manual_seed(1234)
        for p in self.parameters():
            p.data.uniform_(-0.2, 0.2)

    def forward(self, text_batch):
        logits = self.word_embeddings(text_batch)
        logits = self.rnn(logits)[0]
        logits = self.hidden2output(logits)
        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):
        logits = self.forward(text_batch).view(-1, self.N)
        predicts = logits.argmax(1)
        return predicts

    def evaluate(self, iterator):
        correct = 0
        total = 0

        pad_id = TAG.vocab.stoi[TAG.pad_token]
        for batch in tqdm(iterator):
            words = batch.text
            tags = batch.tag.view(-1)
            tags_pred = self.predict(words)

            for i in range(tags.size(0)):
                if tags[i]!=self.pad_state_id:
                    total+=1
                    if tags[i]==tags_pred[i]:
                        correct+=1
        if total == 0:
            return 0
        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**.

[17]:
```
# 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}')
```

```
  0%|          | 0/214 [00:00<?, ?it/s]

  0%|          | 0/29 [00:00<?, ?it/s]

Epoch: 0 Loss: 563.7037 Validation accuracy: 0.7541
```

```
  0%|          | 0/214 [00:00<?, ?it/s]

  0%|          | 0/29 [00:00<?, ?it/s]

Epoch: 1 Loss: 231.5850 Validation accuracy: 0.8721

  0%|          | 0/214 [00:00<?, ?it/s]

  0%|          | 0/29 [00:00<?, ?it/s]

Epoch: 2 Loss: 132.9385 Validation accuracy: 0.9283

  0%|          | 0/214 [00:00<?, ?it/s]

  0%|          | 0/29 [00:00<?, ?it/s]

Epoch: 3 Loss: 88.2456 Validation accuracy: 0.9430

  0%|          | 0/214 [00:00<?, ?it/s]

  0%|          | 0/29 [00:00<?, ?it/s]

Epoch: 4 Loss: 66.9518 Validation accuracy: 0.9498

  0%|          | 0/214 [00:00<?, ?it/s]

  0%|          | 0/29 [00:00<?, ?it/s]

Epoch: 5 Loss: 54.1702 Validation accuracy: 0.9566

  0%|          | 0/214 [00:00<?, ?it/s]

  0%|          | 0/29 [00:00<?, ?it/s]

Epoch: 6 Loss: 45.3603 Validation accuracy: 0.9623

  0%|          | 0/214 [00:00<?, ?it/s]

  0%|          | 0/29 [00:00<?, ?it/s]

Epoch: 7 Loss: 38.7343 Validation accuracy: 0.9653

  0%|          | 0/214 [00:00<?, ?it/s]

  0%|          | 0/29 [00:00<?, ?it/s]

Epoch: 8 Loss: 33.5869 Validation accuracy: 0.9698

  0%|          | 0/214 [00:00<?, ?it/s]

  0%|          | 0/29 [00:00<?, ?it/s]

Epoch: 9 Loss: 29.5124 Validation accuracy: 0.9728

  0%|          | 0/214 [00:00<?, ?it/s]

  0%|          | 0/30 [00:00<?, ?it/s]

Training accuracy: 0.981
Test accuracy:     0.970
```

6 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.

[18]:
```
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.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]
        self.embedding_size = embedding_size
        self.hidden_size = hidden_size

        # Create essential modules
        self.word_embeddings = nn.Embedding(self.V, embedding_size) # Lookup␣
↪layer
        self.lstm = nn.LSTM(input_size=embedding_size, hidden_size=hidden_size)
        self.hidden2output = nn.Linear(hidden_size, self.N)

        # Create loss function
        pad_id = self.tag.vocab.stoi[self.tag.pad_token]
        self.loss_function = nn.CrossEntropyLoss(reduction='sum',␣
↪ignore_index=pad_id)

        # initialize parameters randomly
        torch.manual_seed(1234)
        for p in self.parameters():
            p.data.uniform_(-0.2, 0.2)

    def forward(self, text_batch):
        logits = self.word_embeddings(text_batch)
        logits = self.lstm(logits)[0]
        logits = self.hidden2output(logits)
        return logits
```

```python
    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):
        logits = self.forward(text_batch).view(-1, self.N)
        predicts = logits.argmax(1)
        return predicts

    def evaluate(self, iterator):
        correct = 0
        total = 0

        pad_id = TAG.vocab.stoi[TAG.pad_token]
        for batch in tqdm(iterator):
            words = batch.text
            tags = batch.tag.view(-1)
            tags_pred = self.predict(words)
```

```
            for i in range(tags.size(0)):
                if tags[i]!=self.pad_state_id:
                    total+=1
                    if tags[i]==tags_pred[i]:
                        correct+=1

        return correct/total
```

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

[19]:
```
# 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}')
```

```
  0%|          | 0/214 [00:00<?, ?it/s]

  0%|          | 0/29 [00:00<?, ?it/s]

Epoch: 0 Loss: 623.2899 Validation accuracy: 0.7080

  0%|          | 0/214 [00:00<?, ?it/s]

  0%|          | 0/29 [00:00<?, ?it/s]

Epoch: 1 Loss: 266.0129 Validation accuracy: 0.8402

  0%|          | 0/214 [00:00<?, ?it/s]

  0%|          | 0/29 [00:00<?, ?it/s]

Epoch: 2 Loss: 187.7253 Validation accuracy: 0.8762

  0%|          | 0/214 [00:00<?, ?it/s]

  0%|          | 0/29 [00:00<?, ?it/s]

Epoch: 3 Loss: 141.7275 Validation accuracy: 0.9081

  0%|          | 0/214 [00:00<?, ?it/s]

  0%|          | 0/29 [00:00<?, ?it/s]

Epoch: 4 Loss: 109.1600 Validation accuracy: 0.9301

  0%|          | 0/214 [00:00<?, ?it/s]

  0%|          | 0/29 [00:00<?, ?it/s]
```

```
Epoch: 5 Loss: 87.4566 Validation accuracy: 0.9388

  0%|          | 0/214 [00:00<?, ?it/s]

  0%|          | 0/29 [00:00<?, ?it/s]

Epoch: 6 Loss: 72.7418 Validation accuracy: 0.9421

  0%|          | 0/214 [00:00<?, ?it/s]

  0%|          | 0/29 [00:00<?, ?it/s]

Epoch: 7 Loss: 62.1800 Validation accuracy: 0.9495

  0%|          | 0/214 [00:00<?, ?it/s]

  0%|          | 0/29 [00:00<?, ?it/s]

Epoch: 8 Loss: 54.1019 Validation accuracy: 0.9527

  0%|          | 0/214 [00:00<?, ?it/s]

  0%|          | 0/29 [00:00<?, ?it/s]

Epoch: 9 Loss: 47.7549 Validation accuracy: 0.9572

  0%|          | 0/214 [00:00<?, ?it/s]

  0%|          | 0/30 [00:00<?, ?it/s]

Training accuracy: 0.965
Test accuracy:     0.958
```

7 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.

[20]:
```
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,
```

```python
            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)
```

[21]:
```python
from functools import partialmethod

tqdm.__init__ = partialmethod(tqdm.__init__, disable=True)



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

ratios = [0.1, 0.3, 0.5, 0.7]
test_sizes = [40, 80, 200, 500]

res_hmm = list(list())
res_rnn = list(list())

for a_idx, ratio in enumerate(ratios):

    res_hmm.append([])
    res_rnn.append([])

    for b_idx, test_size in enumerate(test_sizes):
        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]

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

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

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

        hmm_train = hmm_tagger.evaluate(train_iter)
        hmm_test = hmm_tagger.evaluate(test_iter)
        res_hmm[a_idx].append((hmm_train, hmm_test))
        # Evaluate model performance
        print(f'HMM with {BATCH_SIZE=}; {ratio=}; {test_size=}')
        print(f'Training accuracy: {res_hmm[a_idx][b_idx][0]:.3f}\n'
              f'Test accuracy:     {res_hmm[a_idx][b_idx][1]:.3f}')


        # RNN
        # 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)

        rnn_train = rnn_tagger.evaluate(train_iter)
        rnn_test = rnn_tagger.evaluate(test_iter)
        res_rnn[a_idx].append((rnn_train, rnn_test))
        # Evaluate model performance
        print(f'RNN with {BATCH_SIZE=}; {ratio=}; {test_size=}')
        print(f'Training accuracy: {res_rnn[a_idx][b_idx][0]:.3f}\n'
              f'Test accuracy:     {res_rnn[a_idx][b_idx][1]:.3f}')
print('FINISHED RUNNING')
```

```
HMM with BATCH_SIZE=20; ratio=0.1; test_size=40
```

```
Training accuracy: 0.912
Test accuracy:     0.857
Epoch: 0 Loss: 1128.8748 Validation accuracy: 0.6392
Epoch: 1 Loss: 741.1990 Validation accuracy: 0.6392
Epoch: 2 Loss: 534.3540 Validation accuracy: 0.6392
Epoch: 3 Loss: 486.2779 Validation accuracy: 0.7080
Epoch: 4 Loss: 456.7896 Validation accuracy: 0.7080
Epoch: 5 Loss: 423.0286 Validation accuracy: 0.7080
Epoch: 6 Loss: 379.7874 Validation accuracy: 0.7347
Epoch: 7 Loss: 333.3264 Validation accuracy: 0.7585
Epoch: 8 Loss: 298.8895 Validation accuracy: 0.7891
Epoch: 9 Loss: 273.7859 Validation accuracy: 0.8151
RNN with BATCH_SIZE=20; ratio=0.1; test_size=40
Training accuracy: 0.811
Test accuracy:     0.830
HMM with BATCH_SIZE=20; ratio=0.1; test_size=80
Training accuracy: 0.920
Test accuracy:     0.883
Epoch: 0 Loss: 1182.3215 Validation accuracy: 0.5242
Epoch: 1 Loss: 813.2898 Validation accuracy: 0.6392
Epoch: 2 Loss: 559.7973 Validation accuracy: 0.6392
Epoch: 3 Loss: 493.2642 Validation accuracy: 0.7080
Epoch: 4 Loss: 463.8060 Validation accuracy: 0.7080
Epoch: 5 Loss: 429.6064 Validation accuracy: 0.7080
Epoch: 6 Loss: 389.1020 Validation accuracy: 0.7080
Epoch: 7 Loss: 344.6482 Validation accuracy: 0.7586
Epoch: 8 Loss: 306.0016 Validation accuracy: 0.8011
Epoch: 9 Loss: 274.9712 Validation accuracy: 0.8146
RNN with BATCH_SIZE=20; ratio=0.1; test_size=80
Training accuracy: 0.815
Test accuracy:     0.803
HMM with BATCH_SIZE=20; ratio=0.1; test_size=200
Training accuracy: 0.905
Test accuracy:     0.882
Epoch: 0 Loss: 1119.9609 Validation accuracy: 0.5729
Epoch: 1 Loss: 777.0579 Validation accuracy: 0.6392
Epoch: 2 Loss: 517.3261 Validation accuracy: 0.7080
Epoch: 3 Loss: 459.5614 Validation accuracy: 0.7080
Epoch: 4 Loss: 434.4949 Validation accuracy: 0.7080
Epoch: 5 Loss: 411.0320 Validation accuracy: 0.7080
Epoch: 6 Loss: 385.4109 Validation accuracy: 0.7080
Epoch: 7 Loss: 351.6840 Validation accuracy: 0.7080
Epoch: 8 Loss: 311.5224 Validation accuracy: 0.7592
Epoch: 9 Loss: 277.0468 Validation accuracy: 0.8060
RNN with BATCH_SIZE=20; ratio=0.1; test_size=200
Training accuracy: 0.810
Test accuracy:     0.804
HMM with BATCH_SIZE=20; ratio=0.1; test_size=500
```

```
Training accuracy: 0.914
Test accuracy:     0.882
Epoch: 0 Loss: 1168.2067 Validation accuracy: 0.6392
Epoch: 1 Loss: 829.1846 Validation accuracy: 0.6392
Epoch: 2 Loss: 559.4784 Validation accuracy: 0.6392
Epoch: 3 Loss: 492.2838 Validation accuracy: 0.7080
Epoch: 4 Loss: 464.0685 Validation accuracy: 0.7080
Epoch: 5 Loss: 439.9941 Validation accuracy: 0.7080
Epoch: 6 Loss: 419.0415 Validation accuracy: 0.7080
Epoch: 7 Loss: 395.1312 Validation accuracy: 0.7080
Epoch: 8 Loss: 364.4520 Validation accuracy: 0.7105
Epoch: 9 Loss: 327.4932 Validation accuracy: 0.7613
RNN with BATCH_SIZE=20; ratio=0.1; test_size=500
Training accuracy: 0.761
Test accuracy:     0.755
HMM with BATCH_SIZE=20; ratio=0.3; test_size=40
Training accuracy: 0.913
Test accuracy:     0.916
Epoch: 0 Loss: 888.1861 Validation accuracy: 0.7080
Epoch: 1 Loss: 461.6044 Validation accuracy: 0.7080
Epoch: 2 Loss: 364.7464 Validation accuracy: 0.7632
Epoch: 3 Loss: 274.2156 Validation accuracy: 0.8223
Epoch: 4 Loss: 218.1587 Validation accuracy: 0.8492
Epoch: 5 Loss: 183.0774 Validation accuracy: 0.8780
Epoch: 6 Loss: 156.6430 Validation accuracy: 0.8969
Epoch: 7 Loss: 135.6992 Validation accuracy: 0.9082
Epoch: 8 Loss: 118.6820 Validation accuracy: 0.9253
Epoch: 9 Loss: 104.5193 Validation accuracy: 0.9326
RNN with BATCH_SIZE=20; ratio=0.3; test_size=40
Training accuracy: 0.937
Test accuracy:     0.942
HMM with BATCH_SIZE=20; ratio=0.3; test_size=80
Training accuracy: 0.913
Test accuracy:     0.910
Epoch: 0 Loss: 835.2517 Validation accuracy: 0.6392
Epoch: 1 Loss: 471.8930 Validation accuracy: 0.7080
Epoch: 2 Loss: 374.3144 Validation accuracy: 0.7485
Epoch: 3 Loss: 279.9462 Validation accuracy: 0.8103
Epoch: 4 Loss: 235.2413 Validation accuracy: 0.8405
Epoch: 5 Loss: 201.4327 Validation accuracy: 0.8615
Epoch: 6 Loss: 172.9968 Validation accuracy: 0.8774
Epoch: 7 Loss: 149.8857 Validation accuracy: 0.8905
Epoch: 8 Loss: 130.1492 Validation accuracy: 0.9189
Epoch: 9 Loss: 114.0299 Validation accuracy: 0.9293
RNN with BATCH_SIZE=20; ratio=0.3; test_size=80
Training accuracy: 0.931
Test accuracy:     0.929
HMM with BATCH_SIZE=20; ratio=0.3; test_size=200
```

```
Training accuracy: 0.915
Test accuracy:     0.904
Epoch: 0 Loss: 818.8136 Validation accuracy: 0.6392
Epoch: 1 Loss: 452.3134 Validation accuracy: 0.7080
Epoch: 2 Loss: 336.9323 Validation accuracy: 0.8030
Epoch: 3 Loss: 257.8126 Validation accuracy: 0.8313
Epoch: 4 Loss: 215.6841 Validation accuracy: 0.8402
Epoch: 5 Loss: 180.7543 Validation accuracy: 0.8744
Epoch: 6 Loss: 151.6772 Validation accuracy: 0.9031
Epoch: 7 Loss: 128.9294 Validation accuracy: 0.9157
Epoch: 8 Loss: 111.5458 Validation accuracy: 0.9196
Epoch: 9 Loss: 98.1087 Validation accuracy: 0.9267
RNN with BATCH_SIZE=20; ratio=0.3; test_size=200
Training accuracy: 0.934
Test accuracy:     0.919
HMM with BATCH_SIZE=20; ratio=0.3; test_size=500
Training accuracy: 0.916
Test accuracy:     0.889
Epoch: 0 Loss: 862.3377 Validation accuracy: 0.6392
Epoch: 1 Loss: 456.5286 Validation accuracy: 0.7080
Epoch: 2 Loss: 356.6419 Validation accuracy: 0.7637
Epoch: 3 Loss: 265.0396 Validation accuracy: 0.8323
Epoch: 4 Loss: 209.5375 Validation accuracy: 0.8500
Epoch: 5 Loss: 176.7355 Validation accuracy: 0.8702
Epoch: 6 Loss: 153.2418 Validation accuracy: 0.8894
Epoch: 7 Loss: 133.7454 Validation accuracy: 0.9142
Epoch: 8 Loss: 117.1827 Validation accuracy: 0.9196
Epoch: 9 Loss: 103.4889 Validation accuracy: 0.9259
RNN with BATCH_SIZE=20; ratio=0.3; test_size=500
Training accuracy: 0.937
Test accuracy:     0.923
HMM with BATCH_SIZE=20; ratio=0.5; test_size=40
Training accuracy: 0.917
Test accuracy:     0.903
Epoch: 0 Loss: 751.4835 Validation accuracy: 0.7080
Epoch: 1 Loss: 386.1452 Validation accuracy: 0.8038
Epoch: 2 Loss: 252.8003 Validation accuracy: 0.8346
Epoch: 3 Loss: 190.2982 Validation accuracy: 0.8857
Epoch: 4 Loss: 143.6523 Validation accuracy: 0.9164
Epoch: 5 Loss: 113.1662 Validation accuracy: 0.9261
Epoch: 6 Loss: 93.3433 Validation accuracy: 0.9337
Epoch: 7 Loss: 79.7567 Validation accuracy: 0.9398
Epoch: 8 Loss: 69.9193 Validation accuracy: 0.9441
Epoch: 9 Loss: 62.1447 Validation accuracy: 0.9489
RNN with BATCH_SIZE=20; ratio=0.5; test_size=40
Training accuracy: 0.955
Test accuracy:     0.931
HMM with BATCH_SIZE=20; ratio=0.5; test_size=80
```

```
Training accuracy: 0.914
Test accuracy:     0.893
Epoch: 0 Loss: 725.9815 Validation accuracy: 0.7080
Epoch: 1 Loss: 400.9359 Validation accuracy: 0.7132
Epoch: 2 Loss: 262.5803 Validation accuracy: 0.8364
Epoch: 3 Loss: 192.9405 Validation accuracy: 0.8746
Epoch: 4 Loss: 148.2961 Validation accuracy: 0.9123
Epoch: 5 Loss: 117.4247 Validation accuracy: 0.9308
Epoch: 6 Loss: 94.6646 Validation accuracy: 0.9342
Epoch: 7 Loss: 79.6025 Validation accuracy: 0.9390
Epoch: 8 Loss: 69.2457 Validation accuracy: 0.9436
Epoch: 9 Loss: 61.5254 Validation accuracy: 0.9486
RNN with BATCH_SIZE=20; ratio=0.5; test_size=80
Training accuracy: 0.955
Test accuracy:     0.932
HMM with BATCH_SIZE=20; ratio=0.5; test_size=200
Training accuracy: 0.915
Test accuracy:     0.898
Epoch: 0 Loss: 764.5997 Validation accuracy: 0.7080
Epoch: 1 Loss: 371.2308 Validation accuracy: 0.7656
Epoch: 2 Loss: 244.8444 Validation accuracy: 0.8313
Epoch: 3 Loss: 188.6355 Validation accuracy: 0.8855
Epoch: 4 Loss: 142.7622 Validation accuracy: 0.9141
Epoch: 5 Loss: 113.6288 Validation accuracy: 0.9271
Epoch: 6 Loss: 93.8153 Validation accuracy: 0.9357
Epoch: 7 Loss: 79.5500 Validation accuracy: 0.9385
Epoch: 8 Loss: 68.9869 Validation accuracy: 0.9427
Epoch: 9 Loss: 61.1614 Validation accuracy: 0.9461
RNN with BATCH_SIZE=20; ratio=0.5; test_size=200
Training accuracy: 0.955
Test accuracy:     0.946
HMM with BATCH_SIZE=20; ratio=0.5; test_size=500
Training accuracy: 0.914
Test accuracy:     0.897
Epoch: 0 Loss: 717.9303 Validation accuracy: 0.7080
Epoch: 1 Loss: 368.1751 Validation accuracy: 0.7657
Epoch: 2 Loss: 248.2462 Validation accuracy: 0.8374
Epoch: 3 Loss: 189.7852 Validation accuracy: 0.8726
Epoch: 4 Loss: 143.2127 Validation accuracy: 0.9159
Epoch: 5 Loss: 112.5131 Validation accuracy: 0.9297
Epoch: 6 Loss: 92.6496 Validation accuracy: 0.9330
Epoch: 7 Loss: 79.0413 Validation accuracy: 0.9395
Epoch: 8 Loss: 69.1193 Validation accuracy: 0.9433
Epoch: 9 Loss: 61.4485 Validation accuracy: 0.9497
RNN with BATCH_SIZE=20; ratio=0.5; test_size=500
Training accuracy: 0.956
Test accuracy:     0.947
HMM with BATCH_SIZE=20; ratio=0.7; test_size=40
```

```
Training accuracy: 0.913
Test accuracy:     0.906
Epoch: 0 Loss: 609.2667 Validation accuracy: 0.7080
Epoch: 1 Loss: 296.9291 Validation accuracy: 0.8261
Epoch: 2 Loss: 193.5704 Validation accuracy: 0.8844
Epoch: 3 Loss: 134.9334 Validation accuracy: 0.9207
Epoch: 4 Loss: 100.3512 Validation accuracy: 0.9336
Epoch: 5 Loss: 79.2951 Validation accuracy: 0.9396
Epoch: 6 Loss: 66.1079 Validation accuracy: 0.9454
Epoch: 7 Loss: 56.7160 Validation accuracy: 0.9501
Epoch: 8 Loss: 49.3875 Validation accuracy: 0.9551
Epoch: 9 Loss: 43.6684 Validation accuracy: 0.9585
RNN with BATCH_SIZE=20; ratio=0.7; test_size=40
Training accuracy: 0.966
Test accuracy:     0.958
HMM with BATCH_SIZE=20; ratio=0.7; test_size=80
Training accuracy: 0.914
Test accuracy:     0.911
Epoch: 0 Loss: 625.4286 Validation accuracy: 0.7080
Epoch: 1 Loss: 278.6634 Validation accuracy: 0.8371
Epoch: 2 Loss: 176.4800 Validation accuracy: 0.9031
Epoch: 3 Loss: 121.0343 Validation accuracy: 0.9270
Epoch: 4 Loss: 91.7554 Validation accuracy: 0.9355
Epoch: 5 Loss: 74.3619 Validation accuracy: 0.9406
Epoch: 6 Loss: 62.6684 Validation accuracy: 0.9490
Epoch: 7 Loss: 53.7471 Validation accuracy: 0.9543
Epoch: 8 Loss: 46.4293 Validation accuracy: 0.9582
Epoch: 9 Loss: 40.6285 Validation accuracy: 0.9615
RNN with BATCH_SIZE=20; ratio=0.7; test_size=80
Training accuracy: 0.971
Test accuracy:     0.976
HMM with BATCH_SIZE=20; ratio=0.7; test_size=200
Training accuracy: 0.914
Test accuracy:     0.900
Epoch: 0 Loss: 628.0239 Validation accuracy: 0.7080
Epoch: 1 Loss: 292.8200 Validation accuracy: 0.8246
Epoch: 2 Loss: 192.3118 Validation accuracy: 0.8870
Epoch: 3 Loss: 132.8370 Validation accuracy: 0.9282
Epoch: 4 Loss: 98.6912 Validation accuracy: 0.9338
Epoch: 5 Loss: 79.1471 Validation accuracy: 0.9421
Epoch: 6 Loss: 66.5269 Validation accuracy: 0.9473
Epoch: 7 Loss: 57.2625 Validation accuracy: 0.9509
Epoch: 8 Loss: 50.1919 Validation accuracy: 0.9575
Epoch: 9 Loss: 44.4749 Validation accuracy: 0.9584
RNN with BATCH_SIZE=20; ratio=0.7; test_size=200
Training accuracy: 0.968
Test accuracy:     0.958
HMM with BATCH_SIZE=20; ratio=0.7; test_size=500
```

```
Training accuracy: 0.915
Test accuracy:     0.904
Epoch: 0 Loss: 618.3158 Validation accuracy: 0.7080
Epoch: 1 Loss: 322.5076 Validation accuracy: 0.8331
Epoch: 2 Loss: 194.7998 Validation accuracy: 0.8957
Epoch: 3 Loss: 132.0925 Validation accuracy: 0.9243
Epoch: 4 Loss: 96.1115 Validation accuracy: 0.9365
Epoch: 5 Loss: 75.9371 Validation accuracy: 0.9420
Epoch: 6 Loss: 63.3239 Validation accuracy: 0.9491
Epoch: 7 Loss: 54.3939 Validation accuracy: 0.9549
Epoch: 8 Loss: 47.5013 Validation accuracy: 0.9580
Epoch: 9 Loss: 42.0076 Validation accuracy: 0.9632
RNN with BATCH_SIZE=20; ratio=0.7; test_size=500
Training accuracy: 0.971
Test accuracy:     0.961
FINISHED RUNNING
```

[22]:
```python
import pandas as pd

df = pd.DataFrame(index=test_sizes, columns=ratios)

for a_idx, ratio in enumerate(ratios):
    for b_idx, test_size in enumerate(test_sizes):
        df.loc[test_size].iloc[a_idx] = f'{res_hmm[a_idx][b_idx][1]:.3f} /␣
 ↪{res_rnn[a_idx][b_idx][1]:.3f}'

print('Values are (HMM / RNN) test accuracies for ratio and data size')
print('Test Size / Ratio')
display(df)
```

```
Values are (HMM / RNN) test accuracies for ratio and data size
Test Size / Ratio
```

	0.1	0.3	0.5	0.7
40	0.857 / 0.830	0.916 / 0.942	0.903 / 0.931	0.906 / 0.958
80	0.883 / 0.803	0.910 / 0.929	0.893 / 0.932	0.911 / 0.976
200	0.882 / 0.804	0.904 / 0.919	0.898 / 0.946	0.900 / 0.958
500	0.882 / 0.755	0.889 / 0.923	0.897 / 0.947	0.904 / 0.961

As we can see, we got some interesting findings:

With changing the test/train ratio of the dataset, we got better results for each data size with HMM when the ratio was 0.1 - meaning small test size compared to the test corpus.

With every other test-train ratio, we get that the RNN did much better, gradually increasing test accuracy as the test part grew larger. From this, we can infer that RNNs are better at generalizing the solution than HMM - especially with less data and bigger test corpus.

As we can't see any major difference by changing the data size, we can't say that changing the overall size impacts performance.

8 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?

Great project - we learned a lot by implementing the different model types.

9 Instructions for submission of the project segment

This project segment should be submitted to Gradescope at `https://rebrand.ly/project2-submit-code` and `https://rebrand.ly/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 `https://rebrand.ly/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 `https://rebrand.ly/project2-submit-pdf`.