Transformers from Scratch with Pytorch

Phuoc Bui, Thuong Dang

November 2023

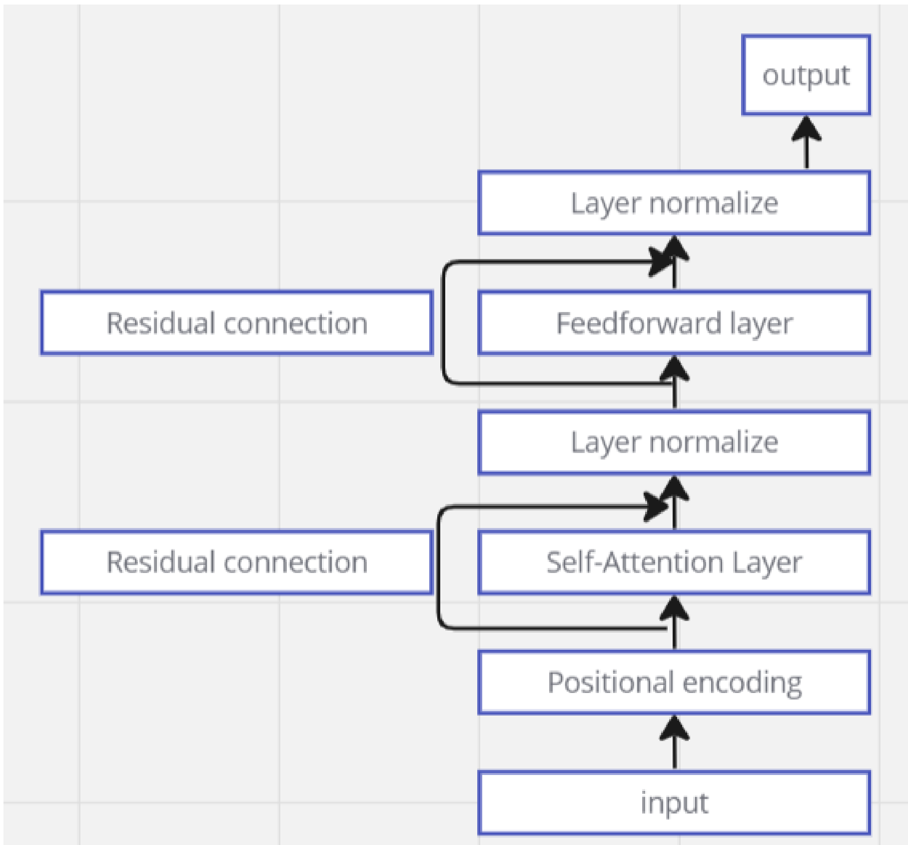
Abstract

This is a documentation for our transformer model from scratch with Pytorch. In our project, we have developed a transformer model for sentiment analysis. The architecture and formulas for transformer blocks are carefully covered in this documentation.

Contents

1 Transformer architecture

Before going to any detail, let us first look at the architecture of a transformer block.



From the picture above, we can see that our input will go through a layer for positions. It then go through the self attention layer, followed by a layer normalize. After that, our data will go through a feedforward layer and a layer normalize before producing the output via the softmax function. There will be several questions:

1. Why transformer models? What are they good for?
2. What is positional encoding?
3. What is self-attention?
4. What is a residual connection?
5. What is a layer normalization?

We will go through the details of those questions. But let us try to answer them quickly. For the first question, previous models such as RNN or LSTM have faced with the *bottleneck* problem, i.e. the context vector contains too much information, and also the vanishing/exploding gradients. An improvement for the bottleneck problem is the *attention mechanism*, which replaces a single context vector by weighted sum of all hidden states (see [1]-Chapter 9 for more details). The vanishing/exploding gradients can be improved by *residual connections*. Residual connections, in short, give us a shortcut through layers and they can reduce the length for the computation of the backpropagation.

Transformers models combine them both, with an important improvement: the *self-attention mechanism*. Moreover, they also solve the positional problem in previous attention models by *positional encoding*. In previous attention models, we can permute our input and the output will also be permuted in the same way. It means the models do not know the order of our input data, and this is not good, for example, for time series applications.

In the next section, let us dive into positional encoding.

2 Positional encoding

As we have said earlier, the attention mechanism does not know the orders of words in an input sentence. For more detail, the reader can look at our second section about the self-attention mechanism. We also note that the content of each section in our documentation is independent and the reader can start reading from Section 2. However, because we want to go with the flow of the model, we will start with positional encoding. To solve the positional problem of attention models, we need to know some important properties of positions:

- Given a sentence, a given position in this sentence uniquely determines a word. It means that for each word in a sentence, its positions must be unique, even if we have the same words in a sentence. For example, in the sentence s = "it is a very very intense situation", the word "very" appears twice, but the positions are different.

- Given two sentences $s_1 = a_1 \cdots a_n w \cdots$, $s_2 = b_1 \cdots b_n w \cdots$ where a_i, b_i, w are words, then the *absolute positions* of w in s_1, s_2 must be the same.

- Given two sentences s_1, s_2 and two words w_1, w_2 in both s_1, s_2, such that $s_1 = \cdots w_1 a_1 \cdots a_n w_2 \cdots$, $s_2 = \cdots w_1 b_1 \cdots b_n w_2 \cdots$, then the *relative position* of w_1 and w_2 in s_1, s_2 must be the same. More generally, for any k and t, the t-th position and the $(t+k)$-th position will be different from a function depending only on k.

- Positions are easy to compute.

Having that in mind, we can understand the positional encoding proposed in the original paper of transformers [3]. Given a word w of position k in a sentence s, we first use a word embedding method to transform this to a vector $v_w = (v_0, \cdots, v_{d-1})$ of length d. We define the *position*

$$P(k, 2i) = \sin\left(\frac{k}{n^{2i/d}}\right), P(k, 2i+1) = \cos\left(\frac{k}{n^{2i/d}}\right)$$

for $0 \le i \le d/2$, and $n = 10000$ in the original paper. And the *positional encoding* is defined to be

$$\text{positional encoding(w)} = v_w + (P(k,0), P(k,1), \cdots)$$

Let us take a look on an example.

Example. Consider the sentence "i am happy", we assume that each word is embedded into a four-dimensional vector space, i.e. a word of position k is represented by a vector $v_k = (v_{k0}, v_{k1}, v_{k2}, v_{k3})$. Consider the word "i", which is of position 0 in the sentence. Then

$$P(0, 2i) = \sin 0 = 0, P(0, 2i+1) = \cos 0 = 1$$

And the positional encoding of the word "i" is

$$(v_{00}, v_{01}, v_{02}, v_{03}) + (0, 1, 0, 1)$$

Next, the word "am" is of position 1, and we have

$$P(1,0) = \sin 1, P(1,1) = \cos 1, P(1,2) = \sin\left(\frac{1}{100}\right), P(1,3) = \cos\left(\frac{1}{100}\right)$$

From this, we have the positional encoding of "am" is

$$(v_{10}, v_{11}, v_{12}, v_{13}) + \left(\sin 1, \cos 1, \sin\frac{1}{100}, \cos\frac{1}{100}\right)$$

Given a sentence s and a word w of position k in s, we will see that the positional encoding satisfies all important properties for position we listed above. First, in order to have the same vector P for two words at different positions k_1, k_2. It means, that for *all* i,

$$\sin\left(\frac{k_1}{n^{2i/d}}\right) = \sin\left(\frac{k_2}{n^{2i/d}}\right)$$

Because of the periodicity of the sin and cos functions, when $i = d/2$, we have

$$k_1 = 2\pi n + k_2,$$

which is impossible because $n = 10000$, and there is no sentence of length 10000.

Second, for the absolute position problem. If a word appears in two sentence in the same absolute position k, it is obvious to see that its positional encoding does not change.

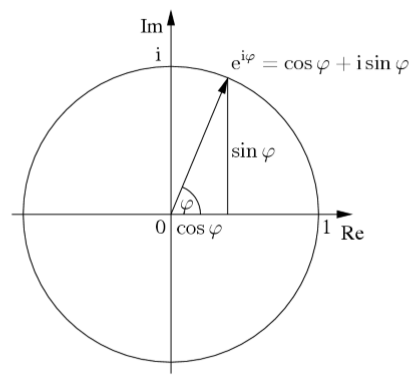
The third problem about relative positions is interesting. Assume that the relative position two words w_1, w_2 in two sentences s_1, s_2 are the same, let's see what we can deduce. We can first see that a pair of (even position, odd position) of the function P at position t can be represented by a complex number

$$e^{i\frac{t}{n^{2i/d}}} = \cos\left(\frac{t}{n^{2i/d}}\right) + i\sin\left(\frac{t}{n^{2i/d}}\right)$$

Here we used Euler's formula

$$e^{i\varphi} = \cos\varphi + i\sin\varphi$$

Here is how we see it in picture



Then at time $k + t$, we have a different complex number

$$e^{i\frac{t+k}{n^{2i/d}}} = \cos\left(\frac{t+k}{n^{2i/d}}\right) + i\sin\left(\frac{t+k}{n^{2i/d}}\right)$$

And we can see that the difference between the two numbers is exactly a *rotation* with angel $\frac{k}{n^{2i/d}}$. Therefore, if the relative positions between w_1 and w_2 in both sentences are k, their difference is exactly a rotation with angel $\frac{k}{n^{2i/d}}$.

The last problem about the computational aspects is also satisfied because computing sin and cos are effectively computed in python libraries, for example, numpy. Here is how positional encoding looks like in codes

```python
# Positional encoding for the whole sentence        You, 24 hours
def positional_encoding(self, inputs):
    pos_encoding = torch.empty(inputs.shape[0],0)
    iter = inputs.shape[1]
    for k in range(iter):
        d = inputs.shape[0]
        pos = torch.zeros((d,1))
        for i in range(d):
            if i % 2 == 0:
                pos[i] = np.sin(k/(self.n**(i/d)))
            if i % 2 == 1:
                pos[i] = np.cos(k/(self.n**((i-1)/d)))
        v_pos = inputs[:, k].reshape(-1,1) + pos
        pos_encoding = torch.cat((pos_encoding, v_pos), dim = 1)
    return pos_encoding
```

3 Self-attention

Self-attention is an instance of attention mechanism. In short, it tells us how much the current input depends on previous inputs. Let us dive into detail. We start with three matrices

- The query matrix W_q.
- The key matrix W_k.
- The value matrix W_v.

Let us understand those matrices this way. Given a tuple of pairs (key, value) of the form $(k_1, v_1), \cdots, (k_T, v_T)$. Given a query q, we will compute how similar between q and each k_i. If the similar between q and k_t is high, it will most likely return v_t. It means, the weight at v_t will also be large. Here is a formula

$$\text{Attention}(q, k, v) = \sum_t \text{similarity}(q, k_t) v_t \tag{1}$$

The similarity function depends on our purposes, but we want it to be non-negative, convex, and $\sum_t \text{similarity}(q, k_t) = 1$. A natural choice is the softmax function.

Let us make it more explicitly, assume that our input X is of dimension $d \times T$. It means that we have T words in our sentence and each word is represented by a d dimensional vector. We can initialize W_q, W_k, W_v of size $h \times d$. We now compute

$$Q = W_q X, K = W_k X, V = W_v X$$

And define

$$\text{Self-attention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$

Let us go deeper into the definition to understand what is going on. First, we need to divide QK^T by $\sqrt{d}$ to prevent exploding by too large exponential values. If we denotes q_i, k_i, v_i rows of W_q, W_k, W_v respectively, and x_i columns of X, then

$$Q = \begin{pmatrix} q_1x_1 & \cdots & q_1x_T \\ \vdots & \vdots & \vdots \\ q_hx_1 & \cdots & q_hx_T \end{pmatrix}$$

$$K = \begin{pmatrix} k_1x_1 & \cdots & k_1x_T \\ \vdots & \vdots & \vdots \\ k_hx_1 & \cdots & k_hx_T \end{pmatrix}$$

$$V = \begin{pmatrix} v_1x_1 & \cdots & v_1x_T \\ \vdots & \vdots & \vdots \\ v_hx_1 & \cdots & v_hx_T \end{pmatrix}$$

And

$$QK^T = \left(\sum_t q_ix_tk_jx_t\right)_{ij},$$

If we apply the softmax function, we can then understand the formula as

$$\text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right) = \left(\sum_t \text{similarity}(q_ix_t, k_jx_t)\right)_{ij}$$

The i-th row in the matrix above is of the form

$$\left(\sum_t \text{similarity}(q_ix_t, k_1x_t), \cdots, \sum_t \text{similarity}(q_ix_t, k_hx_t)\right)$$

And the j-th column in V is of the form

$$\begin{pmatrix} v_1x_j \\ \vdots \\ v_hx_j \end{pmatrix}$$

Multipylying them, we get

$$\left(\text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V\right)_{ij} =$$

$$= \sum_t \text{similarity}(q_ix_t, k_1x_t)(v_1x_j) + \cdots + \sum_t \text{similarity}(q_ix_t, k_hx_t)(v_hx_j)$$

This is very similar to the form of (1) if we consider $q_i x_t$ as a query, $k_u x_t$ as a key and $v_u x_j$ as a value. To summarize, the self attention formula tells us *how much the current input x_j depends on other inputs*. However, the formula does not tell us about the *positions*. If we permute x_j and x_l, the j-th row and the l-th rows of the self-attention matrix will also be permuted as we only change from $v_p x_j$ to $v_p x_l$. And as we already discussed in Section 1, positional encoding can solve this problem.

For time series applications, we want to predict next steps based on previous steps, and not the other way around. And to deal with this, we can modify a bit by adding a masked matrix M to QK^T and then compute softmax. The masked matrix M is of size $h \times h$ and its upper part (not including the diagonal) are all $-\infty$ (so that after the softmax function, those values become 0) and other entries are 0. The self-attention can be now defined

$$\text{Self-attention(Q, K, V)} = \text{softmax}\left(\frac{QK^T}{\sqrt{d}} + M\right)V$$

In codes, we can create the masked matrix in Pytorch as follows

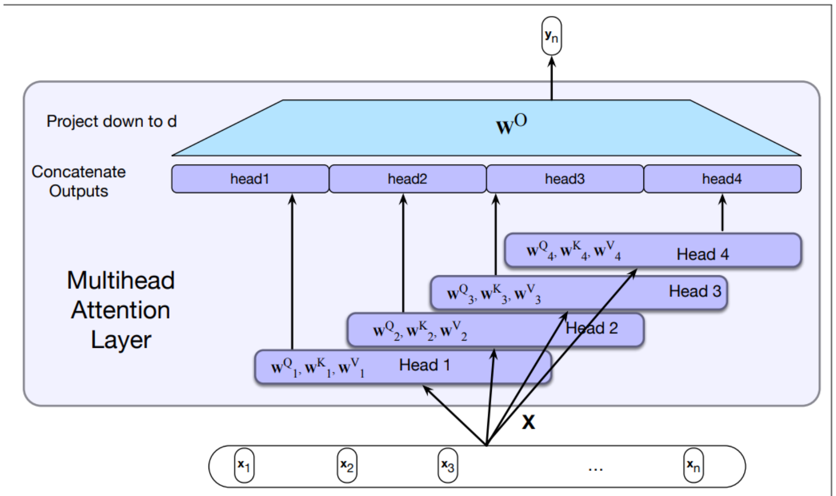
```
# Create the mask matrix with upper part is -infinity
# and other entries are 0
# First we create a matrix filled with - infinity at all entries
matrix = torch.full((self.n_heads*hidden_size, self.n_heads * hidden_size), -float('inf'))
# triu = triangular upper part, we want to fill in upper part with - infinity
matrix = torch.triu(matrix)
# Replace the diagonal by 0
matrix = matrix.fill_diagonal_(0)
self.mask = matrix
```

And here are sample codes for self-attention function

```
def self_attention(mask, Q, K, V):
    att = (Q @ K.T + mask)/np.sqrt(Q.shape[0])
    return (torch.softmax(att, dim = 0) @ V)
```

4 Multihead attention

We have finished the most two important ideas of the transformer mechanism. The multihead attention can help us program parallelly and it gives more parameters to the model. We can look at a picture taken from [1].



Let n be the number of heads, we will initialize n tupe of matrices (W_{qi}, W_{ki}, W_{vi}). Given an input X, each tupe will give us an output

$$h_i = \text{Self-attention}(Q_i, K_i, V_i)$$

And we will *concatenate* them

$$h = h_1 \oplus \cdots \oplus h_n$$

To project h down to the dimension we want, we will multiply it with a matrix W_o. Here is how we do it in codes. We first want to stack parameters for heads and initialize the matrix W_o

```
self.Wq = torch.empty(0, input_size)
self.Wk = torch.empty(0, input_size)
self.Wv = torch.empty(0, input_size)
for i in range(self.n_heads):
    Wq = torch.rand(hidden_size, input_size)/1000
    self.Wq = torch.cat((self.Wq, Wq), dim = 0)
    Wk = torch.rand(hidden_size, input_size)/1000
    self.Wk = torch.cat((self.Wk, Wk), dim = 0)
    Wv = torch.rand(hidden_size, input_size)/1000
    self.Wv = torch.cat((self.Wv, Wv), dim = 0)

self.Wq.requires_grad = True
self.Wk.requires_grad = True
self.Wv.requires_grad = True

# Initialize the matrix Wo for the projection of the multi head layer output
self.Wo = torch.rand(hidden_size, self.n_heads * hidden_size)/1000
self.Wo.requires_grad = True
```

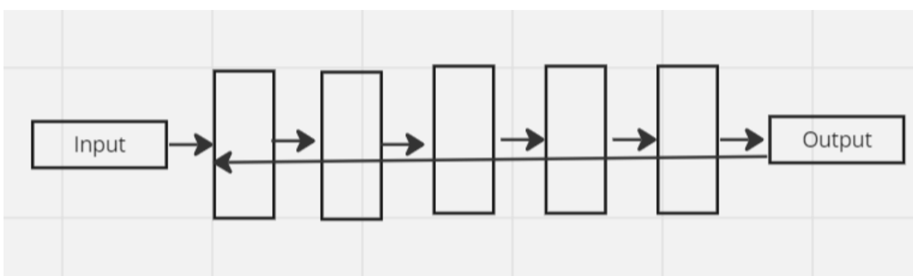
After that, we can define the multihead attention function

```python
# Multi-head attention
def multi_head_attention(self, inputs):
    q = self.Wq @ inputs
    k = self.Wk @ inputs
    v = self.Wv @ inputs
    attention = self_attention(self.mask, q, k, v)
    output = self.Wo @ attention
    return output
```

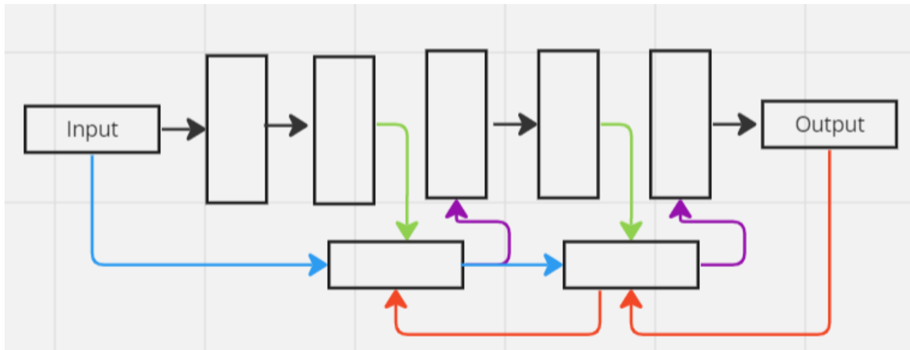
5 Residual connections

In deep networks, a problem we always have to deal with is vanishing/exploding gradients. The more layers our network have, the more possibility we have to deal with these problems. There are several ways to deal with this. We can clip the values whenever it is too small or too large. Another solution is to cut off the partial derivatives after long enough steps. In this section, we will discuss a solution for these problems by residual connections.

Let us look at the usual deep network in the picture below



For each layer, we will randomly initialize weights and when we have too many layers, the final input for the output contains a lot of random noises. Moreover, the backpropagation method have to go through all the way back to update parameters, and it can cause the vanishing/exploding gradients if the path is too long. An improvement for this is *residual connections*.



In a model with residual connections, after several layers, we *add/concatenate the output and the input* and make it become the input for the next layer. By this way, the random noise at the final input is reduced. Let us see why it can also reduce the vanishing/exploding gradient problem. Assume that we have a network with 3 layers. Through the first layer, we can compute

$$\text{output}_1 = W_1\,\text{input}$$

In a residual connection network, the input of layer 2 is

$$\text{input}_2 = \text{output}_1 + \text{input}$$

Next, we have the output of the second layer is

$$\text{output}_2 = W_2\,\text{input}_2$$

Again, the input for the third layer is

$$\text{input}_3 = \text{input}_2 + \text{output}_2$$

Finally, the output for the last layer is

$$\text{output} = \text{softmax}(W_3\,\text{input}_3)$$

After defining the loss function L, for the backward method, to update the parameter W_1, for example, we need to compute

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial\,\text{input}_3}\frac{\partial\,\text{input}_3}{\partial W_1} = \frac{\partial L}{\partial\,\text{input}_3}\frac{\partial(\text{input}_2 + \text{output}_2)}{\partial W_1} \tag{2}$$

Note that in usual networks, the only term we have is

$$\frac{\partial L}{\partial\,\text{input}_3}\frac{\partial\,\text{output}_2}{\partial W_1} = \frac{\partial L}{\partial\,\text{input}_3}\frac{\partial\,\text{output}_2}{\partial\,\text{input}_2}\frac{\partial\,\text{input}_2}{\partial W_1} = \frac{\partial L}{\partial\,\text{input}_2}\frac{\partial\,\text{input}_2}{\partial W_1}$$

And what happens if the term $\frac{\partial L}{\partial\,\text{input}_2}$ is very small? If that happens, it is most likely that the network will not update W_1. However, in (2), we still have the term $\frac{\partial\,\mathrm{input}_2}{\partial W_1}$, and we can cut off the remaining term if it is too small and obtain an approximation

$$\frac{\partial L}{\partial W_1} \approx \frac{\partial L}{\partial\,\mathrm{input}_3}\frac{\partial\,\mathrm{input}_2}{\partial W_1}$$

In practice, residual connections help the models converge much faster than the usual ways. The reader can take a look on [3] for more detail. However, there is one problem we need to solve: the shape of the input and the output can be different. One way to deal with this is to change the shape of the input by multiplying it with a learn-able matrix. And for our transformer model, we will add the input data to the output of the self attention layer, and it will become the input for the layer normalization.

6 Layer normalization

For layer normalization, we do not have much to say, it is just the z-score of a matrix. Recall, to compute the z-score of a matrix along its rows, we will first compute the mean and the standard deviation std of each row, and then compute z-score = (matrix - mean)/std. Here is how it is done in codes

```
def z_score(matrix):
    mean = torch.mean(matrix, dim = 1, keepdim = True)
    std = torch.std(matrix, dim = 1, keepdim = True)
    z_score_vector = (matrix-mean)/std
    return z_score_vector
```

7 Experiments

We have finished explaining the theory behind transformer, and here is our experiment. The project is to build a transformer model for the sentiment analysis task. The source of our data can be found at [4]. It consists of simple sentences with label True for positive feeling and False for negative feeling.

For vector embedding method, we use the one hot encoding. The reader can use different methods such as Word2vec or GloVe, but it is not our main purposes, and we want to keep it simple.

The input for the forward method of the transformer class is a matrix of a whole sentence and each row represents a word embedding vector. Here is our forward method:

```python
def forward(self, inputs):
    # Inputs is a whole sentence at a time
    # And this sentence is represented by a matrix
    # The matrix consists of row vectors, each row is a word embedding
    np_inputs = np.array(inputs)
    torch_inputs = torch.tensor(np_inputs)
    torch_inputs = torch_inputs.T        You, yesterday • version 2
    torch_inputs = torch_inputs.to(torch.float32)
    pos_inputs = self.positional_encoding(torch_inputs)
    output_self_attention = self.multi_head_attention(pos_inputs)
    output_layer_norm_self = self.layer_norm_self_attention(output_self_attention, pos_inputs)
    output_feed_forward = self.feed_forward(output_layer_norm_self)
    output_layer_norm_feed = self.layer_norm_feed_forward(output_feed_forward, output_layer_norm_self)
    y_pred = torch.softmax(output_layer_norm_feed, dim = 0)
    return y_pred
```

And for the backward method, we will use torch.backward() to update parameters with requires_grad = True. Note that we need to define loss = torch.nn.BCEWithLogitsLoss() and optimizer = torch.optim.SGD([parameters], learning_rate) outside of loops and they are initialized only once. At each step, we compute the loss L = loss(y_pred, y_true) and update parameters by the optimizer.step() method. Also, do *not* forget optimizer.zero_grad() after each step, as it will add up otherwise. Here are our fit and process function, the inputs of those are

- The list X consists of matrices of all sentences. Each element of X is a matrix of a sentence.
- The list y consists of true labels from the data.

```python
def fit(self, X, y, max_iter = 201, learning_rate = 0.001):
    self.loss = nn.BCEWithLogitsLoss()
    self.optimizer = opt.SGD([self.Wq, self.Wk, self.Wv, self.Wo,
                              self.Wnorm_self, self.bnorm_self, self.gamma_self, self.beta_self,
                              self.Wf, self.bf,
                              self.Wnorm_feed, self.bnorm_feed, self.gamma_feed, self.beta_feed],
                              lr = learning_rate)
    for i in range(max_iter):
        accuracy = self.process(X, y, run_backward=True)
        if(i % 20 == 0):
            print(f"Step: {i}")
            print(f"accuracy for training data: {accuracy}")
```

```python
def process(self, X, y, run_backward = False):
    accuracy = 0
    for x, y_true in zip(X,y):
        # x is a matrix of a sentence
        probs = self.forward(x)
        # True label
        true_index = int(y_true)
        # Accuracy
        accuracy += int(torch.argmax(probs) == true_index)
        if run_backward:
            y_true_torch = torch.zeros((2,1))
            y_true_torch[true_index] = 1
            L = self.loss(probs, y_true_torch)
            self.optimizer.zero_grad()
            L.backward()
            self.optimizer.step()
     # Accuracy
    return float(accuracy/len(X))
```

And here are the results of our model

```
PS C:\Users\Thuong Dang\Desktop\Data Science\Projects\transformer_from_scratch_with_pytorch> python experiments.py
Step: 0
accuracy for training data: 0.43103448275862066
Step: 20
accuracy for training data: 0.5517241379310345
Step: 40
accuracy for training data: 0.6551724137931034
Step: 60
accuracy for training data: 0.7068965517241379
Step: 80
accuracy for training data: 0.6896551724137931
Step: 100
accuracy for training data: 0.5517241379310345
Step: 120
accuracy for training data: 0.6896551724137931
Step: 140
accuracy for training data: 0.8620689655172413
Step: 160
accuracy for training data: 0.8620689655172413
Step: 180
accuracy for training data: 0.8620689655172413
Step: 200
accuracy for training data: 0.8620689655172413
Accuracy for test set: 0.95
```

```
PS C:\Users\Thuong Dang\Desktop\Data Science\Projects\transformer_from_scratch_with_pytorch> python sentiment_prediction.py -t "i am bad"
Negative feeling
PS C:\Users\Thuong Dang\Desktop\Data Science\Projects\transformer_from_scratch_with_pytorch> python sentiment_prediction.py -t "it is happy"
The text 'it is happy' has words that are not in dictionary of the data file (data/train_data.csv). Please try another words.
PS C:\Users\Thuong Dang\Desktop\Data Science\Projects\transformer_from_scratch_with_pytorch> python sentiment_prediction.py -t "it is sad"
The text 'it is sad' has words that are not in dictionary of the data file (data/train_data.csv). Please try another words.
PS C:\Users\Thuong Dang\Desktop\Data Science\Projects\transformer_from_scratch_with_pytorch> python sentiment_prediction.py -t "this is good"
Positive feeling
PS C:\Users\Thuong Dang\Desktop\Data Science\Projects\transformer_from_scratch_with_pytorch> python sentiment_prediction.py -t "this is happy"
Positive feeling
```

```
PS C:\Users\Thuong Dang\Desktop\Data Science\Projects\transformer_from_scratch_with_pytorch> python sentiment_prediction.py -t "i am not at all happy"
Negative feeling
PS C:\Users\Thuong Dang\Desktop\Data Science\Projects\transformer_from_scratch_with_pytorch> python sentiment_prediction.py -t "i am not at all bad"
Positive feeling
PS C:\Users\Thuong Dang\Desktop\Data Science\Projects\transformer_from_scratch_with_pytorch> python sentiment_prediction.py -t "this was not happy"
Negative feeling
```

We hope you enjoy this documentation and our project. We very much welcome any comment or suggestion for our work. Below, you will find the list of references we have used throughout our project.

REFERENCES

[1] Dan Jurafsky and James H. Martin, *Speech and Language Processing*, 3rd edition, available at `https://web.stanford.edu/~jurafsky/slp3/`.

[2] Rupesh Kumar Srivastava, Klaus Greff,Jurgen Schmidhuber, *Highway networks*, available at `https://arxiv.org/pdf/1505.00387.pdf`.

[3] Ashish Vaswani et al., *Attention is all you need*, available at `https://arxiv.org/pdf/1706.03762.pdf`.

[4] Victor Zhou's dataset for sentiment analysis, available at `https://github.com/vzhou842/rnn-from-scratch/blob/master/data.py`