Deep Learning HW3

April 3, 2023

Q1.1

a. Attention $(K, Q, V) = softmax\left(\left(QK^\top\right)/\sqrt{d_K}\right)V$ where $\beta = 1/\sqrt{d_k}$

b. β controls the haddees of the output. If β is very large (∞) then the query will act as a one-hot encoded mechanism. On the contrary if β is close to 0 then it takes a combination (weighted) of the Key values. $\beta = 1/\sqrt{d_k}$ is used in Attention in all you need, where K is the size of the key vector, and d_k is its dimensionality.

c. If we make the β very large (∞) then the $QK^\top$ term would behave like a one-hot encoded setup. This would mean that one of the V 's would be preserved. Out of the four types of attention, we can go for hard self-attention of hard cross-attention.

To preserve the information we can use gating mechanisms as in GRUS and LSTMS, in fully connected architectures.

d) We should pick a β close to 0 to ensure that the information is diffused. Hence in this situation, the output is a spread version of the value vectors. We are referring to soft self/cross-attention. This can be done with fully-connected architectures by using a Kernel (ID) with equal weights.

e) Consider the row $\times$ column multiplication of $QK^\top$. In the case of a small perturbation, each of the inputs to the softmax would be incremented by some scaled version of ϵ.

Effectively, irrespective of the value of β, a small perturbation would only create a small/insignificant change to the output.

f. On the other hand if the perturbation is very large then the corresponding key might end up dominating the output. This can lead the attention mechanism to focus more on the value associated with the key.

Q1.2

a. We can calculate Attention; as

$$Attention_i = softmax\left(\left(QK^\top/\sqrt{d_K}\right)V\right.$$

And the overall attention is given by [Attention 1, Attention 2 ... Attention m] Where m is the number of attention heads and commas represent that the attentions are stacked together.

b. In Conv1D the parameter groups interacts with different parts of the input and these captured interactions are then concatenated. Similar to this multi-head attention attends to different aspects of the input sequence. These

interactions are then concatenated together.

Q1.3 a.

$\{Q_i\}_1^h = \{w_{q_i} c\}_1^h$

$\{K_i\}_1^h = \{w_{k_i} c\}_1^h$

$\{V_i\}_1^h = \{w_{v_i} C\}_1^h$

$\text{Attention}_i = softmax\left(Q_i K_i^\top / \sqrt{d_{k_i}}\right) v_i$
$H = [\ \text{Attention}_1, , ..., Attention\text{h}]$

b. The positional embeddings are represented by $\vec{P}_t = \begin{bmatrix} \sin(w_k t) & i = 2k \\ \cos(w_k t) & i = 2k+1 \end{bmatrix}$

This enables some interesting properties, Corresponding to the Least Significant Bit analogy in binary numbers, the frequency at $i = 0$ reverts or fluctuates most frequently. Moreover, it enables a smooth decrease in the differences between two positional encodings w.r.t i.

Now, we don't use positional encoding in tasks that are predicting sets or miss any inherent notion of relative positions, for example, a predictive model. However, on the other hand, if the underlying task needs an understanding of the position, for example, language-based tasks like sentiment analysis or language modeling, we would need positional encodings.

c.

A self-attention layer can behave like an identity layer if the attention weights are such that each input position only attends to itself. This can happen when the attention weights for each position are set to 1 for the corresponding input and 0 for all other inputs.

If we have a sequence of inputs X = [x1, x2, x3]. The self-attention layer computes the attention weights using the queries, keys, and values derived from these inputs. If the attention weights matrix A looks like this:

A = [[1, 0, 0], [0, 1, 0], [0, 0, 1]], then Y = A * X = X

As for a permutation layer, the attention weights matrix would need to have a single 1 in each row and column, but not along the diagonal.

A = [[0, 1, 0], [0, 0, 1], [1, 0, 0]], then Y = A * X = [x2, x3, x1]

d.

A self-attention layer can behave like a "running" linear layer if the attention weights at each output position are computed as a linear combination of the input positions.

Suppose we have a sequence of inputs X = [x1, x2, x3, x4, x5]. If the attention weights matrix A

A = [[1/3, 1/3, 1/3, 0, 0], [0, 1/3, 1/3, 1/3, 0], [0, 0, 1/3, 1/3, 1/3]], then

Y = A * X = [1/3*(x1 + x2 + x3), 1/3*(x2 + x3 + x4), 1/3*(x3 + x4 + x5)]

The proper name for such a layer is a convolutional layer with a kernel size of 3 and stride of 1.

e.

A = [[1/3, 1/3, 1/3, 0, 0], [0, 1/3, 1/3, 1/3, 0], [0, 0, 1/3, 1/3, 1/3]], then

Y = A * X = [1/3*(x1 + x2 + x3), 1/3*(x2 + x3 + x4), 1/3*(x3 + x4 + x5)]

Q1.4

a.

Attention Mechanism: While RNNs/LSTMs use a single hidden representation modified recurrently, Transformers leverage multiple hidden representations, one for each input.

Vanishing Gradient Problem: Since Transformers do not depend on recurrence and calculate attention scores between any two hidden representations at once, their inductive bias is more aligned to calculate long-term and short term dependencies. The robustness is evident from the time complexity of mapping the relation between any two hidden states – it is independent of the distance between any two hidden states in the sequence for Transformers.

Positional Encodings: While Transformers utilize positional encodings to retain the input structure, RNNs and LSTMs consume the input sequentially and hence do not require positional embeddings.

Parallelizable: The inherent construction of Transformers enables highly parallelized processing while RNNs and LSTMs need structural optimizations and additional assumptions to ensure parallel processing of the inputs.

Layer Architecture: While RNNs and LSTMs follow a recurrent paradigm, Transformers rely on a feed-forward architecture.

b.

1. Input embeddings: Self-attention requires input which can be represented as a 1D embedding for the smallest unit of input (example: image patch, word or character)

2. Positional embeddings: Since self-attention does not capture positional information, the positional embeddings for hidden layer are explicitly specified. They can be factored as a vector addition as in the original Transformers paper or concatenated with the input embedding.

3. Key (K), Query (Q) and Value (V): Each hidden layer maintains its corresponding key, query and value vector, initialized by the input embeddings and their product with a weight matrix. For self-attention, the dimension of all the three vectors is same. The query vector of an input is projected onto the key matrix all the inputs. These projection scores are normalized using softmax and then used to calculate the value as the (attention) weighted sum of value vectors of all the other inputs.

4. Output: The calculated value vectors are used for downstream layers.

Self-attention captures long and short-term dependencies in the input. Because of its ability to capture relations in the input all at once, self-attention enables Transformers to be highly parallelizable. Moreover, attention helps generate contextualized embeddings instead of static representations because each hidden representation is dependent on its interactions with other hidden representations. Finally, it enables interpretability for relations in a sequence. In the original paper self-attention is used to calculate the representations of the source and target languages for machine translation.

c.

In the paper “Attention is all you need”, multi-head attention serves as an extension of self-attention. It entails multiple heads, each with its own set of key, query and value matrices. This enables the Transformer to model different types of relationships between the input tokens, whereas in a single-head self-attention all the different types of relationships in tokens are averaged out. Each separately calculated head is used to weigh the value matrix just as in self-attention described in part b, additionally each of these heads are concatenated together to generate the final representation. The model enforces the capture of different types of relationships by using all the heads in parallel. Hence, multi-head attention can be summarized as 1. Multiple Heads 2. Self-attention and 3. Concatenation. Overall, the benefits of multi-head attention are:

1. Diversity: Capture diverse relationships
2. Expressiveness: Expressive representations
3. Parallelization: Each of the heads is further parallelizable

d.

Input: The output of the self-attention layer is fed element wise into the feed-forward layer

Structure: The feedforward layer has a fully-connected architecture with two layers – one hidden layer with ReLU activation and one output layer, hence the output is given by

$FFN(x) = \max(0, xW1 + b1)W2 + b2$

Moreover, the input and output dimension is 512 for the Transformer-base to enable residual connects, hidden layer is 2048.

Output: Each input token has its own corresponding FFN. They are all initialized with the same weights (identical) however computations are independent. Moreover, a token-wise residual connection is added to the output of each FFN.

These are the advantages of using an FFN

1. Complexity: The ReLU activation introduces non-linearity
2. Complements the Multi-head attention: Enables the model to capture relations between the generated output value representations for each token
3. Parallelization: The independent position-wise FFN enables parallelization

e.

Layer normalization is used to normalize the output of each token individually. It entails 3 steps:

1. Computing the mean and standard deviation
2. Normalization y = (x - mean)/ standard deviation
3. Then y is scaled with two learnable parameters ay + b

It is used after the FC layer and the self-attention layer in the encoder. Similarly, it is used in the FC and the self-attention layer in the decoder. Moreover, it is used after the encoder input to the decoder.

There are 3 major advantages of using layer normalization:

1. Training stability: Handles internal covariate shift
2. Faster convergence
3. Gradient flow: Layer normalization layer along with the residual connection enables smooth gradient flow

Q1.5

a. Vision transformer divides the input image into image patches and then projects each image patch onto a flattened 2D vector, this removes some of the inductive biases inherent to CNNs, which is why the ViT uses positional encodings. Where as CNNs operate directly on a 2D image, enable inductive biases of locality and translational equivariance.

Hence the CNN works with hierarchical local features and the ViT works with both local and global features together.

The Vision transformer uses CNNs for dividing the image into patches, where the stride = kernel size for non-overlapping patches.

b.

- Application domain:

The original Transformer was designed for natural language processing (NLP) tasks, such as machine translation and text classification. The Vision Transformer is designed for computer vision tasks, such as image classification and object detection.

- Input data processing:

In the original Transformer, the input data are sequences of tokens from a text corpus. These tokens are first embedded into continuous vectors and then processed through the self-attention mechanism in the Transformer layers.

In the Vision Transformer, the input data are images. The images are first divided into non-overlapping patches, which are then linearly embedded into continuous vectors. These vectors are treated as tokens and processed through the self-attention mechanism in the Transformer layers.

- Positional encoding:

Both the original Transformer and ViT use positional encoding to provide information about the relative positions of the input tokens. In the original Transformer, the positional encoding is added to the input embeddings before they are fed into the Transformer layers.

In the ViT, a learnable positional embedding is used instead of a fixed function-based positional encoding.

- MLP activation:

The original transformer uses ReLU to introduce non-linearity, while ViT uses GeLU for non-linearity in the MLP layer.

c.

Both the original Transformer and ViT use positional encoding to provide information about the relative positions of the input tokens. In the ViT, a learnable positional embedding is used instead of a fixed function-based positional encoding as in the original Transformer (uses sinusoidal positional embeddings).