Assignment 4 : LSTM and Attention Mechanism

This assignment is composed of the following parts

```
1. LSTM and its variants
    - Vanilla LSTM
    - Coupled Gate LSTM
    - Peephole LSTM
    - BiLSTM
2. Attention Mechanism
    - General Attention
    - Self-Attention
```

Starting from BiLSTM part we will be working on a sequence classification model which has LSTM as the Encoder (and attention mechanisms before the output)

Code for preparing the dataset for this assignment

In [1]:

```
import torchtext
import torch
from torch import nn

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

#make our work comparable if restarted the kernel
SEED = 1234
torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True

#uncomment this if you are not using puffer
import os
os.environ['http_proxy'] = 'http://192.41.170.23:3128'
os.environ['https_proxy'] = 'http://192.41.170.23:3128'

from torchtext.datasets import IMDB
train_iter, test_iter = IMDB(split=('train', 'test'))

#pip install spacy
#python -m spacy download en_core_web_sm
from torchtext.data.utils import get_tokenizer
tokenizer = get_tokenizer('spacy', language='en_core_web_sm')

from torchtext.vocab import build_vocab_from_iterator
def yield_tokens(data_iter):
    for _, text in data_iter:
        yield tokenizer(text)

vocab = build_vocab_from_iterator(yield_tokens(train_iter), specials=['<unk>', '<pad
vocab.set_default_index(vocab["<unk>"])

#https://github.com/pytorch/text/issues/1350
from torchtext.vocab import FastText
fast_vectors = FastText('simple')

fast_embedding = fast_vectors.get_vecs_by_tokens(vocab.get_itos()).to(device)
# vocab.get_itos() returns a list of strings (tokens), where the token at the i'th p
# get_vecs_by_tokens gets the pre-trained vector for each string when given a list o
# therefore pretrained_embedding is a fully "aligned" embedding matrix
```

```
cuda
```

Defining Hyperparameters

In [2]:

```
input_dim = len(vocab)
hidden_dim = 256
embed_dim = 300
output_dim = 1

pad_idx = vocab['<pad>']
num_layers = 2
bidirectional = True
dropout = 0.5

batch_size = 32
num_epochs = 3
lr=0.0001
```

Code for preparing Train and Test Loader

In [3]:

```
text_pipeline = lambda x: vocab(tokenizer(x))
label_pipeline = lambda x: 1 if x == 'pos' else 0

from torch.utils.data import DataLoader
from torch.nn.utils.rnn import pad_sequence #++

def collate_batch(batch):
    label_list, text_list, length_list = [], [], []
    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)
        length_list.append(processed_text.size(0))  #++<-----packed padded sequences
    #criterion expects float labels
    return torch.tensor(label_list, dtype=torch.float64), pad_sequence(text_list, pa

from torch.utils.data.dataset import random_split
from torchtext.data.functional import to_map_style_dataset

train_iter, test_iter = IMDB()
train_dataset = to_map_style_dataset(train_iter)
test_dataset = to_map_style_dataset(test_iter)
num_train = int(len(train_dataset) * 0.95)
split_train_, split_valid_ = \
    random_split(train_dataset, [num_train, len(train_dataset) - num_train])

train_loader = DataLoader(split_train_, batch_size=batch_size,
                              shuffle=True, collate_fn=collate_batch)
valid_loader = DataLoader(split_valid_, batch_size=batch_size,
                              shuffle=True, collate_fn=collate_batch)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
                             shuffle=True, collate_fn=collate_batch)

#explicitly initialize weights for better learning
def initialize_weights(m):
    if isinstance(m, nn.Linear):
        nn.init.xavier_normal_(m.weight)
        nn.init.zeros_(m.bias)
    elif isinstance(m, nn.RNN):
        for name, param in m.named_parameters():
            if 'bias' in name:
                nn.init.zeros_(param)
            elif 'weight' in name:
                nn.init.orthogonal_(param) #<---here

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

def train(model, loader, optimizer, criterion):
    epoch_loss = 0
    epoch_acc = 0
    model.train() #useful for batchnorm and dropout
```

```
    for i, (label, text, text_length) in enumerate(loader):
        label = label.to(device) #(batch_size, )
        text = text.to(device) #(batch_size, seq len)

        #predict
        predictions = model(text, text_length) #output by the fc is (batch_size, 1),
        predictions = predictions.squeeze(1)

        #calculate loss
        loss = criterion(predictions, label)
        acc = binary_accuracy(predictions, label)

        #backprop
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

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

        if i == 10:
            break

    return epoch_loss / len(loader), epoch_acc / len(loader)

def evaluate(model, loader, criterion):
    epoch_loss = 0
    epoch_acc = 0
    model.eval()

    with torch.no_grad():
        for i, (label, text, text_length) in enumerate(loader):
            label = label.to(device) #(batch_size, )
            text = text.to(device) #(batch_size, seq len)

            predictions = model(text, text_length)
            predictions = predictions.squeeze(1)

            loss = criterion(predictions, label)
            acc = binary_accuracy(predictions, label)

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

            if i == 10:
                break

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

1). LSTM

We have learned in class that LSTM was designed to avoid the long term dependency problem as well as to helps with the problem of vanishing and exploding gradients.

The key to LSTM is the cell state and the 3 gates to 'protect' and 'control' the cell states.

We will now look in to the components inside and implement them line by line :)

The expected shape of LSTM input is SHAPE : (bs, seq_len, input_dim)

For **EACH** time step of our sequence, these are the operations inside LSTM cell.

The first step in our LSTM is to decide what information we're going to throw away from the previous cell state. This decision is made by a sigmoid layer called the "forget gate layer." It looks at $\mathbf{h}_{t-1}$ and $\mathbf{x}_t$, and outputs a number between 0 and 1. A 1 represents "completely keep this" while a 0 represents "completely get rid of this."

$$\mathbf{f}_t = \sigma(\mathbf{W}_f \ \mathbf{h}_{t-1} + \mathbf{U}_f \ \mathbf{x}_t + \mathbf{b}_f)$$

The next step is to decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the "input gate layer" decides which values we'll update.

$$\mathbf{i}_t = \sigma(\mathbf{W}_i \ \mathbf{h}_{t-1} + \mathbf{U}_i \ \mathbf{x}_t + \mathbf{b}_i)$$

Next, a tanh layer creates a vector of new 'candidate' values, $\tilde{\mathbf{c}}_t$ (aka. $\mathbf{g}_t$), that could be added to the state. In the next step, we'll combine these two to create an update to the state.

$$\mathbf{g}_t = \mathbf{tanh} \ (\mathbf{W}_g \ \mathbf{h}_{t-1} + \mathbf{U}_g \ \mathbf{x}_t + \mathbf{b}_g)$$

It's now time to update the old cell state, $\mathbf{c}_{t-1}$, into the new cell state $\mathbf{c}_t$. The previous steps already decided what to do, we just need to actually do it. We multiply the old state by f_t, forgetting the things we decided to forget earlier. Then we add $\mathbf{i}_t \circ \mathbf{g}_t$. This is the new candidate values, scaled by how much we decided to update each state value.

$$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \mathbf{g}_t$$

Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanh (to push the values to be between −1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

$$\mathbf{o}_t = \sigma(\mathbf{W}_o \ \mathbf{h}_{t-1} + \mathbf{U}_o \ \mathbf{x}_t + \mathbf{b}_o)$$

$$\mathbf{h}_t = \mathbf{o}_t \circ \tanh \ (\mathbf{c}_t)$$

In conclusion, these are the formula that we need to implement in our LSTM_cell class :

$$\mathbf{f}_t = \sigma(\mathbf{W}_f \ \mathbf{h}_{t-1} + \mathbf{U}_f \ \mathbf{x}_t + \mathbf{b}_f)$$

$$\mathbf{i}_t = \sigma(\mathbf{W}_i \ \mathbf{h}_{t-1} + \mathbf{U}_i \ \mathbf{x}_t + \mathbf{b}_i)$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_o \ \mathbf{h}_{t-1} + \mathbf{U}_o \ \mathbf{x}_t + \mathbf{b}_o)$$

$$\mathbf{g}_t = \tanh \ (\mathbf{W}_g \ \mathbf{h}_{t-1} + \mathbf{U}_g \ \mathbf{x}_t + \mathbf{b}_g)$$

$$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \mathbf{g}_t$$

$$\mathbf{h}_t = \mathbf{o}_t \circ \tanh \ (\mathbf{c}_t)$$

where

$\mathbf{h}_{t-1}$ is the hidden state from the previous time step [SHAPE : (bs, hidden_dim)] << no seq_len because it is only at time step t-1

$\mathbf{x}_t$ is the input of the current time step [SHAPE : (bs, hidden_dim)] << no seq_len because it is only at time step t

$\mathbf{W}$ are the weights that would be multiply with the hidden states [SHAPE : (hidden_dim, hidden_dim)]

$\mathbf{U}$ are the weights that would be multiply with the inputs [SHAPE : (input_dim, hidden_dim)]

$\mathbf{b}$ are the biases that would be added to the values before they are passed to sigmoid or tanh [SHAPE : (hidden_dim)]

$\circ$ is the Hadamard product as known as element-wise multiplication

** $\tilde{\mathbf{c}}_t$ and $\mathbf{g}_t$ can be used interchangeably

1.1) Please implement the following LSTM_cell class

In [4]:

```
import math
class LSTM_cell(nn.Module):
    def __init__(self, input_dim: int, hidden_dim: int):
        super().__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim

        # initialise the trainable Parameters
        # These should be torch Parameter which is trainable ! (not just a simple te
        self.W_i = nn.Parameter(torch.Tensor(hidden_dim,hidden_dim))
        self.U_i = nn.Parameter(torch.Tensor(input_dim,hidden_dim))
        self.b_i = nn.Parameter(torch.Tensor(hidden_dim))

        self.W_f = nn.Parameter(torch.Tensor(hidden_dim,hidden_dim))
        self.U_f = nn.Parameter(torch.Tensor(input_dim,hidden_dim))
        self.b_f = nn.Parameter(torch.Tensor(hidden_dim))

        self.W_g = nn.Parameter(torch.Tensor(hidden_dim,hidden_dim))
        self.U_g = nn.Parameter(torch.Tensor(input_dim,hidden_dim))
        self.b_g = nn.Parameter(torch.Tensor(hidden_dim))

        self.W_o = nn.Parameter(torch.Tensor(hidden_dim,hidden_dim))
        self.U_o = nn.Parameter(torch.Tensor(input_dim,hidden_dim))
        self.b_o = nn.Parameter(torch.Tensor(hidden_dim))
        self.init_weights()

    def init_weights(self):
        stdv = 1.0 / math.sqrt(self.hidden_dim)
        for weight in self.parameters():
            weight.data.uniform_(-stdv, stdv)

    def forward(self, x, init_states=None):
        """
        x.shape =  (bs, seq_len, input_dim)
        """
        bs, seq_len, _ = x.shape
        output = []

        # initialize the hidden state and cell state for the first time step
        if init_states is None:
            h_t  = torch.zeros(bs, self.hidden_dim).to(x.device)
            c_t  = torch.zeros(bs, self.hidden_dim).to(x.device)
        else:
            h_t, c_t = init_states

        # For each time step of the input x, do ...
        for t in range(seq_len):
            x_t = x[:, t, :] # get x data of time step t (SHAPE: (batch_size, input_

            i_t = torch.sigmoid(x_t @ self.U_i + h_t @ self.W_i + self.b_i) # SHAPE:
            f_t = torch.sigmoid(x_t @ self.U_f + h_t @ self.W_f + self.b_f)# SHAPE:
            g_t = torch.tanh(x_t @ self.U_g + h_t @ self.W_g + self.b_g) # SHAPE: (b
            o_t = torch.sigmoid(x_t @ self.U_o + h_t @ self.W_o + self.b_o) # SHAPE:
            c_t = f_t * c_t + i_t * g_t # SHAPE: (batch_size, hidden_dim)
            h_t = o_t * torch.tanh(c_t) # SHAPE: (batch_size, hidden_dim)
            output.append( h_t.unsqueeze(0)) # reshape h_t to (1, batch_size, hidden

        output = torch.cat(output, dim=0) # concatenate h_t of all time steps into S
        output = output.transpose(0, 1).contiguous() # just transpose to SHAPE :(seq
```

```
        return output, (h_t, c_t)
```

Run this cell to check if your LSTM Cell can run

In [5]:

```
my_LSTM_cell = LSTM_cell(embed_dim, hidden_dim).to(device)

test_data = torch.ones((batch_size, 100, embed_dim)).to(device)
output, (h_t, c_t) = my_LSTM_cell(test_data)

assert output.shape == torch.Size([32, 100, 256])
assert h_t.shape    == torch.Size([32, 256])
assert c_t.shape    == torch.Size([32, 256])
```

Variants of LSTM

Many variants of LSTM have been developed which are slightly different from Vanilla/Basic LSTM that we have just implemented above

- Peephole LSTM

One popular LSTM variant, introduced by Gers & Schmidhuber (2000), is adding “Peephole Connections.” This means that we let all the gate layers look at the cell state.

$$\mathbf{f}_t = \sigma(\mathbf{W}_f\ \mathbf{h}_{t-1} + \mathbf{U}_f\ \mathbf{x}_t + \mathbf{P}_f\ \mathbf{c}_t + \mathbf{b}_f)$$

$$\mathbf{i}_t = \sigma(\mathbf{W}_i\ \mathbf{h}_{t-1} + \mathbf{U}_i\ \mathbf{x}_t + \mathbf{P}_i\ \mathbf{c}_t + \mathbf{b}_i)$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_o\ \mathbf{h}_{t-1} + \mathbf{U}_o\ \mathbf{x}_t + \mathbf{P}_o\ \mathbf{c}_t + \mathbf{b}_o)$$

$$\mathbf{g}_t = \tanh(\mathbf{W}_g\ \mathbf{h}_{t-1} + \mathbf{U}_g\ \mathbf{x}_t + \mathbf{b}_g)$$

$$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \mathbf{g}_t$$

$$\mathbf{h}_t = \mathbf{o}_t \circ \tanh(\mathbf{c}_t)$$

We can see that every gate now has $\mathbf{c}_t$ as their input. And we also have 3 new parameters; $\mathbf{P}_f$, $\mathbf{P}_i$ and $\mathbf{P}_o$ which has the same shape as $\mathbf{W}$.

- Coupled LSTM

Another variation is to use Coupled forget and input gates. Instead of separately deciding what to forget and what we should add new information to, we make those decisions together. We only forget when we’re going to input something in its place. We only input new values to the state when we forget something older. The different is very simple. The input gate is now $(1 - \mathbf{f}_t)$

$$\mathbf{f}_t = \sigma(\mathbf{W}_f\ \mathbf{h}_{t-1} + \mathbf{U}_f\ \mathbf{x}_t + \mathbf{b}_f)$$

$$\mathbf{i}_t = (1 - \mathbf{f}_t)$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_o\ \mathbf{h}_{t-1} + \mathbf{U}_o\ \mathbf{x}_t + \mathbf{b}_o)$$

$$\mathbf{g}_t = \tanh(\mathbf{W}_g\ \mathbf{h}_{t-1} + \mathbf{U}_g\ \mathbf{x}_t + \mathbf{b}_g)$$

$$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \mathbf{g}_t$$

$$\mathbf{h}_t = \mathbf{o}_t \circ \tanh\left(\mathbf{c}_t\right)$$

1.2) Modify the 'LSTM_cell' class from 1.1 such that we can choose to use Vanilla / Peephole / Coupled LSTM

In [6]:

```
# <Put your modified 'new_LSTM_cell' class here>

class new_LSTM_cell(nn.Module):
    def __init__(self, input_dim: int, hidden_dim: int, lstm_type: str):
        super().__init__()

        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.lstm_type = lstm_type

        # initialise the trainable Parameters
        # These should be torch Parameter which is trainable ! (not just a simple te
        self.W_i = nn.Parameter(torch.Tensor(hidden_dim,hidden_dim))
        self.U_i = nn.Parameter(torch.Tensor(input_dim,hidden_dim))
        self.b_i = nn.Parameter(torch.Tensor(hidden_dim))
        self.P_i = nn.Parameter(torch.Tensor(hidden_dim,hidden_dim))

        self.W_f = nn.Parameter(torch.Tensor(hidden_dim,hidden_dim))
        self.U_f = nn.Parameter(torch.Tensor(input_dim,hidden_dim))
        self.b_f = nn.Parameter(torch.Tensor(hidden_dim))
        self.P_f = nn.Parameter(torch.Tensor(hidden_dim,hidden_dim))

        self.W_g = nn.Parameter(torch.Tensor(hidden_dim,hidden_dim))
        self.U_g = nn.Parameter(torch.Tensor(input_dim,hidden_dim))
        self.b_g = nn.Parameter(torch.Tensor(hidden_dim))

        self.W_o = nn.Parameter(torch.Tensor(hidden_dim,hidden_dim))
        self.U_o = nn.Parameter(torch.Tensor(input_dim,hidden_dim))
        self.b_o = nn.Parameter(torch.Tensor(hidden_dim))
        self.P_o = nn.Parameter(torch.Tensor(hidden_dim,hidden_dim))


        self.init_weights()

    def init_weights(self):
        stdv = 1.0 / math.sqrt(self.hidden_dim)
        for weight in self.parameters():
            weight.data.uniform_(-stdv, stdv)

    def forward(self, x, init_states=None):
        """
        x.shape =  (batch_size, sequence_size, input_size)

        """

        bs, seq_len, _ = x.shape
        output = []

        # initialize the hidden state and cell state for the first time step
        if init_states is None:
            h_t  = torch.zeros(bs, self.hidden_dim).to(x.device)
            c_t  = torch.zeros(bs, self.hidden_dim).to(x.device)
        else:
            h_t, c_t = init_states

        # For each time step of the input x, do ...
        for t in range(seq_len):
            x_t = x[:, t, :] # get x data of time step t (SHAPE: (batch_size, input_
```

```
            if self.lstm_type == "vanilla":
                i_t = torch.sigmoid(x_t @ self.U_i + h_t @ self.W_i + self.b_i)
                f_t = torch.sigmoid(x_t @ self.U_f + h_t @ self.W_f + self.b_f)
                g_t = torch.tanh(x_t @ self.U_g + h_t @ self.W_g + self.b_g)
                o_t = torch.sigmoid(x_t @ self.U_o + h_t @ self.W_o + self.b_o)
                c_t = f_t * c_t + i_t * g_t
                h_t = o_t * torch.tanh(c_t)
                #print("vanilla")

            elif self.lstm_type == "peephole":
                i_t = torch.sigmoid(x_t @ self.U_i + h_t @ self.W_i + c_t @ self.P_i
                f_t = torch.sigmoid(x_t @ self.U_f + h_t @ self.W_f + c_t @ self.P_f
                g_t = torch.tanh(x_t @ self.U_g + h_t @ self.W_g + self.b_g)
                o_t = torch.sigmoid(x_t @ self.U_o + h_t @ self.W_o + c_t @ self.P_c
                c_t = f_t * c_t + i_t * g_t
                h_t = o_t * torch.tanh(c_t)
                #print("peephole")

            elif self.lstm_type == "coupled":
                f_t = torch.sigmoid(x_t @ self.U_f + h_t @ self.W_f + self.b_f)
                i_t = (1 - f_t)
                g_t = torch.tanh(x_t @ self.U_g + h_t @ self.W_g + self.b_g)
                o_t = torch.sigmoid(x_t @ self.U_o + h_t @ self.W_o + self.b_o)
                c_t = f_t * c_t + i_t * g_t
                h_t = o_t * torch.tanh(c_t)
                #print("coupled")

            else:
                raise ValueError('lstm type must be one of the followings: "vanilla'


            output.append( h_t.unsqueeze(0))

        output = torch.cat(output, dim=0)
        output = output.transpose(0, 1).contiguous()

        return output, (h_t, c_t)
```

Run this cell to check if all types of your LSTM Cells can run

In [7]:

```
Vanilla_LSTM_cell = new_LSTM_cell(embed_dim, hidden_dim, lstm_type = 'vanilla').to(c
test_data = torch.ones((batch_size, 100, embed_dim)).to(device)
output, (h_t, c_t) = Vanilla_LSTM_cell(test_data)
assert output.shape == torch.Size([32, 100, 256])
assert h_t.shape    == torch.Size([32, 256])
assert c_t.shape    == torch.Size([32, 256])

Coupled_LSTM_cell = new_LSTM_cell(embed_dim, hidden_dim, lstm_type = 'coupled').to(c
test_data = torch.ones((batch_size, 100, embed_dim)).to(device)
output, (h_t, c_t) = Coupled_LSTM_cell(test_data)
assert output.shape == torch.Size([32, 100, 256])
assert h_t.shape    == torch.Size([32, 256])
assert c_t.shape    == torch.Size([32, 256])

Peephole_LSTM_cell = new_LSTM_cell(embed_dim, hidden_dim, lstm_type = 'peephole').tc
test_data = torch.ones((batch_size, 100, embed_dim)).to(device)
output, (h_t, c_t) = Peephole_LSTM_cell(test_data)
assert output.shape == torch.Size([32, 100, 256])
assert h_t.shape    == torch.Size([32, 256])
assert c_t.shape    == torch.Size([32, 256])
```

BiLSTM model for sequence classification

We now have the basic variants of LSTM cells. But what about Bidirectional LSTM. How do we implement that?

The answer is simple. We create **2 LSTM cells** then pass our normal input to one of them, and pass the **flipped** input to the other. (reverse the order of sequence)

Then we take the last hidden state from the 2 LSTM (one would be the hidden state at the last word of the sentence and another at the first word of the sentence) and concatenate them. Like this we have information of the sequence from both directions!

Formally these are the formula

$$\overrightarrow{\mathbf{h}}_t = LSTM(\mathbf{x}_t, \overrightarrow{\mathbf{h}}_{t-1})$$

$$\overleftarrow{\mathbf{h}}_t = LSTM(\mathbf{x}_t, \overleftarrow{\mathbf{h}}_{t+1})$$

$$\mathbf{h}_t = \sigma(\mathbf{W}_y[\overrightarrow{\mathbf{h}}_t; \overleftarrow{\mathbf{h}}_t] + \mathbf{b}_y)$$

Then we should pass $\mathbf{h}_t$ to another Linear Layer to get the output for binary classification.

1.3) Implement the following 'BiLSTM_model' class

It should be a model for sequence classification which only has BiLSTM as its encoder and a Linear Layer for outputting the binary classification class decision.

(Let's use our 'vanilla' LSTM_cell)

In [8]:

```
class BiLSTM_model(nn.Module):
    def __init__(self, input_dim: int, embed_dim: int, hidden_dim: int, output_dim:
        super().__init__()
        self.num_directions = 2
        self.embedding = nn.Embedding(input_dim, embed_dim, padding_idx=pad_idx)
        self.input_dim = input_dim
        self.embed_dim = embed_dim
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim


        self.forward_lstm   =  LSTM_cell(input_dim = embed_dim, hidden_dim = hidden_
        self.backward_lstm  =  LSTM_cell(input_dim = embed_dim, hidden_dim = hidden_


        # These should be torch Parameters
        self.W_h =  nn.Parameter(torch.Tensor(hidden_dim *self.num_directions , outp
        self.b_h =  nn.Parameter(torch.Tensor(hidden_dim * self.num_directions))# SH

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

        self.init_weights()

    def init_weights(self):
        stdv = 1.0 / math.sqrt(self.hidden_dim)
        for weight in self.parameters():
            weight.data.uniform_(-stdv, stdv)

    def forward(self, text, text_lengths):
        embedded      = self.embedding(text) # SHAPE : (batch_size, seq_len, embed_c

        embedded_flip =  torch.flip(embedded, [1])  # SHAPE : (batch_size, seq_len,

        output_forward, (hn_forward, cn_forward)    = self.forward_lstm(embedded) #
        output_backward, (hn_backward, cn_backward) = self.backward_lstm(embedded_fl

        concat_hn = torch.hstack((hn_forward,hn_backward))  # SHAPE : (batch_size, h
        ht        = torch.sigmoid( concat_hn @ self.W_h + self.b_h) # SHAPE : (batch



        return self.fc(ht)
```

Run this cell to show that you can train the model with your BiLSTM Model

In [9]:

```
import torch.optim as optim
bilstm = BiLSTM_model(input_dim, embed_dim, hidden_dim, output_dim).to(device)
bilstm.apply(initialize_weights)
bilstm.embedding.weight.data = fast_embedding

optimizer = optim.Adam(bilstm.parameters(), lr=lr) #<----changed to Adam
criterion = nn.BCEWithLogitsLoss() #combine sigmoid with binary cross entropy

train_losses = []
train_accs = []
valid_losses = []
valid_accs = []

for epoch in range(num_epochs):
    train_loss, train_acc = train(bilstm, train_loader, optimizer, criterion)
    valid_loss, valid_acc = evaluate(bilstm, valid_loader, criterion)

    train_losses.append(train_loss)
    train_accs.append(train_acc)
    valid_losses.append(valid_loss)
    valid_accs.append(valid_acc)

    print(f'Epoch: {epoch+1:02} | Train Loss: {train_loss:.3f} | Train Acc: {train_a
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')

del bilstm
del optimizer
del criterion
```

```
Epoch: 01 | Train Loss: 0.010 | Train Acc: 0.82%
         Val. Loss: 0.194 |  Val. Acc: 13.75%
Epoch: 02 | Train Loss: 0.010 | Train Acc: 0.80%
         Val. Loss: 0.193 |  Val. Acc: 13.12%
Epoch: 03 | Train Loss: 0.010 | Train Acc: 0.73%
         Val. Loss: 0.191 |  Val. Acc: 14.14%
```

As you can see, the 6 equations in our LSTM cell means there are at least 6 matrix multiplication operations for each time step in each of our input sequence, which is A LOT. **But pytorch has the optimized version of LSTM which is much more efficient so let's use that in the next parts.**

2.) Attention Mechanism

The attention mechanism was first born to help memorize long source sentences in neural machine translation (NMT). Rather than building a single context vector out of the encoder's last hidden state, the attention mechanism creates shortcuts between the context vector and the entire source input. The weights of these shortcut connections are customizable for each output element. While the context vector has access to the entire input sequence, we don't need to worry about forgetting. The alignment between the source and target is learned and controlled by the context vector. Essentially the context vector consumes three pieces of information:

```
- encoder hidden states
- decoder hidden states
- alignment between source and target
```

This is the same mechanism that we have learned in class, which is actually called 'Cross Attention'.

However, in this assignment we are making a classification model so we only have the encoder hidden states and our target would be the class decision.

General Attention Mechanism

First, we will be creating an LSTM + General Attention model for classification. Which will be a little bit different from what Prof has taught in class, such that we only have an encoder and we don't have any decoder. In this task, we are going to use LSTM as the encoder and use General Attention before we output the class decision.

Our General Attention mechanism is going to capture how the last encoder hidden state (aka. the 'queries') 'relates' to the other hidden states in the sequence (a.k.a. the 'keys'). (how much our classification decision is related to each of the hidden states) Then we will scale the output (a.k.a. the 'values') according to the Attention Weights (computed from the Alignment Scores), in order to retain focus on words that are relevant to the query. In doing so, it produces an attention output that we will input to a fully connected layer for the result of our classification task.

These are the steps we need to implement :

We will pass our data through LSTM first then pass the outputs of LSTM to the General Attention mechanism.

1. Get the components we need for our Attention Mechanism ('query', 'keys' and 'values')

- Get the last encoder hidden states ($\mathbf{h}_N$) = last hidden state of last LSTM layer

```
- Hint : can be found in 'hn'
- Should be of shape [bs, hidden dim * num_directions]
- a.k.a. 'query'
```

- Get the hidden states of every time step from the last layer of LSTM ($\mathbf{H}$)

```
- Hint : can be found in 'output'
- Should be of shape  [bs, seq len, hidden_dim * num_directions]
- a.k.a. 'keys'
- This will be matched with our h_t to get the Attention Scores.
```

- Get the hidden states of every time step from the last layer of LSTM ($\mathbf{H}$)

```
- Hint : can be found in 'output'
- Should be of shape  [bs, seq len, hidden_dim * num_directions]
- a.k.a. 'values'
- This will be weighted by Attention Weights to get the Context.
```

- - In our case, we are implementing Attention in Classification model so our 'keys' and 'values' are the same thing

2. Calculate Alignment Scores:

Calculate the Alignment Scores by matching the **'query'** with each of the **'keys'**. This matching operation is computed as the **dot product** of our specific 'query' with each of the hidden states or the 'key' vector. This is to get the scores of how 'related' the 'query' is to each 'key' or each hidden state.

$\mathbf{e}_t = [\mathbf{h}_N^T \, \mathbf{h}_1, \mathbf{h}_N^T \, \mathbf{h}_2, \ldots, \mathbf{h}_N^T \, \mathbf{h}_N] \in \mathbb{R}^N$

where

$\mathbf{h}_1, \ldots, \mathbf{h}_N \in \mathbb{R}^h; \mathbf{H} \in \mathbb{R}^{N,h}$

Hint : We can multiply our 'query' with all of the 'keys' at once by using the matrix form of the 'query' ($\mathbf{H}$) (we have to keep shape of batch size at the first dimension so **torch.bmm** might come in handy !)

Hint2: Alignment Scores should be of shape : [batch_size, seq_len, 1]

3. Calculate Attention Weights :

We pass the Alignment Scores through a **softmax** operation to convert the scores into probabilities called the 'Attention Weights' This method is called **soft-attention** which help make the model smooth and differentiable.

$\alpha_t = \text{softmax}(\mathbf{e}_t) \in \mathbb{R}^N$

Hint : our softmaxed Attention Weights should still have the same shape as Alignment Score

4. Calculate Context Vector :

Use the Attention Weight to scale the output **'values'** to get the 'context vector'. In this example, the 'values' is the same as the 'keys' which is the hidden states of every time step from the last layer of LSTM. A context vector is a **weighted sum** of the value vectors, V_ki.

$\mathbf{c}_t = \mathbf{H}^T \alpha_t \in \mathbb{R}^h$

Hint : Again, we can use the matrix form to get the weighted sum in one operation. The resulting context should be of shape [bs, hidden_size * num_directions]

5. Finally, we use this Context Vector as the output of our Attention Mechanism

2.1) Implement the following LSTM + General Attention class

In [10]:

```
import torch.nn as nn
from torch.nn import functional as F

class LSTM_GAtt(nn.Module):
    def __init__(self, input_dim: int, embed_dim: int, hidden_dim: int, output_dim:
        super().__init__()

        self.embedding = nn.Embedding(input_dim, embed_dim, padding_idx=pad_idx)

        # let's use pytorch's LSTM
        self.lstm = nn.LSTM(embed_dim,
                            hidden_dim,
                            num_layers=num_layers,
                            bidirectional=bidirectional,
                            dropout=dropout,
                            batch_first=True)

        # Linear Layer for binary classification
        self.fc = nn.Linear(hidden_dim * 2, output_dim)

        self.softmax = nn.Softmax(dim=-1)

    def attention_net(self, lstm_output, hn):

        h_t      = hn.clone().detach() # last hidden state of last layer (Hint : car
        # use torch.clone to copy tensors safely
        H_keys   = lstm_output.clone().detach() # hidden states of every time step i
        H_values = lstm_output.clone().detach() # hidden states of every time step i

        alignment_score   = torch.bmm(H_keys,h_t.unsqueeze(2)) # SHAPE : (bs, seq_le

        soft_attn_weights = self.softmax(alignment_score) # SHAPE : (bs, seq_len, 1)

        context           = torch.bmm(H_keys.reshape(H_keys.shape[0], H_keys.shape[2

        return context

    def forward(self, text, text_lengths):

        embedded = self.embedding(text) # SHAPE : (batch_size, seq_len, embed_dim)

        lstm_output, (hn, cn) = self.lstm(embedded)

        # This is how we concatenate the forward hidden and backward hidden from Pyt
        hn = torch.cat((hn[-2,:,:], hn[-1,:,:]), dim = 1)

        attn_output = self.attention_net(lstm_output, hn)

        return self.fc(attn_output)
```

Run this cell to show that you can train the model with your LSTM_GAtt Model

In [11]:

```
g_attmodel = LSTM_GAtt(input_dim, embed_dim, hidden_dim, output_dim).to(device)
g_attmodel.apply(initialize_weights)
g_attmodel.embedding.weight.data = fast_embedding

optimizer = optim.Adam(g_attmodel.parameters(), lr=lr) #<----changed to Adam
criterion = nn.BCEWithLogitsLoss() #combine sigmoid with binary cross entropy

train_losses = []
train_accs = []
valid_losses = []
valid_accs = []

for epoch in range(num_epochs):
    train_loss, train_acc = train(g_attmodel, train_loader, optimizer, criterion)
    valid_loss, valid_acc = evaluate(g_attmodel, valid_loader, criterion)

    train_losses.append(train_loss)
    train_accs.append(train_acc)
    valid_losses.append(valid_loss)
    valid_accs.append(valid_acc)

    print(f'Epoch: {epoch+1:02} | Train Loss: {train_loss:.3f} | Train Acc: {train_a
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')

del g_attmodel
del optimizer
del criterion
```

```
Epoch: 01 | Train Loss: 0.017 | Train Acc: 0.73%
         Val. Loss: 0.276 |  Val. Acc: 13.20%
Epoch: 02 | Train Loss: 0.015 | Train Acc: 0.70%
         Val. Loss: 0.233 |  Val. Acc: 13.20%
Epoch: 03 | Train Loss: 0.011 | Train Acc: 0.80%
         Val. Loss: 0.198 |  Val. Acc: 14.30%
```

Self Attention Mechanism

Self-attention, also known as intra-attention, is an attention mechanism relating different positions of a single sequence in order to compute a representation of the same sequence. The self-attention mechanism allows the inputs to interact with each other ("self") and find out who they should pay more attention to ("attention"). The outputs are aggregates of these interactions and attention scores.

It has been shown to be very useful in machine reading, abstractive summarization, or image description generation.

You might have noticed from the previous part that the are 3 main vector/matrix in the attention mechanism, which are 'queries', 'keys' and 'value'. Self-attention also need the same elements but we only have 'self' for the model to consider so 'queries', 'keys' and 'value' is all made from our input. Other steps are very similar to General Attention.

Same with the previous part, we will pass our data through LSTM first then pass the outputs of LSTM to the Self Attention mechanism.

1. Get the components we need for our Attention Mechanism

Make **3 copies** of $\mathbf{H}$ (hidden states of every time step from the last layer of LSTM)

```
- Hint : can be found in 'output'
- Should be of shape  [bs, seq len, hidden_dim * num_directions]
```

2. Initialize 3 Linear Layers :

```
- Initialize 3 Linear Layer called 'lin_Q', 'lin_K', 'lin_V'

- input_dim = hidden_dim * num_direction
- output_dim = hidden_dim * num_direction
```

3. Pass each copy of lstm_output through each of the Linear Layer.

Pass each copy of $\mathbf{H}$ through each of the Linear Layer so that we have learnable weights for generating the queries, keys and values

$\mathbf{Q} = \mathbf{H}^T\ \mathbf{W}_q + \mathbf{b}_q \in \mathbb{R}^{N,h}$

$\mathbf{K} = \mathbf{H}^T\ \mathbf{W}_k + \mathbf{b}_k \in \mathbb{R}^{N,h}$

$\mathbf{V} = \mathbf{H}^T\ \mathbf{W}_v + \mathbf{b}_v \in \mathbb{R}^{N,h}$

*Hint: Expected SHAPE : (bs, seq_len, n_hidden * num_directions)

4. Calculate Alignment Scores:

```
- Matching the 'query' with the 'keys'.
```

$\text{AlignmentScore} = \mathbf{Q}\ \mathbf{K}^T \in \mathbb{R}^{N,N}$

```
- Hint: Our 'query' and 'keys' are both matrix so you might want to use 'tor
ch's matrix multiplication'.
- Expected SHAPE : (bs, seq_len, seq_len) because we want to match each time
step in self with each time step of itself
```

5. Padding Mask

Since there are many padding tokens in our input sequence. It would be inefficient to leave them as is. Please implement 'pad_mask' which will replace the Alignment Scores with -1e9 where the input sequence is the padding token.

**Skipping this step will not affect the next parts :)

6. Calculate Attention Weights :

```
- Pass the Alignment Scores through Softmax
```

$\text{Attention Weights} = \text{softmax}(\text{AlignmentScore}) \in \mathbb{R}^{N,N}$

7. Calculate the Context Vector :

```
- Multiply the Attention Weights with the 'values' to get the Context vector
of SHAPE : (bs, seq_len, hidden_dim * num_directions)
- Then do 'Sequence Length Reduction' to aggregate the dimension of seq_len
 into 1.
- You can choose between averaging or sum.
- Finally, Context vector should have SHAPE : (bs, hidden_dim * num_directio
ns)
```

$$\text{Context Vector} = \text{Attention Weights } \mathbf{V} \in \mathbb{R}^h$$

8. Finally, we use this Context Vector as the output of our Attention Mechanism

2.2) Implement the following LSTM + Self Attention class

In [12]:

```
class LSTM_SelfAtt(nn.Module):
    def __init__(self, input_dim, embed_dim, hidden_dim, output_dim, len_reduction):
        super().__init__()

        self.embedding = nn.Embedding(input_dim, embed_dim, padding_idx=pad_idx)
        self.len_reduction = len_reduction

        # let's use pytorch's LSTM
        self.lstm = nn.LSTM(embed_dim,
                           hidden_dim,
                           num_layers=num_layers,
                           bidirectional=bidirectional,
                           dropout=dropout,
                           batch_first=True)

        self.softmax        = nn.LogSoftmax(dim=1)

        self.lin_Q = nn.Linear(hidden_dim * 2,hidden_dim * 2 )
        self.lin_K = nn.Linear(hidden_dim * 2,hidden_dim * 2 )
        self.lin_V = nn.Linear(hidden_dim * 2,hidden_dim * 2 )

        # Linear Layer for binary classification
        self.fc     = nn.Linear(hidden_dim * 2, output_dim)

    def self_attention_net(self, lstm_output):

        Q = self.lin_Q(lstm_output.clone().detach()) # SHAPE : (bs, seq_len, n_hidde
        K = self.lin_K(lstm_output.clone().detach()) # SHAPE : (bs, seq_len, n_hidde
        V = self.lin_V(lstm_output.clone().detach()) # SHAPE : (bs, seq_len, n_hidde

        alignment_score = torch.bmm(Q,K.transpose(1,2)) # SHAPE : (bs, seq_len, seq_

        # apply padding mask
        if self.mask:
            batch_size, seq_len = self.text.size()
            pad_mask = self.text.data.eq(0).unsqueeze(1)
            alignment_score.masked_fill_(pad_mask.expand(batch_size, seq_len, seq_le


        soft_attn_weights = self.softmax(alignment_score)

        context = torch.bmm(soft_attn_weights,V)  # SHAPE : (bs, seq_len, hidden_dim

        # Do Mean or Sum len reduction
        if self.len_reduction == "mean":
            len_reduced_context = torch.mean(context, dim=1), soft_attn_weights.cpu(
        elif self.len_reduction == "sum":
            len_reduced_context =  torch.sum(context, dim=1), soft_attn_weights.cpu(

        return len_reduced_context

    def forward(self, text, text_lengths, mask =True):

        self.mask = mask
        self.text = text

        embedded = self.embedding(text) # SHAPE : (batch_size, seq_len, embed_dim)

        lstm_output, (hn, cn) = self.lstm(embedded)
```

```
        # This is how we concatenate the forward hidden and backward hidden from Pyt
        hn = torch.cat((hn[-2,:,:], hn[-1,:,:]), dim = 1)

        attn_output, attention = self.self_attention_net(lstm_output)

        return self.fc(attn_output)
```

Run this cell to show that you can train the model with your LSTM_SelfAtt Model

In [13]:

```
self_attmodel = LSTM_SelfAtt(input_dim, embed_dim, hidden_dim, output_dim, len_reduc
self_attmodel.apply(initialize_weights)
self_attmodel.embedding.weight.data = fast_embedding

optimizer = optim.Adam(self_attmodel.parameters(), lr=lr) #<----changed to Adam
criterion = nn.BCEWithLogitsLoss() #combine sigmoid with binary cross entropy

train_losses = []
train_accs = []
valid_losses = []
valid_accs = []

for epoch in range(num_epochs):
    train_loss, train_acc = train(self_attmodel, train_loader, optimizer, criterion)
    valid_loss, valid_acc = evaluate(self_attmodel, valid_loader, criterion)

    train_losses.append(train_loss)
    train_accs.append(train_acc)
    valid_losses.append(valid_loss)
    valid_accs.append(valid_acc)

    print(f'Epoch: {epoch+1:02} | Train Loss: {train_loss:.3f} | Train Acc: {train_a
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')

del self_attmodel
del optimizer
del criterion
```

```
Epoch: 01 | Train Loss: 0.961 | Train Acc: 0.75%
         Val. Loss: 10.062 |  Val. Acc: 14.61%
Epoch: 02 | Train Loss: 0.785 | Train Acc: 0.71%
         Val. Loss: 10.893 |  Val. Acc: 14.06%
Epoch: 03 | Train Loss: 0.441 | Train Acc: 0.74%
         Val. Loss: 7.228 |  Val. Acc: 14.06%
```

2.3) In our implementation we have used 'Dot-Product' to calculate our Alignment Scores. Give 2 examples of Alignment score functions and their formula (other than the ones in our lecture)

Write your answer here

1. Additive

$\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \mathbf{v_a}^T \tanh(\mathbf{W}_a[\mathbf{s}_t; \mathbf{h}_i])$

2. General

$\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \mathbf{s_t}^T \mathbf{W}_a \mathbf{h}_i$ where $\mathbf{W}_a$ is a trainable weight matrix in the attention layer.

Ref : [https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html (https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html)](https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html)

2.4) As explained in the implementation we used softmax to calculate the 'Soft' Attention weights. Explain how to 'Hard' attention works.

Write your answer here

Using hard attention only returns the value which had the maximum output from the attention mechanism, instead of taking a weighted average.

Reference & Further Readings:

LSTM :

- https://colah.github.io/posts/2015-08-Understanding-LSTMs/ (https://colah.github.io/posts/2015-08-Understanding-LSTMs/)
- https://medium.com/@raghavaggarwal0089/bi-lstm-bc3d68da8bd0 (https://medium.com/@raghavaggarwal0089/bi-lstm-bc3d68da8bd0)

Attention :

- https://arxiv.org/pdf/1409.0473.pdf (https://arxiv.org/pdf/1409.0473.pdf)
- https://towardsdatascience.com/illustrated-self-attention-2d627e33b20a (https://towardsdatascience.com/illustrated-self-attention-2d627e33b20a)
- https://machinelearningmastery.com/the-attention-mechanism-from-scratch/#:~:text=In%20essence%2C%20when%20the%20generalized,the%20others%20in%20the%20se (https://machinelearningmastery.com/the-attention-mechanism-from-scratch/#:~:text=In%20essence%2C%20when%20the%20generalized,the%20others%20in%20the%20se
- https://blog.floydhub.com/attention-mechanism/#bahdanau-att (https://blog.floydhub.com/attention-mechanism/#bahdanau-att)
- https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html (https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html)
- https://www.analyticsvidhya.com/blog/2019/11/comprehensive-guide-attention-mechanism-deep-learning/ (https://www.analyticsvidhya.com/blog/2019/11/comprehensive-guide-attention-mechanism-deep-learning/)

In [ ]:

```

```