MARBOUH OUMAIMA
oumaima.marbouh@polytechnique.edu

Lab session # 3
ALTEGRAD 2021

12/03/21

1 Question 1

In this Lab we are using a transformer-based mechanisms with self-attention. While implementing the architecture we are using the square mask to prevent the attention mechanism of the transformer from "cheating" when training, that's why we use the nn.TransformerEncoder abstraction that doesn't allow "peaking ahead" when deriving self attention, that means if we are at position t, and we want to find the self attention for the current token, we are not allowed to consider tokens at times t+1,...., T_x.

If we have the following sequence as an input for our decoder: "I love it", then the expected prediction for the token at position one ("I") is the token at the next position ("love"). Similarly the expected prediction for the tokens "I love" is "it".

We do not want the attention mechanism to share any information regarding the token at the next positions, when giving a prediction using all the previous tokens.

To ensure that this is done, we mask future positions (setting them to -inf) before the softmax step in the self-attention calculation.

If we retake the standard notation for transformers, here is the Z matrix that is passed to fully connected layers:

$$Z = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{1}$$

where d_k is the dimension of the encoding. We can see that by applying a mask whose values are inf for future tokens, the softmax function considers their values really close to zero and ignores their contribution for self-attention at time t.
When it comes to positional encoding, since we are not using any sequential-type architecture for our network (LSTM,RNN, GRU...) that inherently take the order of words into account, the transformer architecture doesn't have any sense of position/order, this may affect the context and the attention mechanism and therefore the results.
In order to solve this we should give the model some sense of order using the positional encoding that is used to equip the input words with their positional information. That is for every word w_t in a sentence $[w_1, ..., w_n]$, calculating the corresponding embedding which is fed to the model is as follows:

$$\psi^{'}(w_t) = \psi(w_t) + \vec{p_t} \tag{2}$$

To make this summation possible, we keep the positional embedding's dimension equal to the word embeddings' dimension i.e.

$$d_{wordembedding} = d_{postionalembedding}$$

2 Question 2

In this lab we use transfer learning, that means we use a network that was trained on a specific classification task different than the problem we have here. For that, we need to change the classification head and put one compatible with the purpose that we want to achieve here, that means instead of creating and tuning a model from scratch we use a pre-trained model but we change the last layer to adapt it to the new problem we are trying to solve.The dataset used, the size of the vocabulary and relations between words is preserved for our case, but the "fine-tuning" phase is precisely designed to fit a different task.

When it comes to the difference between the language modelling and classification tasks. Language modeling is the task of predicting the next word or character in a document. This technique can be used to train language models that can further be applied to a wide range of natural language tasks like text generation, text classification, and question answering.

The common types of language modeling techniques involve:

- N-gram Language Models
- Neural Langauge Models

A language model attempts to learn the structure of natural language through hierarchical representations, both at the level of word representation, and meaning/semantics of a sequence in a particular context. Almost always, language modelling is ”generative”, meaning that it aims to predict the next word given a previous sequence of words(many-to-many models).

On the other hand, classification maps the input sequence into a finite set of classes, like for example sentiment analysis (positive or negative). The set of classes for language modelling is also fixed, but much larger. In this sense, the model types are more of type many-to-one, and is not generative, the prediction is made on the sentence as a whole, without intermediate predictions.

3 Question 3

The number of trainable parameters for both cases is equal to:

$$num\_params = num\_params\_base + num\_params\_classifier$$

As we have noted, each encoder has 8 “heads” which allow to characterize different relationships between tokens. In the end, each “head” produces a matrix of Z context vectors (the number of rows is equal to the number of tokens and the number of columns is the dimensionality of the Q, K and V matrices). These Z matrices are concatenated and then multiplied by a 0 matrix (the parameters of which will be learned by training) which will make it possible to find a single output Z matrix which will be input to the Encoder Feed-Forward as we can see in the representation for an encoder of a transformer:

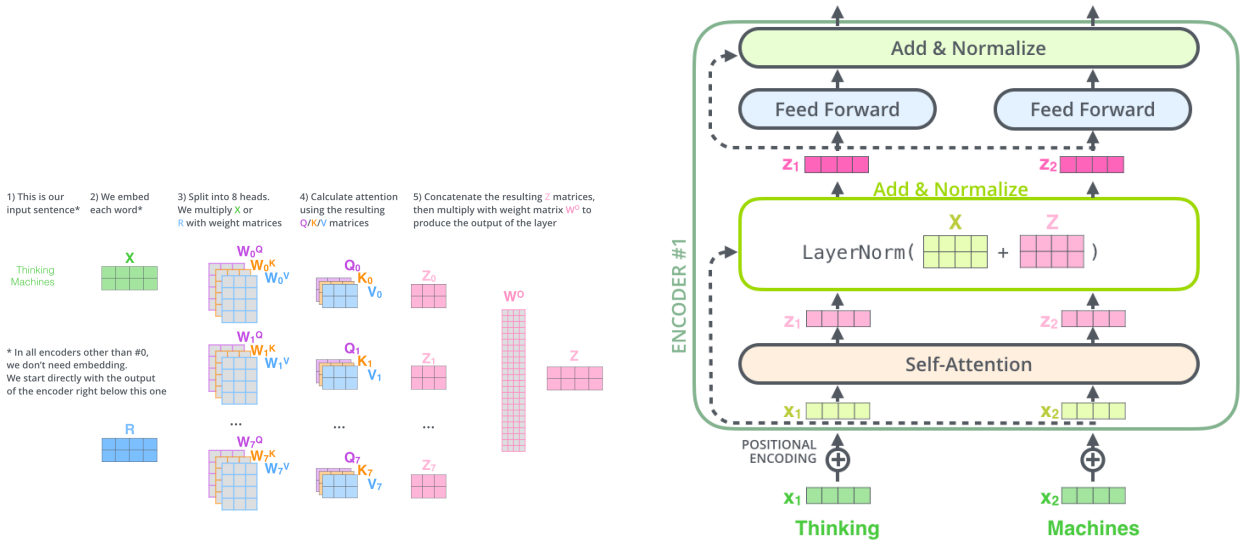

Figure 1: Transformer Encoder

Let’s calculate the number of trainable parameters for the base model. Let’s define our parameters for the model: $n_{token} = 100$(vocab size); $n_{hid} = 200$(hidden dimension of the transformer layers); $n_{layers} = 4$(number of transformer layers in the encoder part); $n_{head} = 2$(attention heads); $n_{nclasses} = 2$ (classes for classification in first example on sentiment analysis). First, the embedding size (nothing is learned in positional encoding):

$$embedding\_weights = n_{token} n_{hid} = 200000 \tag{3}$$

By looking at the internals of the Pytorch implementation of the TransformerEncoderLayer, we can see that since we do not specify values for the dimension of K and V , the weights for the input part of the attention mechanism take the shape of (3 nhid, nhid) (with 3 nhid for the bias) and (nhid, nhid) for the output part of the attention mechanism (with nhid for the bias, which yields:

$$num\_weights\_attention = 3nhid(nhid + 1) + nhid(nhid + 1) = 160800 \tag{4}$$

We then have the FC layer with dimension (nhid, nhid) and bias nhid in our case. This is the the same for the 2 attention heads:

$$num\_weights\_FC = (nhid(nhid + 1))2 = 80400 \tag{5}$$

We finally have the LayerNorm with nhid = 200 parameters for both the self-attention mechanism and the feed forward part, and for both attention heads. In total, for 1 encoder layer, we have:

$$\begin{aligned} num\_weights\_encoder\_layer &= num\_weights\_attention + num\_weights\_FC + num\_weights\_layernorm \\ &= 160800 + 80400 + (20022) \\ &= 242000 \end{aligned} \tag{6}$$

Since we have 4 encoder layers, and taking into account the embedding, here is the total number of trainable paramters for the base model:

$$\begin{aligned} num\_weights\_base\_model &= num\_weights\_encoder\_layer4 + embeddingweights \\ &= 968000 + 20000 \\ &= 988000 \end{aligned} \tag{7}$$

- For the classification, we have a linear layer of size (nhid, nclasses) (and a bias) which here would represent:

$$\begin{aligned} num\_weights\_with\_classifier &= num\_weights\_base\_model + 2002 + 2weights \\ &= 988402 \end{aligned} \tag{8}$$

- For the language modelling, we have a linear layer of size (nhid, ntoken) (and a bias) which here would represent:

$$\begin{aligned} num\_weights\_with\_language\_model &= num\_weights\_base\_model + 200100 + 100 \\ &= 1008100 \end{aligned} \tag{9}$$

4 Question 4

The following plot represent the accuracy plot for the pre-trained model, and the model trained from scratch:

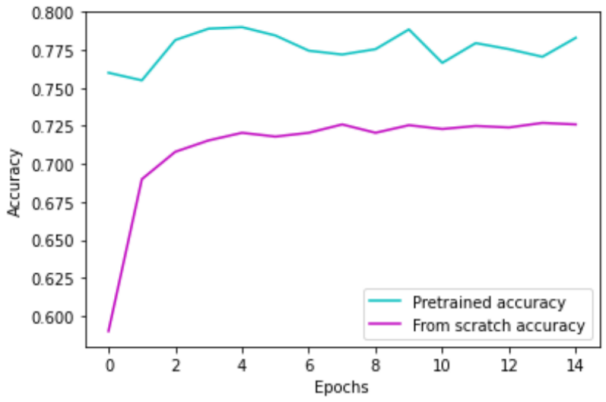

Figure 2: Accuracy VS Epochs for the pretrained and from scratch model

We can observe a few things:

- The training accuracy is starting a lot higher (25%) for the pre-trained model than for the model trained from scratch. This is expected since the weights are initially close to zero for the model trained from scratch, so one epoch step on an unknown classification problem should yield a lower accuracy than one epoch concentrated on the last classification head (for the pre-trained model).
- After approximately 10 epochs, both accuracies plateau at a certain threshold. For the pre-trained model,this is the case because after improving a bit the accuracy for the first few epochs, the network doesn't find any improvement for the given classification task. For the model trained from scratch, the network has learned quite enough for the given classification task, and is becoming limited.

- The pre-trained model achieves a higher accuracy (80%) compared to the model trained from scratch (75%). One possible explanation is that given that the pre-trained model was trained on a much larger dataset, and for longer, it was able to pick up word semantics/relationships that are deeper than the ones learned from scratch.

5 Question 5

The approach we have used here has the objective function of learning general language representations, with a unidirectional language model. we already saw that the architecture chosen with the nn.TransformerEncoderLayer uses a mask so that every token can only attend to previous tokens in the self-attention layers of the Transformer. The major drawback of this choice of unidirectionality is that for some context and tasks, it could harm the results even with fine tuning. the approach proposed in [1] uses a bi-directional language model, with a masked language model in the pre-training phase, where each token is randomly masked from the input, and the objective is to predict the vocabulary id based only on the context. Having this richness of attention and modified training objective allows the model to generalize better for a broader variety of tasks.

References

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.