Lab 4 Report: Pruning

Team Jobless
Jiyang Tang (jiyangta), Ran Ju (ranj), Tinglong Zhu (tinglonz), Xinyu Lu (xinyulu2)

1 Part I: Magnitude pruning on SST2

1.1 Hardware and OS Information

We used the Apple M1 Pro chip of 16 cores, and the RAM is 16 GB. The detailed information is as follows.

1. CPU
   (a) Chipset Model: Apple M1 Pro
   (b) Total Number of Cores: 16
   (c) Vendor: Apple(0x106b)
   (d) Metal Support: Metal 3
2. RAM
   (a) Memory: 16GB
   (b) Type: LPDDR5
   (c) Manufacturer: Hynix
3. OS
   (a) Version: macOS Sonoma 14.1.1
4. Python environment
   (a) Python==3.10

1.2 Experiment Results

1.2.1 Base Model

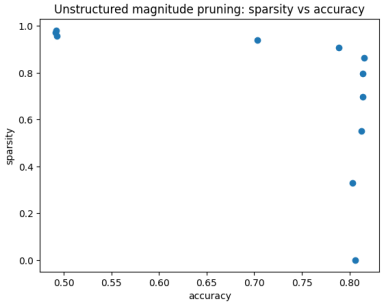
We used exactly the same hyperparameters for the SST2 task as specified in the lab instructions, and the input size is 5000, where we use the most frequent 5000 words as the vocabulary. We didn't apply stopword removal or other preprocessing techniques.

The results of the base model is shown in table 1. The total number of parameters is 1412354.

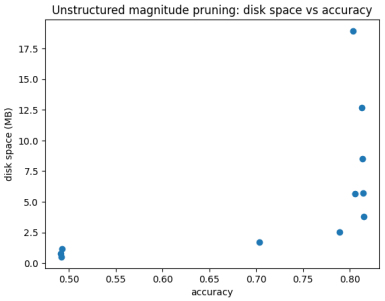
1.2.2 Unstructured Magnitude Pruning

After the first round of pruning, the sparsity result is shown in table 2.

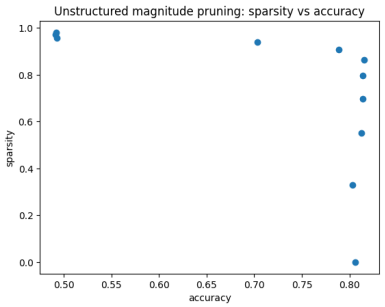
The results of repeated unstructured magnitude pruning is shown in table 3. The three plots are shown in figure 1.



(a) sparsity vs accuracy



(b) disk space vs accuracy



(c) latency vs accuracy

Figure 1: Unsrtuctured magnitude pruning.

1.2.3 Iterative Magnitude Pruning

The results of iterative magnitude pruning is shown in table 4. The three plots are shown in figure 2.

1.3 Discussion

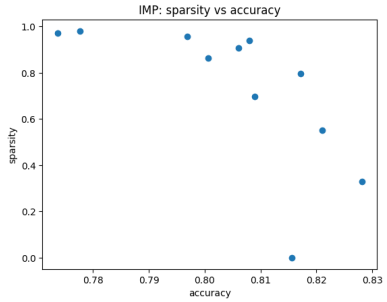
Both methods show a decrease in accuracy as sparsity increases. This trend is expected because removing weights from a network generally leads

Sparsity (%)	Accuracy	Latency (s)	Parameter Count	Disk Size (MB)
0.00000	0.819643	0.000100	1412354	5.651761

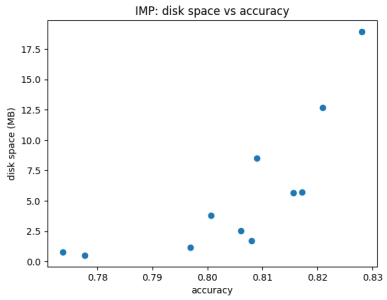
Table 1: Results of Base Model

Parameter	Sparsity (%)
mlp.0.bias	0.0
mlp.0.weight	35.5646875
mlp.2.bias	0.0
mlp.2.weight	8.1268310546875
mlp.4.bias	0.0
mlp.4.weight	7.98187255859375
mlp.6.bias	0.0
mlp.6.weight	7.421875
all_pruned_parameters	33.000019835872324
overall model	32.98202858490152

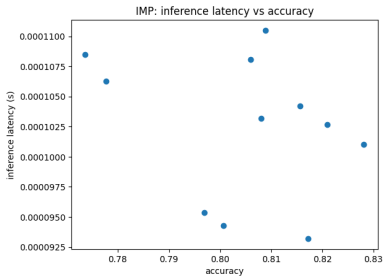
Table 2: First Round of Pruning Sparsity Results



(a) sparsity vs accuracy



(b) disk space vs accuracy



(c) latency vs accuracy

Figure 2: Iterative magnitude pruning.

to a loss of information which can degrade performance. However, the iterative magnitude pruning with rewinding shows a slower decline in accuracy. Even when the sparsity increases to 98%, the accuracy for iterative magnitude pruning is 77%, while that for repeated unstructured magnitude pruning drops significantly to 51%. It could indicate that rewinding helps the network better retain its performance at higher sparsity.

As the sparsity increases, the disk size first increases and then decreases. It is as expected because if we only prune a small proportion of parameters, the sparse representation stores more data than dense representation, since it needs to store both the index and the value. When the sparsity reaches a threshold, the disk size starts to decrease. Therefore, it is more memory saving to use sparse representation when the sparsity is high.

And as the accuracy decreases (with increased pruning and sparsity), the disk size decreases. It is expected as in traditional scenarios higher accuracy often requires mode model parameters, hence larger disk space. However, for the repeated unstructured magnitude pruning, when the model's disk size is tiny, the accuracy decreases significantly. This might indicate that the repeated unstructured pruning method is removing weights that are contributing to correct prediction, while the remaining parameters cannot make the network strong enough to make correct predictions. In the iterative magnitude pruning with rewinding plot, the trend is less pronounced, suggesting that rewinding may lead to a more compact and efficient model that retains higher accuracy with less disk space

usage.

We expected that with the sparsity increases and accuracy decreases, the latency will also decrease gradually. However, for both methods, we did not see a clear pattern between latency and accuracy. There is a small fluctuation of the latency when the model changes. We hypothesize that it is related to the lower implantation of the PyTorch framework and hardware support, which could be that we are not actually using the sparse format when we are doing the inference.

2 Part II: Your Model, Device, and Data

2.1 L1 Unstructured Pruning of the Whole Model

Our model is built for the Speech Translation task. The system is based on an Encoder-Decoder structure.

The hardware specification is shown below:

1. Laptop: Dell G15 5511
2. CPU
   (a) 11th Gen Intel i7-11800H, 2.30GHz
   (b) Speed 3.35 GHz
   (c) 8 cores, 16 logical processors
   (d) L1 cache: 640 KB
   (e) L2 cache: 10.0 MB
   (f) L3 cache: 24.0 MB
3. Memory: 32GB, dual-channel DDR4
4. OS: Windows 10 Education 22H2, OS Build 19045.3570
5. Python environment:
   (a) Python==3.10
   (b) PyTorch==1.12.1
   (c) ESPnet==202308
6. Test data is loaded into memory beforehand.

For this experiment, we did not iteratively prune the entire model with 33% of the remaining parameters each time. Since we found that the performance degradation is huge when the sparsity is becoming larger and larger, and iteratively pruning the model would cause the sparsity of the model to rapidly approach 0.9. The BLEU score is less than 10 when the sparsity is large or equal to 0.5. And when the sparsity is large or equal to 0.9, the BLEU score becomes zero, meaning that the output of the model makes no sense. Thus conduct the experiment on the sparsity.

From the experiment results (Table. 5) and the 3 plots (Figure. 3), we can get to the following conclusions:

1. Only when the model's sparsity is larger than 0.8, the model's disk space usage will be smaller than the original model, while when the sparsity is small (such as 0.1, 0.2) the model's disk space consumption will far larger than that of the original model. We consider it to be the same reason as mentioned in Part I.

2. When the model's sparsity is getting larger and larger, the model's latency in fact increases. Since the model's performance degrades as the sparsity gets larger, the autoregressive model's stop predictor will becoming worse and worse, thus when the model's sparsity is getting larger, the model is more likely not the stop at the right point and will continuously generate output until reaches the maximum length. Thus, the latency increases as the sparsity gets larger.

3. For the BLEU score vs. sparsity. We could see that the BLEU score of the model whose sparsity is less or equal to 0.2 is really close to that of the original model. We think that it is the true pruning, which prunes out some of the unnecessary or redundant parameters (having small magnitude). But when the sparsity of the model gets larger and larger, more and more the important parameters are pruned out, which causes huge performance degradation. If we retrain the model while pruning, we might be able to achieve higher sparsity while keeping an acceptable performance loss.

2.2 Sensitive Analysis of Different Components of the Model

Here we conduct experiments on unstructured pruning of different components:

1. encoder/decoder's:
   (a) feed forward layers (ff)
   (b) point/depth-wise convolution layer + batch normalization (conv)
   (c) self-attention, including the linear transformations (self_attn)
   (d) layer normalizations (norm)

Iteration	Sparsity (%)	Accuracy	Latency (s)	Disk Size (MB)
0	0.00000	0.819643	0.000100	5.651761
1	32.9820	0.818973	0.000099	18.921815
2	55.0800	0.816071	0.000101	12.679703
3	69.8856	0.805804	0.000100	8.497687
4	79.8053	0.803348	0.000098	5.695511
5	86.4516	0.796652	0.000106	3.818199
6	90.9045	0.791071	0.000098	2.560343
7	93.8881	0.729464	0.000100	1.717527
8	95.8870	0.774107	0.000100	1.152919
9	97.2263	0.682143	0.000097	0.774487
10	98.1236	0.510491	0.000104	0.521175

Table 3: Results of Unstructured Magnitude Pruning

Iteration	Sparsity (%)	Accuracy	Latency (s)	Disk Size (MB)
0	0.00000	0.815625	0.000104	5.651761
1	32.9820	0.828125	0.000101	18.921815
2	55.0800	0.820982	0.000103	12.679703
3	69.8856	0.808929	0.000110	8.497623
4	79.8053	0.817188	0.000093	5.695575
5	86.4516	0.800670	0.000094	3.818199
6	90.9045	0.806027	0.000108	2.560343
7	93.8881	0.808036	0.000103	1.717527
8	95.8870	0.796875	0.000095	1.152983
9	97.2263	0.773661	0.000108	0.774743
10	98.1236	0.777679	0.000106	0.521175

Table 4: Results of Iterative Magnitude Pruning

Experiment ID	Sparsity	BLEU	Latency (s)	Disk Size (MB)
0	0	56.59119	1.406278	220
1	0.1	56.59119	1.392705	842
2	0.2	56.59119	1.324602	753
3	0.33	29.50234	1.43281	639
4	0.4	26.2691	0.81771	577
5	0.5	3.379674	6.36895	490
6	0.6	2.747578	6.283118	402
7	0.7	1.349908	6.568361	314
8	0.8	0.679364	6.227498	226
9	0.9	0	6.228	138

Table 5: Results of L1 Unstructured Pruning (Whole Model)

2. Encoder's subsampling module (modules that reduces sequence length)

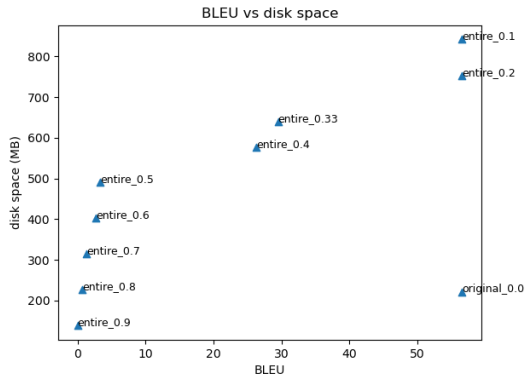
3. Decoder's cross attention module

4. Decoder's embedding module.

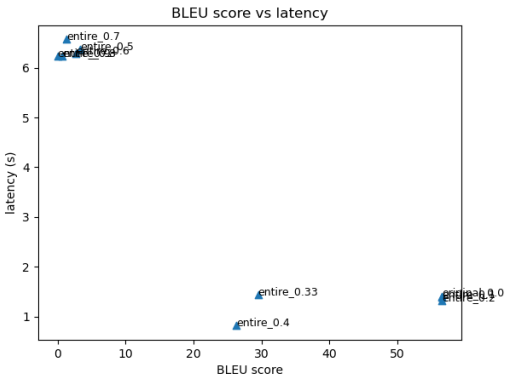
For the modules listed above, we conducted experiments pruning each module (L1 unstructured pruning) at 0.33, 0.5, 0.7, 0.9 sparsity. The plot below shows all results in a single plot; the BLEU score vs. model size plots for each module are shown in the Appendix.

From the plot (Figure. 4), we may be able to get to the conclusion that:

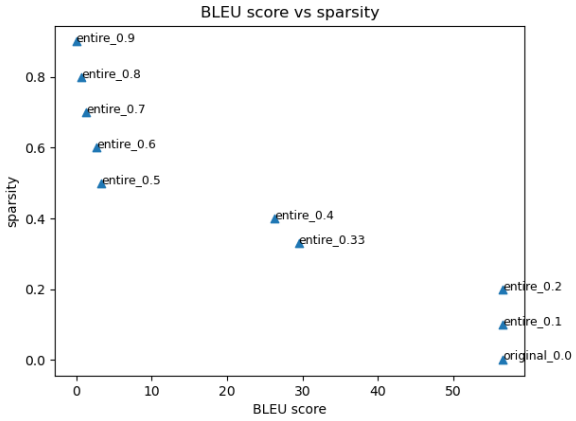
1. When decoder's module gets pruned, the model will likely have a large performance



(a) BLEU vs. Disk Space (MB)



(b) BLEU vs. Latency (s)



(c) BLEU vs. Sparsity

Figure 3: Comparative Plots For Different Sparsity

degradation. One explanation for this is that the decoder is larger than the encoder, and here the decoder acted like a machine translation module + ASR Decoder, thus the decoder is supposed to have a larger model space compared to encoder, which means that the decoder theoretically needs a lot of parameters. Even if we only prune 33% of the decoder's parameters, it is likely to cause underfitting.

2. It seems that pruning about 33% or even a little bit more parameters of the encoder will not cause a huge performance loss compared to the original model. We think that it might be due to the fact that here the encoder plays a lower-level acoustic feature extraction role, which theoretically does not need so many parameters. Thus pruning 30% of them will not greatly affect the performance.

3. An interesting thing here is that even if we pruned about 70% of the self-attention module of the encoder, the model could still achieve a BLEU score which is higher than 20. We think the reason is similar to the point we mentioned just above, the encoder here is not responsible for most of the semantic feature extraction, thus its attention module does not need a large model space.

Based on the experiment results and our analysis we designed an optimal solution that balances the sparsity and model's performance, which is listed below in a module, sparsity format below:

1. Encoder_subsampling, 0.9
2. Encoder_conv, 0.7
3. Encoder_ff, 0.5
4. Encoder_attn, 0.5
5. Encoder_embed, 0.5
6. Encoder_self_attn, 0.5
7. Encoder_cross_attn, 0.33

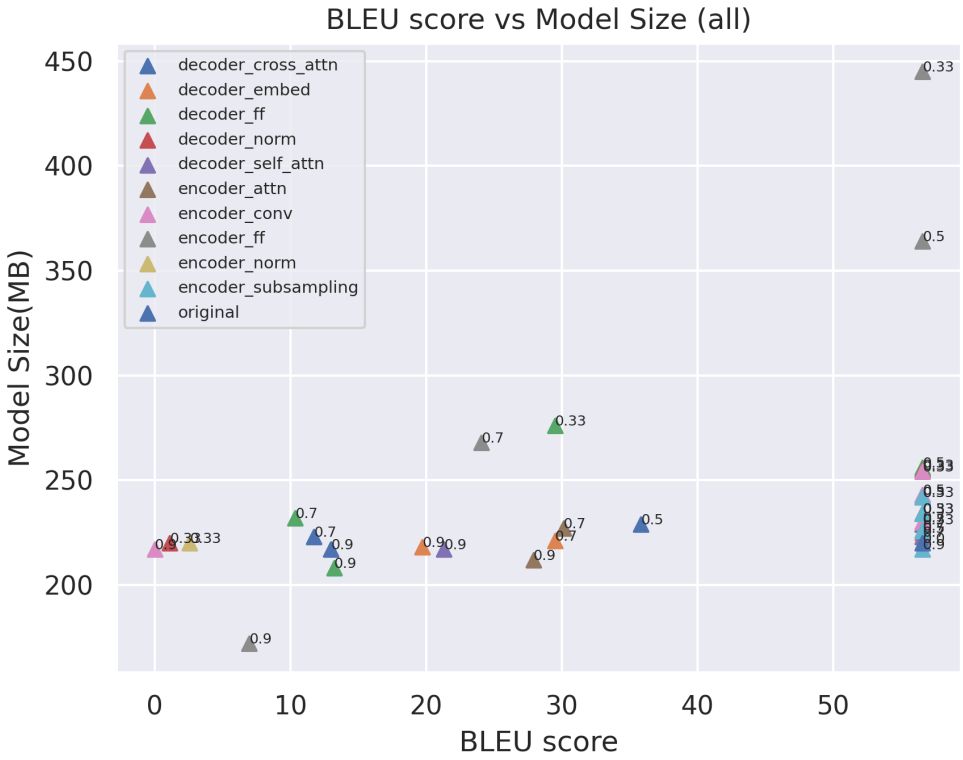


Figure 4: Comparative Plots For Different Sparsity on Different Modules

This solution can achieve a BLEU score of about 42.

2.3 ONNX runtime

Here we use the classes in espnet_onnx (https://github.com/espnet/espnet_onnx) to get the onnx-optimized version of our model. Since the original model recipe has a lot of operations like lambda, **kwargs, etc. which are invalid operations of torch.onnx.export with dynamic_shapes=True (we need it to generate models that can accept inputs of different sizes). If we want to use the torch toolkits, then we have to rewrite most of the classes in espnet, which is a huge workload. Thus, here we took advantage of the espnet_onnx, whose classes written in a torch.onnx-valid format.

The experiment results are listed below (Table. 6):

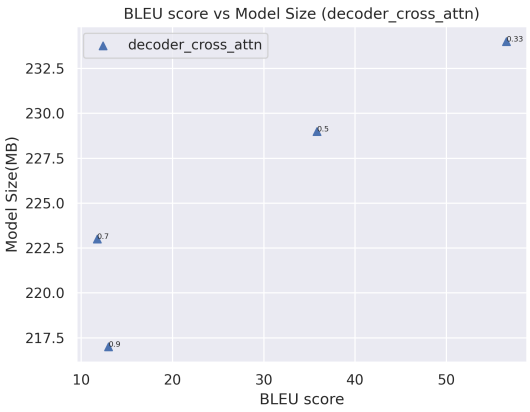
Here the original means that the original model, while the optimal refers to the optimal solution mentioned in the above section. We could see that ONNX can significantly reduce the inference time of no matter the original model or the pruned model.

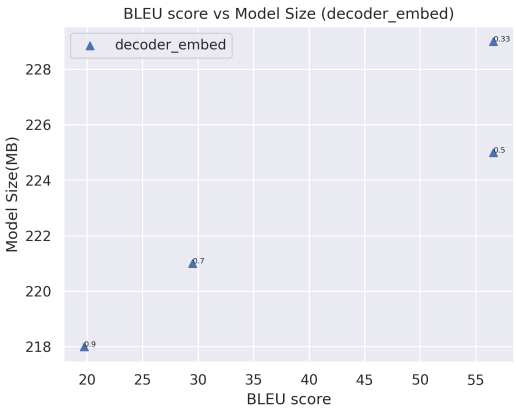
Model Type	ONNX	latency (s)
optimal	TRUE	0.65
optimal	FALSE	0.88
original	TRUE	0.98
original	FALSE	1.41

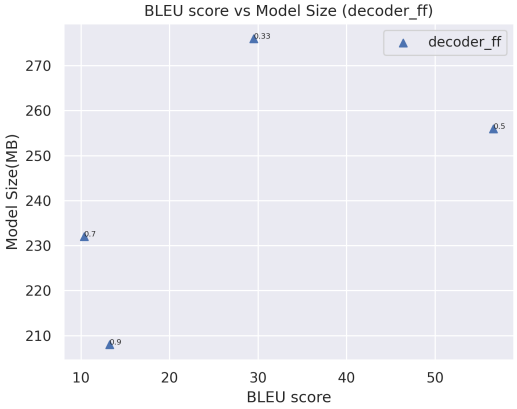
Table 6: ONNX vs. PyTorch

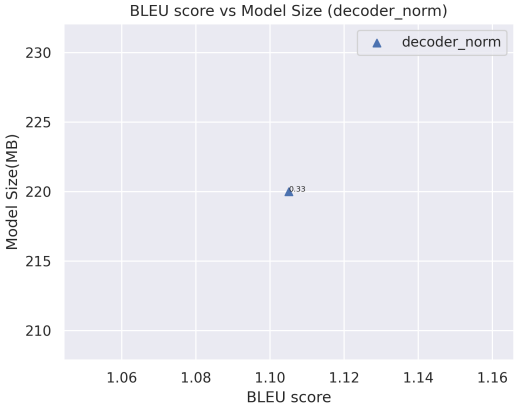
3 Appendix

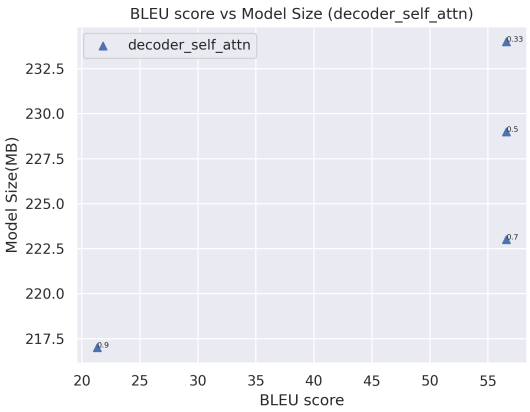
The BLEU score vs. model size plots for each module of different sparsity are shown in this section.

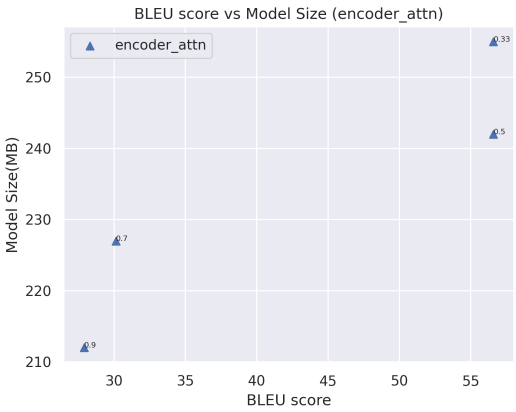

BLEU score vs Model Size (decoder_cross_attn)
decoder_cross_attn
Model Size(MB)
BLEU score
0.33
0.5
0.7
0.9

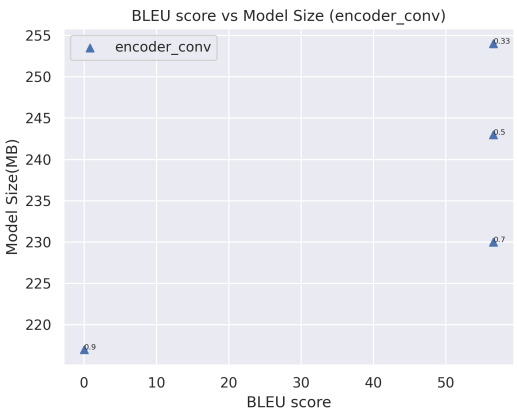

BLEU score vs Model Size (decoder_embed)
decoder_embed
Model Size(MB)
BLEU score
0.33
0.5
0.7
0.9

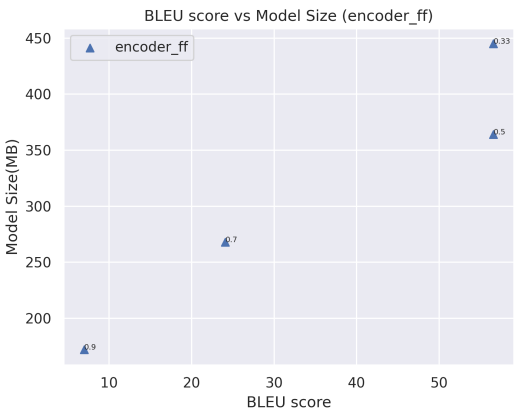

BLEU score vs Model Size (decoder_ff)
decoder_ff
Model Size(MB)
BLEU score
0.33
0.5
0.7
0.9

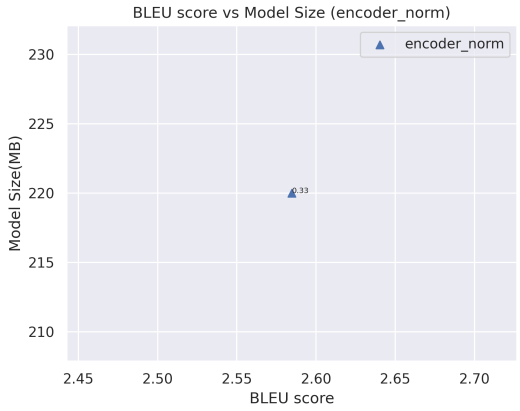

BLEU score vs Model Size (decoder_norm)
decoder_norm
Model Size(MB)
BLEU score
0.33

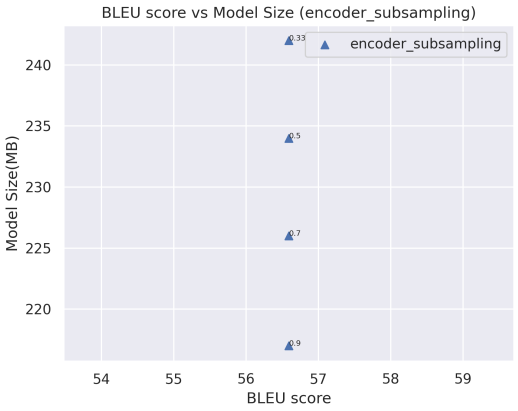

BLEU score vs Model Size (decoder_self_attn)
decoder_self_attn
Model Size(MB)
BLEU score
0.33
0.5
0.7
0.9


BLEU score vs Model Size (encoder_attn)
encoder_attn
Model Size(MB)
BLEU score
0.33
0.5
0.7
0.9


BLEU score vs Model Size (encoder_conv)
encoder_conv
Model Size(MB)
BLEU score
0.33
0.5
0.7
0.9


BLEU score vs Model Size (encoder_ff)
encoder_ff
Model Size(MB)
BLEU score
0.33
0.5
0.7
0.9


BLEU score vs Model Size (encoder_norm)
encoder_norm
Model Size(MB)
BLEU score
230
225
220
215
210
2.45
2.50
2.55
2.60
2.65
2.70


BLEU score vs Model Size (encoder_subsampling)
encoder_subsampling
Model Size(MB)
BLEU score
240
235
230
225
220
54
55
56
57
58
59