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 [1]:
```
!pip install torchtext==0.14.0
```

```
Collecting torchtext==0.14.0
  Downloading torchtext-0.14.0-cp310-cp310-manylinux1_x86_64.whl (2.0 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.0/2.0 MB 22.3 MB/s eta 0:00:00
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from torchtext==0.14.0) (4.66.1)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from torchtext==0.14.0) (2.31.0)
Collecting torch==1.13.0 (from torchtext==0.14.0)
  Downloading torch-1.13.0-cp310-cp310-manylinux1_x86_64.whl (890.1 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 890.1/890.1 MB 1.9 MB/s eta 0:00:00
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from torchtext==0.14.0) (1.23.5)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.10/dist-packages (from torch==1.13.0->torchtext==0.14.0) (4.5.0)
Collecting nvidia-cuda-runtime-cu11==11.7.99 (from torch==1.13.0->torchtext==0.14.0)
  Downloading nvidia_cuda_runtime_cu11-11.7.99-py3-none-manylinux1_x86_64.whl (849 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 849.3/849.3 kB 73.9 MB/s eta 0:00:00
Collecting nvidia-cudnn-cu11==8.5.0.96 (from torch==1.13.0->torchtext==0.14.0)
  Downloading nvidia_cudnn_cu11-8.5.0.96-2-py3-none-manylinux1_x86_64.whl (557.1 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 557.1/557.1 MB 1.4 MB/s eta 0:00:00
Collecting nvidia-cublas-cu11==11.10.3.66 (from torch==1.13.0->torchtext==0.14.0)
  Downloading nvidia_cublas_cu11-11.10.3.66-py3-none-manylinux1_x86_64.whl (317.1 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 317.1/317.1 MB 4.3 MB/s eta 0:00:00
Collecting nvidia-cuda-nvrtc-cu11==11.7.99 (from torch==1.13.0->torchtext==0.14.0)
  Downloading nvidia_cuda_nvrtc_cu11-11.7.99-2-py3-none-manylinux1_x86_64.whl (21.0 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21.0/21.0 MB 55.9 MB/s eta 0:00:00
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-packages (from nvidia-cublas-cu11==11.10.3.66->torch==1.13.0->torchtext==0.14.0) (67.7.2)
Requirement already satisfied: wheel in /usr/local/lib/python3.10/dist-packages (from nvidia-cublas-cu11==11.10.3.66->torch==1.13.0->torchtext==0.14.0) (0.41.2)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.14.0) (3.3.0)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.14.0) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.14.0) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.14.0) (2023.7.22)
Installing collected packages: nvidia-cuda-runtime-cu11, nvidia-cuda-nvrtc-cu11, nvidia-cublas-cu11, nvidia-cudnn-cu11, torch, torchtext
  Attempting uninstall: torch
    Found existing installation: torch 2.1.0+cu118
    Uninstalling torch-2.1.0+cu118:
      Successfully uninstalled torch-2.1.0+cu118
  Attempting uninstall: torchtext
    Found existing installation: torchtext 0.16.0
    Uninstalling torchtext-0.16.0:
      Successfully uninstalled torchtext-0.16.0
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 conflict
s.
torchaudio 2.1.0+cu118 requires torch==2.1.0, but you have torch 1.13.0 which is incompatible.
torchdata 0.7.0 requires torch==2.1.0, but you have torch 1.13.0 which is incompatible.
torchvision 0.16.0+cu118 requires torch==2.1.0, but you have torch 1.13.0 which is incompatible.
Successfully installed nvidia-cublas-cu11-11.10.3.66 nvidia-cuda-nvrtc-cu11-11.7.99 nvidia-cuda-runtime-cu11-11.7.99 nvidia-cudnn-cu11-8.5.0.96 torch-1.13.0 torchtext-0.14.0
```

In [2]:
```
!pip install torchtext==0.6
```

```
Collecting torchtext==0.6
  Downloading torchtext-0.6.0-py3-none-any.whl (64 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 64.2/64.2 kB 1.8 MB/s eta 0:00:00
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from torchtext==0.6) (4.66.1)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from torchtext==0.6) (2.31.0)
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages (from torchtext==0.6) (1.13.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from torchtext==0.6) (1.23.5)
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages (from torchtext==0.6) (1.16.0)
Collecting sentencepiece (from torchtext==0.6)
  Downloading sentencepiece-0.1.99-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (1.3 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.3/1.3 MB 23.4 MB/s eta 0:00:00
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.6) (3.3.0)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.6) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.6) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.6) (2023.7.22)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6) (4.5.0)
Requirement already satisfied: nvidia-cuda-runtime-cu11==11.7.99 in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6) (11.7.99)
Requirement already satisfied: nvidia-cudnn-cu11==8.5.0.96 in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6) (8.5.0.96)
Requirement already satisfied: nvidia-cublas-cu11==11.10.3.66 in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6) (11.10.3.66)
Requirement already satisfied: nvidia-cuda-nvrtc-cu11==11.7.99 in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6) (11.7.99)
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-packages (from nvidia-cublas-cu11==11.10.3.66->torch->torchtext==0.6) (67.7.2)
Requirement already satisfied: wheel in /usr/local/lib/python3.10/dist-packages (from nvidia-cublas-cu11==11.10.3.66->torch->torchtext==0.6) (0.41.2)
Installing collected packages: sentencepiece, torchtext
  Attempting uninstall: torchtext
    Found existing installation: torchtext 0.14.0
    Uninstalling torchtext-0.14.0:
      Successfully uninstalled torchtext-0.14.0
Successfully installed sentencepiece-0.1.99 torchtext-0.6.0
```

In [3]:
```python
import torch
from torchtext.data  import Field, TabularDataset, BucketIterator, Iterator, LabelField
SEED = 12345

torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True

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

In [4]:
```python
from torchtext 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, 34.2MB/s]
```

In [5]:
```python
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 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.

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 [6]:
```python
MAX_VOCAB_SIZE = 25_000

TEXT.build_vocab(train_data,
                 max_size = MAX_VOCAB_SIZE,
                 vectors = "glove.6B.100d", #word embedding 6B- 6 billion train_data, 100D- output_dim
                 unk_init = torch.Tensor.normal_)

LABEL.build_vocab(train_data)
```

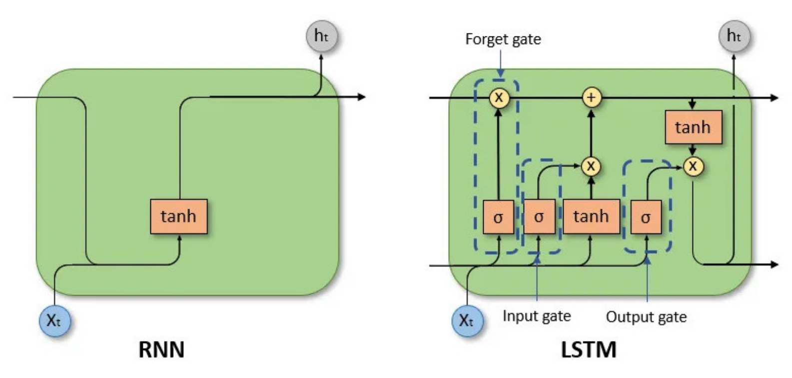
```
.vector_cache/glove.6B.zip: 862MB [02:39, 5.40MB/s]
100%|█████████▉| 399999/400000 [00:15<00:00, 25976.59it/s]
```

In [7]:
```python
BATCH_SIZE = 64

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

train_iterator, valid_iterator, test_iterator = BucketIterator.splits( #minimumal padding and clustering of similar size/length
    (train_data, valid_data, test_data),
    batch_size = BATCH_SIZE,
    sort_within_batch = True,
    device = device)
```

Build the Model

Different RNN Architecture

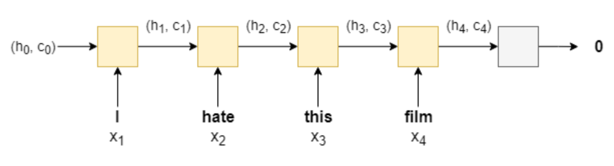
We'll be using a different RNN architecture called a Long Short-Term Memory (LSTM). Why is an LSTM better than a standard RNN? Standard RNNs suffer from the vanishing gradient problem.



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. 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. $\hat{y} = f(h_T)$.

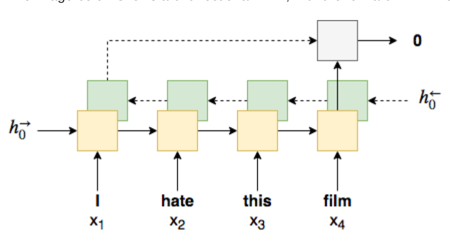
Bidirectional RNN

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. $\hat{y} = f(h_T^\rightarrow, h_T^\leftarrow)$

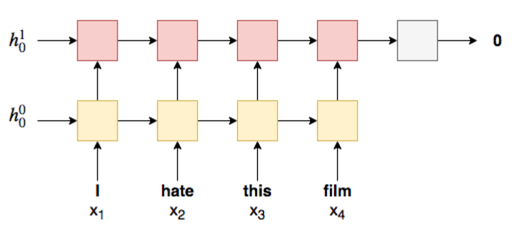
The image below shows a bi-directional RNN, with the forward RNN in orange, the backward RNN in green and the linear layer in silver.



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 [8]:
```python
import torch.nn as nn

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

        super().__init__()

        self.hidden_dim= hidden_dim
        self.embedding_dim= embedding_dim
        self.n_layers= n_layers

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

        #LSTM
        self.rnn = nn.LSTM(embedding_dim, hidden_dim, n_layers, bidirectional= bidirectional) #Test Loss: 0.365 | Test Acc: 84.36%

        #GRU
        # self.rnn = nn.GRU(embedding_dim, hidden_dim, n_layers, bidirectional= bidirectional) #Test Loss: 0.266 | Test Acc: 88.95%

        self.fc = nn.Linear(hidden_dim*2 if bidirectional else hidden_dim, 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]

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

        #GRU
        # output, hidden= self.rnn(embedded)

        #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) if self.rnn.bidirectional else hidden[-1,:,:])

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

        return self.fc(hidden)
```

In [9]:
```python
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 [10]:
```python
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 [11]:
```python
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 [12]:
```python
model.embedding.weight.data.copy_(pretrained_embeddings)
```

Out[12]:
```
tensor([[-1.4798,  0.4873, -3.0128,  ..., -0.2419, -0.8106,  1.0837],
        [-0.7432,  0.7603,  0.6474,  ...,  1.1088, -1.2302, -1.0391],
        [-0.0382, -0.2449,  0.7281,  ..., -0.1459,  0.8278,  0.2706],
        ...,
        [ 0.2110, -0.2472,  0.6508,  ..., -0.1627,  0.4507, -1.1627],
        [ 0.2818,  0.7171,  0.2196,  ...,  0.3584,  0.8843,  0.6610],
        [ 0.2447, -0.3031,  0.6721,  ...,  0.0657, -0.1565, -0.2624]])
```

In [13]:
```python
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,  0.0000],
        [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
        [-0.0382, -0.2449,  0.7281,  ..., -0.1459,  0.8278,  0.2706],
        ...,
        [ 0.2110, -0.2472,  0.6508,  ..., -0.1627,  0.4507, -1.1627],
        [ 0.2818,  0.7171,  0.2196,  ...,  0.3584,  0.8843,  0.6610],
        [ 0.2447, -0.3031,  0.6721,  ...,  0.0657, -0.1565, -0.2624]])
```

Train the Model

In [14]:
```python
import torch.optim as optim

#optimizer = optim.SGD(model.parameters(), lr=1e-3)
optimizer = optim.AdamW(model.parameters(), lr=1e-4) #AdamW-> Adam with weight decay
```

In [15]:
```python
criterion = nn.BCEWithLogitsLoss()

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

In [16]:
```python
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 [17]:
```python
from tqdm import tqdm
```

In [18]:
```python
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 [19]:
```python
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 [20]:
```python
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 [21]:
```python
N_EPOCHS = 20

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 [00:28<00:00,  9.56it/s]
100%|██████████| 118/118 [00:04<00:00, 24.80it/s]
Epoch: 01 | Epoch Time: 0m 33s
	Train Loss: 0.688 | Train Acc: 54.14%
	 Val. Loss: 0.673 |  Val. Acc: 57.27%
100%|██████████| 274/274 [00:28<00:00,  9.68it/s]
100%|██████████| 118/118 [00:04<00:00, 25.53it/s]
Epoch: 02 | Epoch Time: 0m 32s
	Train Loss: 0.636 | Train Acc: 63.59%
	 Val. Loss: 0.554 |  Val. Acc: 71.17%
100%|██████████| 274/274 [00:29<00:00,  9.40it/s]
100%|██████████| 118/118 [00:04<00:00, 24.79it/s]
Epoch: 03 | Epoch Time: 0m 33s
	Train Loss: 0.590 | Train Acc: 69.41%
	 Val. Loss: 0.619 |  Val. Acc: 70.41%
100%|██████████| 274/274 [00:29<00:00,  9.23it/s]
100%|██████████| 118/118 [00:04<00:00, 24.43it/s]
Epoch: 04 | Epoch Time: 0m 34s
	Train Loss: 0.576 | Train Acc: 70.88%
	 Val. Loss: 0.504 |  Val. Acc: 76.47%
100%|██████████| 274/274 [00:29<00:00,  9.14it/s]
100%|██████████| 118/118 [00:04<00:00, 24.98it/s]
Epoch: 05 | Epoch Time: 0m 34s
	Train Loss: 0.537 | Train Acc: 73.60%
	 Val. Loss: 0.574 |  Val. Acc: 75.60%
100%|██████████| 274/274 [00:29<00:00,  9.21it/s]
100%|██████████| 118/118 [00:04<00:00, 24.49it/s]
Epoch: 06 | Epoch Time: 0m 34s
	Train Loss: 0.525 | Train Acc: 74.67%
	 Val. Loss: 0.498 |  Val. Acc: 75.23%
100%|██████████| 274/274 [00:29<00:00,  9.15it/s]
100%|██████████| 118/118 [00:04<00:00, 24.89it/s]
Epoch: 07 | Epoch Time: 0m 34s
	Train Loss: 0.502 | Train Acc: 75.95%
	 Val. Loss: 0.484 |  Val. Acc: 76.68%
100%|██████████| 274/274 [00:29<00:00,  9.14it/s]
100%|██████████| 118/118 [00:04<00:00, 24.64it/s]
Epoch: 08 | Epoch Time: 0m 34s
	Train Loss: 0.482 | Train Acc: 77.12%
	 Val. Loss: 0.483 |  Val. Acc: 79.58%
100%|██████████| 274/274 [00:29<00:00,  9.19it/s]
100%|██████████| 118/118 [00:04<00:00, 24.86it/s]
Epoch: 09 | Epoch Time: 0m 34s
	Train Loss: 0.476 | Train Acc: 77.69%
	 Val. Loss: 0.429 |  Val. Acc: 80.42%
100%|██████████| 274/274 [00:29<00:00,  9.20it/s]
100%|██████████| 118/118 [00:04<00:00, 24.25it/s]
Epoch: 10 | Epoch Time: 0m 34s
	Train Loss: 0.631 | Train Acc: 66.57%
	 Val. Loss: 0.679 |  Val. Acc: 65.55%
100%|██████████| 274/274 [00:29<00:00,  9.14it/s]
100%|██████████| 118/118 [00:04<00:00, 24.93it/s]
Epoch: 11 | Epoch Time: 0m 34s
	Train Loss: 0.610 | Train Acc: 66.46%
	 Val. Loss: 0.565 |  Val. Acc: 71.60%
100%|██████████| 274/274 [00:30<00:00,  9.10it/s]
100%|██████████| 118/118 [00:04<00:00, 24.17it/s]
Epoch: 12 | Epoch Time: 0m 35s
	Train Loss: 0.633 | Train Acc: 63.61%
	 Val. Loss: 0.646 |  Val. Acc: 62.93%
100%|██████████| 274/274 [00:30<00:00,  9.09it/s]
100%|██████████| 118/118 [00:04<00:00, 24.87it/s]
Epoch: 13 | Epoch Time: 0m 34s
	Train Loss: 0.606 | Train Acc: 67.03%
	 Val. Loss: 0.566 |  Val. Acc: 72.54%
100%|██████████| 274/274 [00:30<00:00,  9.13it/s]
100%|██████████| 118/118 [00:04<00:00, 24.00it/s]
Epoch: 14 | Epoch Time: 0m 34s
	Train Loss: 0.497 | Train Acc: 76.28%
	 Val. Loss: 0.451 |  Val. Acc: 79.54%
100%|██████████| 274/274 [00:29<00:00,  9.14it/s]
100%|██████████| 118/118 [00:04<00:00, 24.91it/s]
Epoch: 15 | Epoch Time: 0m 34s
	Train Loss: 0.461 | Train Acc: 79.09%
	 Val. Loss: 0.474 |  Val. Acc: 81.44%
100%|██████████| 274/274 [00:30<00:00,  9.10it/s]
100%|██████████| 118/118 [00:04<00:00, 24.07it/s]
Epoch: 16 | Epoch Time: 0m 35s
	Train Loss: 0.455 | Train Acc: 79.24%
	 Val. Loss: 0.576 |  Val. Acc: 79.23%
100%|██████████| 274/274 [00:30<00:00,  9.10it/s]
100%|██████████| 118/118 [00:04<00:00, 24.90it/s]
Epoch: 17 | Epoch Time: 0m 34s
	Train Loss: 0.425 | Train Acc: 81.24%
	 Val. Loss: 0.457 |  Val. Acc: 79.52%
100%|██████████| 274/274 [00:30<00:00,  9.07it/s]
100%|██████████| 118/118 [00:04<00:00, 23.97it/s]
Epoch: 18 | Epoch Time: 0m 35s
	Train Loss: 0.407 | Train Acc: 82.21%
	 Val. Loss: 0.364 |  Val. Acc: 84.16%
100%|██████████| 274/274 [00:30<00:00,  9.12it/s]
100%|██████████| 118/118 [00:04<00:00, 24.78it/s]
Epoch: 19 | Epoch Time: 0m 34s
	Train Loss: 0.399 | Train Acc: 82.48%
	 Val. Loss: 0.371 |  Val. Acc: 84.34%
100%|██████████| 274/274 [00:30<00:00,  9.08it/s]
100%|██████████| 118/118 [00:04<00:00, 23.98it/s]
Epoch: 20 | Epoch Time: 0m 35s
	Train Loss: 0.397 | Train Acc: 82.62%
	 Val. Loss: 0.346 |  Val. Acc: 85.03%
```

In [22]:
```python
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 [00:15<00:00, 25.03it/s]
Test Loss: 0.365 | Test Acc: 84.36%
```

- Biderectional
- MultiLayer
- MultiLayer+ Biderctioanl
- GRU inplace of LSTM

The rnn model with GRU performs slightly better. The loss and accuracy metrics can been seen in **Class RNN cell**.