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

May 22, 2023

[ ]:
```
# 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/cs236299-2023-spring/project2.git .tmp
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

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

1 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.

1.1 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.

1.2 Setup

[ ]:
```
import copy
import math
import matplotlib.pyplot as plt
import random
import csv

import wget
import torch
import torch.nn as nn
import datasets

from datasets import load_dataset
from tokenizers import Tokenizer
from tokenizers.pre_tokenizers import WhitespaceSplit
from tokenizers.processors import TemplateProcessing
from tokenizers import normalizers
from tokenizers.models import WordLevel
from tokenizers.trainers import WordLevelTrainer
from transformers import PreTrainedTokenizerFast

from tqdm.auto 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)
```

1.3 Loading data

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

[ ]:
```
# 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)
```

1.4 Data preprocessing

We again use `datasets` and `tokenizers` to load data and convert words to indices in the vocabulary.

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]`.

[ ]:
```
for split in ['train', 'dev', 'test']:
    in_file = f'data/atis.{split}.txt'
    out_file = f'data/atis.{split}.csv'

    with open(in_file, 'r') as f_in:
        with open(out_file, 'w') as f_out:
            text, tag = [], []
            writer = csv.writer(f_out)
            writer.writerow(('text','tag'))
            for line in f_in:
                if line.strip() == '':
                    writer.writerow((' '.join(text), ' '.join(tag)))
                    text, tag = [], []
                else:
                    token, label = line.split('\t')
                    text.append(token)
                    tag.append(label.strip())
```

Let's take a look at what each data file looks like.

[ ]:
```
shell('head "data/atis.train.csv"')
```

We use `datasets` to prepare the data.

[ ]:
```
atis = load_dataset('csv', data_files={'train':'data/atis.train.csv', \
                                       'val': 'data/atis.dev.csv', \
                                       'test': 'data/atis.test.csv'})
atis
```

[ ]:
```
train_data = atis['train']
val_data = atis['val']
test_data = atis['test']

train_data.shuffle(seed=seed)
```

We build tokenizers from the training data to tokenize both text and tag and convert them into word ids.

[ ]:
```
MIN_FREQ = 3
unk_token = '[UNK]'
pad_token = '[PAD]'
bos_token = '<bos>'

def train_tokenizers(dataset, min_freq):
    text_tokenizer = Tokenizer(WordLevel(unk_token=unk_token))
    text_tokenizer.pre_tokenizer = WhitespaceSplit()
    text_tokenizer.normalizer = normalizers.Lowercase()

    text_trainer = WordLevelTrainer(min_frequency=min_freq,␣
 ↪special_tokens=[pad_token, unk_token,bos_token])
    text_tokenizer.train_from_iterator(dataset['text'], trainer=text_trainer)
    text_tokenizer.post_processor = TemplateProcessing(single=f"{bos_token}␣
 ↪$A", special_tokens=[(bos_token, text_tokenizer.token_to_id(bos_token))])

    tag_tokenizer = Tokenizer(WordLevel(unk_token=unk_token))
    tag_tokenizer.pre_tokenizer = WhitespaceSplit()

    tag_trainer = WordLevelTrainer(special_tokens=[pad_token, unk_token,␣
 ↪bos_token])

    tag_tokenizer.train_from_iterator(dataset['tag'], trainer=tag_trainer)

    tag_tokenizer.post_processor = TemplateProcessing(single=f"{bos_token} $A",␣
 ↪special_tokens=[(bos_token, tag_tokenizer.token_to_id(bos_token))])
    return text_tokenizer, tag_tokenizer

text_tokenizer, tag_tokenizer = train_tokenizers(train_data, MIN_FREQ)
```

We use `datasets.Dataset.map` to convert text into word ids. As shown in lab 1-5, first we need to wrap `tokenizer` with the `transformers.PreTrainedTokenizerFast` class to be compatible with the `datasets` library.

[ ]:
```
hf_text_tokenizer = PreTrainedTokenizerFast(tokenizer_object=text_tokenizer,␣
 ↪pad_token=pad_token, unk_token=unk_token, bos_token=bos_token)
```

```
hf_tag_tokenizer = PreTrainedTokenizerFast(tokenizer_object=tag_tokenizer,␣
↪pad_token=pad_token, unk_token=unk_token, bos_token=bos_token)
```

```
[ ]: def encode(example):
         example['input_ids'] = hf_text_tokenizer(example['text']).input_ids
         example['tag_ids'] = hf_tag_tokenizer(example['tag']).input_ids
         return example

     train_data = train_data.map(encode)
     val_data = val_data.map(encode)
     test_data = test_data.map(encode)
```

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

```
[ ]: # Compute size of vocabulary
     text_vocab = text_tokenizer.get_vocab()
     tag_vocab = tag_tokenizer.get_vocab()
     vocab_size = len(text_vocab)
     num_tags = len(tag_vocab)

     print(f"Size of English vocabulary: {vocab_size}")
     print(f"Number of tags: {num_tags}")
```

1.5 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 prepended `<bos>` for both text and tag. `Tokenizers` 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: {bos_token}
     Initial tag id:     {tag_vocab[bos_token]}
     """)
```

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

```
[ ]: print(f"""
     Pad tag string: {pad_token}
     Pad tag id:     {tag_vocab[pad_token]}
     """)
```

To load data in batched tensors, we use `torch.utils.data.DataLoader` for data splits, which enables us to iterate over the dataset under a given `BATCH_SIZE`. For the test set, we use a batch size of 1, to make the decoding implementation easier.

[ ]:
```
BATCH_SIZE = 32     # batch size for training and validation

# Defines how to batch a list of examples together
def collate_fn(examples):
    batch = {}
    bsz = len(examples)
    input_ids, tag_ids = [], []
    for example in examples:
        input_ids.append(example['input_ids'])
        tag_ids.append(example['tag_ids'])

    max_length = max([len(word_ids) for word_ids in input_ids])

    tag_batch = torch.zeros(bsz, max_length).long().fill_(tag_vocab[pad_token]).
 ↪to(device)
    text_batch = torch.zeros(bsz, max_length).long().
 ↪fill_(text_vocab[pad_token]).to(device)
    for b in range(bsz):
        text_batch[b][:len(input_ids[b])] = torch.LongTensor(input_ids[b]).
 ↪to(device)
        tag_batch[b][:len(tag_ids[b])] = torch.LongTensor(tag_ids[b]).to(device)

    batch['tag_ids'] = tag_batch
    batch['input_ids'] = text_batch
    return batch

def get_iterators(train_data, val_data, test_data):
    train_iter = torch.utils.data.DataLoader(train_data,
                                              batch_size=BATCH_SIZE,
                                              shuffle=True,
                                              collate_fn=collate_fn)
    val_iter = torch.utils.data.DataLoader(val_data,
                                            batch_size=BATCH_SIZE,
                                            shuffle=False,
                                            collate_fn=collate_fn)
    test_iter = torch.utils.data.DataLoader(test_data,
                                             batch_size=BATCH_SIZE,
                                             shuffle=False,
                                             collate_fn=collate_fn)
    return train_iter, val_iter, test_iter

train_iter, val_iter, test_iter = get_iterators(train_data, val_data, test_data)
```

Now, we can iterate over the dataset. We used 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.

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

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

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

     # Extract the first sentence in the batch, both text and tags
     first_sentence = batch['input_ids'][0]
     first_tags = batch['tag_ids'][0]

     # Print out the first sentence, as token ids and as text
     print("First sentence in batch")
     print(f"{first_sentence}")
     print(f"{hf_text_tokenizer.decode(first_sentence)}\n")

     print("First tags in batch")
     print(f"{first_tags}")
     print(f"{hf_tag_tokenizer.decode(first_tags)}")
```

The goal of this project is to predict the sequence of tags `batch['tag_ids']` given a sequence of words `batch['input_ids']`.

2 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:

```
[ ]: def count_tags(iterator):
       tag_counts = torch.zeros(len(tag_vocab), device=device)

       for batch in iterator:
         tags = batch['tag_ids'].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[pad_token]] = 0
       return tag_counts

     tag_counts = count_tags(train_iter)
```

```
for tag_id in range(len(tag_vocab)):
  print(f'{tag_id:3}  {hf_tag_tokenizer.decode(tag_id):30}{tag_counts[tag_id].
↪item():3.0f}')
```

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:

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

3 HMM for sequence labeling

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

3.1 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).

3.2 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,

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

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

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

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

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

> where the i and j are clear from context.

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

3.2.1 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.

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

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

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

> You'll want to go ahead and implement this part now, and test it below, before moving on to the next goal.
>
> Remember that the training data is being delivered to you batched.

3.2.2 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.

3.3 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 $\widehat{\mathbf{q}} \in Q^T$ given a sequence of words $\mathbf{w} \in \mathcal{V}^T$.

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

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]`.

3.3.1 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.

3.4 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.

[ ]:
```
class HMMTagger():
  def __init__ (self, hf_text_tokenizer, hf_tag_tokenizer):
    self.hf_text_tokenizer = hf_text_tokenizer
    self.hf_tag_tokenizer = hf_tag_tokenizer

    self.V = len(self.hf_text_tokenizer)    # vocabulary size
    self.N = len(self.hf_tag_tokenizer)     # state space size

    self.initial_state_id = self.hf_tag_tokenizer.bos_token_id
    self.pad_state_id = self.hf_tag_tokenizer.pad_token_id
    self.pad_word_id = self.hf_text_tokenizer.pad_token_id

  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.
    ...

    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.

    ...

    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.

    ...

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

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

[ ]:
```
# Instantiate and train classifier
hmm_tagger = HMMTagger(hf_text_tokenizer, hf_tag_tokenizer)
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}')
```

4 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.

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

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.

4.1 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.

4.2 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`.

[ ]:
```
class RNNTagger(nn.Module):
  ...
```

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**.

[ ]:
```
# Instantiate and train classifier
rnn_tagger = RNNTagger(hf_text_tokenizer, hf_tag_tokenizer, 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}')
```

5 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.

[ ]:
```
class LSTMTagger(nn.Module):
  ...
```

Run the cell below to train an LSTM, and evaluate it. A proper implementation should reach about **94%+ accuracy**.

[ ]:
```
# Instantiate and train classifier
lstm_tagger = LSTMTagger(hf_text_tokenizer, hf_tag_tokenizer,␣
 ↪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}')
```

6 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.

```
[ ]: 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)

     atis = load_dataset('csv', data_files={'train':'data/atis.train.csv', \
                                            'val': 'data/atis.dev.csv', \
                                            'test': 'data/atis.test.csv'})
     train_data = atis['train']
     test_data = atis['test']

     # Subsample
     train_data = train_data.shuffle(seed=seed)
     train_data = train_data.select(list(range(len(train_data)))[:int(math.
      ↪floor(len(train_data)*ratio))])
     test_data = test_data.shuffle(seed=seed)
     test_data = test_data.select(list(range(len(test_data)))[:test_size])

     # Rebuild vocabulary
     text_tokenizer, tag_tokenizer = train_tokenizers(train_data, MIN_FREQ)

     # Encode data
     hf_text_tokenizer = PreTrainedTokenizerFast(tokenizer_object=text_tokenizer,␣
      ↪pad_token=pad_token)

     hf_tag_tokenizer = PreTrainedTokenizerFast(tokenizer_object=tag_tokenizer,␣
      ↪pad_token=pad_token)

     def encode(example):
         example['input_ids'] = hf_text_tokenizer(example['text']).input_ids
```

```
    example['tag_ids'] = hf_tag_tokenizer(example['tag']).input_ids
    return example

train_data = train_data.map(encode)
test_data = test_data.map(encode)

# Create iterators
train_iter, val_iter, test_iter = get_iterators(train_data, val_data, test_data)
```

[ ]:
```
...
```

[ ]:
```
...
```

[ ]:
```
...
```

Type your answer here, replacing this text.

7 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?

Type your answer here, replacing this text.

8 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`.