Instructions

Use this as a reference [link text]

Change this code in such a way that:

1. it has 3 LSTM layers
2. it has used a for loop to do so in the forward function
3. the dropout value used is 0.2
4. trained on the text that is reversed (for example "my name is Rohan" becomes "Rohan is name my"
5. achieves 87% or more accuracy once done, share the Github link as well (after training on Google Colab, move the file to GitHub).

```
import torch
import random
import spacy
from torchtext import data, datasets
import torch.nn as nn
import torch.optim as optim

SEED = 2812
torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True

text = data.Field(tokenize = 'spacy', include_lengths = True)
label = data.LabelField(dtype = torch.float)
```

```
#load the IMDb dataset.
train_data, test_data = datasets.IMDB.splits(text, label)
```

```
aclImdb_v1.tar.gz:   0%|          | 98.3k/84.1M [00:00<01:28, 947kB/s]downloading aclImdb_v1.tar.gz
aclImdb_v1.tar.gz: 100%|██████████| 84.1M/84.1M [00:01<00:00, 42.6MB/s]
```

```
#reverse training text data in-place
for i in range(len(train_data.examples)):
  vars(train_data.examples[i]).get('text').reverse()
```

```
# create the validation set from our training set.
train_data, valid_data = train_data.split(random_state = random.seed(SEED))
```

```
# build vocabulary with pre-trained global embedding

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 [06:38, 2.17MB/s]
100%|█████████▉| 399490/400000 [00:15<00:00, 26488.77it/s]
```

```
#Another thing for packed padded sequences all of the tensors within a batch
#need to be sorted by their lengths. This is handled in the iterator by setting
#sort_within_batch = True.

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

```
class RNN(nn.Module):

    #parts list for building blocks
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim,
                 n_layers, bidirectional, dropout, pad_idx):

        super().__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim,
                                      padding_idx = pad_idx)
        #bidirectional = False
        self.rnns = nn.ModuleList([nn.LSTM(embedding_dim, hidden_dim,
                                           bidirectional=bidirectional)])
        # LSTM layers = 3
        for _ in range(n_layers - 1):
          self.rnns.append(nn.LSTM(hidden_dim, hidden_dim,
                                   bidirectional=bidirectional))

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

        self.dropout = nn.Dropout(dropout)

    #step-by-step manual for assembling building blocks
    def forward(self, text, text_lengths):

        #text = [sent len, batch size]

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

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

        #pack sequence
        packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, text_lengths)

        #stack multiple (3) LSTM layers with dropouts
        x = packed_embedded
        for rnn in self.rnns:
```

```
            _, (x, _) = rnn(x)
            x = self.dropout(x)

        #x = last hidden states [1, batch size, hid dim]
        hidden = x.squeeze(0)

        return self.fc(hidden)
```

```
#define model constants

INPUT_DIM = len(text.vocab)
EMBEDDING_DIM = 100
HIDDEN_DIM = 256
OUTPUT_DIM = 1
N_LAYERS = 3
#changed from True to False
BIDIRECTIONAL = False
#changed from 0.5 to 0.2
DROPOUT = 0.2
PAD_IDX = text.vocab.stoi[text.pad_token]
```

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

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

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

```
RNN(
```

```
  (embedding): Embedding(25002, 100, padding_idx=1)
  (rnns): ModuleList(
    (0): LSTM(100, 256)
    (1): LSTM(256, 256)
    (2): LSTM(256, 256)
  )
  (fc): Linear(in_features=256, out_features=1, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
The model has 3,919,721 trainable parameters
```

```
pretrained_embeddings = text.vocab.vectors
print(pretrained_embeddings.shape)
```

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

```
#copy pre-trained embeddings from vocabulary to model
model.embedding.weight.data.copy_(pretrained_embeddings)
```

```
tensor([[ 0.4229, -0.5757, -0.0617,  ...,  0.4862, -1.3053,  1.3924],
        [ 0.6612, -1.0053, -1.7353,  ...,  0.3116, -0.2421, -1.1424],
        [-0.0382, -0.2449,  0.7281,  ..., -0.1459,  0.8278,  0.2706],
        ...,
        [-0.1509, -0.2923,  0.4006,  ..., -0.1604,  0.1807, -0.6672],
        [-0.6806,  0.4531, -0.0683,  ..., -0.0388,  0.4975, -0.0208],
        [ 0.2735, -0.1130,  0.2871,  ..., -0.8155, -0.0639,  0.9330]])
```

```
#zero weights for <unk> and <pad> tokens

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],
```

```
        ...,
        [-0.1509, -0.2923,  0.4006,  ..., -0.1604,  0.1807, -0.6672],
        [-0.6806,  0.4531, -0.0683,  ..., -0.0388,  0.4975, -0.0208],
        [ 0.2735, -0.1130,  0.2871,  ..., -0.8155, -0.0639,  0.9330]])
```

Train the Model

```
#instantiate optimizer
optimizer = optim.Adam(model.parameters())
```

```
#instantiate loss function
criterion = nn.BCEWithLogitsLoss()
```

```
#place the model and criterion on the GPU (if available)
model = model.to(device)
criterion = criterion.to(device)
```

```
#define the accuracy for training, validation and testing
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
```

```
#define the training
def train(model, iterator, optimizer, criterion):

    epoch_loss = 0
    epoch_acc = 0

```

```
    model.train()

    for batch in iterator:

        optimizer.zero_grad()

        text, text_lengths = batch.text

        #runtime type is GPU but  model expects CPU tensor
        text_lengths = text_lengths.cpu()

        predictions = model(text, text_lengths).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 iterator:

            text, text_lengths = batch.text
```

```

            #runtime type is GPU but model expects CPU tensor
            text_lengths = text_lengths.cpu()

            predictions = model(text, text_lengths).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)
```

```
# define how to calculate time required per epoch

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 = 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(), 'tut2-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%|█████████▉| 399490/400000 [00:30<00:00, 26488.77it/s]Epoch: 01 | Epoch Time: 0m 9s
	Train Loss: 0.682 | Train Acc: 56.58%
	 Val. Loss: 0.645 |  Val. Acc: 62.96%
Epoch: 02 | Epoch Time: 0m 9s
	Train Loss: 0.626 | Train Acc: 65.12%
	 Val. Loss: 0.653 |  Val. Acc: 65.77%
Epoch: 03 | Epoch Time: 0m 9s
	Train Loss: 0.676 | Train Acc: 53.34%
	 Val. Loss: 0.681 |  Val. Acc: 53.75%
Epoch: 04 | Epoch Time: 0m 9s
	Train Loss: 0.687 | Train Acc: 52.06%
	 Val. Loss: 0.693 |  Val. Acc: 49.77%
Epoch: 05 | Epoch Time: 0m 9s
	Train Loss: 0.547 | Train Acc: 69.93%
	 Val. Loss: 0.349 |  Val. Acc: 85.47%
Epoch: 06 | Epoch Time: 0m 9s
	Train Loss: 0.277 | Train Acc: 89.46%
	 Val. Loss: 0.338 |  Val. Acc: 86.63%
Epoch: 07 | Epoch Time: 0m 9s
	Train Loss: 0.198 | Train Acc: 92.82%
	 Val. Loss: 0.303 |  Val. Acc: 88.55%
Epoch: 08 | Epoch Time: 0m 9s
	Train Loss: 0.148 | Train Acc: 94.70%
	 Val. Loss: 0.336 |  Val. Acc: 87.95%
Epoch: 09 | Epoch Time: 0m 9s
	Train Loss: 0.109 | Train Acc: 96.32%
	 Val. Loss: 0.368 |  Val. Acc: 87.28%
Epoch: 10 | Epoch Time: 0m 9s
	Train Loss: 0.081 | Train Acc: 97.27%
	 Val. Loss: 0.402 |  Val. Acc: 88.34%
Epoch: 11 | Epoch Time: 0m 9s
	Train Loss: 0.061 | Train Acc: 97.96%
```

```
         Val. Loss: 0.386 |  Val. Acc: 87.95%
Epoch: 12 | Epoch Time: 0m 9s
        Train Loss: 0.048 | Train Acc: 98.51%
         Val. Loss: 0.464 |  Val. Acc: 87.97%
Epoch: 13 | Epoch Time: 0m 9s
        Train Loss: 0.047 | Train Acc: 98.51%
         Val. Loss: 0.475 |  Val. Acc: 86.44%
Epoch: 14 | Epoch Time: 0m 9s
        Train Loss: 0.029 | Train Acc: 99.13%
         Val. Loss: 0.555 |  Val. Acc: 88.14%
Epoch: 15 | Epoch Time: 0m 9s
        Train Loss: 0.025 | Train Acc: 99.22%
         Val. Loss: 0.570 |  Val. Acc: 88.00%
Epoch: 16 | Epoch Time: 0m 9s
        Train Loss: 0.023 | Train Acc: 99.33%
         Val. Loss: 0.514 |  Val. Acc: 88.00%
Epoch: 17 | Epoch Time: 0m 9s
        Train Loss: 0.020 | Train Acc: 99.42%
         Val. Loss: 0.587 |  Val. Acc: 87.97%
Epoch: 18 | Epoch Time: 0m 9s
        Train Loss: 0.014 | Train Acc: 99.53%
         Val. Loss: 0.699 |  Val. Acc: 87.60%
Epoch: 19 | Epoch Time: 0m 9s
        Train Loss: 0.015 | Train Acc: 99.60%
         Val. Loss: 0.669 |  Val. Acc: 87.55%
Epoch: 20 | Epoch Time: 0m 9s
        Train Loss: 0.011 | Train Acc: 99.69%
```

```python
# test model using testing dataset
model.load_state_dict(torch.load('tut2-model.pt'))

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

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

```
Test Loss: 0.346 | Test Acc: 86.31%
```

```python
nlp = spacy.load('en')

def predict_sentiment(model, sentence):
    model.eval()
```

```
    tokenized = [tok.text for tok in nlp.tokenizer(sentence)]
    indexed = [text.vocab.stoi[t] for t in tokenized]
    length = [len(indexed)]
    tensor = torch.LongTensor(indexed).to(device)
    tensor = tensor.unsqueeze(1)
    length_tensor = torch.LongTensor(length)
    prediction = torch.sigmoid(model(tensor, length_tensor))
    return prediction.item()
```

```
#test a positive sentence
predict_sentiment(model, "This film is very good")
```

```
0.9891530275344849
```

Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.