Junyu Yin G2101985L

Question One

(a)

All the results are shown in the table below.

# of layers \ d_{model}	512	1024	2048
1	2102784	6301696	20990976
2	4205568	12603392	41981952
3	6308352	18905088	62972928
4	8411136	25206784	83963904
5	10513920	31508480	104954880
6	12616704	37810176	125945856
7	14719488	44111872	146936832
8	16822272	50413568	167927808
9	18925056	56715264	188918784
10	21027840	63016960	209909760
11	23130624	69318656	230900736
12	25233408	75620352	251891712

Table 1. The total number of parameters in the transformer encoder layer with respect to different hyper-parameter values.

When d_{model} is fixed, the number of parameters increases linearly with the number of layers. And when the number of layers is fixed, the number of parameters increases approximately quadratically with d_{model}.

In fact, the following formula demonstrates clearly how the number of parameters changes with respect to the hyper-parameter values.

$$\#\text{ of parameters} = N\left(4d_{model}^2 + 2d_{model}\text{ffn\_dim} + 9d_{model} + \text{ffn\_dim}\right)$$

(b)

I agree with this statement. Multiple heads in multi-head attention work like multiple channels in traditional CNNs. This mechanism forces a model to attend to different representation subspaces. The computation of different heads can be performed independently and in parallel. And the results from all heads are summarized in next step to get a more comprehensive ones. This is just like an ensemble of heads.

(c)

For decoder training, transformer architecture can perform parallel training when using the teacher-forcing mechanism, while recurrent models must use the previous hidden states, so they can only be trained serially. For decoder inference, both two architectures can only generate outputs sequentially.

In short, these two architectures can be said to be completely different, though both can be used to handle seq2seq tasks.

(d)

The transformer architecture uses positional encoding to inject the sequence information into word embeddings. And during running, it uses the self-attention mechanism to capture sequence

information by attending to all positions in an input sentence.

(e)

For the implementation, I followed the given example code here. Except for the the hyper-parameters $(N, d_{model}, h, d_k, ffn\_dim)$ discussed above in the assignment and the batch size, all other configurations are kept totally same with the given example codes.

More specifically, I set the batch size to 128 here for fast training and performed hyper-parameter search on the rest hyper-parameters. The results are shown in the table below.

$(N, d_{model}, h, d_k, ffn\_dim)$	validation perplexity	test perplexity
(6,512,8,64,1024)	1018.83	961.56
(4,512,8,64,1024)	979.38	923.20
(4,512,8,64,512)	979.42	922.51
(4,256,8,32,512)	1001.89	947.08
(2,512,8,64,512)	978.33	922.78
(2,512,8,64,256)	343.14	319.39
(2,256,8,32,256)	976.30	920.82
(2,512,8,64,128)	503.93	467.18
(1,512,8,64,256)	369.26	341.18

Table 2. The results of the hyper-parameter search.

Due to the time limit, I only did the above hyper-parameter searches. And it can be seen that the best model is given by $(N = 2, d_{model} = 512, h = 8, d_k = 64, ffn\_dim = 256)$. Then I explored the effect of scaling on perplexity, the results are as follows.

$(N, d_{model}, h, d_k, ffn\_dim)$	use scaling	no scaling
(2,512,8,64,256)	343.14	699.65
(2,512,8,64,128)	503.93	735.19
(1,512,8,64,256)	369.26	375.25

Table 3. The effect of scaling on validation perplexity.

From the table above, we can see that no using scaling causes the perplexity to increase. So this operation helps the model to perform better.