**Agenda item:** 9.6

**Source:** Qualcomm Inc.

**Title: [FS\_AI4Media] Intermediate data and split inference evaluation for real-time translation scenario**

**Document for** Discussion andAgreement

# Introduction

During the SA4 meeting #129-e, a new test scenario was added to the evaluation permanent document in [1]. The evaluation scenario covers the use case of language translation during a call and is representative of the IMS integration of AI/ML.

In this contribution, we provide an additional script and an evaluation of the split inference based on configuration scenario 2.

# Implementation

Neural machine translation has significantly advanced in recent years, with Transformer models achieving state-of-the-art results. In this contribution, we describe a concise and effective implementation that we provide in GitHub of such a model using PyTorch. We detail the core components and their functionalities, to facilitate the subsequent usage in describing the different deployment options and split configurations.

## 2.1 Model Architecture

The core of the sequence-to-sequence (Seq2Seq) language translation model is the Translator class, which utilizes the torch.nn.Transformer module. This module comprises an encoder and a decoder, both composed of stacked self-attention layers and feed-forward networks. The architecture can be summarized as follows:

* **Embedding Layers:** Both source and target language tokens are embedded into a continuous vector space using TokenEmbedding layers.
* **Positional Encoding:** PositionalEncoding is applied to the embeddings to inject information about the position of each token in the sequence, crucial for the Transformer to capture word order.
* **Transformer Encoder:** The encoder processes the source sentence by passing it through multiple layers of self-attention and feed-forward networks. This generates a contextualized representation of the input sequence.
* **Transformer Decoder:** The decoder attends to the encoder's output and generates the target sentence token by token. It uses masked self-attention to prevent attending to future tokens, ensuring the translation process remains autoregressive.
* **Output Layer:** A linear layer (fc\_out) projects the decoder's output to the target vocabulary size, followed by a softmax function to produce probabilities for each target token.

The following figure depicts the architecture of the model:



## 2.2 Key Components

**Token Embedding (TokenEmbedding)**

This module converts input tokens into dense vector representations. It employs an embedding layer (nn.Embedding) to map each token to its corresponding embedding vector. The embeddings are then scaled by the square root of the embedding dimension for stability.

**Positional Encoding (PositionalEncoding)**

Since the Transformer architecture processes sequences in parallel, it lacks inherent understanding of word order. PositionalEncoding addresses this by adding positional information to the token embeddings. It uses a combination of sine and cosine functions with different frequencies to encode the position of each token.

**Masking**

Two types of masks are employed in the system:

* **Padding Mask:** Handles variable-length sequences by masking out padding tokens during attention calculations. This prevents the model from attending to irrelevant padding tokens.
* **Subsequent Mask:** Used in the decoder to prevent attending to future tokens during training. This ensures the model generates the target sequence autoregressively.

**Transformer Module (torch.nn.Transformer)**

The torch.nn.Transformer module provides the core functionality of the Transformer architecture. It handles the encoder-decoder structure, self-attention mechanisms, and feed-forward networks. This module significantly simplifies the implementation and allows for efficient training and inference.

**Training and Inference**

During training, the model is optimized to minimize the cross-entropy loss between the predicted and target token sequences. The loss function used calculates the distance between the embedding of the source and the target tokens. Label smoothing is used to increase flexibility of the model.

The scripts have tools to download additional language data sets and to prepare them for training and inference. The training script makes use of underlying GPUs for faster training. Pretrained models for English-German and English-French are included.

The code is available on the 5G-MAG hosted GitHub repository rt-ai-ml-evaluation-framework: [5G-MAG/rt-ai-ml-evaluation-framework (github.com)](https://github.com/5G-MAG/rt-ai-ml-evaluation-framework/tree/main) under scripts/translation folder. Trained models for English-French and English-German language pairs are available under the models folder and are stored using LFS. These models were trained locally with only 100 epochs and a limited vocabulary, so the quality of the translation should not be compared to commercially available translation models and tools. Additional languages can be easily added and trained using the provided scripts.

# Configuration Scenarios

During an IMS call, one of the users may request the activation of the real-time translation functionality. The process will then involve real-time speech-to-text, followed by translation to the target language, and finally a text-to-speech rendering. The translation task itself may be broken down into the following steps, as depicted in the figure above:

* Tokenization and embedding
* Encoding
* Decoding
* Generation

Tokenization, embedding, and encoding are highly dependent on the source language. Decoding and generation are highly dependent on the target language. Based on this, the following configurations are possible:

**Configuration 1: Fulling processing at MF/MRF**



**Configuration 2: Split processing at UE1 and UE2**



**Configuration 3: Split processing at UE1, MF/MRF, and UE2**



Other variations of these configurations are also possible but we suggest to focus on these three configurations in the evaluation.

For the realization of the text-to-speech and speech-to-text functions, we suggest using the Web Speech API by W3C, which enjoys reasonable support on existing browsers.

# Split Inference Evaluation

## 4.1 Evaluation of Configuration Scenario 2

We have implemented configuration 2, which can be found under “translate\_split.py”. The intermediate data is a float tensor of dimensions [NUM\_TOKENS, EMBEDDING\_SIZE]. In a real-time conversation, it is expected that NUM\_TOKENS will be quite low to enable real-time translation. A value between 1 and 40 would be reasonable. The EMBEDDING\_SIZE is typically 256 or 512.

The following table shows an approximate breakdown of the average number of syllables spoken per second for different languages:

| **Language** | **Syllables per Second** | **Estimated Words per Second (Average)** |
| --- | --- | --- |
| **Japanese** | ~7.84 | ~3-4 |
| **Spanish** | ~7.82 | ~3-4 |
| **Italian** | ~6.99 | ~2-3 |
| **French** | ~7.18 | ~2-3 |
| **English** | ~6.19 | ~2-3 |
| **Arabic** | ~6.10 | ~2-3 (depending on dialect) |
| **Mandarin Chinese** | ~5.18 | ~2 (dense information per word) |
| **German** | ~5.97 | ~2-3 |
| **Vietnamese** | ~4.39 | ~1.5-2 |
| **Thai** | ~4.70 | ~2 |

Note that the number of tokens roughly corresponds to the number of syllables in a sentence. Using this information, we can estimate that the resulting bitrate will typically range from 4kbps to 16 kbps.

Note: in this calculation, the NUM\_TOKENS is not relevant. Instead only the number of syllables, which is equivalent the number of tokens that are generated per second is relevant. The history is not assumed to be sent but can be built on both sides simultaneously by accumulating the generated toekens.

# Proposal

We propose to document the content of section 4 in the evaluation PD. We also suggest to continue the evaluation of configuration 3 as well.

# References

[1] 3GPP S4-241716, Evaluation Permanent Document v0.7.0