2 - Improving Performance

In the previous notebook, we got the fundamentals down for sentiment analysis. In this notebook, we'll actually get decent results.

We will use:

- bidirectional RNN
- multi-layer RNN

This will allow us to achieve ~84% test accuracy.

Preparing Data

In [ ]:

```
!pip install torchtext==0.10.0
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.d
ev/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 M
B)
     |████████████████████████████████| 831.4 MB 2.7 kB/s
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist
-packages (from torchtext==0.10.0) (4.64.1)
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/dis
t-packages (from torchtext==0.10.0) (1.21.6)
Requirement already satisfied: typing-extensions in /usr/local/lib/p
ython3.7/dist-packages (from torch==1.9.0->torchtext==0.10.0) (4.1.
1)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python
3.7/dist-packages (from requests->torchtext==0.10.0) (2.10)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.2
1.1 in /usr/local/lib/python3.7/dist-packages (from requests->torcht
ext==0.10.0) (1.24.3)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/p
ython3.7/dist-packages (from requests->torchtext==0.10.0) (3.0.4)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/
python3.7/dist-packages (from requests->torchtext==0.10.0) (2022.6.1
5)
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 accoun
t 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
```

In [ ]:

```
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)
```

In [ ]:

```
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, 33.8M
B/s]
```

In [ ]:

```
import random

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

Next is the use of pre-trained word embeddings. Now, instead of having our word embeddings initialized randomly, they are initialized with these pre-trained vectors. We get these vectors simply by specifying which vectors we want and passing it as an argument to `build_vocab` . `TorchText` handles downloading the vectors and associating them with the correct words in our vocabulary.

Here, we'll be using the `"glove.6B.100d" vectors"` . `glove` is the algorithm used to calculate the vectors, go [here (https://nlp.stanford.edu/projects/glove/)](https://nlp.stanford.edu/projects/glove/) for more. `6B` indicates these vectors were trained on 6 billion tokens and `100d` indicates these vectors are 100-dimensional.

You can see the other available vectors [here (https://github.com/pytorch/text/blob/master/torchtext/vocab.py#L113)](https://github.com/pytorch/text/blob/master/torchtext/vocab.py#L113).

The theory is that these pre-trained vectors already have words with similar semantic meaning close together in vector space, e.g. "terrible", "awful", "dreadful" are nearby. This gives our embedding layer a good initialization as it does not have to learn these relations from scratch.

In [ ]:

```
MAX_VOCAB_SIZE = 25_000

TEXT.build_vocab(train_data,
                 max_size = MAX_VOCAB_SIZE,
                 vectors = "glove.6B.100d",
                 unk_init = torch.Tensor.normal_)

LABEL.build_vocab(train_data)
```

```
.vector_cache/glove.6B.zip: 862MB [02:39, 5.39MB/s]
100%|██████████| 399999/400000 [00:13<00:00, 29738.57it/s]
```

In [ ]:

```
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,
    sort_within_batch = True,
    device = device)
```

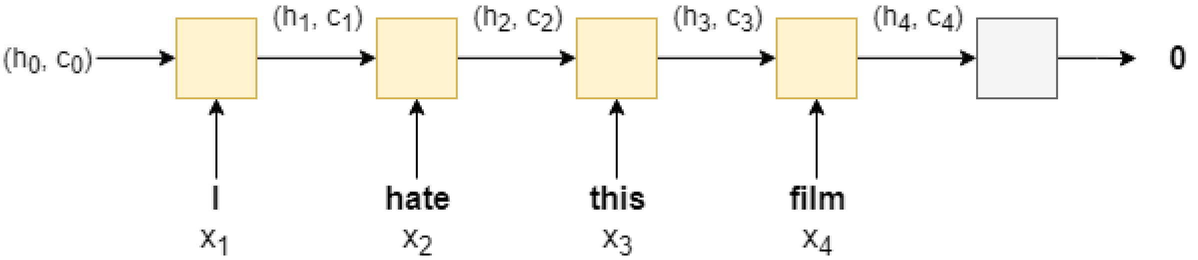
Build the Model

Different RNN Architecture

We'll be using a different RNN architecture called a Long Short-Term Memory (LSTM). Why is an LSTM better than a standard RNN? Standard RNNs suffer from the vanishing gradient problem (https://en.wikipedia.org/wiki/Vanishing_gradient_problem). LSTMs overcome this by having an extra recurrent state called a *cell*, c - which can be thought of as the "memory" of the LSTM - and the use use multiple *gates* which control the flow of information into and out of the memory. For more information, go here (https://colah.github.io/posts/2015-08-Understanding-LSTMs/). We can simply think of the LSTM as a function of x_t, h_t and c_t, instead of just x_t and h_t.

$$(h_t, c_t) = \text{LSTM}(x_t, h_t, c_t)$$

Thus, the model using an LSTM looks something like (with the embedding layers omitted):



The initial cell state, c_0, like the initial hidden state is initialized to a tensor of all zeros. The sentiment prediction is still, however, only made using the final hidden state, not the final cell state, i.e. $\hat{y} = f(h_T)$.

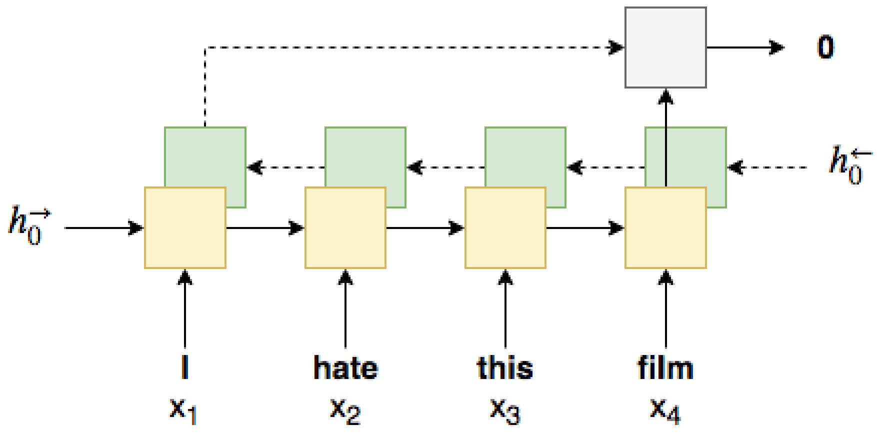
Bidirectional RNN

The concept behind a bidirectional RNN is simple. As well as having an RNN processing the words in the sentence from the first to the last (a forward RNN), we have a second RNN processing the words in the sentence from the **last to the first** (a backward RNN). At time step t, the forward RNN is processing word x_t, and the backward RNN is processing word x_{T-t+1}.

In PyTorch, the hidden state (and cell state) tensors returned by the forward and backward RNNs are stacked on top of each other in a single tensor.

We make our sentiment prediction using a concatenation of the last hidden state from the forward RNN (obtained from final word of the sentence), $h_T^\rightarrow$, and the last hidden state from the backward RNN (obtained from the first word of the sentence), $h_T^\leftarrow$, i.e. $\hat{y} = f(h_T^\rightarrow, h_T^\leftarrow)$

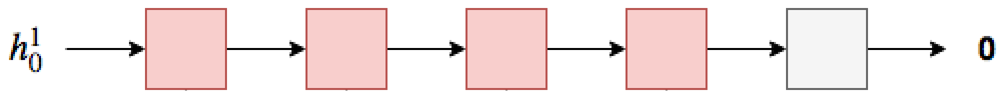
The image below shows a bi-directional RNN, with the forward RNN in orange, the backward RNN in green and the linear layer in silver.



Multi-layer RNN

Multi-layer RNNs (also called *deep RNNs*) are another simple concept. The idea is that we add additional RNNs on top of the initial standard RNN, where each RNN added is another *layer*. The hidden state output by the first (bottom) RNN at time-step t will be the input to the RNN above it at time step t. The prediction is then made from the final hidden state of the final (highest) layer.

The image below shows a multi-layer unidirectional RNN, where the layer number is given as a superscript. Also note that each layer needs their own initial hidden state, h_0^L.



In [ ]:

```
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, n_laye
rs,
                 bidirectional, dropout, pad_idx):

        super().__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx = p
ad_idx)### CODE HERE ###

        self.rnn = nn.LSTM(embedding_dim,
                           hidden_dim,
                           num_layers=n_layers,
                           bidirectional=bidirectional,
                           dropout=dropout)### CODE HERE ###

        self.fc = nn.Linear(hidden_dim * 2, output_dim)### CODE HERE ###

        self.dropout = nn.Dropout(dropout)

    def forward(self, text):

        #text = [sent len, batch size]

        embedded = self.dropout(self.embedding(text))

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

        output, (hidden, cell) = self.rnn(embedded) ### CODE HERE ###

        #output = [sent len, batch size, hid dim * num directions]
        #output over padding tokens are zero tensors

        #hidden = [num layers * num directions, batch size, hid dim]
        #cell = [num layers * num directions, batch size, hid dim]

        #concat the final forward (hidden[-2,:,:]) and backward (hidden[-1,:,:])
hidden layers
        #and apply dropout

        hidden = self.dropout(torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim =
1))

        #hidden = [batch size, hid dim * num directions]

        return self.fc(hidden)
```

In [ ]:

```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 100
HIDDEN_DIM = 256
OUTPUT_DIM = 1
N_LAYERS = 2
BIDIRECTIONAL = True
DROPOUT = 0.5
PAD_IDX = TEXT.vocab.stoi[TEXT.pad_token]

model = RNN(INPUT_DIM,
            EMBEDDING_DIM,
            HIDDEN_DIM,
            OUTPUT_DIM,
            N_LAYERS,
            BIDIRECTIONAL,
            DROPOUT,
            PAD_IDX)
```

We'll print out the number of parameters in our model.

Notice how we have almost twice as many parameters as before!

In [ ]:

```
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 4,810,857 trainable parameters
```

The final addition is copying the pre-trained word embeddings we loaded earlier into the `embedding` layer of our model.

We retrieve the embeddings from the field's vocab, and check they're the correct size, ***[vocab size, embedding dim]***

In [ ]:

```
pretrained_embeddings = TEXT.vocab.vectors

print(pretrained_embeddings.shape)
```

```
torch.Size([25002, 100])
```

We then replace the initial weights of the `embedding` layer with the pre-trained embeddings.

Note: this should always be done on the `weight.data` and not the `weight` !

In [ ]:

```
model.embedding.weight.data.copy_(pretrained_embeddings)
```

Out[ ]:

```
tensor([[-0.1117, -0.4966,  0.1631,  ...,  1.2647, -0.2753, -0.132
5],
        [-0.8555, -0.7208,  1.3755,  ...,  0.0825, -1.1314,  0.399
7],
        [-0.0382, -0.2449,  0.7281,  ..., -0.1459,  0.8278,  0.270
6],
        ...,
        [ 0.1068, -0.0572, -0.5956,  ...,  2.1442,  1.2027,  0.394
7],
        [ 0.3749, -0.0187, -0.3940,  ..., -0.5277,  0.0937, -1.115
2],
        [ 0.1787,  0.1934, -0.0216,  ..., -0.1655,  0.3625, -0.225
6]])
```

In [ ]:

```
UNK_IDX = TEXT.vocab.stoi[TEXT.unk_token]

model.embedding.weight.data[UNK_IDX] = torch.zeros(EMBEDDING_DIM)
model.embedding.weight.data[PAD_IDX] = torch.zeros(EMBEDDING_DIM)

print(model.embedding.weight.data)
```

```
tensor([[ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.000
0],
        [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.000
0],
        [-0.0382, -0.2449,  0.7281,  ..., -0.1459,  0.8278,  0.270
6],
        ...,
        [ 0.1068, -0.0572, -0.5956,  ...,  2.1442,  1.2027,  0.394
7],
        [ 0.3749, -0.0187, -0.3940,  ..., -0.5277,  0.0937, -1.115
2],
        [ 0.1787,  0.1934, -0.0216,  ..., -0.1655,  0.3625, -0.225
6]])
```

Train the Model

In [ ]:

```
import torch.optim as optim

optimizer = optim.Adam(model.parameters(), lr=1e-3)
```

In [ ]:

```
criterion = nn.BCEWithLogitsLoss()

model = model.to(device)
criterion = criterion.to(device)
```

In [ ]:

```
def binary_accuracy(preds, y):

    #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
```

In [ ]:

```
from tqdm import tqdm
```

In [ ]:

```
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)
```

In [ ]:

```
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)
```

In [ ]:

```
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
```

In [ ]:

```
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 [00:31<00:00,  8.64it/s]
100%|██████████| 118/118 [00:04<00:00, 27.85it/s]

Epoch: 01 | Epoch Time: 0m 35s
	Train Loss: 0.693 | Train Acc: 52.76%
	 Val. Loss: 0.683 |  Val. Acc: 58.13%

100%|██████████| 274/274 [00:32<00:00,  8.49it/s]
100%|██████████| 118/118 [00:04<00:00, 27.98it/s]

Epoch: 02 | Epoch Time: 0m 36s
	Train Loss: 0.689 | Train Acc: 54.11%
	 Val. Loss: 0.698 |  Val. Acc: 49.03%

100%|██████████| 274/274 [00:32<00:00,  8.40it/s]
100%|██████████| 118/118 [00:04<00:00, 27.24it/s]

Epoch: 03 | Epoch Time: 0m 36s
	Train Loss: 0.616 | Train Acc: 64.48%
	 Val. Loss: 0.784 |  Val. Acc: 61.06%

100%|██████████| 274/274 [00:33<00:00,  8.23it/s]
100%|██████████| 118/118 [00:04<00:00, 26.91it/s]

Epoch: 04 | Epoch Time: 0m 37s
	Train Loss: 0.424 | Train Acc: 81.53%
	 Val. Loss: 0.327 |  Val. Acc: 86.42%

100%|██████████| 274/274 [00:33<00:00,  8.22it/s]
100%|██████████| 118/118 [00:04<00:00, 26.78it/s]

Epoch: 05 | Epoch Time: 0m 37s
	Train Loss: 0.343 | Train Acc: 85.73%
	 Val. Loss: 0.299 |  Val. Acc: 88.39%
```

In [ ]:

```
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 [00:14<00:00, 27.31it/s]

Test Loss: 0.307 | Test Acc: 87.58%
```

OBSERVATIONS

- Just by changing the optimiser from *SGD* to *Adam* improves the model accuracy from 50% to 87%
- Unidirectional, 2-layer LSTM, with Adam optimiser gives 83% accuracy
- Bidirectional, 1-layer LSTM, with Adam optimiser gives 84% accuracy,epoch time = 0m14s
- Bidirectional, 3-layer LSTM, with Adam optimiser gives 84% accuracy, epoch time = 1m3s - Bidirectional, 4-layer LSTM, with Adam optimiser gives 69% accuracy, epoch time = 1m30s - Unidirectional, 4-layer LSTM, with Adam optimiser gives 68% accuracy, epoch time = 40s - Unidirectional, 3-layer LSTM, with Adam optimiser gives 50% accuracy, epoch time = 0m30s

CONCLUSION

- As we increase number of layers of RNN, it increases training time, but does not necessarily guarantee better accuracy
- Bidirectional LSTM results in better accuracy, but training time also increases.
- Among the performed experiments, 2 layer LSTM, Bidirectional with Adam optimiser performs the best with accuracy upto 87%.