Implementing RNNs for sequence modeling in PyTorch

Dr. Huiping Cao

Predicting the sentiment of IMDb movie reviews

- The **sentiment analysis** is concerned with analyzing the expressed opinion of a sentence or a text document.
- We will implement a **multilayer RNN** for sentiment analysis using a **many-to-one architecture.**

Preparing the movie review data

- Import the necessary modules and read the data from torchtext as follows:
  - Version issue of torchtext (torch 1.9.0, torchtext 0.10.0 works on google Colab; Other version may not work)

```
from torchtext.datasets import IMDB

# Step 1: load and create the datasets
train_dataset = IMDB(split='train')
test_dataset = IMDB(split='test')
```

IMDB dataset

- Each set has 25,000 samples.
- Each sample of the datasets consists of two elements
  - the sentiment label representing the target label we want to predict
    - neg refers to negative sentiment
    - pos refers to positive sentiment,
  - the movie review text (the input features).
    - The text component of these movie reviews is sequences of words
- The RNN model classifies each sequence as a positive (1) or negative (0) review.

Preprocess steps

- 1. Split the training dataset into separate training and validation partitions.
- 2. Identify the **unique words** in the training dataset
- 3. Map each **unique word to a unique integer** and encode the review text into encoded integers (an index of each unique word)
- 4. Divide the dataset into **mini-batches** as input to the model

Preprocess step 1: creating a training and validation partition from the train_dataset

- The original training dataset contains 25,000 examples.
- 20,000 examples are randomly chosen for training, and 5,000 for validation.

```
import torch
import torch.nn as nn
from torch.utils.data.dataset import random_split

torch.manual_seed(1)
train_dataset, valid_dataset = random_split(list(train_dataset), [20000, 5000])
```

Preprocess step 2: find the unique words (tokens) in the training dataset

- To prepare the data for input to an NN, we need to encode it into numeric values.
- While finding unique tokens is a process for which we can use Python datasets, it can be more efficient to use the **Counter class from the collections package**, which is part of Python's standard library.
  - Instantiate a new Counter object (token_counts) that will collect the unique word frequencies.
  - We are only interested in the set of unique words and won't require the word counts.
- To split the text into words (or tokens), we will reuse the **tokenizer function**.
  - Removes HTML markups as well as punctuation and other non-letter characters

Preprocess step 2

```
## Step 2: find unique tokens (words)
import re
from collections import Counter, OrderedDict

def tokenizer(text):
        text = re.sub('<[^>]*>', '', text)
        emoticons = re.findall(\
                '(?::|;|=)(?:-)?(?:\)|\(|D|P)', text.lower())
        text = re.sub('[\W]+', ' ', text.lower()) +\
                ' '.join(emoticons).replace('-', ’’)
        tokenized = text.split()
        return tokenized

token_counts = Counter()
for label, line in train_dataset:
        tokens = tokenizer(line)
        token_counts.update(tokens)
print('Vocab-size:', len(token_counts))
```

Vocab-size: 69023

Preprocess step 3: map each unique word to a unique integer

- Manually using a Python dictionary
  - the keys are the unique tokens (words) and the value associated with each key is a unique integer.
- The torchtext package already provides a class, Vocab, which we can use to create such a mapping and encode the entire dataset.

Preprocess step 3

- Create a vocab object by passing the ordered dictionary mapping tokens to their corresponding occurrence frequencies
  - the ordered dictionary is the sorted token_counts.
- Prepend two **special tokens** to the vocabulary – the padding and the unknown token
- Test: Convert an example input text into a list of integer values

```
## Step 3: encoding each unique token into integers
from torchtext.vocab import vocab

sorted_by_freq_tuples = sorted(token_counts.items(),
key=lambda x: x[1], reverse=True)

ordered_dict = OrderedDict(sorted_by_freq_tuples)
vocab = vocab(ordered_dict)
vocab.insert_token("<pad>", 0)
vocab.insert_token("<unk>", 1)
vocab.set_default_index(1)

#demonstrate how to use the vocab object
print([vocab[token] for token in ['this', 'is', 'an', 'example']])
```

```
[11, 7, 35, 457]
```

Special tokens: tokens not in the training data

- There might be some tokens in the validation or testing data that are not present in the training data and are thus not included in the mapping.
- If we have *q* tokens (that is, the size of token_counts passed to Vocab, which in this case is 69,023), then all tokens that haven't been seen before will be assigned the integer 1 (a placeholder for the unknown token).
- The index 1 is reserved for unknown words.
- Another reserved value is the integer 0, which serves as a placeholder, a so-called *padding token*.

Preprocess step 3

- Define the text_pipeline function to transform each text in the dataset accordingly
- Define label_pipeline function to convert each label to 1 or 0

```
Step 3-A: define the functions for transformation

text_pipeline = lambda x: [vocab[token] for token in tokenizer(x)]

label_pipeline = lambda x: 1. if x == 'pos' else 0.
```

Step 3-B: wrap the encode and transformation function

- Generate batches of samples using DataLoader
- Pass the data processing pipelines declared previously to the argument collate_fn.
- Wrap the text encoding and label transformation function into the collate_batch function.
- ➔ converted sequences of words into sequences of integers, and labels of pos or neg into 1 or 0.

```
## Step 3-B: wrap the encode and transformation function
def collate_batch(batch):
    label_list, text_list, lengths = [], [], []
    for _label, _text in batch:
        label_list.append(label_pipeline(_label))
        processed_text = torch.tensor(text_pipeline(_text), dtype=torch.int64)
        text_list.append(processed_text)
        lengths.append(processed_text.size(0))
    label_list = torch.tensor(label_list)
    lengths = torch.tensor(lengths)
    padded_text_list = nn.utils.rnn.pad_sequence(text_list, batch_first=True)
    return padded_text_list, label_list, lengths

## Take a small batch
from torch.utils.data import DataLoader

dataloader = DataLoader(train_dataset, batch_size=4, \
        shuffle=False, collate_fn=collate_batch)
```

Make all the sequences in a mini-batch have the same length

- Issue: the sequences currently have different lengths.
- Although, in general, RNNs can handle sequences with different lengths, we still need to make sure that all the sequences in a mini-batch have the same length to store them efficiently in a tensor.
- PyTorch provides an efficient method, pad_sequence(), which will automatically pad the consecutive elements that are to be combined into a batch with placeholder values (0s) so that all sequences within a batch will have the same shape.
  - Pad-sequence is utilized in the collate_batch function
- Example:
  - Given 4 samples with lengths 165, 86, 218, and 145
  - The functions will combine these four examples into a single batch and use the maximum size of these examples
  - The minibatch's shape is [4, 218], it means that the other three examples (in this batch are padded as much as necessary to match this size.

Preprocess step 4: divide all three datasets into data loaders

- Divide all three datasets into data loaders with a batch size of 32.

```
batch_size = 32

train_dl = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, collate_fn=collate_batch)
valid_dl = DataLoader(valid_dataset, batch_size=batch_size, shuffle=False, collate_fn=collate_batch)
test_dl = DataLoader(test_dataset, batch_size=batch_size, shuffle=False, collate_fn=collate_batch)
```

Optional preprocess step: feature embedding

- **Feature embedding** is an optional but highly recommended preprocessing step that is used to reduce the dimensionality of the word vectors.
- Preprocess steps
  - Generated sequences of the same length. The elements of these sequences were integer numbers that corresponded to the *indices* of unique words.
- These word indices can be converted into input features in several different ways.
- One naive way is to apply one-hot encoding to convert the indices into vectors of zeros and ones.
  - Each word will be mapped to a vector whose size is the number of unique words in the entire dataset.
  - Issue: given that the number of unique words (the size of the vocabulary) can be in the order of 104 – 105, which will also be the number of our input features, a model trained on such features may suffer from the **curse of dimensionality**.

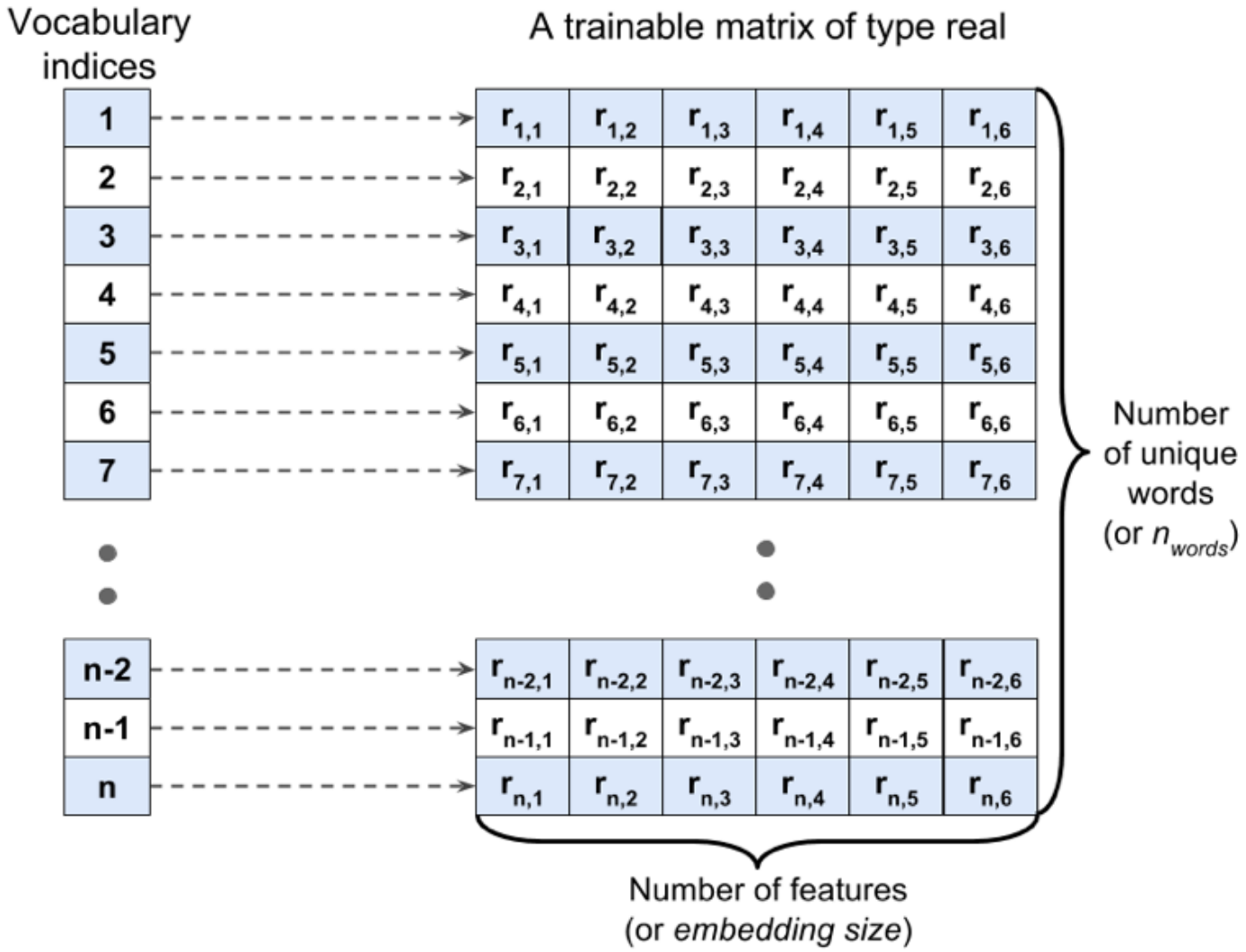
Feature imbedding

- A more elegant approach is to map each word to a vector of a fixed size with real-valued elements (not necessarily integers).
- Use finite-sized vectors to represent an infinite number of real numbers.
  - In theory, we can extract infinite real numbers from a given interval, for example [–1, 1].
- Feature embedding: a feature-learning technique that we can utilize here to automatically learn the salient features to represent the words in our dataset.

Embedding matrix

- Given the number of unique words, n_{words}, we can select the size of the embedding vectors (a.k.a., embedding dimension) to be much smaller than the number of unique words ($embedding\_dim << n_{words}$) to represent the entire vocabulary as input features.
- Given a set of tokens of size $n + 2$, an embedding matrix of size $(n + 2) \times embedding\_dim$ will be created where
  - each row of this matrix represents numeric features associated with a token
  - When an integer index, *i*, is given as input to the embedding, it will look up the corresponding row of the matrix at index *i* and return the numeric features.
- The embedding matrix serves as the input layer to our NN models

Embedding matrix



Creating embedding

- Creating an embedding layer can simply be done using nn.Embedding.
  - num_embeddings: the unique integer values that the model will receive as input
  - padding_idx indicates the token index for padding (here, 0) will not contribute to the gradient updates during training.
- The input to this model (embedding layer) must have rank 2 with the dimensionality *batchsize* × *input_length*
- Example: an input sequence in the mini-batch could be <1, 5, 9, 2>, where each element of this sequence is the index of the unique words.
- The output will have the dimensionality *batchsize* × *input_length* × *embedding_dim.*

```
embedding = nn.Embedding(num_embeddings=10,
embedding_dim=3,padding_idx=0)

# a batch of 2 samples of 4 indices each
text_encoded_input = torch.LongTensor([[1,2,4,5],[4,3,2,0]])
print(embedding(text_encoded_input))
```

```
tensor([[[-0.4651, -0.3203, 2.2408],
         [ 0.3824, -0.3446, -0.3531],
         [-0.0251, -0.5973, -0.2959],
         [ 0.8356, 0.4025, -0.6924]],
        [[-0.0251, -0.5973, -0.2959],
         [ 0.9124, -0.4643, 0.3046],
         [ 0.3824, -0.3446, -0.3531],
         [ 0.0000, 0.0000, 0.0000]]],
           grad_fn=<EmbeddingBackward>)
```

Building an RNN model

- Using the nn.Module class, we can combine the embedding layer, the recurrent layers of the RNN, and the fully connected non-recurrent layers.
- For the recurrent layers, we can use
  - RNN: a regular RNN layer, that is, a fully connected recurrent layer
  - LSTM: a long short-term memory RNN, which is useful for capturing the long-term dependencies
  - GRU: a recurrent layer with a gated recurrent unit

Building an RNN model for the sentiment analysis task

- We will create an RNN model for sentiment analysis
  - an embedding layer producing word embeddings of feature size 20 (embed_dim=20).
  - a recurrent layer of type LSTM
  - a fully connected layer as a hidden layer
  - another fully connected layer as the output layer, which will return a single class-membership probability value via the logistic sigmoid activation as the prediction.

RNN model definition

```
class RNN(nn.Module):
        def __init__(self, vocab_size, embed_dim, rnn_hidden_size, fc_hidden_size):
                super().__init__()
                self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=0)
                self.rnn = nn.LSTM(embed_dim, rnn_hidden_size, batch_first=True)
                self.fc1 = nn.Linear(rnn_hidden_size, fc_hidden_size)
                self.relu = nn.ReLU()
                self.fc2 = nn.Linear(fc_hidden_size, 1)
                self.sigmoid = nn.Sigmoid()

        def forward(self, text, lengths):
                out = self.embedding(text)
                out = nn.utils.rnn.pack_padded_sequence(\
                out, lengths.cpu().numpy(),
                enforce_sorted=False, batch_first=True)
                out, (hidden, cell) = self.rnn(out)
                out = hidden[-1, :, :]
                out = self.fc1(out)
                out = self.relu(out)
                out = self.fc2(out)
                out = self.sigmoid(out)
                return out
```

Initialize the model

- Set model parameters for the given dataset

```
vocab_size = len(vocab)
embed_dim = 20
rnn_hidden_size = 64
fc_hidden_size = 64
torch.manual_seed(1)
model = RNN(vocab_size, embed_dim, rnn_hidden_size, fc_hidden_size)
print(model)
```

```
RNN( (embedding):
        Embedding(69025, 20, padding_idx=0)
        (rnn): LSTM(20, 64, batch_first=True)
        (fc1): Linear(in_features=64, out_features=64, bias=True)
        (relu): ReLU()
        (fc2): Linear(in_features=64, out_features=1, bias=True)
        (sigmoid): Sigmoid() )
```

Define train function

- Develop the train function to train the model on the given dataset for one epoch
- Return the classification accuracy and loss

```
def train(dataloader):
        model.train()
        total_acc, total_loss = 0, 0
        for text_batch, label_batch, lengths in dataloader:
                optimizer.zero_grad()
                pred = model(text_batch, lengths)[:, 0]
                loss = loss_fn(pred, label_batch)
                loss.backward()
                optimizer.step()
                total_acc += ((pred >= 0.5).float() == label_batch).float().sum().item()
                total_loss += loss.item()*label_batch.size(0)
        return total_acc/len(dataloader.dataset), total_loss/len(dataloader.dataset)
```

Define the evaluate function

- The evaluate function to measure the model's performance on a given dataset.

```
def evaluate(dataloader):
        model.eval()
        total_acc, total_loss = 0, 0
        with torch.no_grad():
            for text_batch, label_batch, lengths in dataloader:
                pred = model(text_batch, lengths)[:, 0]
                loss = loss_fn(pred, label_batch)
                total_acc += ((pred>=0.5).float() == label_batch).float().sum().item()
                total_loss += loss.item()*label_batch.size(0)
        return total_acc/len(dataloader.dataset), total_loss/len(dataloader.dataset)
```

Define the loss function and optimazer

- Loss function: For a binary classification with a single class-membership probability output, we use the binary cross-entropy loss (BCELoss)

- Optimizer: Adam optimizer

```
loss_fn = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Train the model

- Train the model for 10 epochs and display the training and validation performances

```
num_epochs = 10
torch.manual_seed(1)
for epoch in range(num_epochs):
        acc_train, loss_train = train(train_dl)
        acc_valid, loss_valid = evaluate(valid_dl)
        print(f'Epoch {epoch} accuracy: {acc_train:.4f}’
        f' val_accuracy: {acc_valid:.4f}')
```

```
Epoch 0 accuracy: 0.5843 val_accuracy: 0.6240
Epoch 1 accuracy: 0.6364 val_accuracy: 0.6870
Epoch 2 accuracy: 0.8020 val_accuracy: 0.8194
Epoch 3 accuracy: 0.8730 val_accuracy: 0.8454
Epoch 4 accuracy: 0.9092 val_accuracy: 0.8598
Epoch 5 accuracy: 0.9347 val_accuracy: 0.8630
Epoch 6 accuracy: 0.9507 val_accuracy: 0.8636
Epoch 7 accuracy: 0.9655 val_accuracy: 0.8654
Epoch 8 accuracy: 0.9765 val_accuracy: 0.8528
Epoch 9 accuracy: 0.9839 val_accuracy: 0.8596
```

Evaluate the model on the test data

- After training this model for 10 epochs, we will evaluate it on the test data:

```
acc_test, _ = evaluate(test_dl)

print(f'test_accuracy: {acc_test:.4f}')
```

```
test_accuracy: 0.8512
```

Bidirectional RNN layer

- We can set the bidirectional configuration of the LSTM to True, which will make the recurrent layer pass through the input sequences from both directions, start to end, as well as in the reverse direction.

- The bidirectional RNN layer makes two passes over each input sequence
  - A forward pass and a reverse or backward pass (note that this is not to be confused with the forward and backward passes in the context of backpropagation)
  - The resulting hidden states of these forward and backward passes are usually concatenated into a single hidden state.

References

- Chapter 15: By Sebastian Raschka , Yuxi (Hayden) Liu , Vahid Mirjalili: Machine Learning with PyTorch and Scikit-Learn, Packt.
- PyTorch RNN: https://pytorch.org/docs/stable/generated/torch.nn.RNN.html