Task_01

October 2, 2022

1 Implementing RNNs

In this task, we will be building a machine learning model to classify sentiment (i.e. detect if a sentence is positive or negative) using PyTorch and TorchText. This will be done on movie reviews, using the IMDb dataset.

1.0.1 Introduction

We'll be using a **recurrent neural network** (RNN) as they are commonly used in analysing sequences. An RNN takes in sequence of words, $X = \{x_1, ..., x_T\}$, one at a time, and produces a *hidden state*, h, for each word. We use the RNN *recurrently* by feeding in the current word x_t as well as the hidden state from the previous word, h_{t-1}, to produce the next hidden state, h_t.

$$h_t = \text{RNN}(x_t, h_{t-1})$$

Once we have our final hidden state, h_T, (from feeding in the last word in the sequence, x_T) we feed it through a linear layer, f, (also known as a fully connected layer), to receive our predicted sentiment, $\hat{y} = f(h_T)$.

Below shows an example sentence, with the RNN predicting zero, which indicates a negative sentiment. The RNN is shown in orange and the linear layer shown in silver. Note that we use the same RNN for every word, i.e. it has the same parameters. The initial hidden state, h_0, is a tensor initialized to all zeros.

Note: some layers and steps have been omitted from the diagram, but these will be explained later.

1.1 Preparing Data

[ ]:
```
!pip install torchtext==0.10.0
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Collecting torchtext==0.10.0
  Downloading torchtext-0.10.0-cp37-cp37m-manylinux1_x86_64.whl (7.6 MB)
     || 7.6 MB 4.7 MB/s
Collecting torch==1.9.0
  Downloading torch-1.9.0-cp37-cp37m-manylinux1_x86_64.whl (831.4 MB)
     || 831.4 MB 2.7 kB/s
```

```
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-
packages (from torchtext==0.10.0) (2.23.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages
(from torchtext==0.10.0) (1.21.6)
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages
(from torchtext==0.10.0) (4.64.1)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.7/dist-packages (from torch==1.9.0->torchtext==0.10.0)
(4.1.1)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in
/usr/local/lib/python3.7/dist-packages (from requests->torchtext==0.10.0)
(1.24.3)
Requirement already satisfied: chardet<4,>=3.0.2 in
/usr/local/lib/python3.7/dist-packages (from requests->torchtext==0.10.0)
(3.0.4)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-
packages (from requests->torchtext==0.10.0) (2.10)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.7/dist-packages (from requests->torchtext==0.10.0)
(2022.6.15)
Installing collected packages: torch, torchtext
  Attempting uninstall: torch
    Found existing installation: torch 1.12.1+cu113
    Uninstalling torch-1.12.1+cu113:
      Successfully uninstalled torch-1.12.1+cu113
  Attempting uninstall: torchtext
    Found existing installation: torchtext 0.13.1
    Uninstalling torchtext-0.13.1:
      Successfully uninstalled torchtext-0.13.1
ERROR: pip's dependency resolver does not currently take into account all
the packages that are installed. This behaviour is the source of the following
dependency conflicts.
torchvision 0.13.1+cu113 requires torch==1.12.1, but you have torch 1.9.0 which
is incompatible.
torchaudio 0.12.1+cu113 requires torch==1.12.1, but you have torch 1.9.0 which
is incompatible.
Successfully installed torch-1.9.0 torchtext-0.10.0
```

[ ]:
```python
import torch
from torchtext.legacy import data

SEED = 1234

torch.manual_seed(SEED)
```

```
torch.backends.cudnn.deterministic = True

TEXT = data.Field(tokenize = 'spacy',
                  tokenizer_language = 'en_core_web_sm')
LABEL = data.LabelField(dtype = torch.float)
```

Another handy feature of TorchText is that it has support for common datasets used in natural language processing (NLP).

The following code automatically downloads the IMDb dataset and splits it into the canonical train/test splits as `torchtext.datasets` objects. It process the data using the `Fields` we have previously defined. The IMDb dataset consists of 50,000 movie reviews, each marked as being a positive or negative review.

[ ]:
```
from torchtext.legacy import datasets

train_data, test_data = datasets.IMDB.splits(TEXT, LABEL)
```

```
downloading aclImdb_v1.tar.gz

aclImdb_v1.tar.gz: 100%|| 84.1M/84.1M [00:02<00:00, 32.9MB/s]
```

We can see how many examples are in each split by checking their length.

[ ]:
```
print(f'Number of training examples: {len(train_data)}')
print(f'Number of testing examples: {len(test_data)}')
```

```
Number of training examples: 25000
Number of testing examples: 25000
```

We can also check an example.

[ ]:
```
print(vars(train_data.examples[0]))
```

```
{'text': ['I', 'love', 'ghost', 'stories', 'in', 'general', ',', 'but', 'I',
'PARTICULARLY', 'LOVE', 'chilly', ',', 'atmospheric', 'and', 'elegantly',
'creepy', 'British', 'period', '-', 'style', 'ghost', 'stories', '.', 'This',
'one', 'qualifies', 'on', 'all', 'counts', '.', 'A', 'naive', 'young', 'lawyer',
'(', '"', 'solicitor', '"', 'in', 'Britspeak', ')', 'is', 'sent', 'to', 'a',
'small', 'village', 'near', 'the', 'seaside', 'to', 'settle', 'an', 'elderly',
',', 'deceased', 'woman', "'s", 'estate', '.', 'It', "'s", 'the', '1920s', ',',
'a', 'time', 'when', 'many', 'middle', '-', 'class', 'Brits', 'go', 'to', 'the',
'seaside', 'on', 'vacation', 'for', '"', 'their', 'health', '.', '"', 'Well',
',', 'guess', 'what', ',', 'there', "'s", 'nothing', '"', 'healthy', '"',
'about', 'the', 'village', 'of', 'Crythin', 'Gifford', ',', 'the', 'creepy',
'site', 'of', 'the', 'elderly', 'woman', "'s", 'hulking', ',', 'brooding',
'Victorian', 'estate', ',', 'which', 'is', 'located', 'on', 'the', 'fringes',
'of', 'a', 'fog', '-', 'swathed', 'salt', 'marsh', '.', 'When', 'the', 'lawyer',
'saves', 'the', 'life', 'of', 'a', 'small', 'girl', '(', 'none', 'of', 'the',
'locals', 'will', 'help', 'the', 'endangered', 'tot', '--', 'you', 'find',
'out', 'why', 'later', 'on', 'in', 'the', 'film', ')', ',', 'he',
```

```
'inadvertently', 'incurs', 'the', 'wrath', 'of', 'a', 'malevolent', 'spirit',
',', 'the', 'woman', 'in', 'black', '.', 'She', 'is', 'no', 'filmy', ',',
'gauzy', 'wraith', ',', 'but', 'a', 'solid', 'black', 'silhouette', 'of',
'malice', 'and', 'evil', '.', 'The', 'viewer', 'only', 'sees', 'her', 'a',
'few', 'times', ',', 'but', 'you', 'feel', 'her', 'malevolent', 'presence',
'in', 'every', 'frame', '.', 'As', 'the', 'camera', 'creeps', 'up', 'on', 'the',
'lawyer', 'while', 'he', "'s", 'reading', 'through', 'legal', 'papers', ',',
'you', 'expect', 'to', 'see', 'the', 'woman', 'in', 'black', 'at', 'any',
'moment', '.', 'When', 'the', 'lawyer', 'goes', 'out', 'to', 'the', 'generator',
'shed', 'to', 'turn', 'on', 'the', 'electricity', 'for', 'the', 'creepy', 'old',
'house', ',', 'the', 'camera', 'snakes', 'in', 'on', 'him', 'and', 'you',
'think', 'she', "'ll", 'pop', 'up', 'there', ',', 'too', '.', 'Waiting', 'for',
'the', 'woman', 'in', 'black', 'to', 'show', 'up', 'is', 'nail', '-',
'bitingly', 'suspenseful', '.', 'We', "'ve", 'seen', 'many', 'elements', 'of',
'this', 'story', 'before(the', 'locked', 'room', 'that', 'no', 'one', 'enters',
',', 'the', 'fog', ',', 'the', 'naive', 'outsider', 'who', 'ignores', 'the',
'locals', "'", 'warnings', ')', 'but', 'the', 'director', 'somehow', 'manages',
'to', 'combine', 'them', 'all', 'into', 'a', 'completely', 'new', '-',
'seeming', 'and', 'compelling', 'ghost', 'story', '.', 'Watch', 'it', 'with',
'a', 'buddy', 'so', 'you', 'can', 'have', 'someone', 'warm', 'to', 'grab',
'onto', 'while', 'waiting', 'for', 'the', 'woman', 'in', 'black', '.', '.',
'.'], 'label': 'pos'}
```

[ ]:
```
import random

train_data, valid_data = train_data.split(random_state = random.seed(SEED))
```

[ ]:
```
print(f'Number of training examples: {len(train_data)}')
print(f'Number of validation examples: {len(valid_data)}')
print(f'Number of testing examples: {len(test_data)}')
```

```
Number of training examples: 17500
Number of validation examples: 7500
Number of testing examples: 25000
```

Next, we have to build a *vocabulary*. This is simply a look up table where every unique word in your data set has a corresponding *index* (an integer).

We do this as our machine learning model cannot operate on strings, only numbers. Each *index* is used to construct a *one-hot* vector for each word. A one-hot vector is a vector where all of the elements are 0, except one, which is 1, and dimensionality is the total number of unique words in your vocabulary, commonly denoted by V.

The number of unique words in our training set is over 100,000, which means that our one-hot vectors will have over 100,000 dimensions! This will make training slow and possibly won't fit onto your GPU (if you're using one).

There are two ways effectively cut down our vocabulary, we can either only take the top n most common words or ignore words that appear less than m times. We'll do the former, only keeping

the top 25,000 words.

What do we do with words that appear in examples but we have cut from the vocabulary? We replace them with a special *unknown* or `<unk>` token. For example, if the sentence was "This film is great and I love it" but the word "love" was not in the vocabulary, it would become "This film is great and I `<unk>` it".

The following builds the vocabulary, only keeping the most common `max_size` tokens.

[ ]:
```
MAX_VOCAB_SIZE = 25_000

TEXT.build_vocab(train_data, max_size = MAX_VOCAB_SIZE)
LABEL.build_vocab(train_data)
```

Why do we only build the vocabulary on the training set? When testing any machine learning system you do not want to look at the test set in any way. We do not include the validation set as we want it to reflect the test set as much as possible.

[ ]:
```
print(f"Unique tokens in TEXT vocabulary: {len(TEXT.vocab)}")
print(f"Unique tokens in LABEL vocabulary: {len(LABEL.vocab)}")
```

```
Unique tokens in TEXT vocabulary: 25002
Unique tokens in LABEL vocabulary: 2
```

Why is the vocab size 25002 and not 25000? One of the addition tokens is the `<unk>` token and the other is a `<pad>` token.

When we feed sentences into our model, we feed a *batch* of them at a time, i.e. more than one at a time, and all sentences in the batch need to be the same size. Thus, to ensure each sentence in the batch is the same size, any shorter than the longest within the batch are padded.

We can also view the most common words in the vocabulary and their frequencies.

[ ]:
```
print(TEXT.vocab.freqs.most_common(20))
```

```
[('the', 201597), (',', 191954), ('.', 164703), ('and', 109139), ('a', 108718),
('of', 100285), ('to', 93202), ('is', 75848), ('in', 61217), ('I', 53700),
('it', 53206), ('that', 48662), ('"', 44466), ("'s", 42941), ('this', 42025),
('-', 37475), ('/><br', 35594), ('was', 34753), ('as', 30313), ('with', 29819)]
```

We can also see the vocabulary directly using either the `stoi` (**s**tring **to** **i**nt) or `itos` (**i**nt **to** **s**tring) method.

[ ]:
```
print(TEXT.vocab.itos[:10])
```

```
['<unk>', '<pad>', 'the', ',', '.', 'and', 'a', 'of', 'to', 'is']
```

We can also check the labels, ensuring 0 is for negative and 1 is for positive.

[ ]:
```
print(LABEL.vocab.stoi)
```

```
defaultdict(None, {'neg': 0, 'pos': 1})
```

The final step of preparing the data is creating the iterators. We iterate over these in the training/evaluation loop, and they return a batch of examples (indexed and converted into tensors) at each iteration.

We'll use a `BucketIterator` which is a special type of iterator that will return a batch of examples where each example is of a similar length, minimizing the amount of padding per example.

We also want to place the tensors returned by the iterator on the GPU (if you're using one). PyTorch handles this using `torch.device`, we then pass this device to the iterator.

[ ]:
```
BATCH_SIZE = 64

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

train_iterator, valid_iterator, test_iterator = data.BucketIterator.splits(
    (train_data, valid_data, test_data),
    batch_size = BATCH_SIZE,
    device = device)
```

1.2 Build the Model

The next stage is building the model that we'll eventually train and evaluate.

The embedding layer is used to transform our sparse one-hot vector (sparse as most of the elements are 0) into a dense embedding vector (dense as the dimensionality is a lot smaller and all the elements are real numbers). This embedding layer is simply a single fully connected layer. As well as reducing the dimensionality of the input to the RNN, there is the theory that words which have similar impact on the sentiment of the review are mapped close together in this dense vector space. For more information about word embeddings, see here.

The RNN layer is our RNN which takes in our dense vector and the previous hidden state h_{t-1}, which it uses to calculate the next hidden state, h_t.

Finally, the linear layer takes the final hidden state and feeds it through a fully connected layer, $f(h_T)$, transforming it to the correct output dimension.

The `forward` method is called when we feed examples into our model.

Each batch, `text`, is a tensor of size ***[sentence length, batch size]***. That is a batch of sentences, each having each word converted into a one-hot vector.

The RNN returns 2 tensors, `output` of size ***[sentence length, batch size, hidden dim]*** and `hidden` of size ***[1, batch size, hidden dim]***. `output` is the concatenation of the hidden state from every time step, whereas `hidden` is simply the final hidden state. We verify this using the `assert` statement. Note the `squeeze` method, which is used to remove a dimension of size 1.

Finally, we feed the last hidden state, `hidden`, through the linear layer, `fc`, to produce a prediction.

[ ]:
```
import torch.nn as nn

class RNN(nn.Module):
```

```
    def __init__(self, input_dim, embedding_dim, hidden_dim, output_dim):

        super().__init__()

        self.embedding = nn.Embedding(input_dim, embedding_dim)

        self.rnn = nn.RNN(embedding_dim, hidden_dim)

        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, text):

        #text = [sent len, batch size]

        embedded = self.embedding(text)

        #embedded = [sent len, batch size, emb dim]

        output, hidden = self.rnn(embedded)

        #output = [sent len, batch size, hid dim]
        #hidden = [1, batch size, hid dim]

        assert torch.equal(output[-1,:,:], hidden.squeeze(0))

        return self.fc(hidden.squeeze(0))
```

[ ]:
```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 100
HIDDEN_DIM = 256
OUTPUT_DIM = 1

model = RNN(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM)
```

Let's also create a function that will tell us how many trainable parameters our model has so we can compare the number of parameters across different models.

[ ]:
```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(model):,} trainable parameters')
```

```
The model has 2,592,105 trainable parameters
```

1.3 Train the Model

[ ]:
```
import torch.optim as optim

optimizer = optim.SGD(model.parameters(), lr=1e-3)
criterion = nn.BCEWithLogitsLoss()
model = model.to(device)
criterion = criterion.to(device)
```

[ ]:
```
def binary_accuracy(preds, y):
    """
    Returns accuracy per batch, i.e. if you get 8/10 right, this returns 0.8,␣
 ↪NOT 8
    """

    #round predictions to the closest integer
    rounded_preds = torch.round(torch.sigmoid(preds))
    correct = (rounded_preds == y).float() #convert into float for division
    acc = correct.sum() / len(correct)
    return acc
```

[ ]:
```
from tqdm import tqdm
```

[ ]:
```
def train(model, iterator, optimizer, criterion):

    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for batch in tqdm(iterator):

        optimizer.zero_grad()

        predictions = model(batch.text).squeeze(1)

        loss = criterion(predictions, batch.label)

        acc = binary_accuracy(predictions, batch.label)

        loss.backward()

        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

[ ]:
```
def evaluate(model, iterator, criterion):

    epoch_loss = 0
    epoch_acc = 0

    model.eval()

    with torch.no_grad():

        for batch in tqdm(iterator):

            predictions = model(batch.text).squeeze(1)

            loss = criterion(predictions, batch.label)

            acc = binary_accuracy(predictions, batch.label)

            epoch_loss += loss.item()
            epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

[ ]:
```
import time

def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

[ ]:
```
N_EPOCHS = 5

best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):

    start_time = time.time()

    train_loss, train_acc = train(model, train_iterator, optimizer, criterion)
    valid_loss, valid_acc = evaluate(model, valid_iterator, criterion)

    end_time = time.time()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
```

```
        torch.save(model.state_dict(), 'tut7-model.pt')

    print(f'Epoch: {epoch+1:02} | Epoch Time: {epoch_mins}m {epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f}%')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')
```

```
100%|| 274/274 [00:15<00:00, 17.72it/s]
100%|| 118/118 [00:01<00:00, 89.82it/s]

Epoch: 01 | Epoch Time: 0m 16s
        Train Loss: 0.694 | Train Acc: 50.02%
         Val. Loss: 0.696 |  Val. Acc: 49.20%

100%|| 274/274 [00:14<00:00, 18.54it/s]
100%|| 118/118 [00:01<00:00, 92.74it/s]

Epoch: 02 | Epoch Time: 0m 16s
        Train Loss: 0.693 | Train Acc: 49.54%
         Val. Loss: 0.696 |  Val. Acc: 50.20%

100%|| 274/274 [00:15<00:00, 18.14it/s]
100%|| 118/118 [00:01<00:00, 93.33it/s]

Epoch: 03 | Epoch Time: 0m 16s
        Train Loss: 0.693 | Train Acc: 50.05%
         Val. Loss: 0.696 |  Val. Acc: 50.66%

100%|| 274/274 [00:15<00:00, 18.15it/s]
100%|| 118/118 [00:01<00:00, 91.81it/s]

Epoch: 04 | Epoch Time: 0m 16s
        Train Loss: 0.693 | Train Acc: 49.51%
         Val. Loss: 0.696 |  Val. Acc: 49.11%

100%|| 274/274 [00:15<00:00, 18.14it/s]
100%|| 118/118 [00:01<00:00, 93.44it/s]

Epoch: 05 | Epoch Time: 0m 16s
        Train Loss: 0.693 | Train Acc: 50.08%
         Val. Loss: 0.696 |  Val. Acc: 50.57%
```

[ ]:
```
model.load_state_dict(torch.load('tut7-model.pt'))

test_loss, test_acc = evaluate(model, test_iterator, criterion)

print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
100%|| 391/391 [00:04<00:00, 92.54it/s]

Test Loss: 0.711 | Test Acc: 46.95%
```

1.4 Next Steps

In the next task, the improvements we will make are: - packed padded sequences - pre-trained word embeddings - different RNN architecture - bidirectional RNN - multi-layer RNN - regularization - a different optimizer

This will allow us to achieve ~84% accuracy.