CS187 Lab 2-5: Sequence labeling with recurrent neural networks

September 22, 2023

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

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

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

shell("""
ls requirements.txt >/dev/null 2>&1
if [ ! $? = 0 ]; then
 rm -rf .tmp
 git clone https://github.com/cs187-2021/lab2-5.git .tmp
 mv .tmp/tests ./
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

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

In the last lab, you saw how to use hidden Markov models (HMMs) for sequence labeling. In this lab, you will use recurrent neural networks (RNNs) for sequence labeling.

In this lab, we consider the task of automatic punctuation restoration from unpunctuated text, which is useful for post-processing transcribed speech from speech recognition systems (since we don't want users to have to utter all punctuation marks). We can formulate this task as a sequence labeling task, predicting for each word the punctuation that should follow. If there's no punctuation following the word, we use a special tag `O` for "other".

The dataset we use is the Federalist papers, but this time we use text without punctuation as our input, and predict the punctuation following each word. An example constructed from the dataset looks like below, which correponds to the punctuated sentence `the powers to make treaties and to send and receive ambassadors , speak their own propriety .`

Token	Label
<bos>	O
the	O
powers	O
to	O
make	O
treaties	O
and	O
to	O
send	O
and	O
receive	O
ambassadors	,
speak	O
their	O
own	O
propriety	.

Preparation and setup

```
[ ]: import copy

import wget
import torch
import torch.nn as nn

import csv
import random

from datasets import load_dataset
from tokenizers import Tokenizer
from tokenizers.pre_tokenizers import WhitespaceSplit
from tokenizers import normalizers
from tokenizers.models import WordLevel
from tokenizers.trainers import WordLevelTrainer
```

```
from transformers import PreTrainedTokenizerFast

from collections import Counter
from tqdm.auto import tqdm

# Fix random seed for replicability
SEED=1234
random.seed(SEED)
torch.manual_seed(SEED)
```

[ ]:
```
## GPU check
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)
```

1 Dataset preparation and exploration

We download the dataset and process it by extracting the text sequences and their corresponding labels, and save it in CSV format.

[ ]:
```
# Prepare to download needed data
def download_if_needed(source, dest, filename):
    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/
 ↪Federalist/"
data_path = "data/"

# Download the files
for filename in ["federalist_tag.train.txt",
                 "federalist_tag.dev.txt",
                 "federalist_tag.test.txt"
                ]:
    download_if_needed(source_path, data_path, filename)

# Read in the dataset, extracting the token sequences and the
# corresponding tag sequences and generate a CSV file of the
# processed data
for split in ['train', 'dev', 'test']:
    in_file = f'data/federalist_tag.{split}.txt'
    out_file = f'data/federalist_tag.{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/federalist_tag.train.csv"')
```

We'll use the HuggingFace `datasets` package to further prepare the data.

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

```
[ ]: # Split out the training, validation, and test sets
     train_data = federalist_dataset['train']
     val_data = federalist_dataset['val']
     test_data = federalist_dataset['test']
```

We build a tokenizer from the training data to tokenize text and convert tokens into word ids.

```
[ ]: # We place a limit on the size of the vocabulary, including only the
     # `MAX_VOCAB_SIZE` most frequent words. All others will become `[UNK]`.
     MAX_VOCAB_SIZE = 5000
     unk_token = '[UNK]'
     pad_token = '[PAD]'

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

     trainer = WordLevelTrainer(vocab_size=MAX_VOCAB_SIZE,␣
      ↪special_tokens=[pad_token, unk_token])
     text_tokenizer.train_from_iterator(train_data['text'], trainer=trainer)
```

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

```
[ ]: # Wrap the tokenizer to allow use with HF datasets
hf_text_tokenizer = PreTrainedTokenizerFast(tokenizer_object=text_tokenizer,
                                            pad_token=pad_token,
                                            unk_token=unk_token)

# encode(example) -- Updates the `example` by tokenizing and encoding the text
# into a list of token ids.
def encode(example):
    return hf_text_tokenizer(example['text'])

# Encode the training, validation, and test sets
train_data = train_data.map(encode)
val_data = val_data.map(encode)
test_data = test_data.map(encode)
```

```
[ ]: # An example from the training dataset, showing the string of tokens,
# the string of tags, and the list of token ids. The token type ids
# and attention mask can be ignored for the time being.
train_data[0]
```

We also need to convert the string of tags into a list of tag ids.

```
[ ]: tag_tokenizer = Tokenizer(WordLevel())
tag_tokenizer.pre_tokenizer = WhitespaceSplit()

tag_trainer = WordLevelTrainer(special_tokens=[pad_token])
tag_tokenizer.train_from_iterator(train_data['tag'], trainer=tag_trainer)

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

# encode_tag(example) -- Updates the `example` by tokenizing and encoding the␣
 ↪tag string
# into a list of tag ids.
def encode_tag(example):
    example['tag_ids'] = hf_tag_tokenizer(example['tag']).input_ids
    return example

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

```
[ ]: # An example from the training dataset, showing the string of tokens,
# the string of tags, and the list of token ids. The token type ids
# and attention mask can be ignored for the time being.
train_data[0]
```

[ ]:
```python
# Print out some stats
text_vocab = hf_text_tokenizer.get_vocab()
tag_vocab = hf_tag_tokenizer.get_vocab()
vocab_size = len(text_vocab)
num_tags = len(tag_vocab)

most_common_tokens = Counter(token
                             for sentence in train_data['text']
                             for token in sentence.split()).most_common(10)
most_common_tags = Counter(tag
                           for sentence_tags in train_data['tag']
                           for tag in sentence_tags.split()).most_common(10)

print(f"Size of vocab: {vocab_size}")
print (f"Most common English words: {most_common_tokens}\n")
print(f"Number of tags: {num_tags}")
print (f"Most common tags: {most_common_tags}")
```

You can see from above that the most common punctuation is comma, on which we will evaluate precision, recall, and F-1 scores later.

We mapped words that are not among the most frequent words (specified by `MAX_VOCAB_SIZE`) to a special unknown token:

[ ]:
```python
unk_index = text_vocab[unk_token]

print (f"Unknown word: {unk_token}\n"
       f"Unknown index: {unk_index}")
```

To facilitate batching sentences of different lengths into the same tensor we also reserved a special padding symbol `[PAD]` for both `text_vocab` and `tag_vocab`.

[ ]:
```python
print (f"Padding token: {pad_token}")
text_pad_index = text_vocab[pad_token]
print (f"Padding text_vocab token id: {text_pad_index}")
tag_pad_index = tag_vocab[pad_token]
print (f"Padding tag_vocab token id: {tag_pad_index}")
```

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`, which is set to be `1` throughout this lab. We still batch the data because other torch functions expect data to be batched.

[ ]:
```python
# We use batch size 1 for simplicity
BATCH_SIZE = 1

# collate(examples) -- Combines a list of examples into a single batch
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

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)
```

Let's take a look at the dataset. Recall from project 1 that there are two different ways of iterating over the dataset, one by iterating over individual examples, the other by iterating over batches of examples.

[ ]:
```
# Iterating over individual examples:
# Note that the words are the original words, so you'd need to manually
# replace them with `[UNK]` if not in the vocabulary.
example = train_data[1]
text = example['text'].split() # a sequence of unpunctuated words
tags = example['tag'].split()  # a sequence of tags indicating the proper␣
 ↪punctuation
print (f'{"TYPE":15}: {"TAG"}')
for word, tag in zip(text, tags):
```

```
  print (f'{word:15}: {tag}')
```

Alternatively, we can produce the data a batch at a time, as in the example below. Note the "shape" of a batch; it's a two-dimensional tensor of size `batch_size x max_length`. (In this case, `batch_size` is 1.) Thus, to extract a sentence from a batch, we need to index by the *first* dimension.

```
[ ]: # Iterating over batches of examples:
     #
     # Note that the collat_fn returns input_ids and tag_ids only, so you
     # need to manually convert them back to strings.
     # Unknown words have been mapped to unknown word ids

     batch = next(iter(train_iter))
     text_ids = batch['input_ids']
     example_text = text_ids[0]
     print (f"Size of first text batch: {text_ids.size()}")
     print (f"First sentence in batch: {example_text}")
     print (f"Mapped back to string: {hf_text_tokenizer.decode(example_text)}")

     print ('-'*20)

     tag_ids = batch['tag_ids']
     example_tags = tag_ids[0]
     print (f"Size of tag batch: {tag_ids.size()}")
     print (f"First sentence in batch: {example_tags}")
     print (f"Mapped back to string: {hf_tag_tokenizer.decode(example_tags,␣
      ↪clean_up_tokenization_spaces=False)}")
```

Given the tokenized tags of an unpunctuated sequence of words, we can easily restore the punctuation:

```
[ ]: def restore_punctuation(word_ids, tag_ids):
       words = hf_text_tokenizer.convert_ids_to_tokens(word_ids)
       tags = hf_tag_tokenizer.convert_ids_to_tokens(tag_ids)
       words_with_punc = []
       for word, tag in zip(words, tags):
         words_with_punc.append(word)
         if tag != 'O':
           words_with_punc.append(tag)
       return ' '.join(words_with_punc)
```

```
[ ]: print(restore_punctuation(example['input_ids'], example['tag_ids']))
```

2 Majority Labeling

Recall from our previous lab that a naive baseline is choosing the majority label for each word in the sequence, where the majority label depends on the word. We've provided an implementation of this baseline for you. The performance of this model should give you a sense of how difficult the punctuation restoration task is.

[ ]:
```
class MajorityTagger():
  def __init__(self):
    """Initializer.
    """
    self.most_common_label_given_word = {}

  def train_all(self, train_iter):
    """Finds the majority label for each word in the training set.
    """
    train_counts_given_word = {}
    for batch in train_iter:
      for example_input_ids, example_tag_ids in zip(batch['input_ids'],␣
↪batch['tag_ids']):
        for word_id, tag_id in zip(example_input_ids, example_tag_ids):
          if word_id not in train_counts_given_word:
            train_counts_given_word[word_id.item()] = Counter([])
          train_counts_given_word[word_id.item()].update([tag_id.item()])

    for word_id in train_counts_given_word:
      # Find the most common
      most_common_label = train_counts_given_word[word_id].most_common(1)[0][0]
      self.most_common_label_given_word[word_id] = most_common_label

  def predict_all(self, test_iter):
    """Predicts labels for each example in test_iter.
       Returns a list of list of strings. The order should be the same as
       in `test_iter.dataset` (or equivalently `test_iter`).
    """
    predictions = []
    for batch in test_iter:
      batch_predictions = []
      for example_input_ids in batch['input_ids']:
        example_tag_ids_pred = []
        for word_id in example_input_ids:
          tag_id_pred = self.most_common_label_given_word[word_id.item()]
          example_tag_ids_pred.append(tag_id_pred)
        batch_predictions.append(example_tag_ids_pred)
      predictions.append(batch_predictions)
    return predictions # batch list -> example list -> tag list

  def evaluate(self, test_iter):
    """Evaluates the overall accuracy, and the precision and recall of comma.
    """
    correct = 0
```

```
    total = 0
    true_positive_comma = 0
    predicted_positive_comma = 0
    total_positive_comma = 0
    comma_id = tag_vocab[',']

    # Get predictions
    predictions = self.predict_all(test_iter)
    assert len(predictions) == len(test_iter)

    for batch_tag_pred, batch in zip(predictions, test_iter):
      for tag_ids_pred, example_tag_ids in zip(batch_tag_pred, batch['tag_ids']):
        assert len(tag_ids_pred) == len(example_tag_ids)
        for tag_id_pred, tag_id in zip(tag_ids_pred, example_tag_ids):
          tag_id = tag_id.item()
          total += 1
          if tag_id_pred == tag_id:
            correct += 1
          if tag_id_pred == comma_id:
            predicted_positive_comma += 1 # predicted positive
          if tag_id == comma_id:
            total_positive_comma += 1     # gold label positive
          if tag_id_pred == comma_id and tag_id == comma_id:
            true_positive_comma += 1      # true positive
    precision_comma = true_positive_comma / predicted_positive_comma
    recall_comma = true_positive_comma / total_positive_comma
    F1_comma = 2. / (1./precision_comma + 1./recall_comma)
    return correct/total, precision_comma, recall_comma, F1_comma
```

Now, we can train our baseline on training data.

[ ]:
```
maj_tagger = MajorityTagger()
maj_tagger.train_all(train_iter)
```

Let's take a look at an example prediction using this simple baseline.

[ ]:
```
# Get all predictions
predictions = maj_tagger.predict_all(test_iter)

# Pick one example
example_id = 2 # the third example
example = test_data[example_id]
prediction = predictions[example_id][0]

print('Ground truth punctuation:')
print(restore_punctuation(example['input_ids'], example['tag_ids']), '\n')
```

```
print('Predicted punctuation:')
print(restore_punctuation(example['input_ids'], prediction))
```

This baseline model clearly grossly underpunctuates. It predicts the tag to be `O` almost all of the time.

We can quantitatively evaluate the performance of the majority labeling tagger, which establishes a baseline that any reasonable model should outperform.

[ ]:
```
accuracy, precision_comma, recall_comma, F1_comma = maj_tagger.
 ↪evaluate(test_iter)
print (f"Overall Accuracy: {accuracy:.4f}. \n"
       f"Comma: Precision: {precision_comma:.4f}. Recall: {recall_comma:.4f}.␣
 ↪F1: {F1_comma:.4f}")
```

Question: You can see that even though the overall accuracy is pretty high, the F-1 score for commas is very low. Why?

Type your answer here, replacing this text.

3 RNN Sequence Tagging

Now we get to the real point, using an RNN model for sequence tagging. We provide a base class `RNNBaseTagger` below, which implements training and evaluation. Throughout the rest of this lab, you will implement three subclasses of this class, using PyTorch functions at different abstraction levels.

[ ]:
```
class RNNBaseTagger(nn.Module):
    def __init__(self):
        super().__init__()
        self.N = ...    # tag vocab size provided by subclass
        self.Vo = ...   # text vocab size provided by subclass

    def init_parameters(self, init_low=-0.15, init_high=0.15):
        """Initialize the parameters of the model. Initial parameter values are
        chosen from a uniform distribution between a lowand a high limit. We
        usually use larger initial values for smaller models. See
        http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf for a more
        in-depth discussion.
        """
        for p in self.parameters():
            p.data.uniform_(init_low, init_high)

    def forward(self, text_batch):
        """Performs forward computation, returns logits.

        Arguments:
          text_batch: a tensor containing word ids of size (bsz=1, seq_len)
```

```
        Returns:
          logits: a tensor of size (1, seq_len, self.N)
        """
        raise NotImplementedError   # You'll implement this in the subclasses.

    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=5, learning_rate=1e-3):
        # 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_iter):
                # Zero the parameter gradients
                self.zero_grad()

                # Input and target
                words = batch["input_ids"]  # 1, seq_len
                tags = batch["tag_ids"]     # 1, seq_len

                # 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_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
```

```python
            print(
                f"Epoch: {epoch} Loss: {epoch_loss:.4f} "
                f"Validation accuracy: {validation_accuracy:.4f}"
            )

    def predict(self, text_batch):
        """Returns the most likely sequence of tags for a sequence of words in␣
↪`text_batch`.

        Arguments:
          text_batch: a tensor containing word ids of size (1, seq_len)
        Returns:
          tag_batch: a tensor containing tag ids of size (1, seq_len)
        """
        raise NotImplementedError   # You'll implement this in the subclasses.

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

        Arguments:
          iterator
        Returns:
          overall accuracy, and precision, recall, and F1 for comma
        """
        correct = 0
        total = 0
        true_positive_comma = 0
        predicted_positive_comma = 0
        total_positive_comma = 0
        comma_id = tag_vocab[',']
        pad_id = tag_vocab[pad_token]
        for batch in tqdm(iterator):
            words = batch['input_ids']       # 1, seq_len
            tags = batch['tag_ids']          # 1, seq_len
            tags_pred = self.predict(words)  # 1, seq_len
            mask = tags.ne(pad_id)
            cor = (tags == tags_pred)[mask]
            correct += cor.float().sum().item()
            total += mask.float().sum().item()
            predicted_positive_comma += (
                (mask * tags_pred.eq(comma_id)).float().sum().item()
            )
            true_positive_comma += (
                (mask * tags.eq(comma_id) * tags_pred.eq(comma_id)).float().
↪sum().item()
            )
```

```
            total_positive_comma += (mask * tags.eq(comma_id)).float().sum().
↪item()

        precision_comma = true_positive_comma / predicted_positive_comma
        recall_comma = true_positive_comma / total_positive_comma
        F1_comma = 2.0 / (1.0 / precision_comma + 1.0 / recall_comma)
        return correct / total, precision_comma, recall_comma, F1_comma
```

3.1 RNN from scratch

In this part of the lab, you will implement the forward pass of an RNN from scratch. **You should implement the `forward` function from scratch and *not* use `nn.RNN`.** We'll make use of this convenient PyTorch module in the next part.

Recall that

$$h_0 = 0 \tag{1}$$
$$h_t = \sigma(\mathbf{U}x_t + \mathbf{V}h_{t-1} + b_h) \tag{2}$$
$$o_t = \mathbf{W}h_t + b_o \tag{3}$$

where we embed each word and use its embedding as x_t, and we use o_t as the output logits. (Again, the final softmax has been absorbed into the loss function so you don't need to implement that.) Note that we added bias vectors b_h and b_o in this lab since we are training very small models. (In large models, having a bias vector matters a lot less.)

You will need to implement both the `forward` function and the `predict` function.

> Hint: You might find `torch.stack` useful for stacking a list of tensors to form a single tensor. You can also use `torch.mv` or `@` for matrix-vector multiplication, `torch.mm` or `@` for matrix-matrix multiplication.

Warning: Training this and later models takes a little while, likely around three minutes for the full set of epochs. You might want to set the number of epochs to a small number (1?) until your code is running well. You should also feel free to move ahead to the next parts while earlier parts are running.

```
[ ]: class RNNTagger1(RNNBaseTagger):
       def __init__(self, text_tokenizer, tag_tokenizer, embedding_size,␣
     ↪hidden_size):
         super().__init__()
         self.text_tokenizer = text_tokenizer
         self.tag_tokenizer = tag_tokenizer
         self.N = len(self.tag_tokenizer)   # tag vocab size
         self.Vo = len(self.text_tokenizer)  # text vocab size
         self.embedding_size = embedding_size
         self.hidden_size = hidden_size

         # Create essential modules
```

```
    self.word_embeddings = nn.Embedding(self.Vo, embedding_size) # Lookup layer
    self.U = nn.Parameter(torch.Tensor(hidden_size, embedding_size))
    self.V = nn.Parameter(torch.Tensor(hidden_size, hidden_size))
    self.b_h = nn.Parameter(torch.Tensor(hidden_size))
    self.sigma = nn.Tanh() # Nonlinear Layer
    self.W = nn.Parameter(torch.Tensor(self.N, hidden_size))
    self.b_o = nn.Parameter(torch.Tensor(self.N))

    # Create loss function
    pad_id = self.tag_tokenizer.pad_token_id
    self.loss_function = nn.CrossEntropyLoss(reduction='sum',␣
↪ignore_index=pad_id)

    # Initialize parameters
    self.init_parameters()

  def forward(self, text_batch):
    """Performs forward, returns logits.

    Arguments:
      text_batch: a tensor containing word ids of size (1, seq_len)
    Returns:
      logits: a tensor of size (1, seq_len, self.N)
    """
    h0 = torch.zeros(self.hidden_size, device=device)
    word_embeddings = self.word_embeddings(text_batch) # 1, seq_len,␣
↪embedding_size
    seq_len = word_embeddings.size(1)
    #TODO: your code below
    logits = ...
    return logits

  def predict(self, text_batch):
    """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 (1, seq_len)
    Returns:
      tag_batch: a tensor containing tag ids of size (1, seq_len)
    """
    #TODO: your code below
    tag_batch = ...
    return tag_batch
```

```
[ ]: # Instantiate and train classifier
rnn_tagger1 = RNNTagger1(hf_text_tokenizer,
                         hf_tag_tokenizer,
                         embedding_size=32,
                         hidden_size=32).to(device)
rnn_tagger1.train_all(train_iter, val_iter, epochs=5, learning_rate=1e-3)
rnn_tagger1.load_state_dict(rnn_tagger1.best_model)

# Evaluate model performance
train_accuracy1, train_p1, train_r1, train_f1 = rnn_tagger1.evaluate(train_iter)
test_accuracy1, test_p1, test_r1, test_f1 = rnn_tagger1.evaluate(test_iter)
print(f'\nTraining accuracy: {train_accuracy1:.3f}, precision: {train_p1:.3f},␣
↪recall: {train_r1:.3f}, F-1: {train_f1:.3f}\n'
      f'Test accuracy: {test_accuracy1:.3f}, precision: {test_p1:.3f}, recall:␣
↪{test_r1:.3f}, F-1: {test_f1:.3f}')
```

```
[ ]: grader.check("rnn1")
```

Did your model outperform the baseline? Don't be surprised if it doesn't: the model is very small and the dataset is small as well.

3.2 RNN forward using `nn.RNN` and explicit loop through time steps

In this part, you will use `nn.RNN` and `nn.Linear` to implement the forward pass:

$$h_0 = 0 \tag{4}$$
$$h_t = \text{nn.RNN}(x_t, h_{t-1}) \tag{5}$$
$$o_t = \text{nn.Linear}(h_t) \tag{6}$$

You will need to implement both the `forward` function and the `predict` function. You'll use the `nn.RNN` function to implement each time step of the RNN, with an explicit `for` loop to step through the time steps. (In the next part, you'll use a single call to `nn.RNN` to handle the entire process!) For the linear projection from RNN outputs to logits, use `self.hidden2output`.

> Hint: you can reuse your `predict` implementation from before if you wrote it in a general way.

```
[ ]: class RNNTagger2(RNNBaseTagger):
       def __init__(self, text_tokenizer, tag_tokenizer, embedding_size,␣
     ↪hidden_size):
         super().__init__()
         self.text_tokenizer = text_tokenizer
         self.tag_tokenizer = tag_tokenizer

         self.N = len(self.tag_tokenizer)   # tag vocab size
         self.Vo = len(self.text_tokenizer)  # text vocab size
```

```
        self.embedding_size = embedding_size
        self.hidden_size = hidden_size

        # Create essential modules
        self.word_embeddings = nn.Embedding(self.Vo, embedding_size) # Lookup layer
        self.rnn = nn.RNN(input_size=embedding_size, hidden_size=hidden_size,␣
↪batch_first=True)
        self.hidden2output = nn.Linear(hidden_size, self.N)

        # Create loss function
        pad_id = self.tag_tokenizer.pad_token_id
        self.loss_function = nn.CrossEntropyLoss(reduction='sum',␣
↪ignore_index=pad_id)

        # Initialize parameters
        self.init_parameters()

    def forward(self, text_batch):
        """Performs forward, returns logits.

        Arguments:
          text_batch: a tensor containing word ids of size (1, seq_len)
        Returns:
          logits: a tensor of size (1, seq_len, self.N)
        """
        # h0 is of shape (num_layers * num_directions, batch, hidden_size),
        # which is (1, 1, hidden_size)
        h0 = torch.zeros(1, 1, self.hidden_size, device=device)
        #TODO: your code below, using an *explicit for-loop*
        logits = ...
        return logits

    def predict(self, text_batch):
        """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 (1, seq_len)
        Returns:
          tag_batch: a tensor containing tag ids of size (1, seq_len)
        """
        #TODO: your code below
        tag_batch = ...
        return tag_batch
```

```
[ ]: # Instantiate and train classifier
rnn_tagger2 = RNNTagger2(hf_text_tokenizer,
                         hf_tag_tokenizer,
                         embedding_size=32,
                         hidden_size=32).to(device)
rnn_tagger2.train_all(train_iter,
                      val_iter,
                      epochs=5,
                      learning_rate=1e-3)
rnn_tagger2.load_state_dict(rnn_tagger2.best_model)

# Evaluate model performance
train_accuracy2, train_p2, train_r2, train_f2 = rnn_tagger2.evaluate(train_iter)
test_accuracy2, test_p2, test_r2, test_f2 = rnn_tagger2.evaluate(test_iter)
print(f'\nTraining accuracy: {train_accuracy2:.3f}, precision: {train_p2:.3f},␣
 ↪recall: {train_r2:.3f}, F-1: {train_f2:.3f}\n'
      f'Test accuracy: {test_accuracy2:.3f}, precision: {test_p2:.3f}, recall:␣
 ↪{test_r2:.3f}, F-1: {test_f2:.3f}')
```

```
[ ]: grader.check("rnn2")
```

3.3 RNN forward using bidirectional `nn.RNN`

Instead of using a `for` loop, we can directly feed the entire sequence to `nn.RNN`:

$$
\begin{aligned}
h_0 &= 0 && (7)\\
H &= \text{nn.RNN}(X, h_0) && (8)\\
O &= \text{nn.Linear}(H) && (9)
\end{aligned}
$$

where X is the concatenation of $x_1, \cdots, x_T$, H is the concatenation of $h_1, \cdots, h_T$, and O is the concatenation of $o_1, \cdots, o_T$.

By using this formulation, our code becomes more efficient, since `nn.RNN` is highly optimized. Besides, we can use bi-directional RNNs by simply passing `bidirectional=True` to the RNN constructor.

The difference between a bidirectional RNN and a unidirectional RNN is that bidirectional RNNs have an additional RNN cell running in the reverse direction:

$$
\begin{aligned}
h'_{T+1} &= 0 && (10)\\
h'_t &= \sigma(\mathbf{U}'x'_t + \mathbf{V}'h'_{t+1} + b'_h) && (11)\\
& && (12)
\end{aligned}
$$

To get the output at step t, a bidirectional RNN simply concatenates h_t and h'_t and projects to produce outputs. The benefit of a bidirectional RNN is that the output at step t takes into account not only words $x_1, \cdots, x_t$, but also $x_{t+1}, \cdots, x_T$.

Implement `forward` and `predict` functions below, using a bidirectional RNN.

```
[ ]: class RNNTagger3(RNNBaseTagger):
    def __init__(
        self, text_tokenizer, tag_tokenizer, embedding_size, hidden_size
    ):
        super().__init__()
        self.text_tokenizer = text_tokenizer
        self.tag_tokenizer = tag_tokenizer

        self.N = len(self.tag_tokenizer)    # tag vocab size
        self.Vo = len(self.text_tokenizer)  # text vocab size
        self.embedding_size = embedding_size
        self.hidden_size = hidden_size

        # Create essential modules
        self.word_embeddings = nn.Embedding(self.Vo, embedding_size)  # Lookup␣
↪layer
        self.rnn = nn.RNN(
            input_size=embedding_size,
            hidden_size=hidden_size,
            batch_first=True,
            bidirectional=True,
        )
        self.hidden2output = nn.Linear(
            hidden_size * 2, self.N
        )  # *2 due to using bi-rnn

        # Create loss function
        pad_id = self.tag_tokenizer.pad_token_id
        self.loss_function = nn.CrossEntropyLoss(reduction="sum",␣
↪ignore_index=pad_id)

        # Initialize parameters
        self.init_parameters()

    def forward(self, text_batch):
        """Performs forward, returns logits.

        Arguments:
          text_batch: a tensor containing word ids of size (1, seq_len)
        Returns:
          logits: a tensor of size (1, seq_len, self.N)
        """
        hidden = None  # equivalent to setting hidden to a zero vector
        # TODO: your code below, without using any for-loops
        logits = ...
```

```
        return logits

    def predict(self, text_batch):
        """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 (1, seq_len)
        Returns:
          tag_batch: a tensor containing tag ids of size (1, seq_len)
        """
        # TODO: your code below
        tag_batch = ...
        return tag_batch
```

[ ]:
```
# Instantiate and train classifier
rnn_tagger3 = RNNTagger3(hf_text_tokenizer,
                         hf_tag_tokenizer,
                         embedding_size=32,
                         hidden_size=32).to(device)
rnn_tagger3.train_all(train_iter, val_iter, epochs=5, learning_rate=1e-3)
rnn_tagger3.load_state_dict(rnn_tagger3.best_model)

# Evaluate model performance
train_accuracy3, train_p3, train_r3, train_f3 = rnn_tagger3.evaluate(train_iter)
test_accuracy3, test_p3, test_r3, test_f3 = rnn_tagger3.evaluate(test_iter)
print(f'\nTraining accuracy: {train_accuracy3:.3f}, precision: {train_p3:.3f},␣
↪recall: {train_r3:.3f}, F-1: {train_f3:.3f}\n'
      f'Test accuracy: {test_accuracy3:.3f}, precision: {test_p3:.3f}, recall:␣
↪{test_r3:.3f}, F-1: {test_f3:.3f}')
```

[ ]:
```
grader.check("birnn")
```

Let's see what our model predicts for the example we used before.

[ ]:
```
# Pick one example
example_id = 2 # the third example
example = test_data[example_id]

# Process strings to word ids
text_tensor = torch.LongTensor([example['input_ids']]).to(device)

# Predict
prediction_tensor = rnn_tagger3.predict(text_tensor)[0]

print ('Ground truth punctuation:')
print(restore_punctuation(example['input_ids'], example['tag_ids']))
```

```
print ('Predicted punctuation:')
print(restore_punctuation(example['input_ids'], prediction_tensor))
```

Question: Did your bidirectional RNN reach a higher F-1 score than unidirectional RNNs? Why?

Type your answer here, replacing this text.

4 Lab debrief

Question: We're interested in any thoughts you have about this lab so that we can improve this lab for later years, and to inform later labs for this year. Please list any issues that arose or comments you have to improve the lab. Useful things to comment on include the following, but you're not restricted to these:

- Was the lab too long or too short?
- Were the readings appropriate for the lab?
- Was it clear (at least after you completed the lab) what the points of the exercises were?
- Are there additions or changes you think would make the lab better?

Type your answer here, replacing this text.

End of lab 2-5

To double-check your work, the cell below will rerun all of the autograder tests.

[ ]:
```
grader.check_all()
```