Project Report: On-device Speech Translation

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

Abstract

This project aims to develop an efficient on-device speech translation system, addressing the challenge of language barriers in a globalized society. Traditional translation methods can be cumbersome, particularly in areas with limited internet access. Our solution emphasizes on-device translation to enhance privacy, reduce latency. The crucial part of our approach is the implementation of model compression techniques, specifically quantization and pruning based on ESPNet, which are crucial for the deployment of efficient real-time systems on devices with limited computational resources. We further refine our system's efficiency by applying the Open Neural Network Exchange (ONNX) optimizations. Our methods resulted in a 62% reduction in latency and a 73% reduction in model size, with a trade-off of a 25% decrease in BLEU score, indicating a significant improvement in efficiency with a modest degradation in translation accuracy. These techniques do not require re-training the model and can be implemented easily using PyTorch and ONNX libraries.

1 Introduction

Language barriers are a common obstacle faced by travelers around the world. In an increasingly globalized society, the ability to communicate with people from diverse linguistic backgrounds is crucial for social interaction, tourism, business, and emergency situations. Traditional phrasebooks and online translation services can be cumbersome, especially in areas with limited internet connectivity.

On-device speech translation aims to bridge this gap. On-device machine learning has gained popularity due to its potential to enhance privacy and reduce latency. Deploying machine learning models on user devices reduces the reliance on centralized servers, improving the user experience and alleviating concerns about data security and privacy. Personal conversations and sensitive information are not transmitted to external servers, reducing the risk of data breaches or unauthorized access. What's more, nowadays edge device has sufficient power to run models offline, and deploying the models on the devices directly is helpful in reducing latency.

In this project, our objective is to build and deploy an end-to-end speech translation system to a laptop based on ESPnet. We try to tackle the challenge of designing and developing targeted methods of pruning and quantization tailored to ESPnet-based model. This approach is essential to ensure the deployment of the model on laptop devices with both acceptable inference latency and performance. We also run experiments to see how the model efficiency will affect the power consumption.

2 Related Works

For speech translations (ST), there are mainly two approaches: cascaded ST system and end-to-end (E2E) ST systems (E2E-ST). For the cascaded ST, the idea is to split the speech translation task into smaller and feasible sub-tasks: speech translation and machine translation (Xu et al., 2023). While E2E-ST aims to solve the speech translation problem with an E2E model. Both approaches have their problems. Cascaded systems have the issue of error accumulation while the E2E ST model is difficult to learn due to the cross-lingual and cross-modal mapping in a single model (Xu et al., 2023).

For E2E-ST, researchers usually use a multi-task framework to train the model, since the cross-lingual and cross-modal mapping make the training process much more challenging compared to training the sub-modules independently (Xu et al., 2023). There are several training strategies: decoupled decoder, decoupled encoder, and two-stream encoder. The decoupled decoder strategy reduces the modeling burden by adding a speech recognition (ASR) decoder in addition to the machine

translation (MT) decoder. For the decoupled encoder strategy, it adds an additional semantic encoder to the original pipeline. Other than directly encoding speech into semantic features, it eases the encoder's burden by first encoding speech into acoustic features and then encoding it into semantic features. Two-stream encoder approach adds a shared encoder, which takes either output from a speech encoder or machine translation text encoder, and its outputs will be used to generate translated text. This aims to map the speech encoder's feature space and the text encoder's feature space into the same one, which makes it easier to train the whole E2E-ST system.

Recently, a lot of work has been done on ASR/MT compression (most of them are quantization and pruning metrics, such as (See et al., 2016)), but little work has been done on ST model compression. We believe these techniques also apply to the ST task.

3 Task Definition and Problem Setup

Model In this project, our investigations focus on E2E-ST. We chose the decoupled decoder scheme mentioned above. In this scheme, the ASR decoder is not required during inference which removes some computation burden. Specifically, we have an speech encoder that extract high-level vector representations from the input speech features (FBank). Then the MT decoder predicts the next token based on the previous token sequence while attending to the encoder output.

Due to the heavy computation requirement for retraining the models, we utilize a pre-trained model from HuggingFace. The details about this model will be specified in later sections.

Dataset For the dataset, we use MuST-C (Di Gangi et al., 2019) for benchmarking experiments. MuST-C is a corpus designed for speech translation, primarily focusing on translating TED talks from English into several other languages. It is a multilingual dataset that provides sentence-level parings of speech and text. We utilize the English-to-Dutch language pairs from its test set to evaluate latency and translation accuracy.

Evaluation metric In order to measure the accuracy-performance trade-offs, it is crucial to employ robust evaluation metrics that effectively encapsulate the quality and efficiency of generated outputs. For our project, we selected BLEU (Bilingual Evaluation Understudy) and latency as our primary metrics. **BLEU** is a widely accepted metric in the translation community, known for measuring the correspondence of n-grams between machine-generated translations and their reference counterparts. A higher BLEU score indicates better performance.

However, while **BLEU score** effectively assesses translation accuracy, it does not evaluate the operational efficiency of the translation system. Therefore, we use **latency** as a significant metric. Latency measures the time taken for the system to produce the translation with the provided input. A lower **latency** indicates a more efficient system that offers faster translations. We also measure Floating-Point Operations per Second (FLOPS) for some experiments where we can get an accurate measurement using GPU and `deepspeed`[1] library. This metric measures how fast calculations are performed and can reflect whether the model is utilizing the underlying hardware efficiently.

Hardware The hardware we are targeting at is a Dell G15 5511 laptop. Its specifications are shown below:

1. 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
2. GPU
   (a) NVIDIA GeForce RTX 3060 Laptop GPU/PCIe/SSE2
   (b) Driver: NVIDIA 470.223.02
   (c) CUDA 12.3
3. Memory: 32GB, dual-channel DDR4
4. OS: Linux 6.2.0-37-generic x86 64-bit, Ubuntu 22.04.3 LTS
5. Python environment:
   (a) Python==3.10
   (b) PyTorch==2.0.1
   (c) ESPnet==202308

[1] https://github.com/microsoft/DeepSpeed

4 Methods

Deploying deep learning models on devices for real-time applications poses a significant challenge due to their computation and storage overhead. To address this challenge, we explore two techniques: quantization and pruning. Quantization reduces the precision of the model's weights in exchange for faster computation and smaller memory footprint. Meanwhile, pruning eliminates parts of the network to reduce the total computation operations. Both techniques aim to reduce the model's size and computational demands but potentially compromise the translation accuracy. We are curious about how much quantization and pruning we can apply to create a lightweight and efficient speech translation system.

Another aspect of our research is the choice of representation and interchange formats for neural network models. There are many software that optimizes the inference process of ML models. Typical optimization techniques include but are not limited to computation graph optimization and multithreading. In addition, these inference engines can decouple our model from the dependency on ESPnet, which makes deployment on smaller devices much more convenient. Our research objective is to export our model to the ONNX format and use ONNX runtime to perform inference. We are interested to see its impact on latency, translation accuracy, and model size.

Furthermore, speech recognition relies on the efficient representation and processing of audio signals. Speech encoders usually use convolution layers to consolidate input features into subsampled high-level representations. This reduces the computation needed since the sequence length is reduced. In this part, we seek to find a method that can increase the subsampling rate without significantly compromising the performance.

4.1 Baseline

To save time, we utilize a pre-trained model[2] from Hugging Face Hub for our experiments. As described in Section 2 and Section 3, this E2E-ST system is an encoder-decoder model with multitask training objectives (Inaguma et al., 2020). The encoder is a Conformer (Gulati et al., 2020) with 12 layers, each having 4 attention heads and 2048 hidden units. The MT decoder is a Transformer (Vaswani et al., 2017) with 6 layers, each containing 4 attention heads and 2048 hidden units. Both the encoder and the decoder use a dropout rate of 0.1 for positional encoding and attention. The beam size is set to 10 during inference.

[2] https://huggingface.co/espnet/brianyan918_mustc-v2_en-de_st_conformer_asrinit_v2_raw_en_de_bpe_tc4000_sp

4.2 Reduce Input Size

We propose two methods to decrease input size. For the first method, we change the hop length of the sliding windowing function when calculating FBank features. The larger the hop length is, the more windows are spread out, and thus the smaller the input size is. However, the sampling rate within each window remains the same, therefore this technique is unlikely to cause huge information loss.

Another way to vary the input size is to reduce the sample rate from 16k to 8k using the ffmpeg toolkit. This can be combined with the first method to further shrink the input size. We aim to explore if subsampling the original wav can lower the input size without significantly harming the performance.

4.3 Quantization

We believe Post-Training Quantization is most suitable for this project, since Quantization-Aware Training (QAT) is not realistic because we do not have sufficient resources to train this model. PTQ is shown to be quite effective for many speech tasks, and we believe with proper implementation and tuning, we can achieve a big reduction of model size and inference latency. We plan to quantize the model using 8-bit integer instead of 16-bit float, because the PyTorch CPU backend does not provide native support for such data types, and our experiments in later sections show that these types actually increase the latency.

In addition, our investigations are focused on dynamic quantization instead of static quantization. Static quantization requires calibration which is difficult to implement due to ESPnet's complexity. During beam search, we need to quantize and dequantize the input and output of the MT decoder for each step. This requires a substantial change to ESPnet's internal code, and we simply didn't have enough time.

4.4 Pruning

For our project, we apply unstructured L1 magnitude pruning to our model. We experiment with global pruning and component-wise pruning with different sparsity to find the best pruning strategy

for this task. Our goal is to reduces the inference latency as much as possible without damaging the translation quality.

4.5 ONNX Runtime

As explained above, we use ONNX to export our models and perform inference using ONNX runtime. We use `espnet_onnx`[3] library to assist us exporting the encoder and the decoder separately. The original ESPnet codebase has a lot of operations like lambda and argument unpacking, which `torch.onnx.export` fails to deal with automatically. In addition, `espnet_onnx` can help us specify the dynamic input dimensions automatically.

4.6 Combining All Techniques

We are interested to see whether combining the techniques mentioned above can further increase the efficiency of our models. We aim to investigate how these techniques affect each other in terms of inference latency and model sizes. For example, combining pruning with ONNX runtime can potentially reduce the number of calculations because ONNX can fuse pruned modules together can increase sparsity even more.

4.7 Measuring Energy Use

We measure the energy use of our model on the two platforms which are laptop CPU and laptop GPU. For laptop CPU, we use CodeCarbon combined with Intel RAPL (Running Average Power Limit) Interface to measure energy consumption. It provides both a high-level estimation of energy use and a more detailed measurement of CPU. This approach can account for the total energy consumed by the system and the specific energy use of model inference. For laptop GPU, we use CodeCarbon combined with PyNVML. In this way, it can directly query the GPU for its power state, which is crucial for an accurate assessment of energy consumption.

We use a test dataset of 100 utterances to benchmark our model. For each experiment, we run the inference on the test dataset four times to get a stable estimate. We repeated the experiment five times to report the mean and standard deviation of the energy consumption per utterance. We set our batch size to 1, since out model is an autoregressive model which can accept input with different lengths. Thus, it is hard for us to implement efficient batch inference (ESPnet does not support batch inference well, either). We start our power measurement when the model starts inferencing the first sample, and stop measuring when the model finishes inferencing all the samples. Since there is very little overhead between the inference of two samples, we decided to ignore this part's energy consumption. We decided not to perform measurements at the wall (since we use PC platform).

[3] (https://github.com/espnet/espnet_onnx)

5 Results and Discussion

In this section, we present our experimental results and discuss their implications. All experiments were conducted using ESPnet (Watanabe et al., 2018).

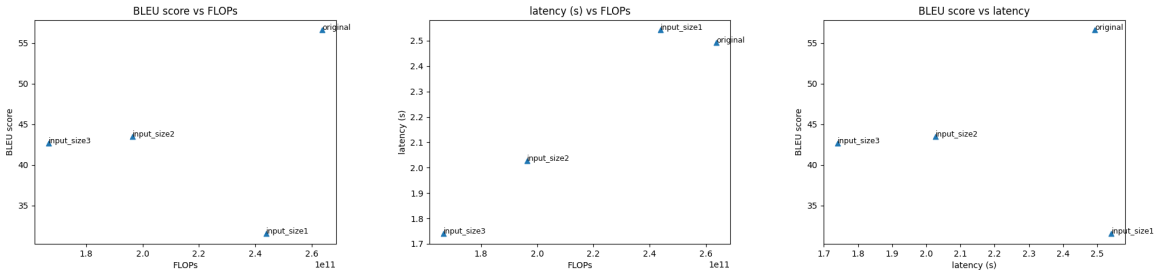
5.1 Varying Input Size

The hop length and sampling rate for each experiment are shown in Table 1. The experiment results for our first method (increase hop length only) are shown in Figure 1. From the BLEU score vs FLOPs plot we observe that as we decrease the input size, the FLOPs decrease as expected. As the FLOPs decrease, we don't see a clear trend of increasing or decreasing BLEU scores. We think the reason is that we didn't re-train the model after changing the hop length. Since the model is trained using a hop length of 160 (16KHz wav), inference with different hop lengths will affect and bring a lot of uncertainty to the BLEU score performance.

The BLEU score vs latency graph looks similar to the BLEU score vs FLOPs one. The latency decreases as we decrease the input size. The BLEU score also drops as we change the input size because of the loss of information brought by increasing the hop length and the mismatch between the training and inference features (led by different hop length).

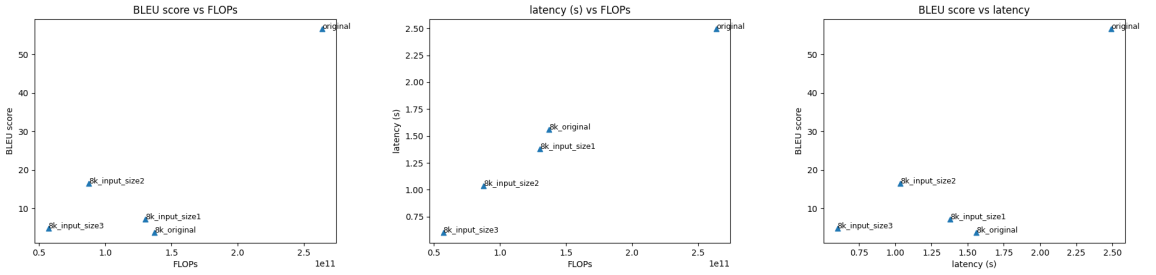
For the relationship between FLOPs and latency, we find that latency increases as the FLOPs increases as expected.

We observe an interesting phenomenon for input_size1: its latency is higher than the original method, but its BLEU score is much lower than that of the original and input_size2 and input_size3. We hypothesize that this is because of the inconsistency between the training and inference features. The model produces bad quality outputs without stopping tokens until it reaches the maximum output length, resulting in higher latency and very low



(a) FLOPs vs. BLEU (b) FLOPs vs. Latency (c) Latency vs. BLEU

Figure 1: Comparative Plots For Different Input Size (Method 1)



(a) FLOPs vs. BLEU (b) FLOPs vs. Latency (c) Latency vs. BLEU

Figure 2: Comparative Plots For Different Input Size (Method 2)

BLEU score.

For the second way of varying input size (first subsample the original wav and then increase the hop length), the experiment results are shown in Figure 2. By examining the graph depicting the relationship between BLEU score and FLOPs, it is apparent that reducing the input size leads to the expected decrease in FLOPs. However, as FLOPs decrease, there isn't a discernible pattern of either an increase or a decrease in the BLEU score. The graph illustrating the relationship between BLEU score and latency closely resembles the one comparing BLEU score to FLOPs. As the input size decreases, latency also decreases. The BLEU score tends to decrease when adjusting the input size due to disparities between the training and inference input sizes, although there isn't a clear, consistent trend observed. When examining the connection between FLOPs and latency, we generally observe that as FLOPs increase, latency also tends to increase.

In the analysis of the experiment results of the two methods, we observe that in terms of the BLEU score, if we decrease the sample rate by half, the BLEU score drops dramatically, except input_size2, the BLEU scores for all other models drop to less than 10. We think the reason is that the dropped samples would cause the inconsistency between the training and inference input features. For example, The model was trained with normal speech rate data, and he may not recognize accelerated speech (subsampling the raw wav and increasing the hop length is similar to accelerate the speech) very well. Failing to correctly recognize the speech will negatively affect the translation quality. In terms of latency, we find that if we decrease the sample rate first, the latency will decrease significantly. It is as expected because there is less computation. In all, we think decreasing the sample rate is not a good strategy for decreasing the input size since the BLEU score drops too much. In comparison, changing hop length directly has a better trade-off between performance and latency.

Model	Hop length
original	160
input_size1	200
input_size2	250
input_size3	300

Table 1: Experiments of Varying Input Size

5.2 Quantization

We performed three sets of experiments:

- Baselines without quantization

Configuration	Q	dtype	Size (MB)	Params	BLEU	Latency(s)
Original	-	float32	220	57,592,032	56.6	1.41
input_size1	-	float32	220	57,592,032	31.5	1.24
input_size2	-	float32	220	57,592,032	43.5	1.10
input_size3	-	float32	220	57,592,032	42.7	0.97
Original	D	qint8	70	57,592,032	35.8	1.20
input_size1	D	qint8	70	57,592,032	35.8	1.06
input_size2	D	qint8	70	57,592,032	43.5	0.96
input_size3	D	qint8	70	57,592,032	37.2	0.92
Original	D	float16	220	57,592,032	56.6	2.05
input_size1	D	float16	220	57,592,032	31.5	1.76
input_size2	D	float16	220	57,592,032	43.5	1.53
input_size3	D	float16	220	57,592,032	42.6	1.40

Table 2: Quantization under different settings.

- Dynamic PTQ with 8-bit (integer) precision
- Dynamic PTQ with 16-bit (float) precision

Figure 4 shows our results with each set of experiments marked with different colors:

There are several interesting conclusions (with and without PTQ, and using different precisions) we can draw from this plot: decreasing the input size reduces the inference latency regardless of whether quantization is enabled.

Dynamic PTQ using 8-bit integer significantly reduces the inference latency. However, it also reduces the BLEU score, especially when the input size isn't reduced (comparing *original* in blue and green). Moreover, the difference in latency is less significant when the input size becomes smaller (for example, comparing *original* and *input_size3* in blue and green). Strangely, *input_size2* with 8-bit integer PTQ achieves higher BLEU score than other reduced input size experiments even without PTQ. We believe somehow this configuration introduces the least precision loss, showing that calibration is very important for PTQ. Overall, this level of decrease in BLEU score is acceptable for our project. But we will try to improve this using static PTQ in the future.

Another interesting finding is that dynamic PTQ using 16-bit float drastically increase the inference latency. Meanwhile, the BLEU scores stay the same as the baselines. The reason is that X86 PyTorch backend doesn't natively support FP16, therefore needs to emulate 16-bit precision using FP32. Also, speech translation models have more diverse layers and that requires more value conversions, causing the latency to increase without the benefit of reduced model size. Since our target platform is X86 CPUs, we will not use this type of quantization in the final product.

5.3 Prunning

5.3.1 Global Prunning

For this experiment (results shown in Table 3), 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.

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

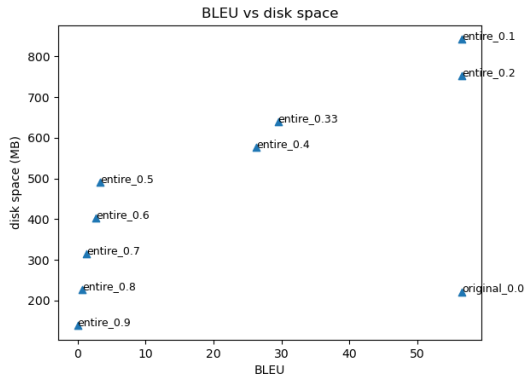
Table 3: Results of Global L1 Unstructured Pruning

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

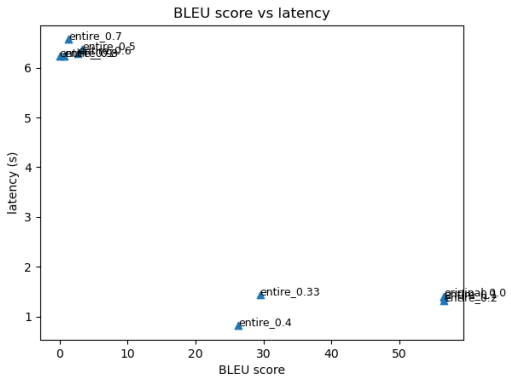
1. Only when the model's sparsity is larger than 0.8, the model's disk space usage will be smaller than the original model, while when the sparsity is small (such as 0.1, 0.2) the model's disk space consumption will far larger than that of the original model. We consider it to be the same reason as mentioned 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 become worse and worse, thus when the model's sparsity is getting larger, the model is more likely not to stop at the right point and will continuously generate output until reaches the maximum length. Thus, the latency increases as the sparsity gets larger.

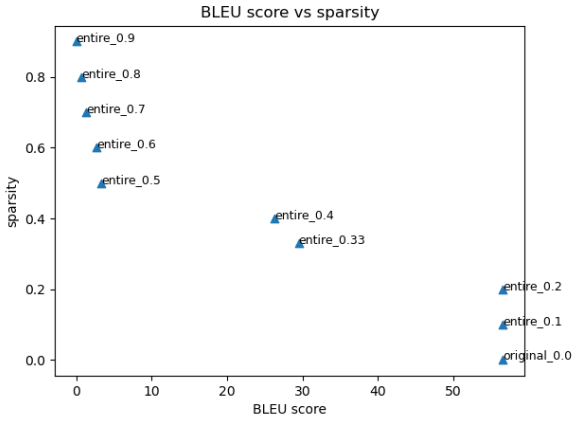
3. For the BLEU score vs. sparsity. We could see that the BLEU score of the model whose sparsity is less or equal to 0.2 is really close to that of the original model. We think that it is



(a) BLEU vs. Disk Space (MB)



(b) BLEU vs. Latency (s)



(c) BLEU vs. Sparsity

Figure 3: BLEU scores, latency and disk space of globally pruned models with different sparsity settings

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.

5.3.2 Sensitive Analysis of Different Components of the Model

Here we conduct experiments on unstructured pruning of different components:

1. Encoder's or 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)
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.

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 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.

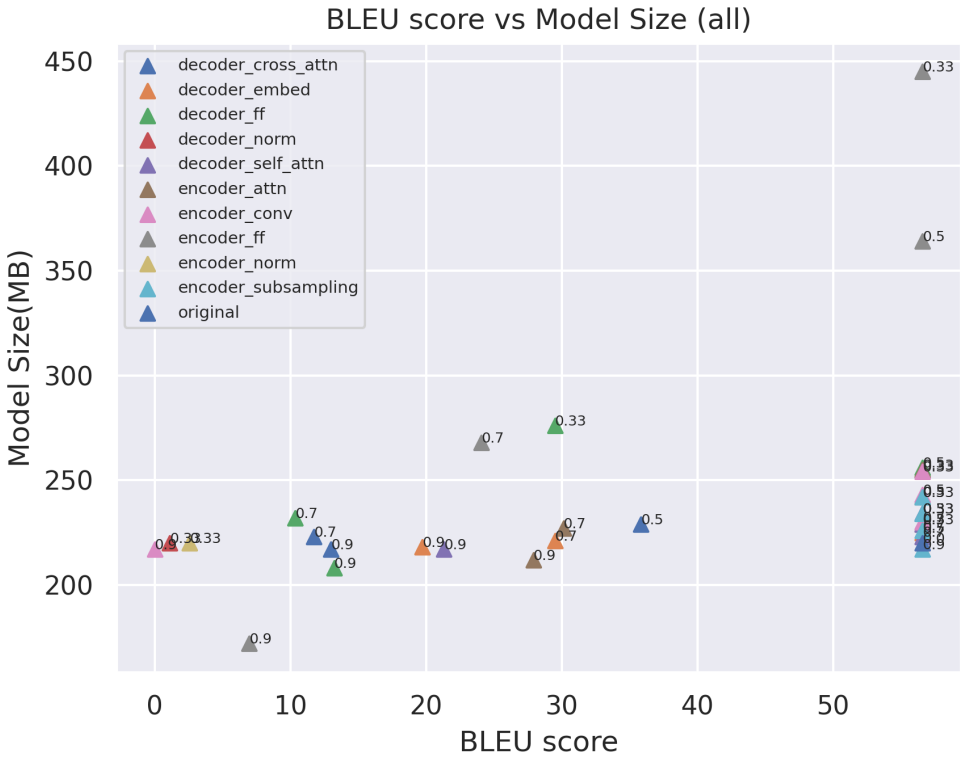


Figure 4: Comparative Plots For Different Sparsity on Different Modules

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 in Table 4. This solution can achieve a BLEU score of about 42.

Component	Sparsity
Encoder_subsampling	0.9
Encoder_conv	0.7
Encoder_ff	0.5
Encoder_attn	0.5
Encoder_embed	0.5
Encoder_self_attn	0.5
Encoder_cross_attn	0.33

Table 4: The optimal pruning strategy based on our experiment results

5.3.3 ONNX runtime

Figure 5 shows the latency and BLEU scores of different combination of ONNX, quantization and pruning. We can see that ONNX consistently reduces the inference time without any reduction of BLEU score, proving the effectiveness of this technique. In addition, most of the performance degradation come from pruning, suggesting further refinement of the sparsity settings is required. Interestingly, quantization using ONNX as a backend does not affect BLEU score, but this is not true if using PyTorch. We conjecture that ONNX's implementation of dynamic PTQ may be more stable than PyTorch's.

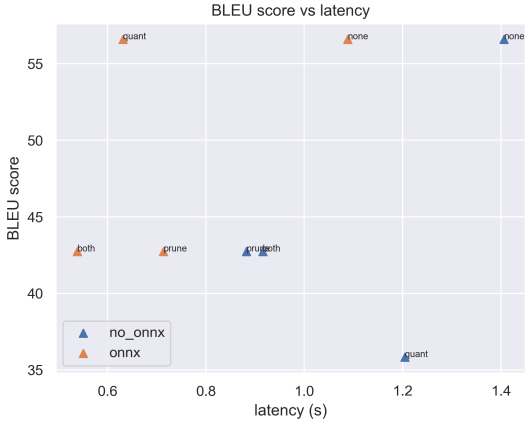


Figure 5: Ablation Study of quantization, pruning, and ONNX techniques

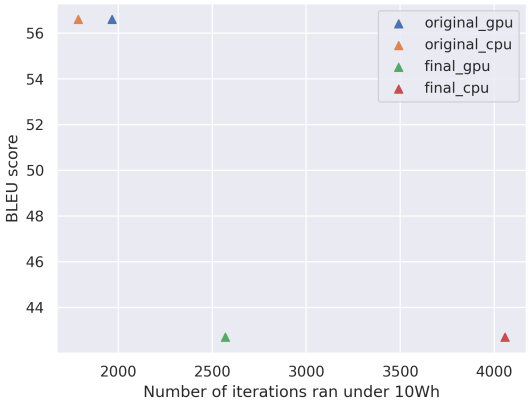


Figure 6: BLEU score vs. number of iterations under 10Wh

5.4 Putting it together

Model Type	BLEU	latency (s)
Original	56.6	1.41
PTQ + Pruning + ONNX	42.7	0.54

Table 5: The most efficient model vs. the original model.

To get the most efficient model based on previous discussion, we apply component-wise pruning with settings shown in Table 4, convert it to ONNX format, and then apply 8-bit integer dynamic PTQ using ONNX runtime. We benchmark this model and the results are shown in Table 5. It's straightforward to see that these techqniues reduce the latency and model size by -62% and -73% respectively, and only led to a -25% relative reduction of the BLEU score. We consider this reduction of BLEU score acceptable since a BLEU score of 42.7 is very high among speech translation benchmarks.

5.5 Measuring Energy Use

The graph shows that for the original model, the GPU is slightly more energy-efficient than the CPU for inference. We think this might because that the GPU's parallel computing mode is more efficient than the CPU's sequential computing mode. Moreover, the 11th gen Intel CPU is not very energy-efficient. Therefore, for the original model, GPU is a better choice for inference.

After applying quantization, pruning, and ONNX optimization, our best model is much more energy-efficient than the original one with a small performance loss. Surprisingly, CPU outperforms GPU in terms of energy efficiency. This could be due to that the ONNX implementation is more optimized for the CPU's Instruction Set Architecture (ISA), or maybe CPU handles low-resource tasks better. However, we do not have much context about the underlying architecture and implementation of these two platforms, so these are only speculations.

5.6 Challenges

In this project, we faced two main challenges. The first is that exporting the model to ONNX is cumbersome, mainly due to the beam search process. We used *espnet_onnx* as a starting point to make this easier, and the encoder and decoder are exported separately. The second challenge is that our experimentation with input size yielded surprising results. If time permits, we need to conduct more ablation studies to examine the reasons for the results for varying input size.

5.7 Insights

In our project, we explored several optimization techniques for the on-device speech translation system based on ESPNet. Our insights revealed that

- Increasing the hop length of the input significantly reduces latency of conformer, while maintaining an acceptable level of performance with minimal information loss. Conversely, downsampling the original waveform led to great latency reduction but resulted in considerable performance degradation and substantial information loss (about 50% compared to increasing hop length).

- Post-training quantization(PTQ) with 8-bit in-

tegers enhanced latency without damaging the BLEU score, whereas PTQ with 16-bit floats increases latency due to the underlying 32-bit float operations in x86 architecture.

- Pruning experiments indicated that while decoder layer are sensitive to pruning, which could lead to poor outputs. Certain layers like LayerNorm will cause a performance drop after pruning. Certain layers like LayerNorm and decoder self-attention could tolerate more aggressive pruning.

5.8 Future Work

In response to the potential decrease in BLEU scores resulting from pruning, our future work involves an in-depth exploration of hyperparameters to identify the most optimal settings. This may include Quantization-Aware training (QAT) and a meticulous search for hyperparameter configurations that mitigate the impact of pruning on model quality.

Additionally, we will explore structured pruning, strategically assessing the removal of specific components to ensure that model efficiency is enhanced without compromising overall performance.

What's more, our project will conduct a comprehensive investigation into the disparities between Torch and ONNX implementations of quantization. By comparing these frameworks, we aim to discern the subtle nuances in their approaches to quantization, providing valuable insights for further model refinement and optimization.

References

Mattia A Di Gangi, Roldano Cattoni, Luisa Bentivogli, Matteo Negri, and Marco Turchi. 2019. Must-c: a multilingual speech translation corpus. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2012–2017. Association for Computational Linguistics.

Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, and Ruoming Pang. 2020. Conformer: Convolution-augmented transformer for speech recognition. In *Interspeech 2020, 21st Annual Conference of the International Speech Communication Association, Virtual Event, Shanghai, China, 25-29 October 2020*, pages 5036–5040. ISCA.

Hirofumi Inaguma, Shun Kiyono, Kevin Duh, Shigeki Karita, Nelson Enrique Yalta Soplin, Tomoki Hayashi, and Shinji Watanabe. 2020. Espnet-st: All-in-one speech translation toolkit. *arXiv preprint arXiv:2004.10234*.

Abigail See, Minh-Thang Luong, and Christopher D Manning. 2016. Compression of neural machine translation models via pruning. *arXiv preprint arXiv:1606.09274*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008.

Shinji Watanabe, Takaaki Hori, Shigeki Karita, Tomoki Hayashi, Jiro Nishitoba, Yuya Unno, Nelson Enrique Yalta Soplin, Jahn Heymann, Matthew Wiesner, Nanxin Chen, Adithya Renduchintala, and Tsubasa Ochiai. 2018. Espnet: End-to-end speech processing toolkit. In *Interspeech 2018, 19th Annual Conference of the International Speech Communication Association, Hyderabad, India, 2-6 September 2018*, pages 2207–2211. ISCA.

Chen Xu, Rong Ye, Qianqian Dong, Chengqi Zhao, Tom Ko, Mingxuan Wang, Tong Xiao, and Jingbo Zhu. 2023. Recent advances in direct speech-to-text translation. *arXiv preprint arXiv:2306.11646*.