project2_sequence

October 6, 2020

[ ]:
```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/cs187-2020/project2.git .tmp
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

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

1 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*, where the tokens in a text are to be labeled with their part of speech (noun, verb, preposition, etc.).

In this segment, you'll 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. 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

`BOS` and `EOS` are special tokens to indicate the beginning and end of the sentence. 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 a recurrent neural network (RNN). By the end of this homework, you should have grasped the pros and cons of both 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. (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.

1.2 Setup

[ ]:
```
import copy
import math
import random

from tqdm import tqdm

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

import matplotlib.pyplot as plt

# 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)
```

1.2.1 Load Data

First, we download the ATIS dataset.

[ ]:
```
!wget -nv -N -P data https://raw.githubusercontent.com/nlp-course/data/master/
↪ATIS/atis.train.txt
!wget -nv -N -P data https://raw.githubusercontent.com/nlp-course/data/master/
↪ATIS/atis.dev.txt
!wget -nv -N -P data https://raw.githubusercontent.com/nlp-course/data/master/
↪ATIS/atis.test.txt
```

1.2.2 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 token `<unk>`.

[ ]:
```
MIN_FREQ = 3

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

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)}")
```

Note that we passed in `init_token="<bos>"` for both fields, which essentially prepends the sequence of words and tags with `<bos>`. This relieves us from estimating the initial distribution of states in HMMs, since we always start from `<bos>`.

[ ]:
```
initial_state_str = TAG.init_token
initial_state_id = TAG.vocab.stoi[TAG.init_token]
print (f"Initial state: {initial_state_str}")
print (f"Initial state id: {initial_state_id}")
```

Now, we can iterate over the dataset using `torchtext`'s iterator. To simplify RNN implementation we use ***batch size 1*** throughout this project.

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

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

batch = next(iter(train_iter))

print (f"First batch of text: {batch.text[0]}")
print (f"Converted back to string: {[TEXT.vocab.itos[i] for i in batch.
 ↪text[0]]}")
```

```
print (f"First batch of tags: {batch.tag}")
print (f"Converted back to string: {[TAG.vocab.itos[i] for i in batch.tag[0]]}")
```

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

1.3 HMM for Sequence Labeling

1.3.1 Notation

First, let's start with some notations. 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.

1.3.2 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,

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

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

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

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.

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.

1.3.3 Sequence labeling with a trained HMM

Now that we're able to train an HMM by estimating the transition matrix A and the emission matrix B, we 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.

[ ]:
```
#TODO
class HMMTagger():
```

```python
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 = tag.vocab.stoi[tag.init_token]

def train_A(self, iterator, delta):
  """Stores A for training dataset `iterator ` and add-`delta` smoothing."""
  #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.
  "your code here"
  A = torch.zeros(self.N, self.N, device=device)
  return A

def train_B(self, iterator, delta):
  """Stores B for training dataset `iterator ` and add-`delta` smoothing."""
  #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.
  "your code here"
  B = torch.zeros(self.N, self.V, device=device)
  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`."""
  #TODO: Add your solution from Goal 1 (b) here.
  #      The returned value should be a list of tag ids.
  "your code here"
  bestpath = []
  return bestpath

def evaluate(self, iterator):
  """Returns the model's performance on a given dataset `iterator`."""
  correct = 0
  total = 0
  for batch in tqdm(iterator):
    words = batch.text[0]
    tags = batch.tag[0]
    tags_pred = self.predict(words)
    for tag_gold, tag_pred in zip(tags, tags_pred):
```

```
        total += 1
        if tag_pred == tag_gold:
          correct += 1
    return correct/total
```

[ ]:
```
#Solution
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 = tag.vocab.stoi[tag.init_token]

  def train_A(self, iterator, delta):
    """Stores A for training dataset `iterator ` and add-`delta` smoothing."""
    # Initialize A
    A = torch.zeros(self.N, self.N, device=device)
    A.fill_(delta)
    # Count A[i][j]: the number of times state j follows state i
    for batch in iterator:
      assert batch.tag.size(0) == 1, 'this implementation only considers batch␣
↪size 1'
      tags = batch.tag[0] # length
      for state, next_state in zip(tags[:-1], tags[1:]):
        A[state][next_state] += 1
    # Normalize A to get probabilities
    A = A / A.sum(-1, keepdim=True)
    return A

  def train_B(self, iterator, delta):
    """Stores B for training dataset `iterator ` and add-`delta` smoothing."""
    # Initialize B
    B = torch.zeros(self.N, self.V, device=device)
    B.fill_(delta)
    # Count B[i][j]: the number of times state i produces word j
    for batch in train_iter:
      assert batch.tag.size(0) == 1, 'this implementation only considers batch␣
↪size 1'
      tags = batch.tag[0] # length
      words = batch.text[0] # length
      for state, word in zip(tags, words):
        B[state][word] += 1

    # Normalize B to get probabilities
    B = B / B.sum(-1).unsqueeze(-1)
    return B
```

```python
  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`."""
    # See Jurafsky & Martin, section 8.4.5
    T = words.size(0) # sequence length
    viterbi = torch.zeros(self.N, T, device=device) # stores partial max for each state
    backpointers = torch.zeros(self.N, T, device=device).long()

    # Initialization step, always start from <bos>
    viterbi[:, 0] = -float('inf')
    viterbi[self.initial_state, 0] = 0

    # Recursion step
    for t in range(1, T):
      word = words[t]
      viterbi[:, t], backpointers[:, t] = (
                      viterbi[:, t-1].unsqueeze(-1) # N, 1
                      + self.log_A                  # N, N
                      + self.log_B[:, word].unsqueeze(0) # 1, N
                     ).max(0) # max returns values, indices

    # Termination step
    bestpathlogprob, bestpathpointer = viterbi[:, T-1].max(0)

    # Follow `backpointers` back in time
    bestpath_reverse = [bestpathpointer,]
    for t in range(T-1, 0, -1):
      bestpathpointer = backpointers[bestpathpointer, t]
      bestpath_reverse.append(bestpathpointer)
    bestpath = reversed(bestpath_reverse)
    return bestpath

  def evaluate(self, iterator):
    """Returns the model's performance on a given dataset `iterator`."""
    correct = 0
    total = 0
    for batch in tqdm(iterator):
      words = batch.text[0]
      tags = batch.tag[0]
      tags_pred = self.predict(words)
      for tag_gold, tag_pred in zip(tags, tags_pred):
```

```
        total += 1
        if tag_pred == tag_gold:
          correct += 1
    return correct/total
```

Putting everything together, you should expect a correct implementation to reach **90% 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):.3f}\n'
      f'Test accuracy:     {hmm_tagger.evaluate(test_iter):.3f}')
```

1.4 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 o_t based on the input x_t and the hidden state of the previous cell h_{t-1}, according to the following equations. (Here, the o_t are the pre-softmax output logits.)

$$
\begin{aligned}
h_t &= Ux_t + Vh_{t-1} + b_h \\
o_t &= Wh_t + b_y
\end{aligned}
$$

The parameters here are the elements of the matrices U, V, and W, and the bias terms b_h and b_y. 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.

Goal 2 (a): RNN training Implement the forward pass of the RNN tagger and the loss function using the starter code below. The training and optimization code is already provided. You will be adding the below three methods:

1. `__init__`: an initializer that takes two `torchtext` fields providing descriptions of the text and tag aspects of examples, as well as an `embedding_size` specifying the size of word embeddings and a `hidden_size` specifying the size of hidden states.
2. `forward`: performs one step of RNN forward computation with current word `word` and previous hidden state `hidden`. This function is expected to return `output` which stores logits,

and the current hidden state `hidden`. Since we only consider batch size 1, `word` is a tensor with 1 element, and `hidden` is a vector of size `hidden_size`.

3. `compute_loss`: computes loss by comparing `output` returned by `forward` to `ground_truth` which stores the true tag id. `output` is a vector of size N (recall that $N = |Q|$), and `ground_truth` is a tensor with a single element. Note that the criterion functions in `torch` expect batched outputs, so you might need to use `output.unsqueeze(0)` to convert it from a 1-D vector to a 2-D matrix (with a single row).

   Hint: You might find `nn.Linear`, `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.

Goal 2 (b) RNN decoding Implement the method `predict` to tag a sequence of `words`.

[ ]:
```
#TODO
class RNNTagger(nn.Module):
  def __init__(self, text, tag, embedding_size, hidden_size):
    super().__init__()
    self.text = text
    self.tag = tag
    # Keep the vocabulary sizes available
    self.N = len(tag.vocab.itos) # |Q|
    self.V = len(text.vocab.itos)  # vocab_size
    self.embedding_size = embedding_size
    self.hidden_size = hidden_size
    #TODO: implement below, create essential modules and loss function
    raise NotImplementedError


  def forward(self, word, hidden):
    """Performs one step of RNN forward, returns current output and hidden state.

    Arguments:
      hidden the hidden state from the previous cell,
      word   the current input word id
    Returns:
      a pair of the current output and the current hidden state
    """
    # TODO: implement below
    return output, hidden

  def compute_loss(self, output, ground_truth):
    """Computes loss."""
    # TODO: implement below
```

```python
    return loss

def train_all(self, train_iter, val_iter, epochs=3, learning_rate=0.001):
    # 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.text[0] # vector with T word ids
        tags = batch.tag[0]   # vector with T tags

        # Run forward pass and compute loss along the way.
        hidden = torch.zeros(self.hidden_size, device=device)
        loss = 0
        for word, tag in zip(words, tags):
          output, hidden = self(word, hidden)
          loss = loss + self.compute_loss(output, tag)

        # Perform backpropagation
        loss.backward()
        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
      print (f'Epoch: {epoch} Loss: {epoch_loss:.4f} '
             f'Validation accuracy: {validation_accuracy:.4f}')

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

```
    #TODO: implement below
    tags = []
    return tags

  def evaluate(self, iterator):
    """Returns the model's performance on a given dataset `iterator`."""
    correct = 0
    total = 0
    for batch in tqdm(iterator):
      words = batch.text[0]
      tags = batch.tag[0]
      tags_pred = self.predict(words)
      for tag_gold, tag_pred in zip(tags, tags_pred):
        total += 1
        if tag_pred == tag_gold:
          correct += 1
    return correct/total
```

[ ]:
```
#Solution
class RNNTagger(nn.Module):
  def __init__(self, text, tag, embedding_size, hidden_size):
    super().__init__()
    self.text = text
    self.tag = tag
    # Keep the vocabulary sizes available
    self.N = len(tag.vocab.itos) # |Q|
    self.V = len(text.vocab.itos)  # vocab_size
    self.embedding_size = embedding_size
    self.hidden_size = hidden_size

    # Create essential modules
    self.word_embeddings = nn.Embedding(self.V, embedding_size) # Lookup layer
    self.input2hidden = nn.Linear(embedding_size, hidden_size)  # U in the above␣
↪illustration
    self.hidden2hidden = nn.Linear(hidden_size, hidden_size)    # V
    self.hidden2output = nn.Linear(hidden_size, self.N)         # W

    # Create loss function
    self.loss_function = nn.CrossEntropyLoss(reduction='sum')


  def forward(self, word, hidden):
    """Performs one step of RNN forward, returns current output and hidden state.

    Arguments:
      hidden the hidden state from the previous cell,
      word   the current input word id
```

```
      Returns:
        a pair of the current output and the current hidden state
      """
      input = self.word_embeddings(word)
      hidden = self.input2hidden(input) + self.hidden2hidden(hidden)
      output = self.hidden2output(hidden)
      return output, hidden

    def compute_loss(self, output, ground_truth):
      return self.loss_function(output.view(1, -1), ground_truth.view(-1))

    def train_all(self, train_iter, val_iter, epochs=3, learning_rate=0.001):
      # 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.text[0] # vector with T word ids
          tags = batch.tag[0]   # vector with T tags

          # Run forward pass and compute loss along the way.
          hidden = torch.zeros(self.hidden_size, device=device)
          loss = 0
          for word, tag in zip(words, tags):
            output, hidden = self(word, hidden)
            loss = loss + self.compute_loss(output, tag)

          # Perform backpropagation
          loss.backward()
          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
      print (f'Epoch: {epoch} Loss: {epoch_loss:.4f} '
             f'Validation accuracy: {validation_accuracy:.4f}')

  def predict(self, words):
    """Returns the most likely sequence of tags for a sequence of `words`."""
    hidden = torch.zeros(self.hidden_size, device=device)
    loss = 0
    tags = []
    for word in words:
      output, hidden = self.forward(word, hidden)
      _, tag = torch.max(output, 0)
      tags.append(tag)
    return tags

  def evaluate(self, iterator):
    """Returns the model's performance on a given dataset `iterator`."""
    correct = 0
    total = 0
    for batch in tqdm(iterator):
      words = batch.text[0]
      tags = batch.tag[0]
      tags_pred = self.predict(words)
      for tag_gold, tag_pred in zip(tags, tags_pred):
        total += 1
        if tag_pred == tag_gold:
          correct += 1
    return correct/total
```

Run the below cell to train an RNN, and evaluate it. A proper implementation should reach **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)
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}')
```

1.5 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`.

Goal 3 (a) Use LSTM instead of RNN Use LSTM instead of RNN, implement the `LSTMTagger` class with the below starter code. What are the expected input arguments of `nn.LSTM`?

Goal 3 (b) Use LSTM for decoding Implement the method `predict` to tag a sequence of `words`.

[ ]:
```
#TODO
class LSTMTagger(nn.Module):
  def __init__(self, text, tag, embedding_size, hidden_size):
    super().__init__()
    self.text = text
    self.tag = tag
    # Keep the vocabulary sizes available
    self.N = len(tag.vocab.itos) # |Q|
    self.V = len(text.vocab.itos)  # vocab_size
    self.embedding_size = embedding_size
    self.hidden_size = hidden_size

    #TODO: implement below, create essential modules and loss function
    raise NotImplementedError

  def forward(self, word, hidden):
    """Performs one step of RNN forward, returns current output and hidden."""
    # TODO: implement below
    return output, hidden

  def compute_loss(self, output, ground_truth):
    # TODO: implement below
    return loss

  def train_all(self, train_iter, val_iter, epochs=3, learning_rate=0.001):
    # 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
```

```python
    running_loss = 0.0
    for batch in tqdm(train_iter):
      # Zero the parameter gradients
      self.zero_grad()

      # Input and target
      words = batch.text[0] # vector with T word ids
      tags = batch.tag[0]   # vector with T tags

      # Run forward pass and compute loss along the way.
      hidden = torch.zeros(1, 1, self.hidden_size, device=device)
      hidden = (hidden, hidden) # LSTM hidden is a tuple of two tensors
      loss = 0
      for word, tag in zip(words, tags):
        output, hidden = self(word, hidden)
        loss = loss + self.compute_loss(output, tag)

      # Perform backpropagation
      loss.backward()
      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
    print (f'Epoch: {epoch} Loss: {epoch_loss:.4f} '
           f'Validation accuracy: {validation_accuracy:.4f}')

def predict(self, words):
  """Returns the most likely sequence of tags for a sequence of `words`."""
  #TODO: implement below
  tags = []
  return tags

def evaluate(self, iterator):
  """Returns the model's performance on a given dataset `iterator`."""
  correct = 0
  total = 0
  for batch in tqdm(iterator):
    words = batch.text[0]
    tags = batch.tag[0]
```

```
      tags_pred = self.predict(words)
      for tag_gold, tag_pred in zip(tags, tags_pred):
        total += 1
        if tag_pred == tag_gold:
          correct += 1
    return correct/total
```

[ ]:
```
#Solution
class LSTMTagger(nn.Module):
  def __init__(self, text, tag, embedding_size, hidden_size):
    super().__init__()
    self.text = text
    self.tag = tag
    # Keep the vocabulary sizes available
    self.N = len(tag.vocab.itos) # |Q|
    self.V = len(text.vocab.itos)  # vocab_size
    self.embedding_size = embedding_size
    self.hidden_size = hidden_size

    # Create essential modules
    self.word_embeddings = nn.Embedding(self.V, embedding_size) # Lookup layer
    self.hidden2output = nn.Linear(hidden_size, self.N)         # W
    # nn.LSTM takes word embeddings as inputs, and outputs hidden states
    # with dimensionality hidden_size.
    self.lstm = nn.LSTM(embedding_size, hidden_size)

    # Create loss function
    self.loss_function = nn.CrossEntropyLoss(reduction='sum')


  def forward(self, word, hidden):
    """Performs one step of RNN forward, returns current output and hidden."""
    input = self.word_embeddings(word)
    input_reshaped = input.view(1, 1, -1) # LSTM expect (seq_len, bsz,␣
↪embedding_size)
    output, hidden = self.lstm(input_reshaped, hidden)
    output = self.hidden2output(output.view(-1))
    return output, hidden

  def compute_loss(self, output, ground_truth):
    return self.loss_function(output.view(1, -1), ground_truth.view(-1))

  def train_all(self, train_iter, val_iter, epochs=3, learning_rate=0.001):
    # 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.text[0] # vector with T word ids
        tags = batch.tag[0]   # vector with T tags

        # Run forward pass and compute loss along the way.
        hidden = torch.zeros(1, 1, self.hidden_size, device=device)
        hidden = (hidden, hidden) # LSTM hidden is a tuple of two tensors
        loss = 0
        for word, tag in zip(words, tags):
          output, hidden = self(word, hidden)
          loss = loss + self.compute_loss(output, tag)

        # Perform backpropagation
        loss.backward()
        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
      print (f'Epoch: {epoch} Loss: {epoch_loss:.4f} '
             f'Validation accuracy: {validation_accuracy:.4f}')

  def predict(self, words):
    """Returns the most likely sequence of tags for a sequence of `words`."""
    hidden = torch.zeros(1, 1, self.hidden_size, device=device)
    hidden = (hidden, hidden)
    loss = 0
    tags = []
    for word in words:
      output, hidden = self.forward(word, hidden)
```

```
      _, tag = torch.max(output, 0)
      tags.append(tag)
    return tags

  def evaluate(self, iterator):
    """Returns the model's performance on a given dataset `iterator`."""
    correct = 0
    total = 0
    for batch in tqdm(iterator):
      words = batch.text[0]
      tags = batch.tag[0]
      tags_pred = self.predict(words)
      for tag_gold, tag_pred in zip(tags, tags_pred):
        total += 1
        if tag_pred == tag_gold:
          correct += 1
    return correct/total
```

Run the below cell to train an LSTM, and evaluate it. A proper implementation should reach **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)
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}')
```

1.6 (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.

The below code 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)
```

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

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

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

[ ]:
```python
#Solution
test_size = 50
accuracy_hmm = []
for ratio in [0.01, 0.1, 0.5, 1.0]:
  random.seed(seed)
  torch.manual_seed(seed)
  train, val, test = tt.datasets.SequenceTaggingDataset.splits(
            fields=fields,
            path='./data/',
            train='atis.train.txt',
            validation='atis.dev.txt',
            test='atis.test.txt')

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

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

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

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

  # Evaluate model performance
```

```
  accuracy_hmm.append(hmm_tagger.evaluate(test_iter))
```

[ ]:
```
#Solution
test_size = 50
accuracy_rnn = []
for ratio in [0.01, 0.1, 0.5, 1.0]:
  random.seed(seed)
  torch.manual_seed(seed)
  train, val, test = tt.datasets.SequenceTaggingDataset.splits(
              fields=fields, path='./data/', train='atis.train.txt',␣
 ↪validation='atis.dev.txt',
              test='atis.test.txt')

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

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

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

  # 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)
  rnn_tagger.load_state_dict(rnn_tagger.best_model)

  accuracy_rnn.append(rnn_tagger.evaluate(test_iter))
```

[ ]:
```
#Solution
plt.plot([0.01, 0.1, 0.5, 1.0], accuracy_rnn, 'r-', label='RNN')
plt.plot([0.01, 0.1, 0.5, 1.0], accuracy_hmm, 'b-', label='HMM')
plt.legend()
plt.show()
```

Surprisingly, RNN even outperforms HMM under low resource scenarios, which is typically not the case on other tasks.

HMM:

Pros: * Training is fast (counting) * Interpretable

Cons: * Worse performance compared to end-to-end neural approaches

RNN:

Pros: * Better performance than HMM with enough data and enough model capacity as shown in the above experiments.

Cons: * Not interpretable * Training is not as efficient