AI6127 Deep Learning for Natural Language Processing

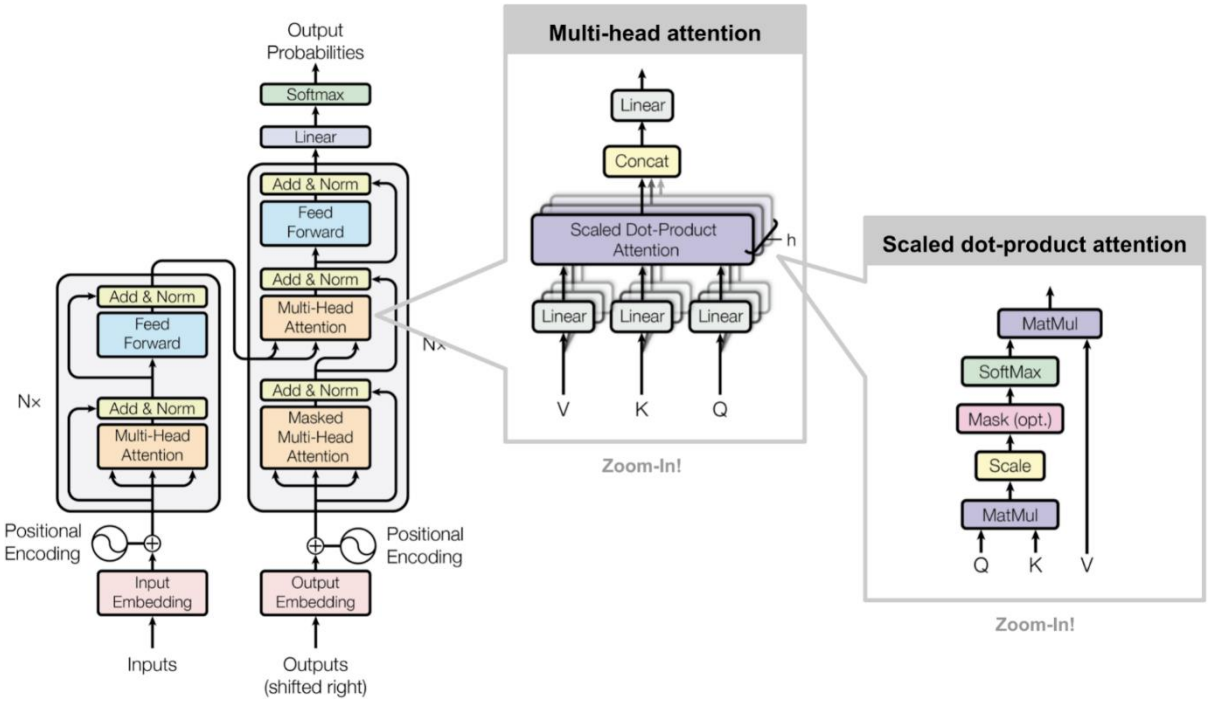
Chen Yongquan (G2002341D)

Nanyang Technological University

Assignment 3

1a.

The original "Attention Is All You Need" paper [1] did not specify if the linear projection matrices for the multi-head attention contain bias parameters. We assume here that the matrices have additional bias parameters.



Feature dimension for each token, $d_{model} = 512$

Number of heads, $h = 8$

Key and value representation size, $d_k = d_v = \frac{d_{model}}{h} = 64$

Hidden representation size of the feed-forward layer, $d_{ff} = 1024$

W: Weights

B: Biases

Encoder Layer ($N = 1$):

$$MultiHead(Q, K, V) = Concat(head_1, \dots, \text{head}_\text{h})\text{W}^\text{O}$$
$$where\ head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

$\dim(W_i^Q) = d_{model} \times d_k = 512 \times 64$
$\dim(B_i^Q) = d_k = 64$
$\dim(W^Q + B^Q) = [(512 \times 64) + 64] \times 8 = 262656$

$\dim(W_i^K) = d_{model} \times d_k = 512 \times 64$
$\dim(B_i^K) = d_k = 64$
$\dim(W^K + B^K) = [(512 \times 64) + 64] \times 8 = 262656$

$\dim(W_i^V) = d_{model} \times d_v = 512 \times 64$
$\dim(B_i^V) = d_v = 64$
$\dim(W^V + B^V) = [(512 \times 64) + 64] \times 8 = 262656$

$\dim(W^O) = hd_v \times d_{model} = (8 \times 64) \times 512 = 262144$
$\dim(B^O) = d_{model} = 512$
$\dim(W^O + B^O) = 262144 + 512 = 262656$

Total parameters for multi-head attention $= 262656 \times 3 + 262656 = 1050624$

$\dim(W^{Add\ \&\ Norm\ 1}) = d_{model} = 512 = 512$
$\dim(B^{Add\ \&\ Norm\ 1}) = d_{model} = 512 = 512$

Total parameters for first sub-layer $= 1050624 + 512 + 512 = 1051648$

Position-wise feed-forward network consists of two linear transformations with a ReLU activation in between [1].

$\dim(W^{ff1}) = d_{model} \times d_{ff} = 512 \times 1024 = 524288$
$\dim(B^{ff1}) = d_{ff} = 1024$

$\dim(W^{ff2}) = d_{ff} \times d_{model} = 1024 \times 512 = 524288$
$\dim(B^{ff1}) = d_{model} = 512$

Total parameters for feed-forward network $= (524288 \times 2) + 1024 + 512 = 1050112$

$\dim(W^{Add\ \&\ Norm\ 2}) = d_{model} = 512 = 512$
$\dim(B^{Add\ \&\ Norm\ 2}) = d_{model} = 512 = 512$

Total parameters for second sub-layer $= 1050112 + 512 + 512 = 1051136$

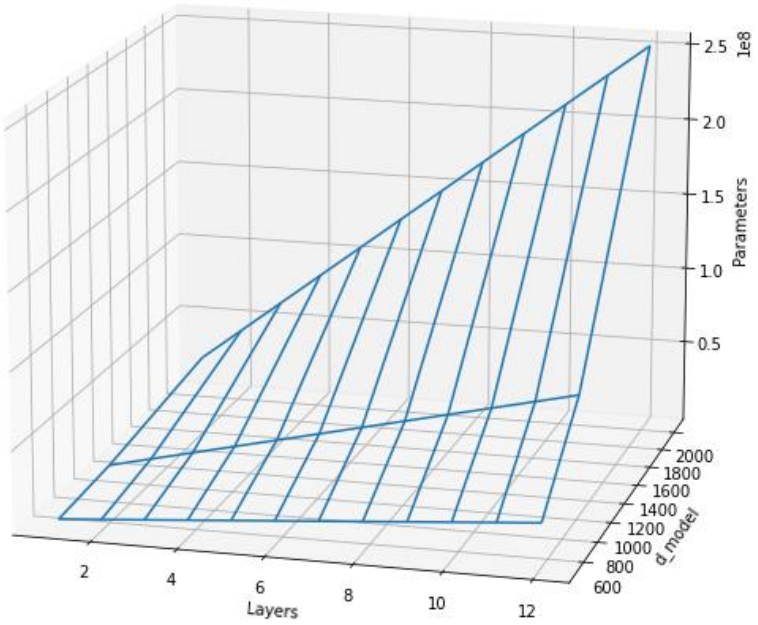
Total parameters for 1 transformer encoder layer $= 1051648 + 1051136 = 2102784$

Implementing this 1 layer in PyTorch:

```
self_attn.in_proj_weight:    linear1.weight:              norm1.weight:
torch.Size([1536, 512])      torch.Size([1024, 512])      torch.Size([512])
Parameters:786432            Parameters:524288            Parameters:512

self_attn.in_proj_bias:      linear1.bias:                norm1.bias:
torch.Size([1536])           torch.Size([1024])           torch.Size([512])
Parameters:1536              Parameters:1024              Parameters:512

self_attn.out_proj.weight:   linear2.weight:              norm2.weight:
torch.Size([512, 512])       torch.Size([512, 1024])      torch.Size([512])
Parameters:262144            Parameters:524288            Parameters:512

self_attn.out_proj.bias:     linear2.bias:                norm2.bias:
torch.Size([512])            torch.Size([512])            torch.Size([512])
Parameters:512               Parameters:512               Parameters:512          Total parameters: 2102784
```

The attention weights for Q, K and V for every head are stored in a single matrix and split among each head by rearranging the matrix such that the smallest dimension is $\frac{d_{model}}{h}$: https://github.com/pytorch/pytorch/blob/0a81034dd09c911c1ef81c22b44a9c527f9db061/torch/nn/functional.py#L4890

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



The table on the left shows how the total number of parameters increases with number of layers and the token representation size, d_{model}. The table is visualized as a 3D wireframe plot above.

We can see that the total number of parameters increases more steeply as d_{model} increases from 512 to 2048 than when number of layers increases from 1 to 12.

This is because the number of layers increases total number of parameters arithmetically, as a multiple of the total number of parameters in one layer. In contrast, the token representation size affects the size of the attention and feed forward sublayers in each encoder layer, as it (or a factor of it) is used as either the input or output dimensions for those sublayers. As such, it acts as a multiplier on the other dimension of the sublayers and total number of parameters can explode exponentially as it increases, especially for the feed forward linear layers.

1b.

Yes, multi-head attention works as an ensemble of heads in the transformer architecture.

Scaled dot-product attention as a single head self-attention allows the model to focus on a single input word as the query and encode a contextual representation of it using all the words in the input sequence as keys. The word embeddings are projected into query, key and value vectors using query, key and value weight matrices.

Multi-head attention expands upon this idea by using as many sets of query, key and value matrices as number of heads. This allows each head to learn to project a different contextual representation of the input word based on its surrounding words. Thus, the multi-head attention acts an ensemble of self-attention heads with their own query, key and value weight matrices which return their own contextual representation of the input query. Their outputs are concatenated and projected back to the d_{model} size using an output weight matrix. Thus, the ensemble of heads allows the model to jointly attend to information from different representation subspaces at different positions i.e. one head can learn to extract verb contextual information, one head learn to extract object information etc. and information from all the heads are concatenated and projected.

1c.

Decoder training and decoder inference in the transformer architecture is different compared to recurrent models like LSTM/GRU in that they have a masked self-attention sublayer and also an encoder-decoder attention sub-layer.

In attentionless recurrent models like LSTM/GRU, the decoder RNN at each step only has access to the previous decoder hidden state and output (or the encoder hidden state for the first decoding step).

In contrast, the encoder-attention sub-layer uses both the output from the previous decoder layer as its query and the output of the encoder as keys and values.

Furthermore, the self-attention sublayer also has access to outputs from all positions in the decoder up to and including the current position. This is accomplished by shifting the outputs right by one position and also masking out the input of the softmax in the scaled dot-product attention. The masking is done by setting to $-\infty$ the values in the input vector corresponding to invalid connections i.e. future decoder outputs.

1d.

The transformer architecture capture sequence information by directly summing positional encodings with the input embeddings at the bottom of the encoder and decoder stacks. Positional encodings can be learned [2] or fixed. In the original paper [1], fixed positional encodings are given by:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

The positional encodings have the same dimension d_{model} as the word embeddings so that they can be directly summed. Thus, with a toy embedding of size 4, the positional encoding at position 1 would be:

$PE_{(1,0=2(0))}$ $= \sin\left(\frac{1}{10000^{\frac{2\times0}{4}}}\right)$ $= 0.84$	$PE_{(1,1=2(0)+1)}$ $= \cos\left(\frac{1}{10000^{\frac{2\times0}{4}}}\right)$ $= 0.54$	$PE_{(1,2=2(1))}$ $= \sin\left(\frac{1}{10000^{\frac{2\times1}{4}}}\right)$ $= 0.01$	$PE_{(1,3=2(1)+1)}$ $= \cos\left(\frac{1}{10000^{\frac{2\times1}{4}}}\right)$ $= 1.0$

1e.

We followed the tutorial from Alexander Rush and integrated it into the project here. We trained a model using the original TransformerModel implemented in the original PyTorch example project to serve as a baseline. Using our integrated transformer, we trained a model using the default hyperparameters specified in the original Attention is All You Need. We then perform hyper-parameter searching by varying the number of layers, the token representation size, the number of heads, and the hidden representation size of the feed-forward layer. We also investigate how label smoothing and scaling within the attention softmax affects the perplexity of the trained model.

These are the hyper-parameters and commands used for each model.

Model_Default: Checkpoint file meant to be used with original code here	python main.py --cuda --model Transformer --epochs 10 --emsize 512 --nhid 1024 --nlayers 6 --nhead 8 --save model_default.pt
Model_1: Default AIAYN $N = 6$ $d_{model} = 512$ $h = 8$ $d_k = d_v = \frac{d_{model}}{h} = 64$ $ffn_{dim} = d_{ff} = 1024$	python main.py --cuda --model Transformer --epochs 10 --emsize 512 --nhid 1024 --nlayers 6 --nhead 8 --save model_1.pt
Model_2: $N = 3$	python main.py --cuda --model Transformer --epochs 10 --emsize 512 --nhid 1024 --nlayers 3 --nhead 8 --save model_2.pt
Model_3: $d_{model} = 256$	python main.py --cuda --model Transformer --epochs 10 --emsize 256 --nhid 1024 --nlayers 6 --nhead 8 --save model_3.pt
Model_4: $h = 4$	python main.py --cuda --model Transformer --epochs 10 --emsize 512 --nhid 1024 --nlayers 6 --nhead 4 --save model_4.pt
Model_5: $d_{ff} = 512$	python main.py --cuda --model Transformer --epochs 10 --emsize 512 --nhid 512 --nlayers 6 --nhead 4 --save model_5.pt
Model_6: Label Smoothing $\epsilon_{ls} = 0.1$	python main.py --cuda --model Transformer --epochs 10 --emsize 512 --nhid 1024 --nlayers 6 --nhead 8 --label-smoothing 0.1 --save model_6.pt
Model_7: No $\frac{1}{\sqrt{d_k}}$ scaling in attention	python main.py --cuda --model Transformer --epochs 10 --emsize 512 --nhid 1024 --nlayers 6 --nhead 8 --no-attention-scaling --save model_7.pt
Model_8: Learning Rate Factor = 2.0	python main.py --cuda --model Transformer --epochs 10 --emsize 512 --nhid 1024 --nlayers 6 --nhead 8 --lr 2.0 --save model_8.pt

The learn rate of the model is varied over the course of training according to the formula:

$$lrate = factor \cdot d_{model}^{-0.5} \cdot \min\left(step\_num^{-0.5} \cdot warmup\_steps^{-1.5}\right)$$

The learn rate specified to the main.py python file is used as the factor term.

The optimizer used is the Adam optimizer with $\beta_1 = 0.9$ and $\beta_2 = 0.98$ and $\epsilon = 10^{-9}$.

We used $warmup\_steps = 4000$.

```
115     model_opt = transformer.NoamOpt(model.src_embed[0].d_model,
116                                     args.lr,
117                                     4000,
118                                     torch.optim.Adam(model.parameters(),
119                                                      lr=0,
120                                                      betas=(0.9, 0.98),
121                                                      eps=1e-9))
```

Because the transformer model in the tutorial by Alexander Rush uses Kullback–Leibler divergence loss for implementing label smoothing and thus uses it for problem optimization, we separately calculate a cross-entropy loss which is not backpropagated and only used for computation of perplexity. Thus, the loss shown in the training logs for our model is actually KL divergence loss while the loss shown in model_default.log is cross entropy loss.

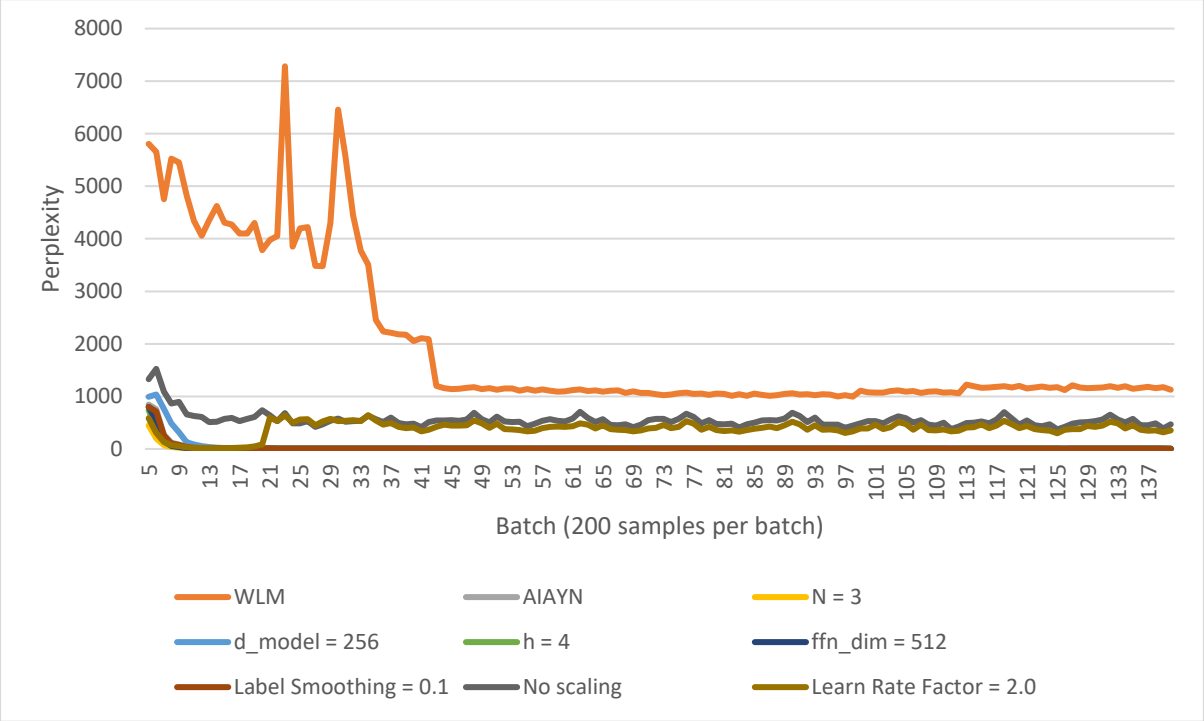
Model	Perplexity on test set
Model_Default	932.66
Model_1: Default AIAYN	3.21
Model_2: $N = 3$	3.04
Model_3: $d_{model} = 256$	3.48
Model_4: $h = 4$	3.47
Model_5: $d_{ff} = 512$	3.73
Model_6: Label Smoothing $\epsilon_{ls} = 0.1$	3.58
Model_7: No $\frac{1}{\sqrt{d_k}}$ scaling in attention	513.60
Model_8: Learning Rate Factor = 2.0	11.76

The best model was achieved by reducing number of layers to 3. We suspect the default model may be able to achieve better performance if trained for more epochs until saturation, while the smaller model was able to converge faster as it had less parameters to optimize.

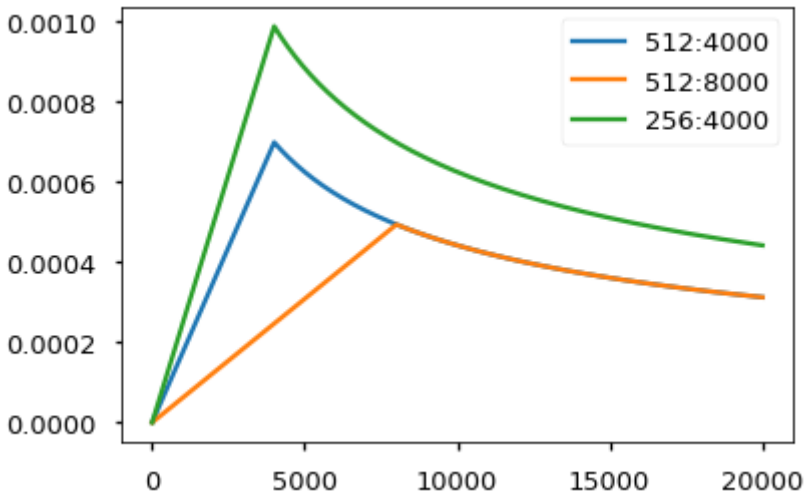
Using label smoothing hurts perplexity but it is used as a form of regularization and should improve generalization performance across a wider range of test data.

Removing scaling in the softmax of the attention function is detrimental to model performance. As the original paper described, for large values of d_k, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients. The scaling is used to counteract this effect.

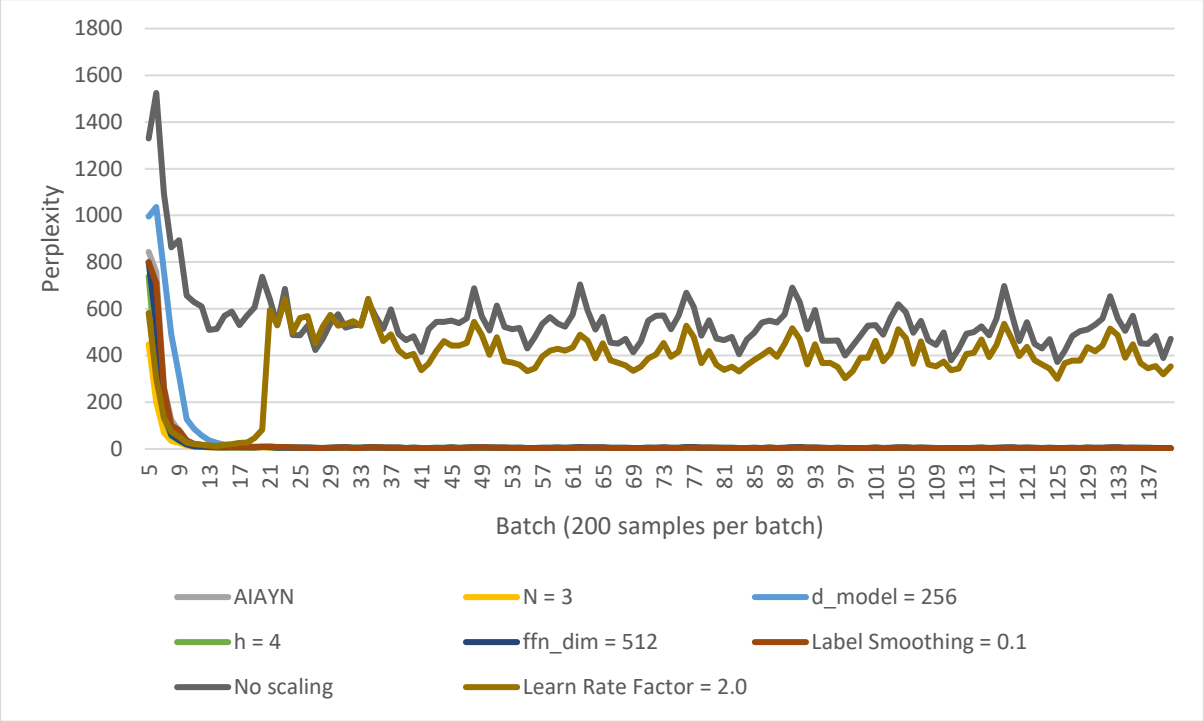
The charts below show the training perplexity for all trained models in different combinations and axes.



The chart shows the training perplexity from the fifth batch onwards and we can see that the original example model performed the worst followed by removing attention scaling and increasing the learning rate factor to 2.0. We note that using the learning rate factor of 2.0 did allowed the model to reach a comparable perplexity with the other models for the first epoch, however learning rate could have became too large in later epochs resulting in model perplexity increasing. This pattern corresponds with the curve of the learning rate function shown below:



The chart below shows the training perplexity of the 8 models trained on our integrated transformer model from the 5th batch onwards.



The chart below shows the training perplexity of our 6 best models from the 15th batch onwards.