```
# 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/cs236299-2020/project2.git .tmp
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

SHOW HIDDEN OUTPUT

%%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}{,|,}

Project 2: Sequence labeling – The slot filling task

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 two 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 both 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 under different amount of training data. Discuss the pros and cons of the HMM approach and the neural approach.

Setup

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

import torch
import torch.nn as nn
import torchtext as tt

from tqdm import tqdm

# 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
```

Load Data

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

```
shell("""
  wget -nv -N -P data https://raw.githubusercontent.com/nlp-236299/data/master/ATIS/atis.t
  wget -nv -N -P data https://raw.githubusercontent.com/nlp-236299/data/master/ATIS/atis.d
  wget -nv -N -P data https://raw.githubusercontent.com/nlp-236299/data/master/ATIS/atis.t
```

```
  wget -nv -N -P data https://raw.githubusercontent.com/nlp-236299/data/master/ATIS/atis.t
""")
```

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>`.

```
## Turn off annoying torchtext warnings about pending deprecations
warnings.filterwarnings("ignore", module="torchtext", category=UserWarning)
```

```
MIN_FREQ = 3

TEXT = tt.data.Field(init_token="<bos>")
TAG = tt.data.Field(init_token="<bos>")
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.

```
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),

Number of tags: 104
Most common tags: [('O', 38967), ('B-toloc.city_name', 3751), ('B-fromloc.city_name', 37
```

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:

```
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:

```
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. This is different from how you've previously used torch, with a batch size of 1, and you'll need to be careful about the shapes of the various tensors that are being manipulated.

```
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.

```
# 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 <

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-to
```

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:

```
tag_counts = torch.zeros(len(TAG.vocab.itos), device=device)

for batch in train_iter:              # 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

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>                       42094
  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
```

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. If we were to just label every token with the `'O'` tag, we'd expect to be right on this proportion of the tokens.

```
majority_baseline_accuracy = (
  tag_counts[TAG.vocab.stoi['O']]
  / (sum(tag_counts)
     - tag_counts[TAG.vocab.stoi[TAG.pad_token]])
)
print(f'Baseline accuracy: {majority_baseline_accuracy:.3f}')
```

```
Baseline accuracy: 0.637
```

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 $Q = \langle Q_1, Q_2, \ldots, Q_N \rangle$ to denote the possible tags, which is the state space of the HMM, and $\mathcal{V} = \langle v_1, v_2, \ldots v_V \rangle$ to denote the vocabulary of word types. We use $q_t \in Q$ to denote the state at time step t (where t varies from 1 to T), and $o_t \in \mathcal{V}$ to denote the observation (word) at time step t.

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,

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

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

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

> As is typical in notating probabilities, we'll use abbreviations
>
> $$P(q_{t+1} \mid q_t) \equiv P(q_{t+1} = Q_j \mid q_t = Q_i)$$
> $$P(o_t \mid q_t) \equiv P(o_t = \mathcal{V}_j \mid q_t = Q_i)$$
>
> 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.

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

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

Using the above definition, implement the method `train_A` in the `HMM` class, 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 $o_t = \mathcal{V}_j$ conditioned on state $q_t = Q_i$.

We can find this by counting as well.

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

Using the above definitions, implement the `train_B` method in the `HMM` class, 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 sequence labeling. Our goal is to find the most probable sequence of tags $\hat{q} \in Q^T$ given a sequence of words $o \in \mathcal{V}^T$.

$$\begin{aligned}\hat{q} &= \underset{q \in Q^T}{\operatorname{argmax}}(P(q \mid o)) \\ &= \underset{q \in Q^T}{\operatorname{argmax}}(P(q, o)) \\ &= \underset{q \in Q^T}{\operatorname{argmax}}\left(\Pi_{t=1}^{T} P(o_{t+1} \mid q_{t+1}) P(q_{t+1} \mid q_t)\right)\end{aligned}$$

where $P(o_{t+1} = \mathcal{V}_j \mid q_{t+1} = Q_i) = B_{ij}$, $P(q_{t+1} = Q_j \mid q_t = Q_i) = A_{ij}$.

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`.

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

> Warning: It may take up to 30 minutes to tag the entire test set depending on your implementation. 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.

```
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]

  def train_A(self, iterator, delta):
    """Stores 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.
    '''
     tag_counts = torch.zeros(len(self.tag.vocab.itos), device=device)
    for batch in iterator:
        for sent_id in range(len(batch)):
            for tag in batch.tag[:, sent_id]:
                tag_counts[tag] += 1
    tg_counts = tag_counts+1
    pads_count = tg_counts[torch.tensor(self.tag.vocab.stoi[self.tag.pad_token]).long()]
    self.log_priors = (tg_counts / (sum(tg_counts)-pads_count)).log()
    '''
    for batch in iterator:
        row_indices = batch.tag.T[:, 0:-1].reshape(-1)
        colume_indices = batch.tag.T[:,1:].reshape(-1)
        pairs = torch.stack([row_indices, colume_indices]).T
        indices, counts = pairs.unique(return_counts=True, dim=0)
        row_indices, colume_indices = indices.T[0], indices.T[1]
        A[row_indices, colume_indices] += counts
    '''
    for batch in iterator:
        for c in range(batch.tag.size(1)):
            for r in range(batch.tag.size(0)-1):
                A[batch.tag[r,c], batch.tag[r+1,c]] += 1
    '''
    A[1,:] = 0
    A[:,1] = 0
    A=(A+delta) / (torch.sum(A, dim=1).view(-1, 1)+self.N*delta)
    return A

  def train_B(self, iterator, delta):
    """Stores 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:
        row_indices = batch.tag.T.reshape(-1)
        colume_indices = batch.text.T.reshape(-1)
        pairs = torch.stack([row_indices, colume_indices]).T
```

```
        indices, counts = pairs.unique(return_counts=True, dim=0)
        row_indices, colume_indices = indices.T[0], indices.T[1]
        B[row_indices, colume_indices] += counts
    '''
    for batch in iterator: # just another method for verification
        for r,c in zip(batch.tag.T.reshape(-1), batch.text.T.reshape(-1)):
            B[r, c] += 1
    '''
    B[1,:] = 0
    B = (B + delta) / (torch.sum(B, dim=1).view(-1,1) + self.V * delta)

    return B

  def train_all(self, iterator, delta=0.01):
    """Stores A and B 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.

    T = len(words)
    T1 = torch.empty((self.N, T), device=device)
    T2 = torch.empty((self.N, T), device=device)

    # Initilaize the tracking tables from first word
    '''
    b_0 = torch.ones(self.N) # we could zeroify every other tag then <bos>
    b_0[2] = self.N          # as much as calculating the lof priors as was done in experi
    b_0 = b_0 / b_0.sum()    # in the A train section in comment. No significant changes i
    '''
    T1[:, 0] = self.log_B[:, words[0]]
    T2[:, 0] = 0

    #Viterbi
    for t in range(1, T):
        temp_v = T1[:, t - 1].view(-1, 1) + self.log_A + self.log_B[:, words[t]]
        T1[:, t], T2[:, t-1] = torch.max(temp_v, dim=0)
    '''
    #Viterbi - just another method for verification
    for t in range(1, T):
        for i in range(self.N):
            temp_vec = T1[:, t - 1] + self.log_A[:, i]
            T2[i, t - 1] = torch.argmax(temp_vec)
            T1[i, t] = self.log_B[i, words[t]] + torch.max(temp_vec)
```

```
    '''

    bestpath = []
    # The last element of the most likely sequence of states
    x = torch.argmax(T1[:, T - 1])
    bestpath.append(x)
    # Backtracking
    for t in reversed(range(T - 1)):
        x = T2[x.long(), torch.tensor(t).long()]
        bestpath.append(x)

    return (list(reversed(bestpath)))

  def evaluate(self, iterator):
    """Returns the model's performance on a given dataset `iterator`."""
    correct = 0
    total = 0
    for batch in tqdm(iterator):
      for sent_id in range(len(batch)):
        words = batch.text[:, sent_id]
        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 can expect a correct implementation to reach about **90% test set accuracy** after running the following cell.

```
# 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):.10f}\n'
      f'Test accuracy:     {hmm_tagger.evaluate(test_iter):.10f}')
```

```
100%|██████████| 214/214 [00:12<00:00, 17.78it/s]
100%|██████████| 30/30 [00:01<00:00, 28.06it/s]Training accuracy: 0.9136448354
Test accuracy:     0.9057787429
```

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 o_t based on the input x_t and the hidden state of the previous cell h_{t-1}, according to the following equations.

$$h_t = \sigma(Ux_t + Vh_{t-1})$$
$$o_t = \mathrm{softmax}(Wh_t)$$

The parameters here are the elements of the matrices U, V, and W, and the bias terms b_h and b_y which we omitted for simplicity. 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, hidden_0)`: Performs the RNN forward computation over a whole `text_batch` (`batch.text` in the above data loading example) starting with the starting hidden state `hidden_0` of size, say, `1 x batch_size x hidden_size`. 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 (which is 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, ground_truth)` : Computes the loss for a batch by comparing `logits` of a batch returned by `forward` to `ground_truth`, which stores the true tag ids for the batch. Thus `logits` is a tensor of size `max_length x batch_size x N`, and `ground_truth` 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. For reshaping tensors, check out the `torch.Tensor.view` method.

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.

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`.

```
class RNNTagger(nn.Module):
    def __init__(self, text, tag, embedding_size, hidden_size):
        super().__init__()
        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_id = tag.vocab.stoi[tag.init_token]
        self.pad_id = tag.vocab.stoi[tag.pad_token]

        vocab_size = len(self.text.vocab)
        self.embedding = torch.nn.Embedding(vocab_size, embedding_size)
        self.rnn = torch.nn.RNN(input_size=embedding_size, hidden_size=hidden_size, num_la
        self.hidden2output = torch.nn.Linear(hidden_size, vocab_size)
        self.loss_function = nn.CrossEntropyLoss(reduction='sum', ignore_index=self.pad_id

    def forward(self, text_batch, hidden_0=None):
        """Returns the most likely sequence of tags for a sequence of words in `text_batch

        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
        word_embeddings = self.embedding(text_batch)  # seq_len, 1, embedding_size
        outs, hidens = self.rnn(word_embeddings, hidden_0)
        logits = self.hidden2output(outs)
        return logits

    def compute_loss(self, logits, ground_truth):
        loss = self.loss_function(logits.view(-1, self.V), ground_truth.view(-1))
        return loss

    def train_all(self, train_itr, val_itr, epochs, learning_rate):
        """Stores A and B for training dataset `iterator`."""
        # Switch the module to training mode
        self.train()
        # Use Adam to optimize the parameters
        optim = torch.optim.Adam(self.parameters(), lr=learning_rate)
        best_validation_accuracy = -float('inf')
        best_model = None
        # Run the optimization for multiple epochs
        for epoch in range(epochs):
            total = 0
            running_loss = 0.0
            for batch in tqdm(train_itr):
                # Zero the parameter gradients
                self.zero_grad()

                # Input and target
                words = batch.text  # seq_len, 1
                tags = batch.tag  # seq_len, 1

                # Run forward pass and compute loss along the way.
                logits = self.forward(words)
```

```
                loss = self.compute_loss(logits, tags)

                # Perform backpropagation
                (loss / words.size(1)).backward()

                # Update parameters
                optim.step()

                # Training stats
                total += 1
                running_loss += loss.item()

            # Evaluate and track improvements on the validation dataset
            validation_accuracy = self.evaluate(val_itr)
            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}')
        return

    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
        """
        self.eval()
        logits = self.forward(words.view(-1,1))
        predicted_words = logits.argmax(dim=2)
        return predicted_words.squeeze()

    def evaluate(self, iterator):
        """Returns the model's performance on a given dataset `iterator`."""
        correct = 0
        total = 0
        for batch in tqdm(iterator):
            for sent_id in range(len(batch)):
                words = batch.text[:, sent_id]
                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_id:  # stop once we hit padding
                        break
                    else:
                        total += 1
                        if tag_pred == tag_gold:
                            correct += 1
        return correct / total
```

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

```
# 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}')
```

```
100%|██████████| 214/214 [00:02<00:00, 102.77it/s]
100%|██████████| 29/29 [00:00<00:00, 42.58it/s]
  5%|▌         | 10/214 [00:00<00:02, 98.93it/s]Epoch: 0 Loss: 699.8854 Validation accu
100%|██████████| 214/214 [00:01<00:00, 108.03it/s]
100%|██████████| 29/29 [00:00<00:00, 42.98it/s]
  5%|▌         | 11/214 [00:00<00:01, 103.88it/s]Epoch: 1 Loss: 290.9520 Validation accu
100%|██████████| 214/214 [00:01<00:00, 107.99it/s]
100%|██████████| 29/29 [00:00<00:00, 43.30it/s]
  5%|▌         | 10/214 [00:00<00:02, 94.32it/s]Epoch: 2 Loss: 179.0184 Validation accu
100%|██████████| 214/214 [00:01<00:00, 108.40it/s]
100%|██████████| 29/29 [00:00<00:00, 42.40it/s]
  5%|▌         | 10/214 [00:00<00:02, 97.58it/s]Epoch: 3 Loss: 120.9971 Validation accu
100%|██████████| 214/214 [00:01<00:00, 108.79it/s]
100%|██████████| 29/29 [00:00<00:00, 40.82it/s]
  5%|▌         | 11/214 [00:00<00:01, 102.92it/s]Epoch: 4 Loss: 90.3452 Validation accur
100%|██████████| 214/214 [00:01<00:00, 107.09it/s]
100%|██████████| 29/29 [00:00<00:00, 42.66it/s]
  5%|▌         | 10/214 [00:00<00:02, 97.93it/s]Epoch: 5 Loss: 71.5639 Validation accur
100%|██████████| 214/214 [00:02<00:00, 106.94it/s]
100%|██████████| 29/29 [00:00<00:00, 41.52it/s]
  5%|▌         | 11/214 [00:00<00:01, 106.85it/s]Epoch: 6 Loss: 59.4664 Validation accur
100%|██████████| 214/214 [00:02<00:00, 104.15it/s]
100%|██████████| 29/29 [00:00<00:00, 41.91it/s]
  5%|▌         | 11/214 [00:00<00:02, 101.46it/s]Epoch: 7 Loss: 51.0500 Validation accur
100%|██████████| 214/214 [00:02<00:00, 106.30it/s]
100%|██████████| 29/29 [00:00<00:00, 42.76it/s]
  5%|▌         | 10/214 [00:00<00:02, 93.68it/s]Epoch: 8 Loss: 44.8281 Validation accur
100%|██████████| 214/214 [00:02<00:00, 105.03it/s]
100%|██████████| 29/29 [00:00<00:00, 42.35it/s]
  2%|▏         | 4/214 [00:00<00:06, 34.10it/s]Epoch: 9 Loss: 40.0092 Validation accura
100%|██████████| 214/214 [00:06<00:00, 33.12it/s]
100%|██████████| 30/30 [00:00<00:00, 43.03it/s]Training accuracy: 0.968
Test accuracy:     0.960
```

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 bits, `h` and `c`. You'll thus need to initialize both of these when performing forward computations.

```
class LSTMTagger(nn.Module):
    def __init__(self, text, tag, embedding_size, hidden_size):
        super().__init__()
        self.text = text
        self.tag = tag
        self.V = len(text.vocab.itos)  # vocabulary size
        self.N = len(tag.vocab.itos)  # state space size
        self.pad_id = tag.vocab.stoi[tag.pad_token]

        vocab_size = len(self.text.vocab)
        self.embedding = torch.nn.Embedding(vocab_size, embedding_size)
        self.lstm = torch.nn.LSTM(input_size=embedding_size, hidden_size=hidden_size, num_
        self.hidden2output = torch.nn.Linear(hidden_size, vocab_size)
        self.loss_function = nn.CrossEntropyLoss(reduction='sum', ignore_index=self.pad_id

    def forward(self, text_batch, hidden_0=None, cell_0=None):
        """Returns the most likely sequence of tags for a sequence of words in `text_batch

        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
        h_c = (hidden_0, cell_0) if (hidden_0,cell_0) != (None, None) else None
        word_embeddings = self.embedding(text_batch)  # seq_len, 1, embedding_size
        outs,_  = self.lstm(word_embeddings, h_c)
        logits = self.hidden2output(outs)
        return logits

    def compute_loss(self, logits, ground_truth):
        loss = self.loss_function(logits.view(-1, self.V), ground_truth.view(-1))
        return loss

    def train_all(self, train_itr, val_itr, epochs, learning_rate):
        """Stores A and B for training dataset `iterator`."""
        # Switch the module to training mode
        self.train()
        # Use Adam to optimize the parameters
```

```
        optim = torch.optim.Adam(self.parameters(), lr=learning_rate)
        best_validation_accuracy = -float('inf')
        best_model = None
        # Run the optimization for multiple epochs
        for epoch in range(epochs):
            total = 0
            running_loss = 0.0
            for batch in tqdm(train_itr):
                # Zero the parameter gradients
                self.zero_grad()

                # Input and target
                words = batch.text  # seq_len, 1
                tags = batch.tag  # seq_len, 1

                # Run forward pass and compute loss along the way.
                logits = self.forward(words)
                loss = self.compute_loss(logits, tags)

                # Perform backpropagation
                (loss / words.size(1)).backward()

                # Update parameters
                optim.step()

                # Training stats
                total += 1
                running_loss += loss.item()

            # Evaluate and track improvements on the validation dataset
            validation_accuracy = self.evaluate(val_itr)
            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}')
        return

    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
        """
        self.eval()
        logits = self.forward(words.view(-1,1))
        predicted_words = logits.argmax(dim=2)
        return predicted_words.squeeze()

    def evaluate(self, iterator):
```

```
        """Returns the model's performance on a given dataset `iterator`."""
        correct = 0
        total = 0
        for batch in tqdm(iterator):
            for sent_id in range(len(batch)):
                words = batch.text[:, sent_id]
                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_id:  # stop once we hit padding
                        break
                    else:
                        total += 1
                        if tag_pred == tag_gold:
                            correct += 1
        return correct / total
```

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

```
# 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}')
```

```
100%|██████████| 214/214 [00:03<00:00, 70.39it/s]
100%|██████████| 29/29 [00:01<00:00, 25.54it/s]
  3%|▌         | 6/214 [00:00<00:03, 56.49it/s]Epoch: 0 Loss: 766.7183 Validation accur
100%|██████████| 214/214 [00:03<00:00, 69.09it/s]
100%|██████████| 29/29 [00:01<00:00, 25.77it/s]
  3%|▌         | 7/214 [00:00<00:02, 69.49it/s]Epoch: 1 Loss: 335.3721 Validation accur
100%|██████████| 214/214 [00:03<00:00, 68.74it/s]
100%|██████████| 29/29 [00:01<00:00, 25.76it/s]
  4%|▌         | 8/214 [00:00<00:02, 71.65it/s]Epoch: 2 Loss: 242.6323 Validation accur
100%|██████████| 214/214 [00:03<00:00, 70.09it/s]
100%|██████████| 29/29 [00:01<00:00, 25.22it/s]
  3%|▌         | 7/214 [00:00<00:02, 69.33it/s]Epoch: 3 Loss: 180.2119 Validation accur
100%|██████████| 214/214 [00:03<00:00, 69.80it/s]
100%|██████████| 29/29 [00:01<00:00, 25.73it/s]
  3%|▌         | 6/214 [00:00<00:03, 58.11it/s]Epoch: 4 Loss: 131.3403 Validation accur
100%|██████████| 214/214 [00:03<00:00, 68.32it/s]
100%|██████████| 29/29 [00:01<00:00, 25.68it/s]
  3%|▌         | 7/214 [00:00<00:03, 61.27it/s]Epoch: 5 Loss: 101.7092 Validation accur
100%|██████████| 214/214 [00:03<00:00, 68.84it/s]
100%|██████████| 29/29 [00:01<00:00, 25.34it/s]
  3%|▌         | 7/214 [00:00<00:03, 66.75it/s]Epoch: 6 Loss: 83.5255 Validation accura
100%|██████████| 214/214 [00:03<00:00, 69.82it/s]
100%|██████████| 29/29 [00:01<00:00, 25.36it/s]
```

```
  3%|▉          | 7/214 [00:00<00:02, 69.42it/s]Epoch: 7 Loss: 71.4472 Validation accura
100%|██████████| 214/214 [00:03<00:00, 69.41it/s]
100%|██████████| 29/29 [00:01<00:00, 25.81it/s]
  4%|▉          | 8/214 [00:00<00:02, 75.67it/s]Epoch: 8 Loss: 62.6736 Validation accura
100%|██████████| 214/214 [00:03<00:00, 70.48it/s]
100%|██████████| 29/29 [00:01<00:00, 25.75it/s]
  1%|           | 2/214 [00:00<00:12, 16.54it/s]Epoch: 9 Loss: 55.7900 Validation accurac
100%|██████████| 214/214 [00:11<00:00, 18.43it/s]
100%|██████████| 30/30 [00:01<00:00, 26.56it/s]Training accuracy: 0.959
Test accuracy:     0.952
```

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

```
def subsample(ratio, test_size=50):
    trn, vl, tst = tt.datasets.SequenceTaggingDataset.splits(
                fields=fields,
                path='./data/',
                train='atis.train.txt',
                validation='atis.dev.txt',
                test='atis.test.txt')

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

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

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

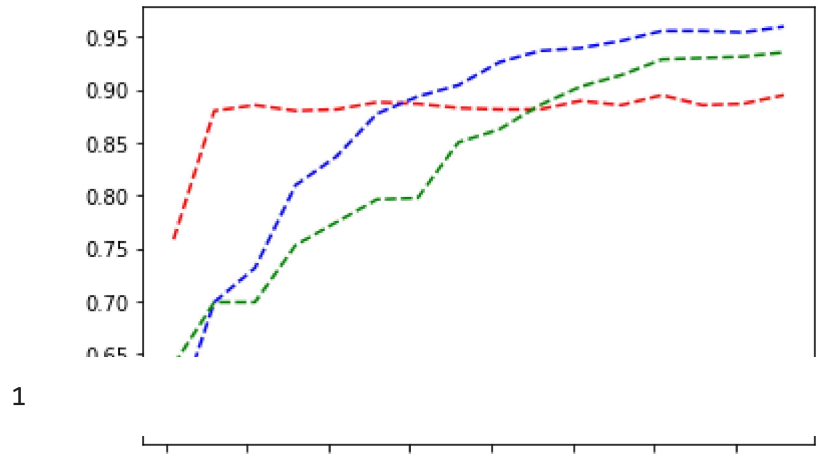
```
    train_itr, val_itr, test_itr = tt.data.BucketIterator.splits(
    (train, val, test),
    batch_size=BATCH_SIZE,
    repeat=False,
    device=device)

    return TEXT, TAG, train_itr, val_itr, test_itr
```

```
# Instantiate and train classifier
start, end, step = 0.01, 0.8, 0.05
rnn_results, hmm_results, lstm_results = [],[],[]
ratio_range = torch.arange(start, end, step)

for ratio in ratio_range:

    TEXT, TAG, train_itr, val_itr, test_itr = subsample(ratio)
    rnn_tagger = RNNTagger(TEXT, TAG, embedding_size=36, hidden_size=36).to(device)
    rnn_tagger.train_all(train_itr, val_itr, epochs=10, learning_rate=0.001)
    rnn_tagger.load_state_dict(rnn_tagger.best_model)

    # Evaluate model performance
    rnn_results.append(rnn_tagger.evaluate(test_itr))

    lstm_tagger = LSTMTagger(TEXT, TAG, embedding_size=36, hidden_size=36).to(device)
    lstm_tagger.train_all(train_itr, val_itr, epochs=10, learning_rate=0.001)
    lstm_tagger.load_state_dict(lstm_tagger.best_model)

    # Evaluate model performance
    lstm_results.append(lstm_tagger.evaluate(test_itr))

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

    # Evaluate model performance

    hmm_results.append(hmm_tagger.evaluate(test_itr))
```

```
100%|██████████| 3/3 [00:00<00:00, 121.58it/s]
100%|██████████| 29/29 [00:00<00:00, 42.25it/s]
100%|██████████| 3/3 [00:00<00:00, 141.38it/s]
 21%|██        | 6/29 [00:00<00:00, 53.70it/s]Epoch: 0 Loss: 1198.0683 Validation acc
100%|██████████| 29/29 [00:00<00:00, 41.98it/s]
100%|██████████| 3/3 [00:00<00:00, 124.64it/s]
 21%|██        | 6/29 [00:00<00:00, 54.10it/s]Epoch: 1 Loss: 1182.4480 Validation acc
100%|██████████| 29/29 [00:00<00:00, 42.26it/s]
100%|██████████| 3/3 [00:00<00:00, 123.34it/s]
 21%|██        | 6/29 [00:00<00:00, 52.55it/s]Epoch: 2 Loss: 1161.4372 Validation acc
100%|██████████| 29/29 [00:00<00:00, 41.12it/s]
100%|██████████| 3/3 [00:00<00:00, 138.86it/s]
 17%|█▋        | 5/29 [00:00<00:00, 44.91it/s]Epoch: 3 Loss: 1148.4037 Validation acc
```

```
100%|██████████| 29/29 [00:00<00:00, 39.86it/s]
100%|██████████| 3/3 [00:00<00:00, 126.94it/s]
 17%|██        | 5/29 [00:00<00:00, 46.49it/s]Epoch: 4 Loss: 1129.2455 Validation acc
100%|██████████| 29/29 [00:00<00:00, 37.37it/s]
100%|██████████| 3/3 [00:00<00:00, 97.30it/s]
 21%|██        | 6/29 [00:00<00:00, 52.95it/s]Epoch: 5 Loss: 1106.3382 Validation acc
100%|██████████| 29/29 [00:00<00:00, 38.79it/s]
100%|██████████| 3/3 [00:00<00:00, 106.59it/s]
 21%|██        | 6/29 [00:00<00:00, 54.00it/s]Epoch: 6 Loss: 1081.2416 Validation acc
100%|██████████| 29/29 [00:00<00:00, 41.13it/s]
100%|██████████| 3/3 [00:00<00:00, 125.70it/s]
 21%|██        | 6/29 [00:00<00:00, 51.88it/s]Epoch: 7 Loss: 1063.2270 Validation acc
100%|██████████| 29/29 [00:00<00:00, 40.90it/s]
100%|██████████| 3/3 [00:00<00:00, 136.33it/s]
 21%|██        | 6/29 [00:00<00:00, 49.06it/s]Epoch: 8 Loss: 1026.7814 Validation acc
100%|██████████| 29/29 [00:00<00:00, 41.84it/s]
100%|██████████| 3/3 [00:00<00:00, 42.95it/s]
100%|██████████| 3/3 [00:00<00:00, 88.08it/s]
  0%|          | 0/29 [00:00<?, ?it/s]Epoch: 9 Loss: 997.9568 Validation accuracy: 0.
100%|██████████| 29/29 [00:01<00:00, 25.35it/s]
100%|██████████| 3/3 [00:00<00:00, 92.01it/s]
 14%|█▍        | 4/29 [00:00<00:00, 35.74it/s]Epoch: 0 Loss: 1210.3504 Validation acc
100%|██████████| 29/29 [00:01<00:00, 25.14it/s]
100%|██████████| 3/3 [00:00<00:00, 81.25it/s]
 14%|█▍        | 4/29 [00:00<00:00, 35.55it/s]Epoch: 1 Loss: 1200.9523 Validation acc
100%|██████████| 29/29 [00:01<00:00, 26.00it/s]
100%|██████████| 3/3 [00:00<00:00, 93.12it/s]
 14%|█▍        | 4/29 [00:00<00:00, 33.48it/s]Epoch: 2 Loss: 1190.2758 Validation acc
100%|██████████| 29/29 [00:01<00:00, 25.82it/s]
100%|██████████| 3/3 [00:00<00:00, 85.70it/s]
 14%|█▍        | 4/29 [00:00<00:00, 34.63it/s]Epoch: 3 Loss: 1181.6237 Validation acc
100%|██████████| 29/29 [00:01<00:00, 26.20it/s]
100%|██████████| 3/3 [00:00<00:00, 86.24it/s]
 14%|█▍        | 4/29 [00:00<00:00, 36.66it/s]Epoch: 4 Loss: 1170.7654 Validation acc
100%|██████████| 29/29 [00:01<00:00, 26.28it/s]
100%|██████████| 3/3 [00:00<00:00, 87.86it/s]
 14%|█▍        | 4/29 [00:00<00:00, 32.20it/s]Epoch: 5 Loss: 1158.4903 Validation acc
100%|██████████| 29/29 [00:01<00:00, 24.90it/s]
100%|██████████| 3/3 [00:00<00:00, 91.59it/s]
 14%|█▍        | 4/29 [00:00<00:00, 33.36it/s]Epoch: 6 Loss: 1140.0977 Validation acc
100%|██████████| 29/29 [00:01<00:00, 24.41it/s]
100%|██████████| 3/3 [00:00<00:00, 96.08it/s]
 14%|█▍        | 4/29 [00:00<00:00, 35.96it/s]Epoch: 7 Loss: 1122.8099 Validation acc
100%|██████████| 29/29 [00:01<00:00, 25.16it/s]
100%|██████████| 3/3 [00:00<00:00, 84.42it/s]
```

```
1 import matplotlib.pyplot as plt
2 # red dashes, blue squares and green triangles
3 plt.plot(ratio_range, hmm_results, 'r--', ratio_range, rnn_results, 'b--', ratio_range, ls
4 plt.show()
```



The experiment:

The models of RNN, LSTM and HMM were trained on the partial datasets accroding to the ratio, in a for-loop manner iterating over different ratio values beggining on 0.01 to 0.8 with steps of 0.05.

Results and conclusions:

It can be seen from the plot, where the **red line** is **HMM**, the** blue line** is **RNN** and the **green line** is **LSTM**, that **the HMM is much more consistent with its result**, **it is better the the RNN and LSTM models till some point**, from which can be concluded that **for small amounts of data it performs better then the neural network models**, which is an expectable result, **because networks need as big as possible amounts of data to optimize efficiently their parameters**.** The RNN and LSTM models are pretty close in their accurancy**.** Generally the LSTM outperforms RNN** in cases were there is a lot data, here we can see mostly that RNN outperforms LSTM, which points on the small amount of data, and effect that also had been seen on the whole dataset. **The RNN outperforms both models, the lstm model on pretty early point of ratio~0.06**, and **the HMM model about ratio~2.6**, which points out as was mentioned that **with enough data it better then an HMM model, but as the amounts of data generally too small, it also outperforms LSTM**. **The LSTM also outperform HMM as expected for a neural network model with more data then the RNN, about ratio~4.6.**

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?

- It was very clear.
- The reading and the previous labs were a good background for the assigment.
- Nothing comes to my mind.

Instructions for submission of the project segment

This project segment should be submitted to Gradescope, 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.

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.

End of project segment 2