

AI VIET NAM – AI COURSE 2025

# Tutorial: Dịch thuật Hoa-Việt (Olympic AI)

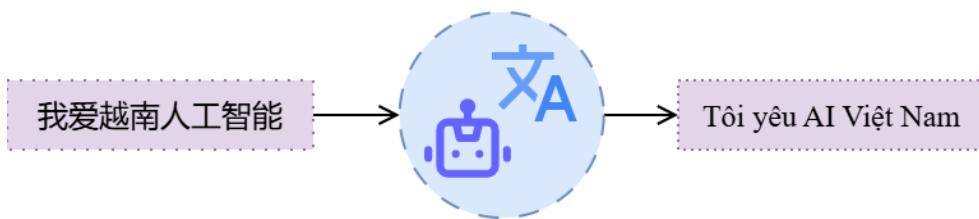
Nguyễn Quốc Thái  
Xin Quý HùngHồ Quang Hiển  
Đinh Quang Vinh

## I. Giới thiệu

Cuộc thi Olympic AI 2025 (OlpAI'25) được Hội Tin học Việt Nam tổ chức lần đầu tiên, dành cho các bạn học sinh sinh viên có đam mê và muốn thử sức với các bài toán trong lĩnh vực Trí tuệ nhân tạo (AI). Tuy mới tổ chức lần đầu, cuộc thi vẫn nhận được nhiều sự quan tâm và tham gia từ các thí sinh khắp cả nước. OlpAI'25 mang đến nhiều đề thi thú vị, nằm trong nhiều lĩnh vực: Học máy (ML), Thị giác máy tính (CV) và Xử lý ngôn ngữ tự nhiên (NLP).

Cuộc thi gồm 2 vòng: Vòng khu vực (theo các miền Bắc - Trung - Nam) và vòng chung kết. Ở vòng đầu tiên, mỗi khu vực sẽ do một trường trong khu vực đó đăng cai. Trong đó, vòng thi miền Bắc sẽ do Học viện Công nghệ Bưu chính Viễn thông (PTIT) đăng cai. Đề thi ở mỗi miền sẽ khác nhau, do đơn vị đăng cai tổ chức.

Ở vòng miền Bắc, các thí sinh cần phải giải quyết 2 bài toán thuộc 2 lĩnh vực khác nhau: CV và NLP. Đối với NLP, chủ đề được đưa ra là **Dịch thuật Hoa - Việt**, một bài toán thuộc nhánh Machine Translation (MT).

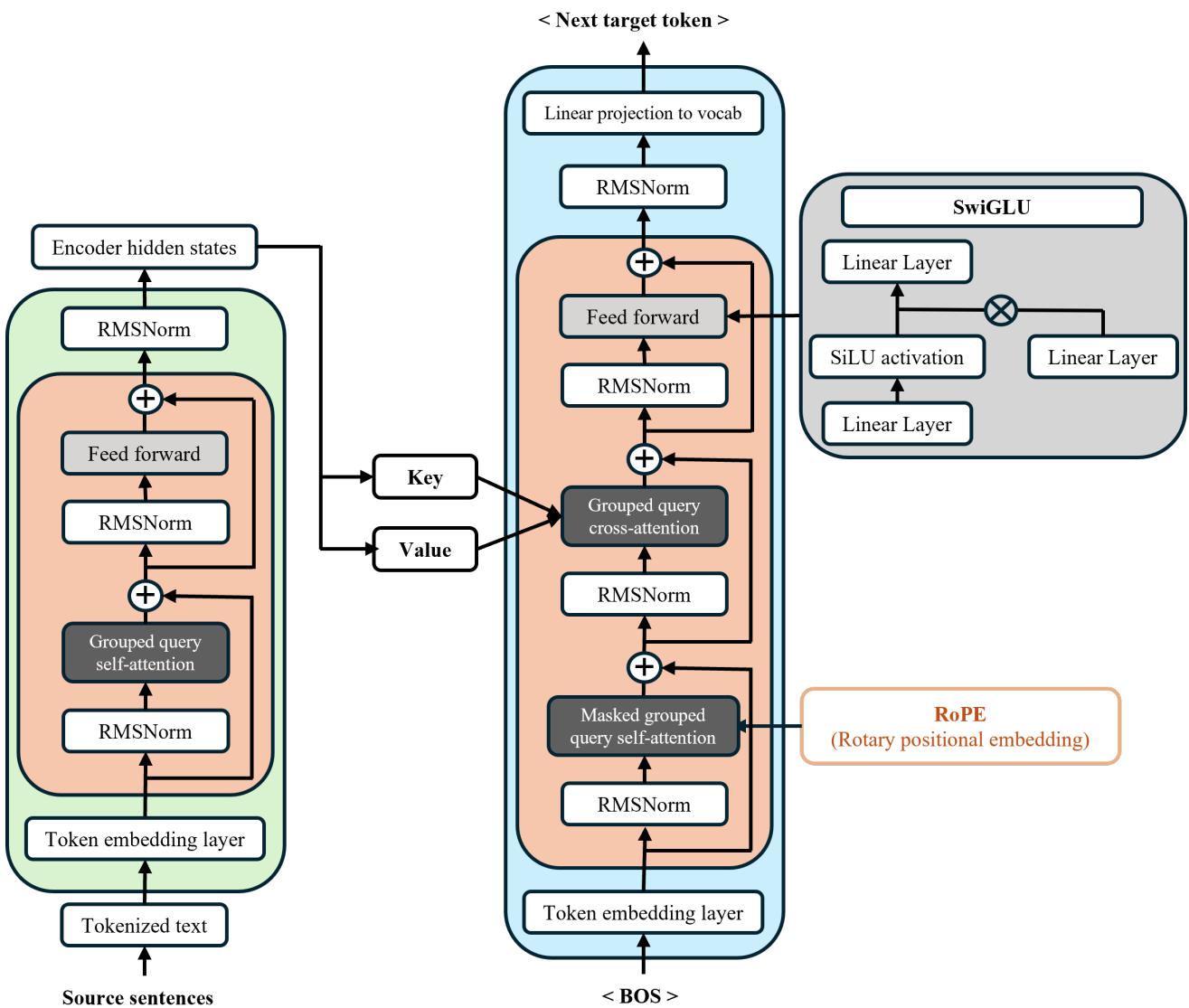


Hình 1: Minh họa đề thi dịch thuật Hoa - Việt.

Đề thi nhấn mạnh tính thực tế của các mô hình dịch thuật Hoa - Việt trước bối cảnh toàn cầu hóa, khi việc giao lưu về kinh tế, văn hóa cũng như học thuật giữa Việt Nam và Trung Quốc ngày càng được đẩy mạnh. Lúc này, các mô hình có khả năng tự động dịch giữa hai ngôn ngữ tiếng Việt và tiếng Trung Quốc càng trở nên hữu ích, có vai trò là công cụ giúp phá vỡ các rào cản về ngôn ngữ. Nhờ vào học sâu (Deep learning), các mô hình này càng trở nên mạnh mẽ, có

thể kể đến các hệ thống dịch thuật lớn như Google Translate, hoặc các mô hình ngôn ngữ lớn (LLMs) đang ngày càng được sử dụng rộng rãi trong tác vụ dịch thuật hằng ngày.

Bài toán MT là một trong những bài kinh điển của lĩnh vực NLP. Ở bài toán này, chúng ta sẽ xây dựng một mô hình có khả năng tự động dịch từ ngôn ngữ này sang ngôn ngữ khác. Cụ thể, với đề bài, các thí sinh sẽ phải huấn luyện một mô hình có khả năng dịch từ Tiếng Trung (Giản thể) sang tiếng Việt.



Hình 2: Mô hình Transformer Encoder-Decoder được xây dựng cho bài toán dịch thuật.

# Mục lục

I.	Giới thiệu	1
II.	Phân tích đề thi	4
II.1.	Thể lệ và yêu cầu của cuộc thi	4
II.2.	Hướng triển khai	5
II.3.	Dữ liệu	5
II.4.	Kiến thức cần có	6
III.	Giới thiệu baseline	8
IV.	Bổ sung kiến thức về Transformer	18
IV.1.	RMSNorm	18
IV.2.	RoPE	19
IV.3.	Grouped-query attention	20
IV.4.	SwiGLU	21
IV.5.	Kiến trúc mô hình Transformer Encoder-Decoder được sử dụng	22
V.	Mô hình Transformer Encoder-Decoder cho bài toán dịch thuật Hoa-Việt	24
VI.	Hướng phát triển và cải tiến mô hình	41
VI.1.	Thử nghiệm đa cấu hình: kiến trúc, từ vựng và lịch huấn luyện	41
VI.2.	Khai thác Back-Translation để mở rộng dữ liệu	41
VI.3.	Regularization nâng cao: R-Drop và tối ưu hoá SAM	42
VI.4.	Mở rộng khung contrastive với cặp dương từ nhiều phiên bản của cùng một câu	43
VI.5.	Khảo sát các chiến lược giải mã: beam search, top- $k$ , top- $p$	43
VI.6.	Vòng lặp suy luận, phân tích lỗi và cải thiện mô hình	44
	Phụ lục	46

## II. Phân tích đề thi

### II.1. Thể lệ và yêu cầu của cuộc thi

Đến với cuộc thi, ban tổ chức sẽ cung cấp các tài nguyên bài thi gồm: Dữ liệu train, dữ liệu test và code baseline.

Dựa vào dữ liệu train, nhiệm vụ của thí sinh là **huấn luyện mô hình dịch Hoa - Việt**, với khả năng nhận đầu vào là một câu tiếng Trung và dịch ra câu tiếng Việt tương ứng.

Về dữ liệu test, ban tổ chức cung cấp 2 loại: **Public test** được nhận vào đầu buổi thi để đánh giá kết quả và **Private test** được mở vào cuối buổi thi.

Ngoài ra, các thí sinh cũng sẽ được cung cấp một file code *baseline*, bao gồm đầy đủ các bước để xây dựng mô hình. Các thí sinh có thể dựa vào *baseline*, phân tích các ưu khuyết điểm để làm cơ sở phát triển cho mô hình của mình.

Kết quả sẽ được đánh giá bằng **SacreBLEU**. Đây là một độ đo được sử dụng phổ biến đối với nhiều bài toán trong NLP, trong đó có MT. SacreBLEU được tính bằng:

$$\text{SacreBLEU} = BP \cdot \exp \left( \sum_{n=1}^N w_n \log p_n \right)$$

Trong đó:

- $p_n$ : Độ chính xác n-gram bậc  $n$ , tính bằng

$$p_n = \frac{\sum_{\text{ngram} \in \text{hyp}_n} \min(\text{count}_{\text{hyp}}(\text{ngram}), \text{max}_{\text{ref}} \text{count}_{\text{ref}}(\text{ngram}))}{\sum_{\text{ngram} \in \text{hyp}_n} \text{count}_{\text{hyp}}(\text{ngram})}$$

- $w_n$ : Trọng số, thường là  $\frac{1}{4}$  cho  $n = 1, 2, 3, 4$ .
- $BP$ : Hệ số phạt độ ngắn, tính bằng

$$BP = \begin{cases} 1, & \text{nếu } c > r \\ \exp(1 - \frac{r}{c}), & \text{nếu } c \leq r \end{cases}$$

- $\text{count}_{\text{hyp}}(\text{ngram})$ : Số lần xuất hiện của n-gram trong bản dịch máy.
- $\text{count}_{\text{ref}}(\text{ngram})$ : Số lần xuất hiện của n-gram trong bản dịch tham chiếu.
- $c$ : Tổng độ dài của bản dịch hệ thống.
- $r$ : Tổng độ dài của bản dịch tham chiếu.

Ngoài ra, điểm SacreBLEU còn được chuẩn hóa (Normalize) theo công thức của cuộc thi, nhưng nhìn chung chúng ta vẫn chỉ quan tâm đến việc đạt SacreBLEU càng cao càng tốt.

Bên cạnh đề thi cơ bản là xây dựng mô hình dịch từ Tiếng Trung sang tiếng Việt, cuộc thi vẫn có một số yêu cầu cơ bản sau:

- Các thí sinh **không được sử dụng các mô hình huấn luyện sẵn** (pre-trained) trong bài làm của mình. Điều này có nghĩa là thí sinh không được sử dụng các mô hình huấn luyện sẵn (pre-trained) như **BERT**, **BART**, ... cho bất kỳ bước xử lý nào trong bài làm của mình. .
- Các thí sinh **không được sử dụng dữ liệu từ bên ngoài** bổ sung cho tập train đã được cung cấp. Điều này có nghĩa là chiến thuật tăng cường dữ liệu bằng cách bổ sung data từ bên ngoài cuộc thi sẽ không được chấp nhận. Tuy nhiên, các thí sinh vẫn **được tạo dữ liệu nhân tạo** (synthetic) dựa trên dữ liệu của ban tổ chức.

## II.2. Hướng triển khai

Đối với bài toán MT, chúng ta có một số cách tiếp cận như:

- **Rule-based**: Dịch thuật theo tập luật được định sẵn từ trước
- **Statistical Machine Translation (SMT)**: Dựa trên xác suất xuất hiện của từ/cụm từ.
- **Neural Machine Translation (NMT)**: Huấn luyện mô hình học sâu

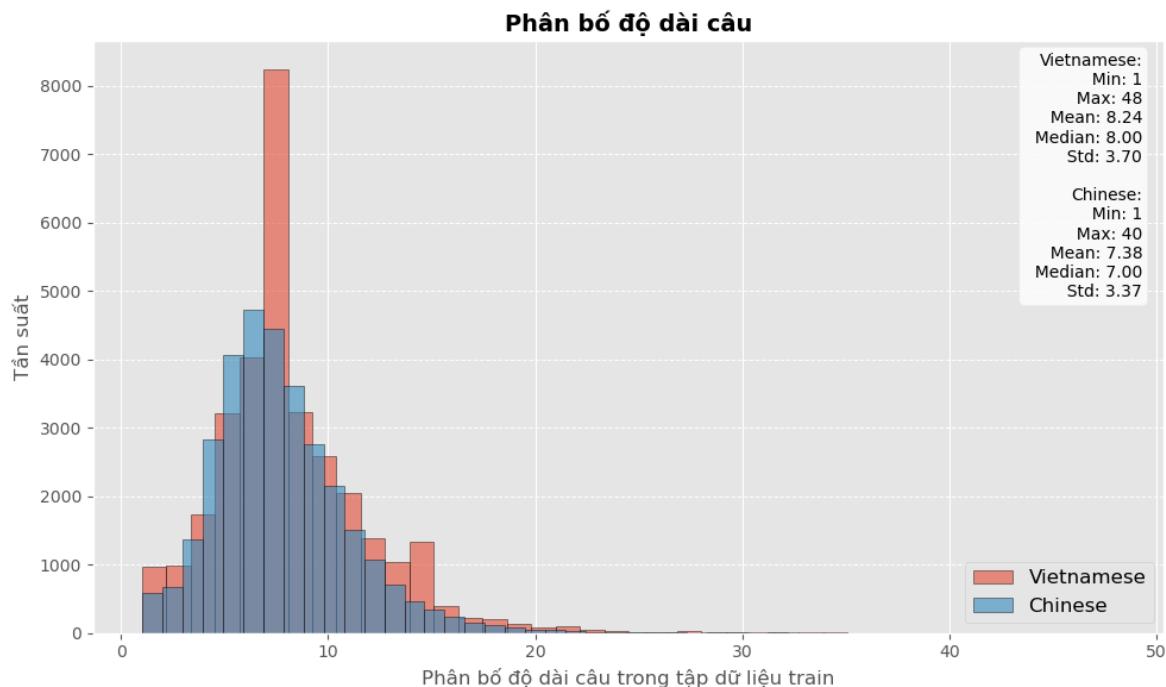
Có thể thấy rằng, cách làm phổ biến cũng như được nhiều thí sinh lựa chọn vẫn là **NMT**. Sự thành công của các mô hình học sâu chính là minh chứng cho sức mạnh của **NMT**, với kết quả tốt hơn nhiều so với hai phương pháp còn lại. Bên cạnh đó, code baseline cũng được triển khai theo hướng NMT. Do đó, để được kết quả cao, chúng ta vẫn nên lựa chọn NMT.

## II.3. Dữ liệu

Bộ dữ liệu được cung cấp được chia làm ba phần:

- **Tập train**: Bao gồm 2 file là **train.zh** và **train.vi**. Trong đó, **train.zh** bao gồm các văn bản tiếng Trung, còn **train.vi** bao gồm các văn bản tiếng Việt.
- **Tập public test**: **public\_test.zh**
- **Tập private test**: **private\_test.zh**

Trong đó, tập huấn luyện bao gồm 32,061 samples. Đối với bài toán MT, đây là số lượng không quá nhỏ nhưng cũng không quá lớn. Các sample khác nhau có độ dài không đều: có những câu rất ngắn (1-2 từ) và những câu rất dài (hơn 40 từ).



Hình 3: Tổng quan về độ dài câu trong tập huấn luyện.

## II.4. Kiến thức cần có

Đối với cuộc thi này, chúng ta cần nắm bắt các kiến thức nền tảng của NLP cũng như bài toán MT, cả về lý thuyết và thực hành.

Về mặt lý thuyết, việc trang bị những kiến thức nền tảng cho bài toán MT sẽ là lợi thế lớn. Những kiến thức này bao gồm:

- **xử lý dữ liệu:** Nắm rõ các kỹ thuật cơ bản của NLP như data normalization hay tokenization (Word-based tokenization, Byte-pair-encoding (BPE),...). Ngoài ra, các kỹ thuật Data augmentation trong bài toán MT cũng quan trọng, giúp mô hình cải thiện khả năng tổng quát và tránh overfit.
- **Xây dựng các mô hình học sâu:** Hiểu một số mô hình cơ bản (RNN, GRU, Transformer). Các mô hình với kiến trúc mạnh mẽ với khả năng học tốt có thể cải thiện kết quả rất nhiều.
- **Kỹ thuật huấn luyện mô hình:** Lựa chọn hàm loss phù hợp (Cross-Entropy, Label Smoothing), sử dụng các optimizer hiện đại (Adam, AdamW, Adagrad, ...), áp dụng learning rate scheduler (warm-up, cosine decay), regularization (dropout, weight decay),...
- **Kỹ thuật Decode:** Biết một số kỹ thuật decode như Greedy decode, Beam search,...

Việc có những kiến thức sâu rộng trong bài toán MT là một lợi thế rất lớn, giúp cạnh tranh với các thí sinh khác. Điều đó đòi hỏi mỗi thí sinh cần liên tục cập nhật các kiến thức mới, tìm hiểu những kỹ thuật hiện đại và mạnh mẽ trong những năm gần đây.

Về mặt thực hành, chúng ta cần có một số nền tảng lập trình nhất định (biết cách sử dụng Python,...) và cần nắm rõ cách sử dụng các thư viện Deep learning như Pytorch, Tensorflow hay Keras.

### III. Giới thiệu baseline

Ở phần này, chúng ta sẽ tìm hiểu và phân tích baseline do ban tổ chức cung cấp. Việc tìm hiểu baseline là vô cùng quan trọng, giúp tạo tiền đề cho những quyết định tiếp theo.

Trước hết, dữ liệu được tokenize bằng phương pháp **BPE**. Tiếp theo, một mô hình Seq2Seq với Encoder và Decoder đều sử dụng các khối GRU được xây dựng và tiến hành huấn luyện. Sau khi đã huấn luyện xong mô hình, baseline decode bằng phương pháp **Greedy decoding** lên tập dữ liệu test.

#### Bước 1: Configuration and Hyperparameters

```

1 # Data paths
2 TRAIN_SRC = "dataset/train/train.zh"
3 TRAIN_TGT = "dataset/train/train.vi"
4 TEST_SRC = "dataset/public_test/public_test.zh"
5 SAVE_DIR = "./checkpoints"
6 SPM_ZH_PREFIX = os.path.join(SAVE_DIR, "spm_zh")
7 SPM_VI_PREFIX = os.path.join(SAVE_DIR, "spm_vi")
8
9 # Model hyperparameters
10 VOCAB_SIZE = 3000
11 EMB_SIZE = 64
12 HID_SIZE = 128
13 BATCH_SIZE = 64
14 EPOCHS = 10
15 LR = 0.01
16 MAX_LEN = 80
17 SEED = 42
18
19 # Device configuration
20 DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
21 print(f"Using device: {DEVICE}")
22
23 # Set random seeds for reproducibility
24 os.makedirs(SAVE_DIR, exist_ok=True)
25 random.seed(SEED)
26 torch.manual_seed(SEED)
27 if torch.cuda.is_available():
28     torch.cuda.manual_seed(SEED)

```

**Giải thích:** Ở phần này, baseline gán giá trị cho các hyperparameters chính trong bài. Trong đây, các thông số đáng chú ý là: `VOCAB_SIZE = 3000`, `EMB_SIZE = 64` và `HID_SIZE = 128`.

**Nhận xét:** Nhìn chung, các thông số này tương đối nhỏ. Một chiến thuật có thể thử nhằm cải thiện baseline chính là tăng giá trị của các hyperparameters kể trên, chẳng hạn như tăng giá trị của `VOCAB_SIZE` để tránh vấn đề Out-of-Vocabulary (OOV), hoặc tăng `EMB_SIZE` và `HID_SIZE` để nâng kích thước mô hình.

## Bước 2: Data processing

```

1 def train_spm(input_file, model_prefix, vocab_size=VOCAB_SIZE):
2     """Train a SentencePiece BPE model."""
3     args = (
4         f"--input={input_file} --model_prefix={model_prefix} --vocab_size={vocab_size}"
5             "
6             "--model_type=bpe --character_coverage=1.0 "
7             "--pad_id=0 --unk_id=1 --bos_id=2 --eos_id=3"
8     )
9     spm.SentencePieceTrainer.Train(args)
10    print(f"Trained SentencePiece model: {model_prefix}.model")
11
12 def load_sp(model_path):
13     """Load a trained SentencePiece model."""
14     sp = spm.SentencePieceProcessor()
15     sp.Load(model_path)
16     return sp
17
18 # Train Chinese tokenizer
19 tmp_zh = os.path.join(SAVE_DIR, "tmp_zh.txt")
20 if not os.path.exists(SPM_ZH_PREFIX + ".model"):
21     with open(tmp_zh, "w", encoding="utf-8") as f:
22         for s in train_src:
23             f.write(s + "\n")
24     train_spm(tmp_zh, SPM_ZH_PREFIX)
25
26 # Train Vietnamese tokenizer
27 tmp_vi = os.path.join(SAVE_DIR, "tmp_vi.txt")
28 if not os.path.exists(SPM_VI_PREFIX + ".model"):
29     with open(tmp_vi, "w", encoding="utf-8") as f:
30         for s in train_tgt:
31             f.write(s + "\n")
32     train_spm(tmp_vi, SPM_VI_PREFIX)
33
34 # Load tokenizers
35 sp_zh = load_sp(SPM_ZH_PREFIX + ".model")
36 sp_vi = load_sp(SPM_VI_PREFIX + ".model")

```

**Giải thích:** Ở phần này, baseline sử dụng thư viện **Sentencepiece** cho quá trình tokenize. Dựa vào biến *args* trong hàm *train\_spm*, ta có dễ dàng nhận thấy những điều sau:

- **model\_type=bpe**: Phương pháp được sử dụng để tokenize là Byte-pair-encoding (BPE).
- **character\_coverage=1.0**: Ép tokenizer sử dụng toàn bộ ký tự trong tập train cho bộ từ điển.
- **pad\_id=0, unk\_id=1, bos\_id=2, eos\_id=3**: Gán ID cho các token đặc biệt trong bài, bao gồm <pad>, <unk>, <bos>, <eos>, lần lượt mang ID 0, 1, 2 và 3.

**Nhận xét:** Việc lựa chọn BPE cho bài toán là hợp lý, do BPE phù hợp với cả tiếng Trung và tiếng Việt.

```

1 class TranslationDataset(Dataset):
2     """Dataset for Chinese-Vietnamese translation pairs."""
3
4     def __init__(self, src, tgt, sp_src, sp_tgt, max_len=MAX_LEN):
5         self.src = src
6         self.tgt = tgt
7         self.sp_src = sp_src
8         self.sp_tgt = sp_tgt
9         self.max_len = max_len
10
11    def __len__(self):
12        return len(self.src)
13
14    def __getitem__(self, idx):
15        # Add BOS (2) and EOS (3) tokens
16        src_ids = [2] + self.sp_src.EncodeAsIds(self.src[idx])[:self.max_len-2] + [3]
17        tgt_ids = [2] + self.sp_tgt.EncodeAsIds(self.tgt[idx])[:self.max_len-2] + [3]
18        return torch.tensor(src_ids), torch.tensor(tgt_ids)
19
20 def collate_fn(batch):
21     """Collate function to pad sequences to the same length."""
22     srcs, tgts = zip(*batch)
23     max_src = max(len(s) for s in srcs)
24     max_tgt = max(len(t) for t in tgts)
25
26     # Pad with 0 (PAD token)
27     src_pad = torch.zeros(len(batch), max_src, dtype=torch.long)
28     tgt_pad = torch.zeros(len(batch), max_tgt, dtype=torch.long)
29
30     for i, (s, t) in enumerate(zip(srcs, tgts)):
31         src_pad[i, :len(s)] = s
32         tgt_pad[i, :len(t)] = t
33
34     return src_pad, tgt_pad
35
36 # Create training dataset and dataloader
37 # Split data: 90% train, 10% validation
38 total_samples = len(train_src)
39 train_size = int(0.9 * total_samples)
40
41 train_src_split = train_src[:train_size]
42 train_tgt_split = train_tgt[:train_size]
43 valid_src = train_src[train_size:]
44 valid_tgt = train_tgt[train_size:]
45
46 dataset = TranslationDataset(train_src_split, train_tgt_split, sp_zh, sp_vi)
47 dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True, collate_fn=
48                         collate_fn, num_workers=4, pin_memory=True)
49 valid_dataset = TranslationDataset(valid_src, valid_tgt, sp_zh, sp_vi)

```

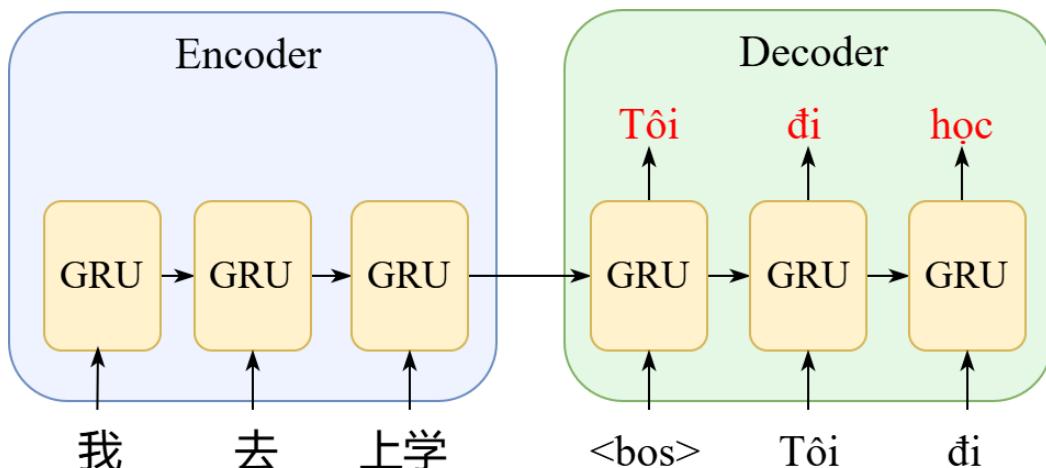
```
50 valid_loader = DataLoader(valid_dataset, batch_size=32, shuffle=False, collate_fn=
                           collate_fn, num_workers=2, pin_memory=True)
```

**Giải thích:** Đoạn code trên sẽ xây dựng bộ dữ liệu

- **Phân chia tập train và val:** Tập train và val được xây dựng từ tập train ban đầu của ban tổ chức, chia ra với tỉ lệ **9:1**.
- **TranslationDataset:** Trước khi đưa vào mô hình, mỗi sample sẽ được bổ sung token `<bos>` và `<eos>` lần lượt báo hiệu cho mô hình rằng đây là khởi đầu và kết thúc của câu. Đây là cách xử lý tiêu chuẩn cho các bài toán NLP.
- **collate\_fn:** Do mô hình cần các sample có kích thước giống nhau là `MAX_LEN`, ta cần điều chỉnh lại độ dài mỗi sample trước khi đưa vào mô hình. Với các câu quá dài (lớn hơn `MAX_LEN`), ta bỏ đi phần phía sau. Với các câu ngắn (nhỏ hơn `MAX_LEN`), ta điều chỉnh độ dài bằng cách thêm các token `<pad>` vào.

**Nhận xét:** Phần xử lý dữ liệu của baseline tương đối đơn giản. Ta cũng nhận thấy rằng baseline không sử dụng kỹ thuật tiền xử lý hay augmentation nào khác. Đây cũng là một phần hạn chế của baseline, khi tập train có kích thước không quá lớn, có thể khiến cho mô hình không có khả năng tổng quát tốt.

### Bước 3: Model Architecture



Hình 4: Mô hình Seq2Seq của baseline

```
1 class EncoderRNN(nn.Module):
2     """GRU-based encoder."""
3
4     def __init__(self, vocab_size, emb_size, hidden_size):
5         super().__init__()
6         self.embedding = nn.Embedding(vocab_size, emb_size, padding_idx=0)
```

```
7     self.dropout = nn.Dropout(0.3)
8     self.rnn = nn.GRU(emb_size, hidden_size, batch_first=True)
9
10    def forward(self, src):
11        emb = self.dropout(self.embedding(src))
12        _, hidden = self.rnn(emb)
13        return hidden
14
15
16 class DecoderRNN(nn.Module):
17     """GRU-based decoder with teacher forcing."""
18
19     def __init__(self, vocab_size, emb_size, hidden_size):
20         super().__init__()
21         self.embedding = nn.Embedding(vocab_size, emb_size, padding_idx=0)
22         self.dropout = nn.Dropout(0.3)
23         self.rnn = nn.GRU(emb_size, hidden_size, batch_first=True)
24         self.fc = nn.Linear(hidden_size, vocab_size)
25
26     def forward(self, input_step, hidden):
27         emb = self.dropout(self.embedding(input_step))
28         output, hidden = self.rnn(emb, hidden)
29         pred = self.fc(output.squeeze(1))
30         return pred, hidden
31
32
33 class Seq2Seq(nn.Module):
34     """Sequence-to-sequence model."""
35
36     def __init__(self, encoder, decoder):
37         super().__init__()
38         self.encoder = encoder
39         self.decoder = decoder
40
41     def forward(self, src, tgt, teacher_forcing_ratio=0.3):
42         batch_size = src.size(0)
43         tgt_len = tgt.size(1)
44         vocab_size = self.decoder.fc.out_features
45
46         # Store outputs
47         outputs = torch.zeros(batch_size, tgt_len, vocab_size).to(src.device)
48
49         # Encode source sentence
50         hidden = self.encoder(src)
51
52         # Start with BOS token
53         input_step = tgt[:, 0].unsqueeze(1)
54
55         # Decode step by step
56         for t in range(1, tgt_len):
57             pred, hidden = self.decoder(input_step, hidden)
58             outputs[:, t] = pred
59
60             # Teacher forcing
```

```

61     teacher_force = random.random() < teacher_forcing_ratio
62     input_step = tgt[:, t].unsqueeze(1) if teacher_force else pred.argmax(1).unsqueeze(1)
63
64     return outputs
65
66 # Initialize model
67 encoder = EncoderRNN(sp_zh.GetPieceSize(), EMB_SIZE, HID_SIZE)
68 decoder = DecoderRNN(sp_vi.GetPieceSize(), EMB_SIZE, HID_SIZE)
69 model = Seq2Seq(encoder, decoder).to(DEVICE)

```

**Giải thích:** Baseline sử dụng mô hình Seq2Seq với các khối Encoder - Decoder, với tổng cộng 919,992 tham số, được xây dựng từ GRU:

- **Encoder:** Xử lý tuần tự các ký tự đầu vào. Hidden state cuối cùng của Encoder đóng vai trò như một *context vector*, chứa thông tin ngữ nghĩa tổng hợp của toàn bộ câu đầu vào
- **Decoder:** sử dụng hidden state này để sinh ra dự đoán. Tại mỗi bước, Decoder GRU nhận vào các thông tin trước đó, bao gồm token được sinh ra từ mô hình hoặc từ dữ liệu huấn luyện, cùng với hidden state, để sinh ra phân phối xác suất dự đoán từ tiếp theo.
- **Dropout:** Khối dropout được sử dụng trong cả **Encoder** và **Decoder**. Đây là kỹ thuật regularization có thể giúp mô hình tổng quát tốt hơn.
- **Teacher forcing:** Kỹ thuật này được thể hiện qua việc mô hình chọn token được sinh ra trong decoder hoặc token trong tập train.

**Nhận xét:** Việc lựa chọn GRU làm cốt lõi cho mô hình có nhiều hạn chế. Mặc dù GRU được đánh giá là tốt hơn RNN trong các bài toán NLP, GRU vẫn kém cạnh so với các mô hình Transformer trong việc nắm bắt ngữ nghĩa của câu, đặc biệt trong các câu dài. Ngoài ra, kích thước của mô hình tương đối nhỏ (chưa đến 1M tham số). Do đó, một cải tiến có thể thực hiện là nâng cấp mô hình.

## Bước 4: Training

```

1 def train_epoch(model, dataloader, criterion, optimizer):
2     """Train for one epoch."""
3     model.train()
4     total_loss = 0
5
6     for src, tgt in dataloader:
7         src, tgt = src.to(DEVICE), tgt.to(DEVICE)
8
9         optimizer.zero_grad()
10        output = model(src, tgt)
11

```

```

12     # Calculate loss (ignore first BOS token)
13     loss = criterion(
14         output[:, 1:].reshape(-1, output.size(-1)),
15         tgt[:, 1:].reshape(-1)
16     )
17
18     loss.backward()
19     torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
20     optimizer.step()
21
22     total_loss += loss.item()
23
24     return total_loss / len(dataloader)
25
26
27 @torch.no_grad()
28 def evaluate_bleu(model, dataloader, sp_tgt):
29     """Evaluate model using SacreBLEU metric."""
30     model.eval()
31     hyps, refs = [], []
32
33     pbar = tqdm(dataloader, desc="Evaluating", leave=False)
34     for src, tgt in pbar:
35         src = src.to(DEVICE)
36
37         # Encode
38         hidden = model.encoder(src)
39
40         # Decode (greedy)
41         input_step = torch.full((src.size(0), 1), 2, dtype=torch.long, device=DEVICE)
42         decoded = [[] for _ in range(src.size(0))]
43
44         for _ in range(MAX_LEN):
45             pred, hidden = model.decoder(input_step, hidden)
46             next_token = pred.argmax(1).unsqueeze(1)
47
48             for i in range(src.size(0)):
49                 decoded[i].append(next_token[i].item())
50
51             input_step = next_token
52
53         # Convert to text
54         for i in range(src.size(0)):
55             ids = decoded[i]
56             if 3 in ids: # Stop at EOS
57                 ids = ids[:ids.index(3)]
58             hyps.append(sp_tgt.DecodeIds(ids))
59
60             ref_ids = tgt[i].tolist()[1:-1] # Remove BOS and EOS
61             refs.append(sp_tgt.DecodeIds([x for x in ref_ids if x not in [0, 1]]))
62
63         bleu = sacrebleu.corpus_bleu(hyps, [refs])
64     return bleu.score
65

```

```

66 # Loss function and optimizer
67 criterion = nn.CrossEntropyLoss(ignore_index=0) # Ignore padding
68 optimizer = optim.Adam(model.parameters(), lr=LR)
69
70 print("Starting training...\n")
71
72 # Training loop with best model saving
73 best_bleu = 0.0
74 best_model_path = os.path.join(SAVE_DIR, "best_model.pt")
75
76 for epoch in range(1, EPOCHS + 1):
77     loss = train_epoch(model, dataloader, criterion, optimizer)
78     bleu = evaluate_bleu(model, valid_loader, sp_vt)
79
80     # Save best model
81     if bleu > best_bleu:
82         best_bleu = bleu
83         torch.save({
84             "epoch": epoch,
85             "model_state_dict": model.state_dict(),
86             "optimizer_state_dict": optimizer.state_dict(),
87             "bleu": bleu,
88             "loss": loss
89         }, best_model_path)
90         print(f"Epoch {epoch:02d} | Loss={loss:.3f} | SacreBLEU={bleu:.2f} Best model
91                         saved!")
92     else:
93         print(f"Epoch {epoch:02d} | Loss={loss:.3f} | SacreBLEU={bleu:.2f}")

```

**Giải thích:** Các chiến thuật training trong baseline gồm:

- **Hàm loss:** Cross Entropy.
- **Optimizer:** Adam.

**Nhận xét:** Nhìn chung, lựa chọn hàm loss và optimizer của baseline có thể được coi là tiêu chuẩn trong bài toán MT. Tuy nhiên, ngoài ra, baseline cũng không sử dụng các kỹ thuật training nào khác như learning rate scheduler, Label Smoothing,... Đây là một thứ mà chúng ta có thể cải thiện.

## Bước 5: Decode

```

1 @torch.no_grad()
2 def translate_test(model, test_src, sp_src, sp_tgt, out_path):
3     """Translate test set and save to CSV."""
4     model.eval()
5     outputs = []
6
7     for s in tqdm(test_src, desc="Translating"):
8         # Tokenize source sentence

```

```

9     src_ids = [2] + sp_src.EncodeAsIds(s)[:MAX_LEN-2] + [3]
10    src_tensor = torch.tensor(src_ids, dtype=torch.long, device=DEVICE).unsqueeze(0
11                                )
12
13    # Encode
14    hidden = model.encoder(src_tensor)
15
16    # Decode
17    input_step = torch.full((1, 1), 2, dtype=torch.long, device=DEVICE)
18    decoded = []
19
20    for _ in range(MAX_LEN):
21        pred, hidden = model.decoder(input_step, hidden)
22        next_token = pred.argmax(1)
23        token = next_token.item()
24
25        if token == 3: # EOS
26            break
27
28        decoded.append(token)
29        input_step = next_token.unsqueeze(1)
30
31    # Decode to Vietnamese text
32    vi_sent = sp_tgt.DecodeIds(decoded)
33    outputs.append(vi_sent)
34
35    # Save to CSV
36    df = pd.DataFrame({"tieng_trung": test_src, "tieng_viet": outputs})
37    df.to_csv(out_path, index=False, encoding="utf-8-sig")
38
39    return out_path

```

**Giải thích:** Sau khi hoàn thành việc luyện mô hình, baseline tiếp tục với phần inference trên các dữ liệu test, trong đó chiến thuật decode được sử dụng là Greedy Decode: Mô hình sẽ lần lượt chọn từ có xác suất cao nhất để dự đoán cho từ tiếp theo.

**Nhận xét:** Kỹ thuật decode này đơn giản và khá yếu so với những kỹ thuật khác như Beam search.

## Nhận xét chung về baseline

Baseline đã thực hiện đầy đủ các quá trình của một bài toán MT. Tuy nhiên, nó vẫn còn nhiều hạn chế:

- **Xử lý dữ liệu:** Ngoài việc sử dụng BPE tokenization, baseline không sử dụng các kỹ thuật nào để xử lý dữ liệu. Kích thước của tập dữ liệu không quá lớn (xấp xỉ 32000 samples), mô hình có thể không tổng quát tốt đối với tập test. Do đó, việc bổ sung các kỹ thuật tiền xử lý dữ liệu hoặc augmentation là vô cùng cần thiết.
- **Mô hình:** Mô hình baseline Seq2Seq sử dụng GRU đơn giản và dễ triển khai. Thế nhưng, nó vẫn tồn tại nhiều hạn chế. GRU tuy tốt hơn RNN khi xử lý các văn bản có kích thước

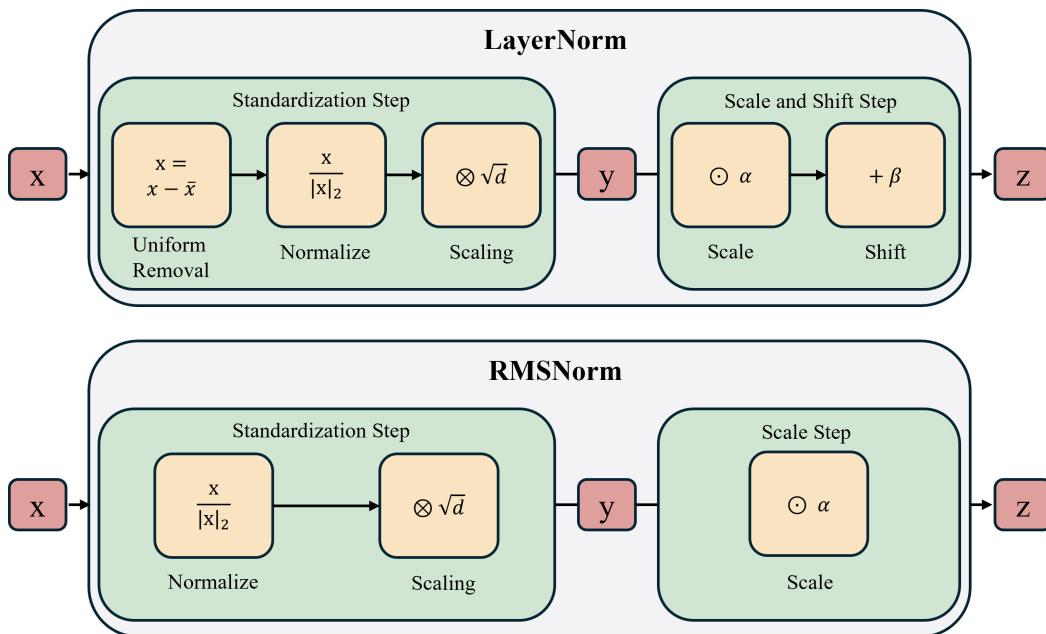
dài nhưng vẫn yếu hơn so với các mô hình như Transformer, vốn được sử dụng khá nhiều hiện nay trong các bài toán MT. Ngoài ra, kích thước mô hình (**VOCAB\_SIZE**, **EMB\_SIZE**, **HID\_SIZE**) vẫn còn tương đối nhỏ. Do đó, để cải thiện baseline, chúng ta có thể thử sử dụng các mô hình có kiến trúc mạnh mẽ và khả năng học tốt hơn.

- **Chiến thuật training:** Baseline đang huấn luyện mô hình với Cross Entropy loss và Adam optimizer. Mặc dù đây là lựa chọn tốt và phổ biến, hiện nay đã có nhiều chiến thuật training khác có thể được áp dụng như AdamW, learning rate scheduler, label smoothing, regularization, để cải thiện mô hình.
- **Chiến thuật decode:** Baseline đang sử dụng Greedy decode. So với các phương pháp Sampling hiện nay như Beam search, Greedy Decode vẫn còn nhiều hạn chế. Đôi khi Greedy Decode có thể dẫn đến những câu vô nghĩa ảnh hưởng đến kết quả

Điểm SacreBLEU của baseline trên tập public\_test là 4.52, tương đối thấp. Từ những nhận xét trên, chúng ta có thể cải thiện baseline ở nhiều khía cạnh (xử lý dữ liệu, thay đổi kiến trúc mô hình, thay đổi kỹ thuật huấn luyện, decode,...).

# IV. Bổ sung kiến thức về Transformer

## IV.1. RMSNorm



Hình 5: Minh họa LayerNorm và RMSNorm.

Trong kiến trúc Transformer gốc, *LayerNorm* chuẩn hoá một vector kích hoạt  $\mathbf{x} \in \mathbb{R}^d$  bằng cách trừ đi trung bình theo chiều kênh rồi chia cho độ lệch chuẩn. Cụ thể, đầu ra được tính theo

$$y = \alpha \odot \frac{\mathbf{x} - \mu(\mathbf{x})}{\sqrt{\sigma^2(\mathbf{x}) + \varepsilon}} + \beta.$$

Cách làm này bảo đảm mỗi chiều có phân phối gần chuẩn, nhưng đồng thời làm thay đổi cả hướng lẫn độ lớn của vector, và đòi hỏi tính cả mean lẫn variance cho từng bước lan truyền.

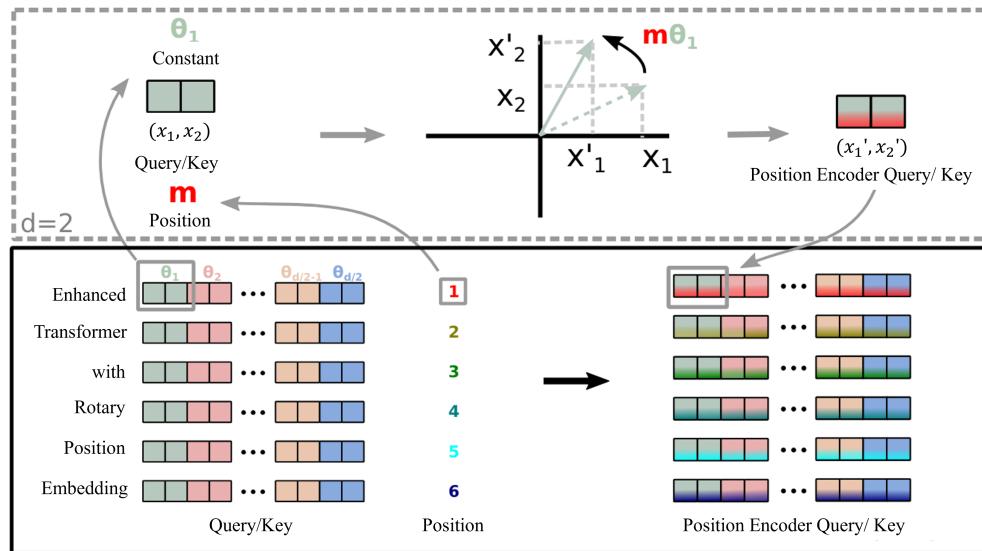
*RMSNorm* loại bỏ bước trừ mean và chỉ chuẩn hoá theo *chuẩn RMS* của vector:

$$y = \alpha \odot \frac{\mathbf{x}}{\text{RMS}(\mathbf{x})}, \quad \text{RMS}(\mathbf{x}) = \sqrt{\frac{1}{d} \sum_{i=1}^d x_i^2 + \varepsilon}.$$

Nhờ vậy, RMSNorm giữ nguyên hướng của  $\mathbf{x}$ , chỉ điều chỉnh độ lớn, đồng thời giảm bớt phép tính (không cần variance và bias  $\beta$ ), dẫn tới chi phí tính toán thấp hơn và ít nhiễu số hơn trên các mô hình rất sâu.

Trong các LLM hiện đại (như LLaMA, Qwen, v.v.), RMSNorm thường được ưa dùng vì giúp ổn định gradient khi kết hợp với sơ đồ *pre-norm* và residual connection, cải thiện độ ổn định huấn luyện ở kích thước lớp ẩn lớn, và thực nghiệm cho thấy có thể đạt chất lượng tương đương hoặc tốt hơn LayerNorm với chi phí tính toán đơn giản hơn.

## IV.2. RoPE



Hình 6: Minh họa cách hoạt động của RoPE.

Trong Transformer gốc, thông tin vị trí thường được đưa vào bằng *positional embedding tuyệt đối*, ví dụ dạng hình sin–cos: một vector  $PE(m) \in \mathbb{R}^d$  phụ thuộc vào chỉ số vị trí  $m$  được cộng trực tiếp vào biểu diễn token:

$$\mathbf{x}'_m = \mathbf{x}_m + PE(m).$$

Cách làm này giúp mô hình phân biệt thứ tự các token, nhưng sự phụ thuộc vào vị trí tuyệt đối khiến việc ngoại suy sang độ dài chuỗi lớn hơn hoặc mô hình hoá quan hệ tương đối (khoảng cách  $m - n$ ) trở nên kém tự nhiên.

*Rotary Position Embedding (RoPE)* thay vì cộng một vector vị trí, sẽ mã hoá vị trí bằng cách xoay các cặp chiều trong không gian 2D phụ thuộc vào chỉ số  $m$ . Cho  $\mathbf{q}_m, \mathbf{k}_m \in \mathbb{R}^d$  là query và key tại vị trí  $m$ , với mỗi cặp chỉ số  $(2i, 2i+1)$  ta định nghĩa góc  $\theta_i$  và áp dụng phép quay:

$$\begin{aligned}\tilde{q}_m^{(2i)} &= q_m^{(2i)} \cos(m\theta_i) - q_m^{(2i+1)} \sin(m\theta_i), \\ \tilde{q}_m^{(2i+1)} &= q_m^{(2i)} \sin(m\theta_i) + q_m^{(2i+1)} \cos(m\theta_i),\end{aligned}$$

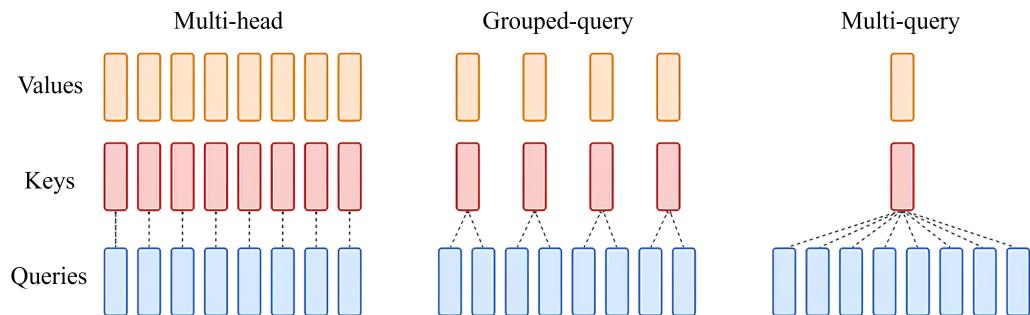
và tương tự cho  $\tilde{\mathbf{k}}_n$ . Khi đó, tích vô hướng trong attention trở thành

$$\tilde{\mathbf{q}}_m^\top \tilde{\mathbf{k}}_n,$$

một hàm phụ thuộc chủ yếu vào *chênh lệch vị trí*  $m - n$ , nhúng thông tin vị trí tương đối trực tiếp vào cơ chế attention mà không cần sửa lại công thức softmax.

Nhờ đặc tính “vị trí tương đối hoá trong tích vô hướng”, RoPE thường cho khả năng *ngoại suy chiều dài chuỗi* tốt hơn so với positional embedding thuần tuý cộng thêm, và tương thích tự nhiên với cơ chế scaled dot-product attention. Thực nghiệm trên nhiều mô hình lớn (LLaMA, Qwen, v.v.) cho thấy RoPE giúp mô hình ổn định hơn trên ngữ cảnh dài, tận dụng tốt hơn thông tin trạng thái, nên ngày càng trở thành lựa chọn mặc định cho các LLM hiện đại.

### IV.3. Grouped-query attention



Hình 7: Mô phỏng 3 dạng attention cơ bản: multi-head, grouped-query và multi-query.

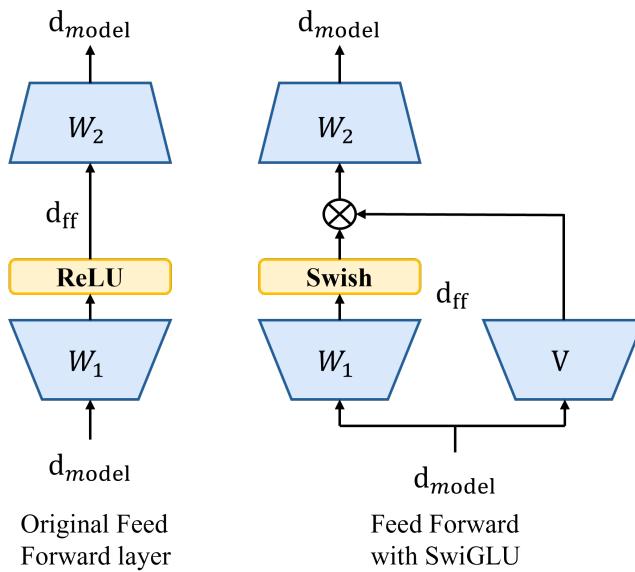
Trong Transformer chuẩn, *multi-head attention* sử dụng  $h$  head với các ma trận riêng cho query, key và value: mỗi head  $i$  có  $(W_Q^{(i)}, W_K^{(i)}, W_V^{(i)}) \in \mathbb{R}^{d_{\text{model}} \times d_h}$ . Ví dụ, với  $h = 16$  và chiều mỗi head  $d_h$ , ta có đủ 16 cặp  $(K, V)$  khác nhau, giúp mô hình học được nhiều kiểu quan hệ song song, nhưng chi phí lưu trữ  $KV$  cache trong giai đoạn suy luận tỉ lệ với  $16 \cdot L \cdot d_h$ , rất tốn bộ nhớ khi  $L$  và  $d_{\text{model}}$  lớn.

*Multi-query attention (MQA)* là một cực trị theo hướng tối ưu hiệu năng: thay vì có 16 cặp  $(K, V)$ , ta chỉ dùng *một* cặp  $(K, V)$  chung cho tất cả head (tức  $h_{\text{kv}} = 1$ ). Khi đó, chi phí KV cache giảm xuống còn  $\sim 1 \cdot L \cdot d_h$ , giúp tăng tốc decoding đáng kể. Tuy nhiên, vì mọi head cùng nhìn vào một không gian key–value duy nhất, mức độ đa dạng hoá biểu diễn có thể suy giảm, và một số mô hình lớn ghi nhận giảm nhẹ về chất lượng khi dùng MQA thuận tuý.

*Grouped-query attention (GQA)* được đề xuất như một điểm cân bằng giữa MHA và MQA. Ta chọn một số nhóm key–value  $h_{\text{kv}}$  sao cho  $1 < h_{\text{kv}} < h$ , ví dụ  $h = 16$  nhưng  $h_{\text{kv}} = 4$ . Khi đó, 16 head query được chia thành 4 nhóm, mỗi nhóm 4 head chia sẻ chung một cặp  $(K, V)$ , nên tổng số cặp  $(K, V)$  còn 4 thay vì 16 (giảm KV cache xuống  $\sim 4 \cdot L \cdot d_h$ ), trong khi vẫn giữ đủ số head query để mô hình hoá nhiều pattern attention khác nhau.

Nhờ giảm đáng kể chi phí lưu trữ và truy cập KV cache (gần với MQA) nhưng vẫn duy trì độ đa dạng không gian key–value tốt hơn, GQA thường cho chất lượng dịch/generative gần ngang multi-head truyền thống, với tốc độ suy luận và footprint bộ nhớ tương đương hoặc tiệm cận MQA. Đây là lý do nhiều LLM hiện đại (chẳng hạn LLaMA 2, Mistral, Qwen, v.v.) ưu tiên dùng *grouped-query attention* trong self-attention và/hoặc cross-attention, đặc biệt trong các ứng dụng yêu cầu serving thời gian thực và context dài.

## IV.4. SwiGLU



Hình 8: Mô phỏng hai dạng FFN thường gặp.

Trong Transformer gốc, tầng *feed-forward* (FFN) thường có dạng hai lớp tuyến tính kèm hàm kích hoạt, mở rộng chiều ẩn từ  $d_{model}$  lên  $d_{ff}$  rồi thu hẹp trở lại:

$$\text{FFN}(\mathbf{x}) = W_2 \phi(W_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2,$$

trong đó  $\phi$  thường là ReLU hoặc GELU. Cấu trúc này đơn giản nhưng ReLU có vùng chết (output bằng 0 cho giá trị âm) và không có cơ chế *gating*, nên đôi khi kém linh hoạt khi mô hình hoá các tương tác phi tuyến tính vi trong LLM lớn.

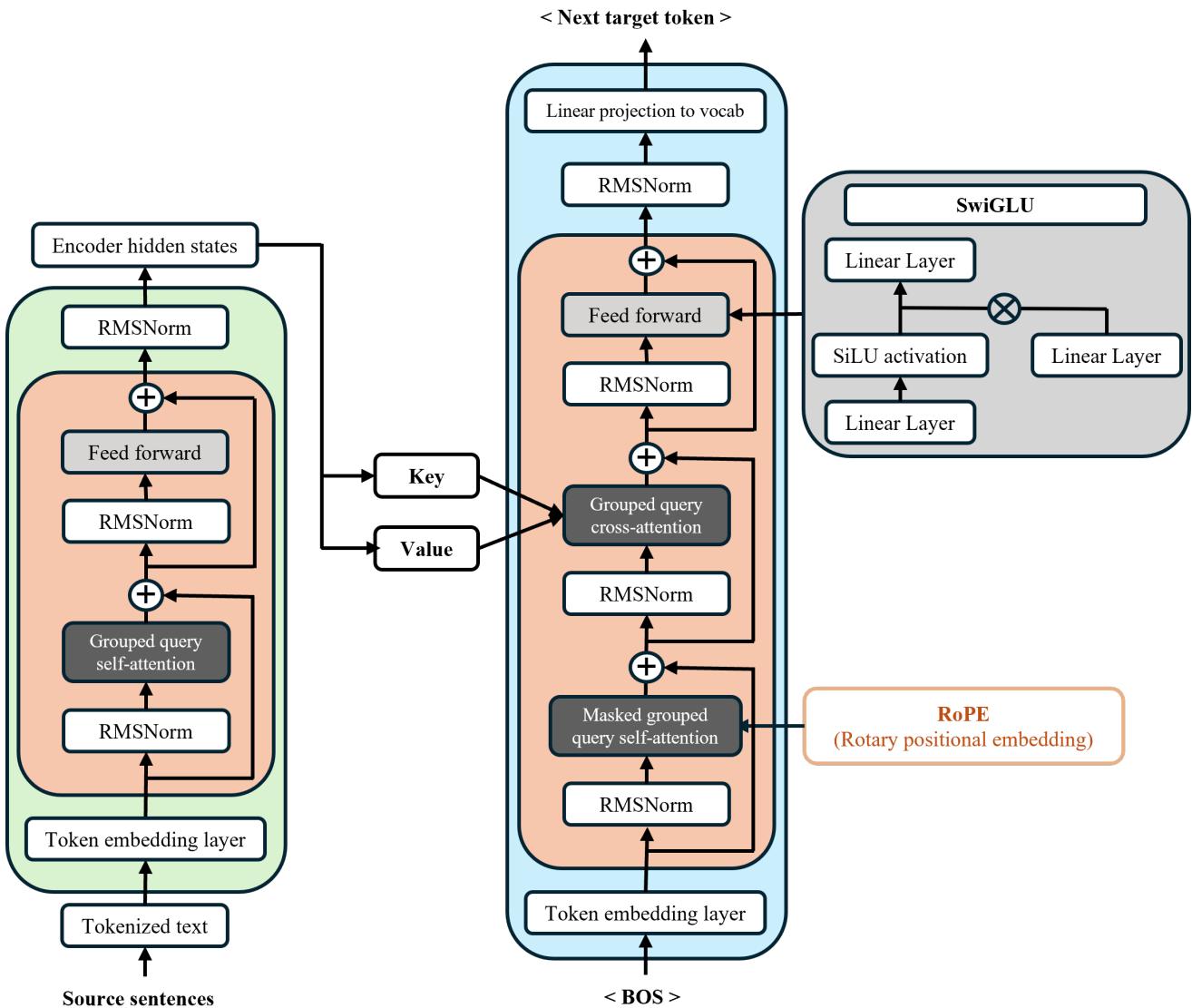
*SwiGLU* kết hợp ý tưởng *gating* (như GLU) với hàm kích hoạt trơn *Swish/SiLU*. Cho hai phép chiếu tuyến tính  $W_1, V \in \mathbb{R}^{d_{ff} \times d_{model}}$ , ta định nghĩa:

$$\begin{aligned} \mathbf{u} &= W_1 \mathbf{x}, & \mathbf{v} &= V \mathbf{x}, \\ \text{SwiGLU}(\mathbf{x}) &= W_2 (\text{SiLU}(\mathbf{u}) \odot \mathbf{v}), \end{aligned}$$

trong đó  $\text{SiLU}(t) = t \sigma(t)$  với  $\sigma(t)$  là hàm sigmoid, và  $\odot$  là phép nhân từng phần tử. Nhánh  $\text{SiLU}(\mathbf{u})$  đóng vai trò *gate* trơn, quyết định kênh nào được khuếch đại hay bị triệt tiêu, trong khi nhánh  $\mathbf{v}$  mang nội dung tuyến tính được điều biến bởi gate này.

Nhờ vừa có hàm kích hoạt trơn (giảm nguy cơ “chết” neuron như ReLU), vừa có cơ chế gating giàu biểu đạt, SwiGLU thường cho gradient ổn định hơn và khả năng mô hình hoá tốt hơn trong tầng FFN. Nhiều LLM hiện đại (PaLM, LLaMA, v.v.) đã chuyển từ FFN dùng ReLU/GELU sang SwiGLU và quan sát được cải thiện rõ rệt về perplexity và chất lượng sinh, đặc biệt ở thiết lập mô hình lớn với chiều  $d_{ff}$  rất cao.

## IV.5. Kiến trúc mô hình Transformer Encoder-Decoder được sử dụng



Hình 9: Kiến trúc mô hình được sử dụng trong bài toán dịch máy.

Mô hình được triển khai là một *Transformer Encoder-Decoder* với RMSNorm, Grouped-query attention kết hợp RoPE và tầng FFN dùng SwiGLU. Dưới đây là mô tả chi tiết vai trò của từng lớp trong quá trình lan truyền tiến.

**Trong Encoder:** Chuỗi nguồn được ánh xạ sang không gian liên tục thông qua *token embedding* và sau đó đi qua nhiều lớp encoder.

- **RMSNorm (pre-norm):** chuẩn hoá biên độ của các vector biểu diễn trước khi đưa vào self-attention, giúp ổn định gradient và duy trì độ lớn tín hiệu khi mô hình sâu.

- **Grouped-query self-attention with RoPE:** cho phép mỗi vị trí quan sát toàn bộ chuỗi nguồn để thu nhận ngữ cảnh toàn cục. RoPE mã hoá quan hệ vị trí tương đối trực tiếp trong không gian attention, còn grouped-query attention giảm chi phí lưu trữ và truy cập key-value mà vẫn giữ nhiều hướng truy vấn (query heads).
- **Residual connection:** giữ lại thông tin tầng trước, hỗ trợ mô hình hội tụ dễ hơn và tránh mất mát thông tin gốc.
- **RMSNorm:** chuẩn hoá lại tín hiệu sau attention nhằm bảo đảm độ ổn định trước khi đi vào tầng phi tuyến.
- **FFN với SwiGLU:** thực hiện các biến đổi phi tuyến độc lập trên từng vị trí. Cơ chế gating của SwiGLU cho phép mô hình chọn lọc các chiều quan trọng, tăng khả năng biểu đạt so với FFN truyền thống.
- **Residual connection:** duy trì dòng thông tin ổn định xuyên suốt nhiều lớp encoder.

Sau khi đi qua toàn bộ encoder, mô hình thu được bộ *contextual representations* giàu ngữ nghĩa, được sử dụng làm bộ nhớ cho decoder.

**Trong Decoder:** Chuỗi đích được shift-right để tạo đầu vào theo cơ chế autoregressive và đi qua nhiều lớp decoder có ba thành phần chính.

- **RMSNorm (pre-norm):** chuẩn bị tín hiệu ổn định trước khi vào self-attention.
- **Grouped-query masked self-attention with causal mask:** mô hình hoá phân bố ngôn ngữ của phía đích dưới điều kiện autoregressive. Causal mask đảm bảo mỗi vị trí chỉ được nhìn các token xuất hiện trước đó trong chuỗi.
- **Residual connection:** giữ thông tin trước attention và hỗ trợ lan truyền ổn định.
- **RMSNorm:** chuẩn hoá biểu diễn để chuẩn bị cho bước tương tác với encoder.
- **Grouped-query cross-attention:** cho phép decoder truy vấn lên toàn bộ encoder representations. Đây là nơi mô hình học cách căn chỉnh (alignment) giữa nguồn và đích để quyết định từ dịch phù hợp.
- **Residual connection:** duy trì sự ổn định của dòng thông tin sau cross-attention.
- **RMSNorm:** chuẩn hoá trước khi áp dụng tầng phi tuyến.
- **FFN với SwiGLU:** tinh chỉnh biểu diễn tại mỗi vị trí đích thông qua biến đổi phi tuyến mạnh, tăng khả năng mô hình hoá đặc trưng ngữ nghĩa phức tạp.
- **Residual connection:** hỗ trợ huấn luyện sâu và giữ thông tin nhất quán.

**Tầng sinh đầu ra:** Biểu diễn cuối của decoder được chuẩn hoá và chiếu lên không gian từ vựng thông qua trọng số embedding dùng chung, tạo thành logits cho token tiếp theo. Trong huấn luyện, các logits này được tối ưu bằng cross-entropy có label smoothing; trong suy luận, chúng được sử dụng để sinh chuỗi đích theo kiểu autoregressive.

# V. Mô hình Transformer Encoder-Decoder cho bài toán dịch thuật Hoa-Việt

Trong phần này, Mô hình dịch thuật Hoa-Việt được triển khai dựa trên kiến trúc *Transformer Encoder-Decoder* hiện đại, kết hợp các cải tiến như RMSNorm, RoPE, Grouped-query attention và tầng FFN-SwiGLU. Cấu trúc này cho phép mô hình nắm bắt ngữ cảnh song ngữ hiệu quả hơn, ổn định trong huấn luyện và đạt kết quả tốt dựa trên đánh giá của ban tổ chức.

## Bước 1: Import thư viện và tải dữ liệu

```

1 !gdown 1KuDLpUaSe0wzDWEhgG0wKXV1_AVmzLK4
2 !unzip ./dataset.zip -d .
3
4 import os
5 import random
6 import sentencepiece as spm
7 import pandas as pd
8 from tqdm import tqdm
9 import matplotlib.pyplot as plt
10 import numpy as np
11 import math
12 from dataclasses import dataclass
13 from typing import List, Tuple, Any, Dict, Optional
14
15 import torch
16 import torch.nn as nn
17 import torch.nn.functional as F
18 from torch.utils.data import Dataset, DataLoader, SubsetRandomSampler
19 import torch.optim as optim
20 import sacrebleu

```

Trong bước đầu tiên, tập dữ liệu được tải về từ Google Drive bằng `gdown` và giải nén ra thư mục làm việc. Tiếp theo, các thư viện cần thiết được import, bao gồm: `pandas`, `numpy`, `matplotlib` và `tqdm` cho xử lý và quan sát dữ liệu; `sentencepiece` cho việc xây dựng tokenizer theo kiểu subword; các module `torch`, `torch.nn`, `Dataset`, `DataLoader` và `optim` cho việc xây dựng và huấn luyện mô hình Transformer; cùng với `sacrebleu` dùng để đánh giá chất lượng dịch máy thông qua chỉ số BLEU.

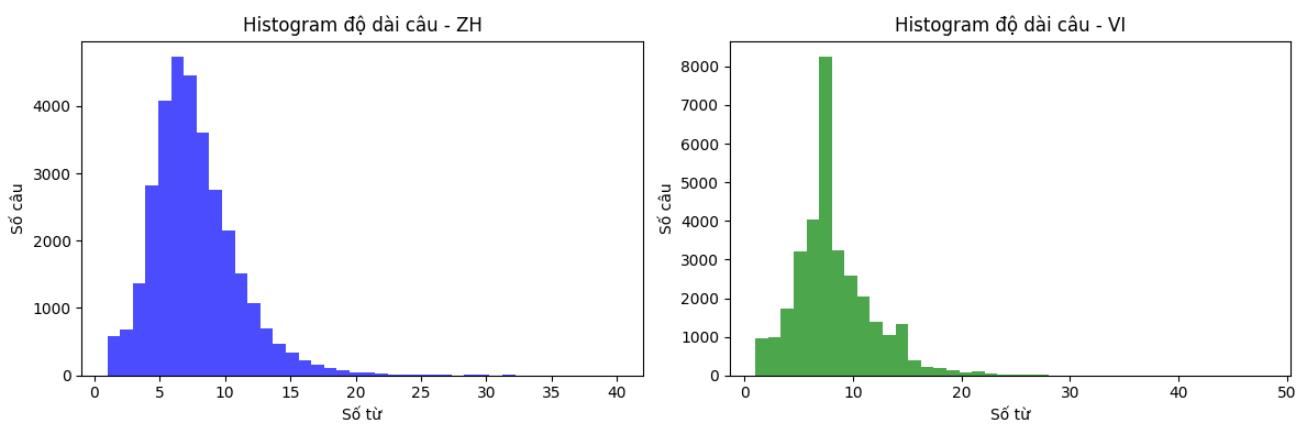
## Bước 2: Khám phá dữ liệu song ngữ

```

1 BASE_DIR = "."
2 TRAIN_DIR = os.path.join(BASE_DIR, "dataset", "train")
3 SRC_FILE = os.path.join(TRAIN_DIR, "train.zh")
4 TGT_FILE = os.path.join(TRAIN_DIR, "train.vi")
5 TOKENIZER_DIR = os.path.join(BASE_DIR, "tokenizer_train32")
6 TOKENIZER_PREFIX = os.path.join(TOKENIZER_DIR, "spm_zh_vi_joint")
7 MAX_TOKENS = 32
8 VOCAB_SIZE = 8000
9 USER_SYMBOLS = ["<2zh>", "<2vi>"]
10
11 os.makedirs(TOKENIZER_DIR, exist_ok=True)
12
13 # ----- LOAD -----
14 def load_lines(path):
15     with open(path, "r", encoding="utf-8") as f:
16         return [l.strip() for l in f if l.strip()]
17
18 zh_lines = load_lines(SRC_FILE)
19 vi_lines = load_lines(TGT_FILE)
20 #Xem thêm ở phần phụ lục

```

Trong bước này, các câu Hoa–Việt thô được nạp từ hai tệp `train.zh` và `train.vi`, kiểm tra lại số lượng để bảo đảm tương ứng 1–1 giữa hai ngôn ngữ. Một vài cặp câu đầu tiên được hiển thị nhằm quan sát nhanh chất lượng dữ liệu, đồng thời độ dài câu (tính theo số từ) được thống kê và trực quan hóa bằng histogram cho cả tiếng Hoa và tiếng Việt, giúp nắm được phân phối độ dài và chuẩn bị cho các bước cắt lọc sau này.



Hình 10: Biểu đồ số lượng từ của mỗi câu khi chưa xử lý

## Bước 3: Xây dựng tokenizer và lọc độ dài câu

```

1 # ----- TRAIN SENTENCEPIECE -----
2 temp_corpus = os.path.join(TOKENIZER_DIR, "temp_corpus.txt")
3 with open(temp_corpus, "w", encoding="utf-8") as fout:
4     for zh, vi in zip(zh_lines, vi_lines):
5         fout.write(zh + "\n")
6         fout.write(vi + "\n")
7
8 spm_args = (
9     f"--input={temp_corpus} ",
10    f"--model_prefix={TOKENIZER_PREFIX} ",
11    f"--vocab_size={VOCAB_SIZE} ",
12    f"--model_type=bpe ",
13    f"--character_coverage=1.0 ",
14    f"--pad_id=0 --unk_id=1 --bos_id=2 --eos_id=3 ",
15    f"--user_defined_symbols={','.join(USER_SYMBOLS)}"
16)
17 spm.SentencePieceTrainer.Train(" ".join(spm_args))
18
19 print(f"Tokenizer saved to {TOKENIZER_PREFIX}.model")
20
21 # ----- LOAD TOKENIZER -----
22 sp = spm.SentencePieceProcessor()
23 sp.Load(f"{TOKENIZER_PREFIX}.model")
24
25 # ----- RELOAD RAW DATA -----
26 def load_lines(path):
27     with open(path, "r", encoding="utf-8") as f:
28         return [l.strip() for l in f if l.strip()]
29
30 zh_lines = load_lines(SRC_FILE)
31 vi_lines = load_lines(TGT_FILE)
32 assert len(zh_lines) == len(vi_lines)
33
34 MAX_TOK = 32
35
36 def tok_len(t, prefix=None):
37     if prefix:
38         t = f"{prefix} {t}"
39     return len(sp.encode(t, out_type=int))
40
41 # Token length BEFORE filtering
42 zh_before = [tok_len(z, "<2vi>") for z in zh_lines]
43 vi_before = [tok_len(v) for v in vi_lines]
44
45 # Filter sample dài
46 filtered_zh, filtered_vi = [], []
47 for z, v in zip(zh_lines, vi_lines):
48     if tok_len(z, "<2vi>") <= MAX_TOK and tok_len(v) <= MAX_TOK:
49         filtered_zh.append(z)
50         filtered_vi.append(v)
51

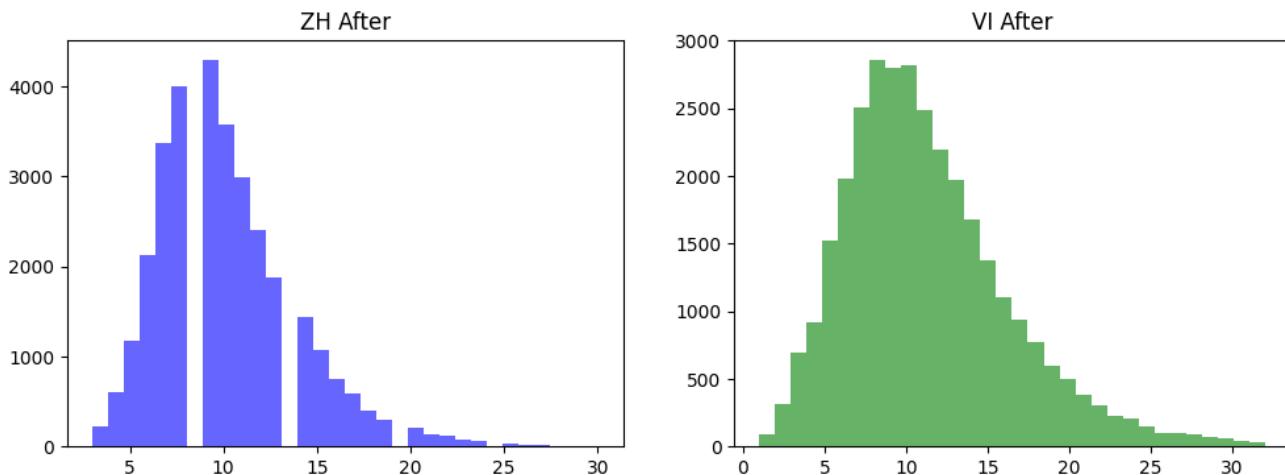
```

```

52 zh_after = np.array([tok_len(z, "<2vi>") for z in filtered_zh])
53 vi_after = np.array([tok_len(v) for v in filtered_vi])
54 #Xem thêm ở phần Phụ Lục

```

Trong bước tiếp theo, một *joint SentencePiece BPE tokenizer* được huấn luyện trên toàn bộ câu Hoa và Việt, trong đó hai token đặc biệt "<2zh>" và "<2vi>" được giữ cố định để đánh dấu ngôn ngữ nguồn. Sau khi tokenizer được huấn luyện và nạp lại, độ dài câu theo số lượng subword token được tính toán cho từng câu, và các cặp câu mà phía Hoa (kèm tiền tố "<2vi>") hoặc phía Việt vượt quá MAX\_TOK = 32 token sẽ bị loại bỏ. Histogram độ dài trước và sau khi lọc được vẽ để kiểm tra lại phân phối, trước khi lưu bộ dữ liệu sạch cuối cùng vào hai tệp train maxlen32.zh và train maxlen32.vi trong thư mục ./clean\_data để sử dụng cho giai đoạn huấn luyện mô hình.



Hình 11: Biểu đồ số lượng token của mỗi câu sau khi xử lý

## Bước 4: Cấu hình mô hình

```

1 # Cấu hình chung cho mô hình Transformer HoaViệt
2 @dataclass
3 class RopeConfig:
4     # Đường dẫn dữ liệu và tokenizer
5     train_src_file: str
6     train_tgt_file: str
7     spm_prefix: str
8
9     # Kiến trúc Transformer
10    vocab_size: int = 8000
11    d_model: int = 768
12    n_heads: int = 12
13    n_kv_heads: int = 4

```

```

14 num_encoder_layers: int = 8
15 num_decoder_layers: int = 8
16 d_ff: int = 3072
17 dropout: float = 0.01
18 max_len: int = 32
19 rope_base: float = 10000.0
20
21 # Token đặc biệt
22 pad_token: str = "<pad>"
23 bos_token: str = "<s>"
24 eos_token: str = "</s>"
25 unk_token: str = "<unk>"
26 zh_token: str = "<2zh>"
27 vi_token: str = "<2vi>"
28
29 # Tham số huấn luyện
30 batch_size: int = 128
31 num_epochs: int = 40
32 lr_base: float = 2e-4
33 warmup_steps: int = 200
34 label_smoothing: float = 0.01
35 grad_clip: float = 1.0
36 span_mask_prob: float = 0.01
37 vi2zh_epoch_ratio: float = 0.7
38
39 # Thiết bị và hệ thống
40 device: torch.device = torch.device(
41     "cuda" if torch.cuda.is_available() else "cpu"
42 )
43 num_workers: int = 8
44 save_dir: str = "./checkpoints_bidirectional"
45 seed: int = 42

```

Lớp RopeConfig gom toàn bộ siêu tham số cốt lõi của mô hình vào một cấu trúc duy nhất: bao gồm đường dẫn đến dữ liệu song ngữ đã làm sạch và tokenizer SentencePiece, các tham số kiến trúc của Transformer (d\_model, số lớp encoder/decoder, số attention heads và KV-heads, kích thước FFN, tham số RoPE), các token đặc biệt dùng trong huấn luyện, cùng các tham số huấn luyện quan trọng như batch\_size, num\_epochs, lr\_base, warmup\_steps, label\_smoothing, grad\_clip, tỉ lệ chọn mẫu Việt→Hoa theo từng epoch và thiết bị thực thi (cuda hoặc cpu).

## Bước 5: Dataset hai chiều và lịch chọn mẫu

```

1 class BidirectionalTranslationDataset(Dataset):
2     """Sinh dữ liệu hai chiều ZHVI với span masking nhẹ."""
3
4     def __init__(
5         self,
6         src_lines: List[str],
7         tgt_lines: List[str],

```

```

8     sp_model: spm.SentencePieceProcessor,
9     config: RopeConfig,
10    is_training: bool = True,
11):
12    self.sp = sp_model
13    self.config = config
14    self.is_training = is_training
15
16    self.pad_id = sp_model.piece_to_id(config.pad_token)
17    self.bos_id = sp_model.piece_to_id(config.bos_token)
18    self.eos_id = sp_model.piece_to_id(config.eos_token)
19    self.unk_id = sp_model.piece_to_id(config.unk_token)
20    self.zh_id = sp_model.piece_to_id(config.zh_token)
21    self.vi_id = sp_model.piece_to_id(config.vi_token)
22
23    self.samples = []
24    if is_training:
25        # Luân phiên gán huống zhvi và vizh
26        for i, (src, tgt) in enumerate(zip(src_lines, tgt_lines)):
27            if i % 2 == 0:
28                self.samples.append(
29                    (self.add_lang_token(src, config.vi_token), tgt, "zh2vi")
30                )
31            else:
32                self.samples.append(
33                    (self.add_lang_token(src, config.zh_token), tgt, "vi2zh")
34                )
35        else:
36            # Validation: chỉ giữ huống zhvi
37            for src, tgt in zip(src_lines, tgt_lines):
38                self.samples.append(
39                    (self.add_lang_token(src, config.vi_token), tgt, "zh2vi")
40                )
41
42        self.zh2vi_indices = [
43            idx for idx, (_, _, d) in enumerate(self.samples) if d == "zh2vi"
44        ]
45        self.vi2zh_indices = [
46            idx for idx, (_, _, d) in enumerate(self.samples) if d == "vi2zh"
47        ]
48
49    def add_lang_token(self, text: str, lang_tok: str) -> str:
50        text = text.strip()
51        if text.startswith("<2vi>") or text.startswith("<2zh>"):
52            return text
53        return f"{lang_tok} {text}"
54
55    def __len__(self):
56        return len(self.samples)
57
58    def __getitem__(self, idx: int):
59        src_text, tgt_text, _ = self.samples[idx]
60        src_ids = self.sp.encode(src_text, out_type=int)[: self.config.max_len]
61        tgt_ids = [self.bos_id] + self.sp.encode(tgt_text, out_type=int) + [self.eos_id]

```

```

62         ]
63     tgt_ids = tgt_ids[: self.config.max_len]
64     src_ids = self.apply_span_masking(src_ids)
65     return torch.tensor(src_ids, dtype=torch.long), torch.tensor(
66         tgt_ids, dtype=torch.long
67     )
68
69     def apply_span_masking(self, src_ids: List[int]) -> List[int]:
70         """Thay ngẫu nhiên một đoạn 12 token thường bằng <unk> với xác suất nhỏ."""
71         if not self.is_training or random.random() > self.config.span_mask_prob:
72             return src_ids
73         src_ids = src_ids.copy()
74         special = {self.pad_id, self.bos_id, self.eos_id, self.zh_id, self.vi_id}
75         maskable = [i for i, t in enumerate(src_ids) if t not in special]
76         if not maskable:
77             return src_ids
78         num_to_mask = min(random.randint(1, 2), len(maskable))
79         start = random.choice(maskable)
80         for i in range(start, min(start + num_to_mask, len(src_ids))):
81             if i in maskable and random.random() < 0.7:
82                 src_ids[i] = self.unk_id
83     return src_ids
84

```

```

1 def select_vi2zh_window(indices: List[int], epoch: int, ratio: float) -> List[int]:
2     """
3     Chọn một "cửa sổ" con của các mẫu vizh cho mỗi epoch
4     sao cho xấp xỉ ratio số mẫu vizh được dùng.
5     """
6
7     if not indices or ratio <= 0:
8         return []
9     total = len(indices)
10    window = max(1, int(math.ceil(total * min(ratio, 1.0))))
11    start = ((epoch - 1) * window) % total
12    end = start + window
13    if end <= total:
14        return indices[start:end]
15    wrap = end - total
16    return indices[start:] + indices[:wrap]
17
18 def build_train_loader(
19     train_dataset: BidirectionalTranslationDataset,
20     epoch: int,
21     config: RopeConfig,
22 ) -> Tuple[DataLoader, int]:
23     """
24     Mỗi epoch: lấy toàn bộ mẫu zhvi,
25     cộng thêm một cửa sổ vizh quay vòng theo epoch.
26     """
27     active = list(train_dataset.zh2vi_indices)
28     vi_slice = select_vi2zh_window(
29         train_dataset.vi2zh_indices, epoch, config.vi2zh_epoch_ratio

```

```

30     )
31     active.extend(vi_slice)
32
33     sampler = SubsetRandomSampler(active)
34     loader = DataLoader(
35         train_dataset,
36         batch_size=config.batch_size,
37         sampler=sampler,
38         # collate_fn, num_workers,... được thiết lập phù hợp với cấu hình.
39     )
40     return loader, len(vi_slice)

```

`BidirectionalTranslationDataset` xây dựng một tập dữ liệu song ngữ hai chiều, trong đó mỗi mẫu gồm câu nguồn, câu đích và nhãn hướng dịch ("zh2vi" hoặc "vi2zh"). Trong chế độ huấn luyện, các cặp câu được duyệt luân phiên: chỉ số chẵn gán hướng Hoa→Việt (gắn thêm token "<2vi>" vào đầu câu nguồn), chỉ số lẻ gán hướng Việt→Hoa (gắn "<2zh>"). Trong chế độ validation, mọi mẫu đều được chuẩn hoá về hướng Hoa→Việt để việc đánh giá tập trung vào chiều dịch chính.

Khi truy cập một phần tử, câu nguồn được mã hoá bằng SentencePiece (kèm language token) và cắt về `max_len`; hàm `apply_span_masking` với xác suất nhỏ chọn một đoạn 1–2 token thường và thay bằng "`<unk>`", giúp mô hình học biểu diễn ổn định hơn dưới nhiễu nhẹ ở phía nguồn. Câu đích được bọc `BOS/EOS`, cắt về `max_len` và cả hai được trả về dưới dạng tensor để đưa vào mô hình.

Hai danh sách chỉ số `zh2vi_indices` và `vi2zh_indices` lưu vị trí các mẫu theo từng hướng dịch. Hàm `select_vi2zh_window` định nghĩa một lịch chọn mẫu cho hướng Việt→Hoa: mỗi epoch chỉ lấy một "cửa sổ" con có kích thước tỉ lệ với `vi2zh_epoch_ratio`, và cửa sổ này được trượt tuần tự qua danh sách chỉ số. Hàm `build_train_loader` xây dựng `DataLoader` huấn luyện bằng cách luôn lấy toàn bộ mẫu Hoa→Việt, sau đó ghép thêm cửa sổ Việt→Hoa tương ứng với epoch hiện tại, giúp ưu tiên học tốt chiều Hoa→Việt nhưng vẫn khai thác đều đặn thông tin nghịch chiều Việt→Hoa trong suốt quá trình huấn luyện.

## Bước 6: Xây dựng khối Transformer với RMSNorm, RoPE, GQA và SwiGLU

```

1 class RMSNorm(nn.Module):
2     def __init__(self, d_model: int, eps: float = 1e-8):
3         super().__init__()
4         self.eps = eps
5         self.weight = nn.Parameter(torch.ones(d_model))
6
7     def forward(self, x: torch.Tensor) -> torch.Tensor:
8         rms = torch.sqrt(torch.mean(x ** 2, dim=-1, keepdim=True) + self.eps)
9         return self.weight * x / rms
10

```

```

11
12 class RoPE(nn.Module):
13     def __init__(self, d_model: int, base: float = 10000.0):
14         super().__init__()
15         inv_freq = 1.0 / (base ** (torch.arange(0, d_model, 2).float() / d_model))
16         self.register_buffer("inv_freq", inv_freq)
17         self._cos = None
18         self._sin = None
19         self._seq_len_cached = 0
20
21     def _maybe_update_cache(self, seq_len: int, device, dtype):
22         if seq_len > self._seq_len_cached or self._cos is None or self._cos.device != device:
23             self._seq_len_cached = seq_len
24             positions = torch.arange(seq_len, device=device, dtype=dtype)
25             freqs = torch.outer(positions, self.inv_freq.to(device))
26             self._cos = freqs.cos()
27             self._sin = freqs.sin()
28
29     def forward(self, x: torch.Tensor, seq_len: Optional[int] = None):
30         seq_len = seq_len or x.size(-2)
31         self._maybe_update_cache(seq_len, x.device, x.dtype)
32         return self._cos[:seq_len], self._sin[:seq_len]
33
34
35     def apply_rope(x: torch.Tensor, cos: torch.Tensor, sin: torch.Tensor) -> torch.Tensor:
36         x1 = x[..., 0::2]
37         x2 = x[..., 1::2]
38         cos = cos.unsqueeze(0).unsqueeze(0)
39         sin = sin.unsqueeze(0).unsqueeze(0)
40         rot1 = x1 * cos - x2 * sin
41         rot2 = x1 * sin + x2 * cos
42         return torch.stack([rot1, rot2], dim=-1).flatten(-2)
43
44
45 class FFN_SwiGLU(nn.Module):
46     def __init__(self, d_model: int, d_ff: int, dropout: float):
47         super().__init__()
48         self.d_ff = d_ff
49         self.linear1 = nn.Linear(d_model, 2 * d_ff, bias=False)
50         self.linear2 = nn.Linear(d_ff, d_model, bias=False)
51         self.dropout = nn.Dropout(dropout)
52
53     def forward(self, x: torch.Tensor) -> torch.Tensor:
54         h = self.linear1(x)
55         g, v = h[..., : self.d_ff], h[..., self.d_ff :]
56         s = g * torch.sigmoid(g)
57         out = self.linear2(s * v)
58         return self.dropout(out)
59
60
61 class GroupedQueryAttentionRoPE(nn.Module):
62     def __init__(self, d_model: int, n_heads: int, n_kv_heads: int, dropout: float,
63                  rope_base: float):

```

```

63     super().__init__()
64     assert n_heads % n_kv_heads == 0
65     self.n_heads = n_heads
66     self.n_kv_heads = n_kv_heads
67     self.n_groups = n_heads // n_kv_heads
68     self.d_k = d_model // n_heads
69     self.W_q = nn.Linear(d_model, n_heads * self.d_k, bias=False)
70     self.W_k = nn.Linear(d_model, n_kv_heads * self.d_k, bias=False)
71     self.W_v = nn.Linear(d_model, n_kv_heads * self.d_k, bias=False)
72     self.W_o = nn.Linear(d_model, d_model, bias=False)
73     self.dropout = nn.Dropout(dropout)
74     self.scale = math.sqrt(self.d_k)
75     self.rope = RoPE(self.d_k, base=rope_base)

76
77     def forward(self, q, k, v, key_padding_mask=None, attn_mask=None):
78         B, T_q = q.size(0), q.size(1)
79         T_k = k.size(1)
80
81         Q = self.W_q(q).view(B, T_q, self.n_heads, self.d_k).transpose(1, 2)
82         K = self.W_k(k).view(B, T_k, self.n_kv_heads, self.d_k).transpose(1, 2)
83         V = self.W_v(v).view(B, T_k, self.n_kv_heads, self.d_k).transpose(1, 2)
84
85         cos_q, sin_q = self.rope(Q, T_q)
86         cos_k, sin_k = self.rope(K, T_k)
87         Q = apply_rope(Q, cos_q, sin_q)
88         K = apply_rope(K, cos_k, sin_k)
89
90         K = K.repeat_interleave(self.n_groups, dim=1)
91         V = V.repeat_interleave(self.n_groups, dim=1)
92
93         scores = torch.matmul(Q, K.transpose(-2, -1)) / self.scale
94         if key_padding_mask is not None:
95             scores = scores.masked_fill(key_padding_mask.unsqueeze(1).unsqueeze(2),
96                                         float("-inf"))
96         if attn_mask is not None:
97             scores = scores.masked_fill(attn_mask.unsqueeze(0).unsqueeze(0), float(
98                                         "-inf"))
98
99         attn = F.softmax(scores, dim=-1)
100        attn = self.dropout(attn)
101        out = torch.matmul(attn, V)
102        out = out.transpose(1, 2).contiguous().view(B, T_q, -1)
103        return self.W_o(out)

104
105
106    class EncoderLayer(nn.Module):
107        def __init__(self, config: RopeConfig):
108            super().__init__()
109            self.ln1 = RMSNorm(config.d_model)
110            self.self_attn = GroupedQueryAttentionRoPE(
111                config.d_model, config.n_heads, config.n_kv_heads, config.dropout, config.
112                                            rope_base
113            )
114            self.ln2 = RMSNorm(config.d_model)

```

```

114     self.ffn = FFN_SwiGLU(config.d_model, config.d_ff, config.dropout)
115     self.dropout = nn.Dropout(config.dropout)
116
117     def forward(self, x, src_pad_mask=None):
118         x1 = self.ln1(x)
119         attn = self.self_attn(x1, x1, x1, key_padding_mask=src_pad_mask)
120         x = x + self.dropout(attn)
121         x2 = self.ln2(x)
122         return x + self.ffn(x2)
123
124
125 class DecoderLayer(nn.Module):
126     def __init__(self, config: RopeConfig):
127         super().__init__()
128         self.ln1 = RMSNorm(config.d_model)
129         self.self_attn = GroupedQueryAttentionRoPE(
130             config.d_model, config.n_heads, config.n_kv_heads, config.dropout, config.
131                                         rope_base
132         )
133         self.ln2 = RMSNorm(config.d_model)
134         self.cross_attn = GroupedQueryAttentionRoPE(
135             config.d_model, config.n_heads, config.n_kv_heads, config.dropout, config.
136                                         rope_base
137         )
138         self.ln3 = RMSNorm(config.d_model)
139         self.ffn = FFN_SwiGLU(config.d_model, config.d_ff, config.dropout)
140         self.dropout = nn.Dropout(config.dropout)
141
142     def forward(self, y, enc_out, tgt_pad_mask=None, tgt_causal_mask=None, src_pad_mask
143                 =None):
144         y1 = self.ln1(y)
145         self_attn = self.self_attn(
146             y1, y1, y1,
147             key_padding_mask=tgt_pad_mask,
148             attn_mask=tgt_causal_mask,
149         )
150         y = y + self.dropout(self_attn)
151
152         y2 = self.ln2(y)
153         cross_attn = self.cross_attn(y2, enc_out, enc_out, key_padding_mask=
154                                         src_pad_mask)
155         y = y + self.dropout(cross_attn)
156
157         y3 = self.ln3(y)
158         return y + self.ffn(y3)
159
160
161 class TransformerModel(nn.Module):
162     def __init__(self, config: RopeConfig, vocab_size: int):
163         super().__init__()
164         self.config = config
165         self.embedding = nn.Embedding(vocab_size, config.d_model, padding_idx=0)
166         self.emb_dropout = nn.Dropout(config.dropout)
167

```

```

164     self.encoder_layers = nn.ModuleList(
165         [EncoderLayer(config) for _ in range(config.num_encoder_layers)])
166     )
167     self.encoder_final_ln = RMSNorm(config.d_model)
168
169     self.decoder_layers = nn.ModuleList(
170         [DecoderLayer(config) for _ in range(config.num_decoder_layers)])
171     )
172     self.decoder_final_ln = RMSNorm(config.d_model)
173
174     self.output_bias = nn.Parameter(torch.zeros(vocab_size))
175     self.emb_scale = math.sqrt(config.d_model)
176     self._init_weights()
177
178     def _init_weights(self):
179         for p in self.parameters():
180             if p.dim() > 1:
181                 nn.init.xavier_uniform_(p)
182
183     def forward(self, src_ids: torch.Tensor, tgt_ids: torch.Tensor) -> torch.Tensor:
184         src_pad = (src_ids == 0)
185         tgt_pad = (tgt_ids == 0)
186
187         tgt_in = tgt_ids[:, :-1]
188         tgt_pad_in = tgt_pad[:, :-1]
189         T = tgt_in.size(1)
190         tgt_causal = torch.triu(
191             torch.ones(T, T, dtype=torch.bool, device=src_ids.device),
192             diagonal=1,
193         )
194
195         src_emb = self.emb_dropout(self.embedding(src_ids) * self.emb_scale)
196         enc_out = src_emb
197         for layer in self.encoder_layers:
198             enc_out = layer(enc_out, src_pad)
199             enc_out = self.encoder_final_ln(enc_out)
200
201         tgt_emb = self.emb_dropout(self.embedding(tgt_in) * self.emb_scale)
202         dec_out = tgt_emb
203         for layer in self.decoder_layers:
204             dec_out = layer(dec_out, enc_out, tgt_pad_in, tgt_causal, src_pad)
205             dec_out = self.decoder_final_ln(dec_out)
206
207         return F.linear(dec_out, self.embedding.weight, self.output_bias)

```

Doạn mã trên hiện thực toàn bộ các khối kiến trúc cốt lõi của mô hình dịch Hoa–Việt. Lớp RMSNorm chuẩn hoá độ lớn vector theo chuẩn căn-trung-bình-bình-phương, giúp ổn định gradient khi số lớp encoder/decoder lớn. Khối RoPE và hàm `apply_rope` mã hoá vị trí bằng phép quay trên từng cặp chiều của query/key, nhưng thông tin vị trí tương đối trực tiếp vào không gian attention.

Lớp FFN\_SwiGLU hiện thực tầng feed-forward với cơ chế SwiGLU: đầu tiên chiều biểu diễn sang

không gian ẩn gấp đôi, sau đó tách thành hai nhánh *gate* và *value*, *gate* được kích hoạt bằng SiLU và dùng để điều biến nhánh *value* trước khi chiếu ngược về *d\_model*. Cách thiết kế này cho phép tầng FFN biểu đạt các quan hệ phi tuyến sâu hơn so với FFN dùng ReLU/GELU.

Lớp `GroupedQueryAttentionRoPE` kết hợp ba ý tưởng: multi-head attention, grouped-query và RoPE. Ở đây, mỗi head query vẫn độc lập, nhưng các head được chia thành các nhóm cùng chia sẻ một cặp key-value, giúp giảm chi phí KV cache trong khi vẫn giữ được nhiều hướng chú ý khác nhau. RoPE được áp dụng lên Q và K trước khi tính attention scores, đảm bảo mô hình tận dụng tốt thông tin thứ tự.

`EncoderLayer` và `DecoderLayer` sắp xếp các khối trên theo sơ đồ pre-norm: mỗi lớp encoder gồm RMSNorm → grouped-query self-attention with RoPE → residual → RMSNorm → FFN-SwiGLU → residual; mỗi lớp decoder gồm RMSNorm → masked grouped-query self-attention with causal mask → residual, tiếp theo là RMSNorm → grouped-query cross-attention lên encoder outputs → residual, và cuối cùng RMSNorm → FFN-SwiGLU → residual.

Lớp `TransformerModel` gắn kết toàn bộ: embedding chung cho cả nguồn và đích, một stack nhiều `EncoderLayer` để tạo contextual representations cho câu nguồn, một stack nhiều `DecoderLayer` để sinh câu đích theo cơ chế autoregressive với causal mask, và một tầng tuyến tính cuối cùng sử dụng lại trọng số embedding để sinh logits trên không gian từ vựng. Đây là hiện thực cụ thể của kiến trúc Transformer Encoder–Decoder được mô tả trong phần lý thuyết cho bài toán dịch Hoa–Việt.

## Bước 7: Hàm măt mát với label smoothing

```

1 class LabelSmoothedCrossEntropyLoss(nn.Module):
2     def __init__(self, smoothing: float = 0.1, ignore_index: int = 0):
3         super().__init__()
4         self.smoothing = smoothing
5         self.ignore_index = ignore_index
6
7     def forward(self, logits: torch.Tensor, targets: torch.Tensor) -> torch.Tensor:
8         vocab_size = logits.size(-1)
9         log_probs = F.log_softmax(logits, dim=-1)
10
11     # Không tính loss trên vị trí padding
12     mask = targets != self.ignore_index
13
14     with torch.no_grad():
15         # Phân phối "mềm" thay vì one-hot
16         true_dist = torch.full_like(
17             log_probs, self.smoothing / (vocab_size - 1)
18         )
19         true_dist.scatter_(1, targets.unsqueeze(1), 1.0 - self.smoothing)
20         true_dist[targets == self.ignore_index] = 0.0
21
22     loss = -(true_dist * log_probs).sum(dim=-1)
23     loss = loss.masked_fill(~mask, 0.0)
24     return loss.sum() / mask.sum().clamp(min=1)

```

Hàm mất mát `LabelSmoothedCrossEntropyLoss` thay thế phân phối one-hot truyền thống bằng một phân phối *mềm*: thay vì gán xác suất 1.0 cho token đích duy nhất và 0.0 cho tất cả token còn lại, một phần nhỏ xác suất (*smoothing*) được phân bổ đều lên các lớp khác nhau trong vocab. Điều này làm giảm độ tự tin cực đoan của mô hình, đóng vai trò như một dạng regularization giúp mô hình bớt overfit và ổn định hơn trên dữ liệu nhiễu.

Trong quá trình tính loss, các logits được đưa qua `log_softmax`, sau đó nhân với phân phối mục tiêu đã được làm mượt `true_dist` để thu được giá trị cross-entropy tại từng vị trí. Tham số `ignore_index` cho phép loại bỏ hoàn toàn các vị trí padding (thường tương ứng với token `<pad>`), bằng cách đặt phân phối mục tiêu về 0 tại đó và không tính các phần tử này vào mẫu số khi chuẩn hoá loss trung bình. Cách hiện thực này đảm bảo hàm mất mát vừa phản ánh tốt chất lượng dự đoán trên token thật, vừa tránh việc padding làm sai lệch giá trị loss trong quá trình huấn luyện.

```
Epoch 39/40
vi→zh coverage: 10598/15140 (~70.0% per epoch)
Train Loss: 0.1666
Valid Loss: 0.8484
Valid BLEU: 63.30
Learning Rate: 0.000045
Epoch 40: 100%|██████████| 101/101 [00:28<00:00, 3.49it/s, loss=0.1691, lr=0.000044]
Epoch 40: 100%|██████████| 101/101 [00:28<00:00, 3.49it/s, loss=0.1691, lr=0.000044]

Epoch 40/40
vi→zh coverage: 10598/15140 (~70.0% per epoch)
Train Loss: 0.1665
Valid Loss: 0.8474
Valid BLEU: 63.35
Learning Rate: 0.000044
✓ Saved checkpoint to ./checkpoints_bidirectional/checkpoint_epoch_40.pt
```

Hình 12: Kết quả các epoch cuối sau khi pre-training.

## Bước 8: Thành phần contrastive và mất mát song ngữ

```
1 import torch.nn as nn
2 import torch.nn.functional as F
3
4 # Projection head: ánh xạ vector ẩn sang không gian embedding đối lập
5 class ProjectionHead(nn.Module):
6     def __init__(self, input_dim: int, proj_dim: int):
7         super().__init__()
8         self.net = nn.Sequential(
9             nn.Linear(input_dim, input_dim),
10            nn.ReLU(),
11            nn.Linear(input_dim, proj_dim),
12        )
13
14     def forward(self, x):
15         # Chuẩn hoá L2 để dùng cosine similarity
```

```
16     return F.normalize(self.net(x), dim=-1)
17
18
19 def encode_context(model, src_ids, pad_id):
20     """Chạy encoder để lấy biểu diễn ngữ cảnh cho toàn bộ câu."""
21     src_pad_mask = (src_ids == pad_id)
22     src_emb = model.embedding(src_ids) * model.emb_scale
23     src_input = model.emb_dropout(src_emb)
24
25     enc_out = src_input
26     for layer in model.encoder_layers:
27         enc_out = layer(enc_out, src_pad_mask)
28     enc_out = model.encoder_final_ln(enc_out)
29     return enc_out
30
31
32 def mean_pool(enc_out, ids, pad_id, special_ids):
33     """Mean pooling trên các token nội dung, bỏ pad + special tokens."""
34     mask = (ids != pad_id)
35     for special in special_ids:
36         mask = mask & (ids != special)
37     mask = mask.float()
38
39     summed = (enc_out * mask.unsqueeze(-1)).sum(dim=1)
40     denom = mask.sum(dim=1, keepdim=True).clamp(min=1.0)
41     return summed / denom
42
43
44 def contrastive_loss(z_a, z_b, tau):
45     """
46     InfoNCE loss hai chiều giữa hai view (z_a, z_b):
47     mỗi câu dùng bản dịch của chính nó là positive, câu khác là negative.
48     """
49     sim = torch.matmul(z_a, z_b.transpose(0, 1)) / tau
50     labels = torch.arange(sim.size(0), device=sim.device)
51
52     loss_i = F.cross_entropy(sim, labels)
53     loss_j = F.cross_entropy(sim.transpose(0, 1), labels)
54     return 0.5 * (loss_i + loss_j)
55
56
57 def compute_crosslingual_loss(
58     model,
59     projection,
60     ids_zh_vi,
61     ids_vi_vi,
62     ids_vi_zh,
63     ids_zh_zh,
64     cl_config,
65     pad_id,
66     special_ids,
67 ):
68     if ids_zh_vi.size(0) < 2:
69         return torch.zeros(1, device=config.device)
```

```

70
71 # 1) Mã hoá 4 view của cùng cặp câu
72 enc_zh_vi = encode_context(model, ids_zh_vi, pad_id)
73 enc_vi_vi = encode_context(model, ids_vi_vi, pad_id)
74 enc_vi_zh = encode_context(model, ids_vi_zh, pad_id)
75 enc_zh_zh = encode_context(model, ids_zh_zh, pad_id)
76
77 # 2) Mean pooling + projection sang không gian đối lập
78 z_zh_vi = projection(mean_pool(enc_zh_vi, ids_zh_vi, pad_id, special_ids))
79 z_vi_vi = projection(mean_pool(enc_vi_vi, ids_vi_vi, pad_id, special_ids))
80 z_vi_zh = projection(mean_pool(enc_vi_zh, ids_vi_zh, pad_id, special_ids))
81 z_zh_zh = projection(mean_pool(enc_zh_zh, ids_zh_zh, pad_id, special_ids))
82
83 # 3) Hai loss đối lập song ngữ: "vi-space" và "zh-space"
84 cl_vi_space = contrastive_loss(
85     z_zh_vi, z_vi_vi, cl_config.contrastive_tau
86 )
87 cl_zh_space = contrastive_loss(
88     z_vi_zh, z_zh_zh, cl_config.contrastive_tau
89 )
90
91 # Trung bình hai không gian
92 return 0.5 * (cl_vi_space + cl_zh_space)

```

Khối `ProjectionHead` đóng vai trò ánh xạ các vector ẩn từ encoder (kích thước `d_model`) sang một không gian embedding đối lập có chiều thấp hơn (`proj_dim`), kèm chuẩn hoá L2 để độ đo tương tự giữa hai câu tương ứng trở thành cosine similarity. Mỗi câu sau khi đi qua encoder được *mean pooling* trên các token nội dung (loại bỏ pad, BOS/EOS và language tokens), rồi đưa qua projection head để thu được một vector biểu diễn duy nhất cho toàn bộ câu.

Hàm `contrastive_loss` hiện thực một biến thể InfoNCE hai chiều: ma trận `sim` chứa cosine similarity (sau khi chia cho nhiệt độ `tau`) giữa từng cặp câu trong batch; mỗi câu sử dụng bản dịch của chính nó làm *positive*, các câu còn lại trong batch làm *negative*. Loss được tính theo cả hai hướng (dùng `z_a` truy vấn `z_b` và ngược lại), sau đó lấy trung bình để cân bằng hai phía.

Hàm `compute_crosslingual_loss` tạo ra bốn view cho mỗi cặp song ngữ: Hoa với token "`<2vi>`" và "`<2zh>`", Việt với token "`<2vi>`" và "`<2zh>`". Các view này lần lượt được mã hoá, pooling và đưa qua projection head, từ đó xây dựng hai không gian đối lập: một “vi-space” (căn chỉnh `zh_vi` với `vi_vi`), một “zh-space” (căn chỉnh `vi_zh` với `zh_zh`). Việc tối ưu trung bình hai InfoNCE loss trong hai không gian này buộc mô hình đưa các câu dịch tương ứng trong Hoa–Việt tiến gần nhau trong không gian ngữ nghĩa chung, tăng chất lượng biểu diễn cross-lingual ngay tại tầng encoder và hỗ trợ tốt hơn cho nhiệm vụ dịch chính.

```

Contrastive Epoch 19/20
CE Loss: 0.1499
CL Loss: 0.0111
Total Loss: 0.1510
Learning Rate: 0.000010
Contrastive Epoch 20: 100% |██████████| 250/250 [01:32<00:00, 2.71it/s, ce=0.1506, cl=0.0000, λ=0.100, lr=0.000010]

```

```

Contrastive Epoch 20/20
CE Loss: 0.1497
CL Loss: 0.0114
Total Loss: 0.1508
Learning Rate: 0.000010

```

Hình 13: Kết quả các epoch cuối sau khi contrastive training.

## Bước 9: Kết quả xếp hạng của model được xây dựng

#	Thí sinh	Ngày	ID	Total Score
1	Không phải TQKhang	2025-11-19 14:59	6067	<b>68.8</b>
2	anhht	2025-11-27 17:02	6965	<b>68.55</b>
3	tvan	2025-11-22 03:35	6471	<b>66.95</b>
4	Hao Buoi Cong	2025-11-20 14:33	6234	<b>65.9</b>
5	quanghien	2025-11-28 11:50	7019	<b>65.71</b>

Hình 14: Kết quả xếp hạng của model trong cuộc thi.

Mô hình Transformer Encoder–Decoder đề xuất, với các cải tiến như RMSNorm, RoPE, grouped-query attention, FFN dùng SwiGLU, huấn luyện hai chiều Hoa–Việt kết hợp *label smoothing* và lịch học *warmup + inverse square root*, đã cho kết quả khả quan trên hệ thống chấm điểm trực tuyến của Ban tổ chức. Sau khi huấn luyện xong, mô hình được dùng để sinh dự đoán cho hai file `public_test.csv` và `private_test.csv` theo đúng định dạng yêu cầu, nộp lên hệ thống và thu được điểm tổng **65.71**, xếp hạng **Top 5** như thể hiện trên bảng xếp hạng. Kết quả này cho thấy thiết kế kiến trúc và quy trình tiền xử lý–huấn luyện là hợp lý, đủ sức cạnh tranh với các lời giải hàng đầu trong cuộc thi dịch máy Hoa–Việt.

# VI. Hướng phát triển và cải tiến mô hình

Dựa trên nền tảng mô hình *Transformer Encoder–Decoder* với RMSNorm, RoPE, Grouped-Query Attention, FFN-SwiGLU và thành phần contrastive song ngữ đã trình bày, mô hình dịch Hoa–Việt hoàn toàn có thể được phát triển thêm theo nhiều hướng khác nhau. Phần này đề xuất một số hướng mở rộng thực nghiệm nhằm cải thiện hơn nữa chỉ số SacreBLEU và chất lượng dịch thực tế.

## VI.1. Thủ nghiêm đa cấu hình: kiến trúc, từ vựng và lịch huấn luyện

Một hướng phát triển tự nhiên là khảo sát có hệ thống các siêu tham số đã được cố định trong cấu hình `RopeConfig`, bao gồm kích thước từ vựng, kích thước mô hình và lịch huấn luyện:

- **Kích thước từ vựng (VOCAB\_SIZE):** Thủ nghiêm các giá trị khác nhau (ví dụ 4k, 8k, 12k, 16k) để đánh đổi giữa: (i) Độ mịn biểu diễn subword (giảm `<unk>` và cải thiện xử lý từ hiếm) và (ii) Độ khó khi tối ưu trên không gian từ vựng quá lớn.
- **Kích thước mô hình (d\_model, d\_ff, số lớp encoder/decoder):** So sánh các cấu hình “base” và “large”:
  - Tăng `d_model` và `d_ff` giúp mô hình có năng lực biểu diễn mạnh hơn.
  - Tăng số lớp encoder/decoder giúp nắm bắt ngữ cảnh sâu hơn, nhưng cần thêm regularization (như label smoothing, R-Drop) để tránh overfitting.
- **Số epoch và lịch học:**
  - Huấn luyện lâu hơn với `num_epochs` lớn hơn, kết hợp `early stopping` trên tập validation.
  - Điều chỉnh `warmup_steps` và dạng scheduler (Noam, cosine decay) để mô hình hội tụ ổn định.
  - Thủ thay đổi `vi2zh_epoch_ratio` để đánh giá mức độ đóng góp của chiều Việt→Hoa vào việc cải thiện chiều Hoa→Việt.

Các thí nghiệm này có thể được tổ chức theo dạng *grid search* hoặc *random search*, sau đó so sánh SacreBLEU trên tập kiểm tra công khai để lựa chọn cấu hình tối ưu.

## VI.2. Khai thác Back-Translation để mở rộng dữ liệu

Bên cạnh bộ song ngữ gốc, một hướng rất hiệu quả là áp dụng *Back-Translation* để tận dụng dữ liệu đơn ngữ tiếng Việt:

- Sử dụng mô hình hiện tại (hoặc một mô hình đơn giản hơn) dịch các câu tiếng Việt đơn ngữ sang tiếng Hoa, tạo thành cặp (`zh_synthetic`, `vi_real`).
- Trộn các cặp câu sinh ra này với bộ song ngữ gốc trong giai đoạn huấn luyện tiếp theo (fine-tuning).

Điểm mạnh của Back-Translation:

- Tận dụng được nguồn dữ liệu đơn ngữ tiếng Việt ngoài tập huấn luyện ban đầu.
- Tăng độ đa dạng về cấu trúc câu, ngữ cảnh và cách diễn đạt ở phía đích (Việt), giúp mô hình học cách sinh câu tự nhiên hơn.
- Có thể kết hợp với chiến lược huấn luyện hai giai đoạn: pre-training với dữ liệu song ngữ gốc, sau đó fine-tuning với tập mở rộng có thêm dữ liệu Back-Translation.

### VI.3. Regularization nâng cao: R-Drop và tối ưu hoá SAM

R-Drop Regularization: Trên nền `LabelSmoothedCrossEntropyLoss`, có thể bổ sung R-Drop để giảm phuơng sai giữa các lần forward pass:

- Với mỗi batch, chạy hai lần forward với cùng đầu vào nhưng khác `dropout mask`.
- Tính thêm một thành phần KL-divergence giữa hai phân phối đầu ra logits:

$$\mathcal{L}_{\text{R-Drop}} = \text{KL}\left(p_{\theta}^{(1)} \| p_{\theta}^{(2)}\right) + \text{KL}\left(p_{\theta}^{(2)} \| p_{\theta}^{(1)}\right),$$

và cộng với loss đích chính.

- Cách này khuyến khích mô hình sinh ra phân phối dự đoán nhất quán hơn, giảm overfitting và thường cải thiện BLEU.

SAM (Sharpness-Aware Minimization): Ngoài các optimizer chuẩn như Adam/AdamW, có thể thử *SAM optimizer*:

- SAM không chỉ tối ưu giá trị loss tại một điểm tham số, mà còn cố gắng tìm các vùng lân cận có loss nhỏ (vùng phẳng).
- Trực giác: nghiệm nằm ở “vùng phẳng” thường có khả năng tổng quát tốt hơn, ít nhạy với nhiễu dữ liệu.
- Trong thực nghiệm, SAM cho thấy hiệu quả trên nhiều bài toán thị giác và ngôn ngữ, nên việc áp dụng cho bài toán dịch Hoa–Việt là hướng đáng khảo sát.

## VI.4. Mở rộng khung contrastive với cặp dương từ nhiều phiên bản của cùng một câu

Phần trước đã xây dựng một loss contrastive song ngữ dựa trên các view khác nhau của cùng một cặp Hoa–Việt (`zh_vi`, `vi_vi`, `vi_zh`, `zh_zh`). Một hướng mở rộng là bổ sung một nhánh contrastive theo tinh thần *self-supervised* đơn ngữ, trong đó cặp dương được xây dựng từ các phiên bản khác nhau của chính một câu:

- Tạo nhiều phiên bản (views) của cùng một câu bằng các phép biến đổi:
  - Span masking mạnh hơn ở phía encoder.
  - Dropout ở tầng embedding hoặc encoder.
  - Round-trip translation đơn giản:  $x \rightarrow \hat{y} \rightarrow \tilde{x}$  (với  $\hat{y}$  là bản dịch tạm), sau đó dùng  $x$  và  $\tilde{x}$  làm hai views.
- Đặt các phiên bản này làm cặp dương trong InfoNCE:

$$\mathcal{L}_{\text{self-CL}} = \text{InfoNCE}\left(z(x^{(1)}), z(x^{(2)})\right),$$

với negative là các câu khác trong batch.

- Kết hợp  $\mathcal{L}_{\text{self-CL}}$  với loss contrastive song ngữ hiện tại và loss dịch:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{MT}} + \lambda_{\text{CL-bilingual}} \mathcal{L}_{\text{CL-bilingual}} + \lambda_{\text{CL-self}} \mathcal{L}_{\text{self-CL}}.$$

Cách làm này giúp encoder học được biểu diễn ổn định hơn cho từng câu riêng lẻ, đồng thời hỗ trợ cho việc căn chỉnh song ngữ trong không gian embedding chung.

## VI.5. Khảo sát các chiến lược giải mã: beam search, top- $k$ , top- $p$

Ở giai đoạn suy luận, ngoài beam search cổ điển, có thể khảo sát thêm các chiến lược giải mã khác nhau để đánh đổi giữa độ chính xác và độ tự nhiên của bản dịch:

- Beam search với điều chỉnh penalty:
  - Thử các giá trị `beam_size` khác nhau (ví dụ 4, 6, 8).
  - Điều chỉnh *length penalty* để tránh hiện tượng sinh câu quá ngắn hoặc quá dài.
- Sampling-based decoding:
  - *Top-k sampling*: tại mỗi bước, chỉ giữ lại  $k$  token có xác suất cao nhất và sample trong tập này, phù hợp khi muốn sinh ra nhiều bản dịch đa dạng.
  - *Top-p (nucleus) sampling*: chọn tập nhỏ nhất các token sao cho tổng xác suất  $\geq p$  (ví dụ  $p = 0,9$ ), sau đó sampling trong tập đó.

- **Kết hợp search và reranking:**

- Sinh ra nhiều candidate bằng beam search hoặc sampling.
- Dùng một language model tiếng Việt (huấn luyện riêng từ tập vi\_lines) để rerank theo độ trôi chảy, chọn bản dịch tự nhiên và phù hợp ngữ pháp hơn.

Các chiến lược này không thay đổi trọng số mô hình, nhưng có thể tác động đáng kể đến chất lượng dịch cuối cùng, đặc biệt về độ mượt và tự nhiên của câu tiếng Việt.

## **VI.6. Vòng lặp suy luận, phân tích lỗi và cải thiện mô hình**

Một bước quan trọng để phát triển mô hình bền vững là xây dựng vòng lặp phản hồi dựa trên kết quả suy luận:

### **1. Suy luận trên tập kiểm tra/validation:**

- Chạy inference trên public\_test.zh và một phần train.zh được dành riêng làm validation nội bộ.
- Lưu lại đồng thời bản dịch của mô hình và bản dịch tham chiếu (khi có).

### **2. Lọc và phân loại các dự đoán sai:**

- Ưu tiên phân tích các câu có BLEU câu-level thấp hoặc chênh lệch lớn về độ dài so với tham chiếu.
- Gắn nhãn thủ công một số nhóm lỗi tiêu biểu:
  - Dịch sai thuật ngữ chuyên ngành.
  - Sai trật tự cú pháp (chủ-vị-tân, vị trí trạng ngữ).
  - Bỏ sót hoặc lặp cụm từ.
  - Xử lý chưa tốt tên riêng, số, đơn vị đo lường.

### **3. Rút ra hướng điều chỉnh mô hình:**

- Nếu xuất hiện nhiều lỗi về thuật ngữ: bổ sung từ điển song ngữ hoặc lớp hậu xử lý (post-processing) cho các trường hợp đặc biệt.
- Nếu bản dịch hay lặp từ/cụm: điều chỉnh chiến lược decoding (thêm penalty cho repetition) hoặc tăng regularization (R-Drop, SAM).
- Nếu lỗi chủ yếu liên quan đến trật tự cú pháp: tăng số lớp encoder/decoder hoặc bổ sung dữ liệu Back-Translation với cấu trúc câu đa dạng hơn.

### **4. Huấn luyện lại hoặc fine-tuning với thông tin mới:**

- Tạo tập “hard examples” gồm những câu mô hình dịch sai nhiều để fine-tuning tập trung.
- Lặp lại bước đo SacreBLEU và phân tích lỗi để đánh giá hiệu quả của từng thay đổi.

Thông qua vòng lặp này, mô hình không chỉ được cải thiện về các chỉ số tổng quan như SacreBLEU, mà còn được tinh chỉnh dần theo những dạng lỗi thực tế thường gặp trong bài toán dịch Hoa–Việt, phù hợp với yêu cầu của đề thi và các ứng dụng triển khai trong môi trường thật.

## Phụ lục

1. **Code:** Các file code được đề cập trong bài có thể được tải tại [tại đây](#).
2. **Dataset:** Bộ dữ liệu có thể được tải tại [tại đây](#).
3. **Contest:** Thẻ lẻ về cuộc thi có thể đọc tại [tại đây](#).
4. **Q&A:** Bạn có thể đặt thêm câu hỏi về nội dung bài đọc trong group Facebook hỏi đáp tại [đây](#). Tất cả câu hỏi sẽ được trả lời trong vòng tối đa 4 giờ.

### AIO\_QAs-Verified

🔒 Private group · 1.4K members



Hình 15: Hình ảnh group facebook AIO Q&A

AI VIET NAM – AI COURSE 2025

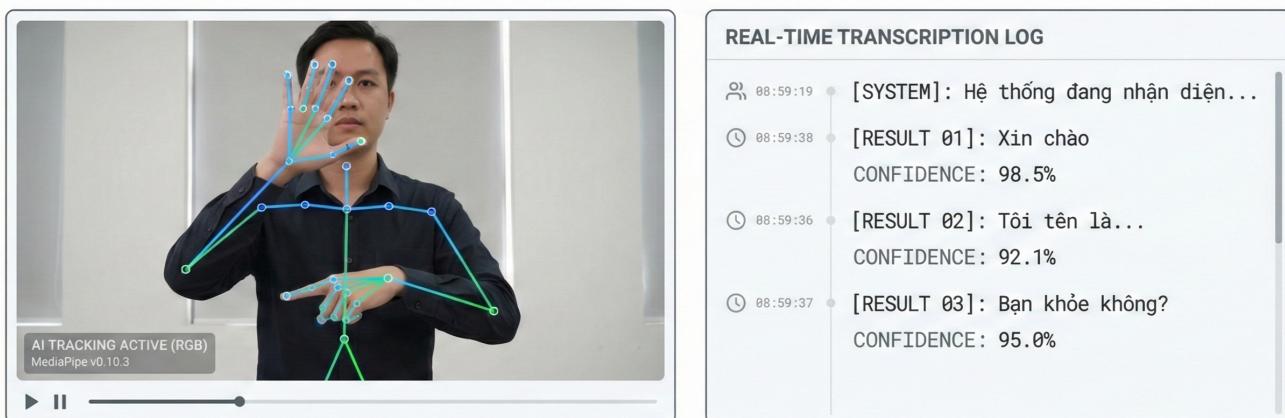
# Tutorial: Nhận diện Ngôn ngữ Ký hiệu (Olympic AI)

Nguyễn Phúc Thịnh  
Trần Hoàng DuyNguyễn Quốc Thái  
Đinh Quang Vinh

## I. Bối cảnh bài toán

Ngôn ngữ ký hiệu (Sign Language) là phương tiện giao tiếp tự nhiên và quan trọng nhất của cộng đồng người khiếm thính. Tuy nhiên, đa số người nghe nói bình thường không được trang bị kiến thức về ngôn ngữ này, tạo nên một rào cản giao tiếp lớn, khiến người khiếm thính gặp nhiều khó khăn trong việc tiếp cận thông tin, giáo dục, y tế và các dịch vụ công. Trong nhiều bối cảnh, việc chuyển đổi ngôn ngữ ký hiệu sang văn bản hoặc giọng nói vẫn chủ yếu dựa vào đội ngũ phiên dịch viên, vốn khan hiếm và tốn kém.

Những năm gần đây, sự phát triển mạnh mẽ của Thị giác máy tính (Computer Vision) và Học sâu (Deep Learning) đã mở ra các hướng tiếp cận mới nhằm thu hẹp khoảng cách này. Các kỹ thuật như ước lượng dáng điệu (Pose Estimation) cho phép máy tính trích xuất chính xác các điểm đặc trưng trên cơ thể, bàn tay và khuôn mặt từ dữ liệu video mà không cần đến các thiết bị đeo hỗ trợ, từ đó tạo tiền đề cho việc xây dựng các hệ thống tự động nhận diện và phiên dịch ngôn ngữ ký hiệu.



Hình 1: Minh họa bài toán nhận diện ngôn ngữ ký hiệu tiếng Việt.

Trong bối cảnh đó, cuộc thi đặt ra cho thí sinh bài toán xây dựng một hệ thống Nhận diện Ngôn ngữ Ký hiệu (Sign Language Recognition - SLR), tận dụng các tiến bộ trên. Cụ thể, nhiệm vụ là phát triển mô hình có thể nhận đầu vào là các video ký hiệu và phân loại chúng thành những nhãn từ hoặc cụm từ tiếng Việt tương ứng.

# Mục lục

I.	Bối cảnh bài toán . . . . .	1
II.	Yêu cầu và thể lệ cuộc thi . . . . .	3
III.	Phân tích dữ liệu . . . . .	5
III.1.	Phân bổ các tập dữ liệu train và test . . . . .	5
III.2.	Tổng quan về dữ liệu . . . . .	6
III.3.	Phân bổ nhãn và mất cân bằng lớp . . . . .	8
IV.	Phân tích baseline . . . . .	10
IV.1.	Xử lý dữ liệu đầu vào . . . . .	10
IV.2.	Kiến trúc mô hình CRNN . . . . .	11
IV.3.	Chiến lược huấn luyện . . . . .	13
V.	Ứng dụng các cải tiến cho dữ liệu . . . . .	13
V.1.	Chuẩn hoá chuỗi video . . . . .	14
V.2.	Augmentation video nhất quán . . . . .	15
V.3.	Lấy mẫu cân bằng cho tập huấn luyện . . . . .	18
VI.	Ứng dụng các cải tiến cho mô hình . . . . .	20
VI.1.	Backbone: ConvNeXt-Tiny . . . . .	21
VI.2.	Mô hình hóa thời gian: Transformer Encoder . . . . .	21
VI.3.	Tổng hợp thông tin: Attention Pooling . . . . .	22
VII.	Cải tiến chiến lược huấn luyện . . . . .	23
VIII.	Các phương pháp mở rộng thêm . . . . .	25
VIII.1.	Phân tích thêm dữ liệu . . . . .	25
VIII.2.	Đổi backbone sang Vision Transformer (ViT) . . . . .	27
VIII.3.	Thay đổi hàm Loss . . . . .	28
	Phụ lục . . . . .	32

## II. Yêu cầu và thể lệ cuộc thi

Các tài nguyên cho bài toán được cung cấp tại [Google Drive](#).

### Mô tả bài toán

Cho một tập dữ liệu gồm các video biểu diễn các ký hiệu tay tương ứng với các từ hoặc cụm từ trong ngôn ngữ ký hiệu Việt Nam. Nhiệm vụ của bạn là xây dựng một mô hình phân loại hình ảnh để nhận diện ký hiệu tương ứng.

- **Đầu vào:** Mô hình nhận đầu vào là  $X = \{I_1, I_2, \dots, I_n\}$ , với mỗi  $I_i$  là một hình ảnh tay đang biểu diễn ký hiệu.
- **Mục tiêu:** Dự đoán nhãn  $y_i$  tương ứng với từ hoặc cụm từ mà tay đó biểu diễn:

$$\hat{y}_i = f_\theta(I_i),$$

trong đó  $f_\theta$  là mô hình học sâu được huấn luyện.

### Dữ liệu và mô hình baseline

- **Dữ liệu huấn luyện:** Bộ video ngắn chứa các ký hiệu tay, được gán nhãn tương ứng (ví dụ: các từ như “An ủi”, “Xin lỗi”, “Cảm ơn”, ...).
- **Dữ liệu kiểm thử:** Tập video chưa biết nhãn, dùng để đánh giá mô hình.
- **Mô hình cơ sở:** Thí sinh được cung cấp sẵn một mô hình cơ sở đã được huấn luyện sẵn trên ImageNet CRNN – một kiến trúc mạng nơ-ron được thiết kế cho nhận dạng hành động trong video.
- **Mô hình khác:** Thí sinh cũng có thể sử dụng các mô hình *pretrained* được huấn luyện trên ImageNet.
- **Tính đa dạng:** Các dữ liệu được chụp trong nhiều điều kiện ánh sáng, góc quay, những người thực hiện khác nhau và màu da khác nhau để đảm bảo tính đa dạng.

### Yêu cầu

- Huấn luyện mô hình phân loại hình ảnh/chuỗi hình ảnh nhận diện chính xác ký hiệu tay.
- Kết quả đầu ra là nhãn của từ hoặc cụm từ tương ứng với video đầu vào.
- Thí sinh được khuyến khích đề xuất thêm bước xử lý đầu vào (*preprocessing*) hoặc cải tiến mô hình để tăng độ chính xác.

## Tiêu chí đánh giá

**Tổng thể (Macro-F1):** Là giá trị trung bình của chỉ số  $F_1$  trên tất cả các lớp, giúp đánh giá cân bằng giữa Precision và Recall.

Với mỗi lớp ( $i$ ), các chỉ số được tính như sau:

$$\text{Precision}_i = \frac{TP_i}{TP_i + FP_i}.$$

$$\text{Recall}_i = \frac{TP_i}{TP_i + FN_i}.$$

$$F_{1,i} = 2 \times \frac{\text{Precision}_i \times \text{Recall}_i}{\text{Precision}_i + \text{Recall}_i}.$$

Trong đó:

- **TP (True Positive):** số mẫu thuộc lớp dương (positive) được dự đoán đúng là positive.
- **TN (True Negative):** số mẫu thuộc lớp âm (negative) được dự đoán đúng là negative.
- **FP (False Positive):** số mẫu thuộc lớp âm nhưng bị mô hình dự đoán nhầm là positive.
- **FN (False Negative):** số mẫu thuộc lớp dương nhưng bị mô hình dự đoán nhầm là negative.

**Chỉ số Macro-F1 được định nghĩa:**

$$\text{Macro-}F_1 = \frac{1}{N} \sum_{i=1}^N F_{1,i},$$

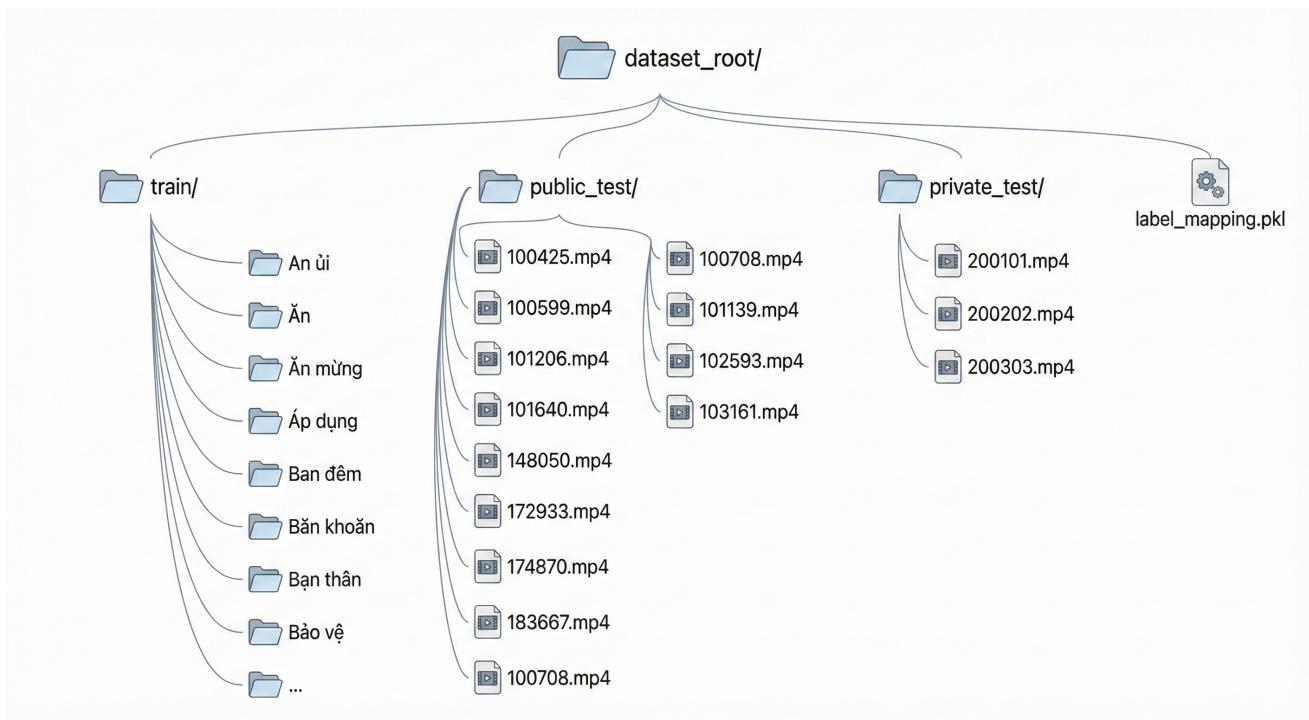
trong đó  $N$  là số lớp.

## III. Phân tích dữ liệu

Trước khi thiết kế mô hình, bước quan trọng đầu tiên là hiểu rõ dữ liệu mà mô hình sẽ học: loại video này là gì, độ dài ra sao, được gán nhãn như thế nào và phân bố lớp có cân bằng hay không. Phần này trình bày một số phân tích khám phá dữ liệu (EDA) ở cả khía cạnh định tính và định lượng, làm nền tảng cho các quyết định thiết kế mô hình và chiến lược xử lý dữ liệu ở các phần sau.

### III.1. Phân bổ các tập dữ liệu train và test

Trong bộ dữ liệu mà ban tổ chức cung cấp, thư mục gốc bao gồm ba thư mục chứa video hành động ngôn ngữ ký hiệu và một tệp ánh xạ nhãn, được minh họa ở Hình 2:



Hình 2: Cấu trúc tổ chức bộ dữ liệu được cung cấp bởi ban tổ chức.

- **train/**: chứa toàn bộ dữ liệu có gán nhãn, dùng để huấn luyện và xây dựng mô hình.
- **public\_test/**: chứa dữ liệu không có nhãn, dùng để sinh dự đoán gửi lên hệ thống đánh giá công khai.
- **private\_test/**: chứa dữ liệu không có nhãn dùng cho đánh giá ẩn (xếp hạng cuối cùng).
- **label\_mapping.pkl**: tệp pickle lưu ánh xạ giữa tên lớp (chuỗi tiếng Việt) và chỉ số nguyên tương ứng được mô hình sử dụng trong quá trình huấn luyện và suy luận.

Trong thư mục **train/**, dữ liệu được tổ chức theo dạng mỗi lớp một thư mục con như minh họa ở Hình 2. Mỗi thư mục con mang tên một từ hoặc cụm từ trong ngôn ngữ ký hiệu tiếng Việt (ví

dụ: “Ăn”, “Ăn mừng”, “An ủi”, ...) và chứa các video thể hiện ký hiệu tương ứng. Tổng cộng có **100 lớp** khác nhau trong tập train.

Về quy mô từng tập dữ liệu:

- Tập **train** gồm 3875 video đã được gán nhãn, phân bố vào 100 thư mục lớp như mô tả ở trên.
- Tập **public\_test** gồm **1630** video không có nhãn đi kèm.
- Tập **private\_test** gồm **2859** video không có nhãn đi kèm.

Như vậy, toàn bộ pipeline xử lý dữ liệu sẽ sử dụng `label_mapping.pkl` để chuyển đổi tên thư mục (tên lớp) thành chỉ số nhãn số nguyên, đọc video từ các thư mục con trong `train/` để huấn luyện và đánh giá mô hình, đồng thời áp dụng cùng quy trình tiền xử lý cho các video trong `public_test/` và `private_test/` khi sinh dự đoán.

### III.2. Tổng quan về dữ liệu

Tập dữ liệu bao gồm các video RGB ngắn, mỗi video tương ứng với một từ hoặc cụm từ trong ngôn ngữ ký hiệu tiếng Việt. Để có cái nhìn trực quan, ta trích xuất một số frame đại diện từ nhiều lớp khác nhau như minh họa trong Hình 3.



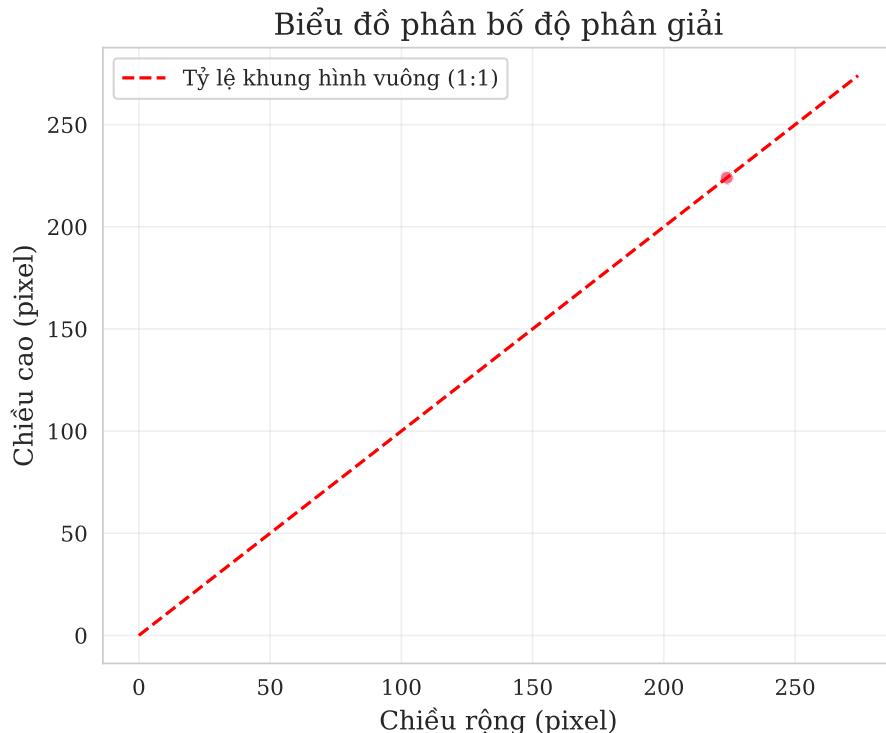
Hình 3: Một số frame trích xuất từ các video thuộc nhiều lớp khác nhau trong tập huấn luyện.

Từ các mẫu frame trực quan, ta rút ra một số đặc điểm chung của dữ liệu:

- **Góc quay và bối cảnh:** Hầu hết video được quay ở góc chính diện, khung hình bán thân (upper body) giúp tập trung vào chuyển động tay và biểu cảm khuôn mặt - hai yếu tố cốt lõi của ngôn ngữ ký hiệu.
- **Tính đa dạng:** Cùng một ký hiệu được thực hiện bởi nhiều người với giới tính, vóc dáng, màu da, trang phục khác nhau. Điều này giúp mô hình học được đặc trưng bất biến thay vì học theo một cá nhân cụ thể.
- **Bối cảnh và ánh sáng:** Đa số video quay trong nhà với phông nền khá đơn giản, song điều kiện ánh sáng vẫn có độ biến thiên nhất định (video sáng rõ, video hơi tối hoặc ám màu). Mô hình do đó cần có khả năng học được sự thay đổi về ánh sáng và màu sắc.

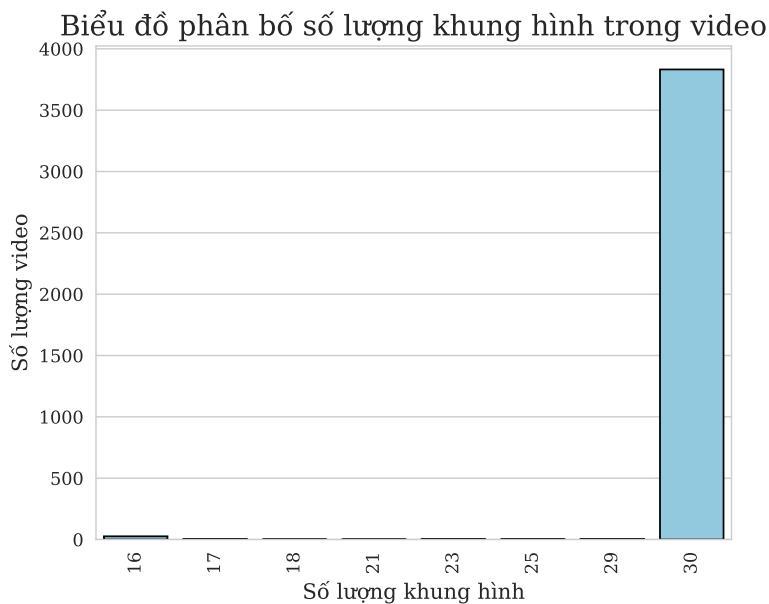
### Đặc điểm không gian và thời gian của video

Khác với bài toán ảnh tĩnh, dữ liệu video mang thông tin ở cả hai chiều: không gian (kích thước khung hình) và thời gian (số lượng frame, fps). Việc chuẩn hóa hai yếu tố này là bước quan trọng trước khi đưa vào mô hình.



Hình 4: Biểu đồ phân bố độ phân giải của các video trong tập train.

Hình 4 cho thấy độ phân giải của các video toàn bộ tập trung quanh tại một giá trị kích thước chuẩn  $224 \times 224$ , phù hợp với model có backbone như ConvNeXt, ... được pretrained trên ImageNet ít gây méo hình và đơn giản hóa bước tiền xử lý.



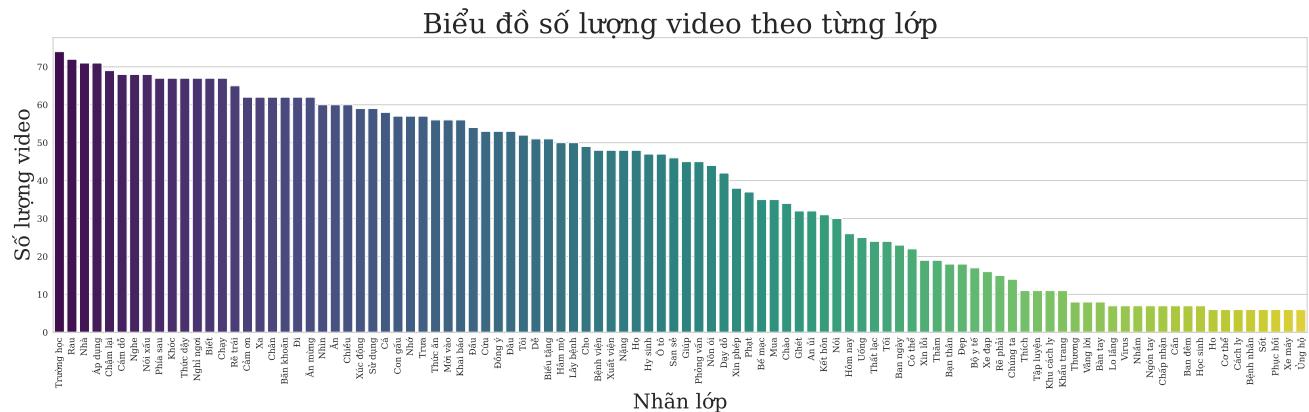
Hình 5: Biểu đồ phân bố số lượng khung hình trong các video huấn luyện.

Quan sát Hình 5, có thể thấy phân bố số lượng khung hình giữa các video gần như “dồn cục” vào một giá trị duy nhất. Cột ở vị trí khoảng 30 frames chiếm áp đảo, trong khi các giá trị còn lại (từ 16-29 frames) chỉ xuất hiện lác đác với tần suất rất nhỏ, gần như khó nhìn thấy trên biểu đồ. Điều này cho thấy chiều dài video trong tập huấn luyện được kiểm soát khá chặt chẽ: phần lớn các mẫu đều có thời lượng tương đương nhau, chỉ tồn tại một số ít ngoại lệ ngắn hơn.

Nói cách khác, về mặt thời gian, bộ dữ liệu huấn luyện có độ biến thiên rất thấp: không có các video quá dài hoặc quá ngắn, và số khung hình của đa số mẫu nằm tập trung quanh một mốc chuẩn duy nhất. Đây là một đặc điểm thuận lợi khi xây dựng pipeline xử lý vì ta không phải đổi mới với sự phân tán mạnh về độ dài chuỗi.

### III.3. Phân bố nhãn và mất cân bằng lớp

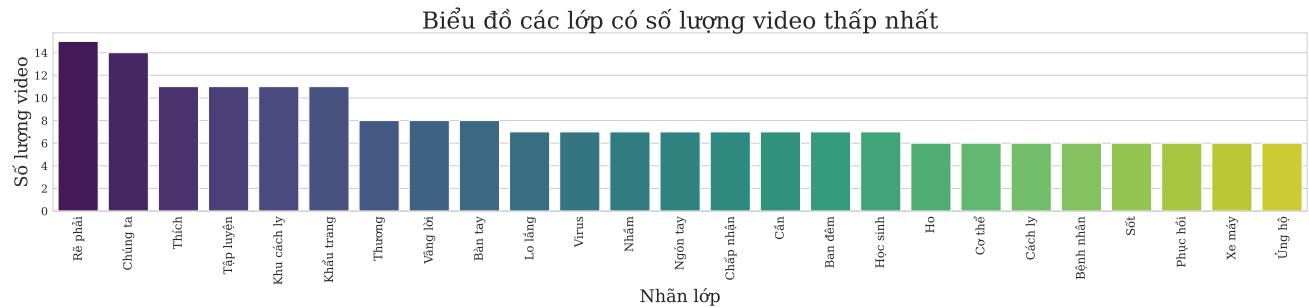
Sau khi hiểu cấu trúc và chất lượng của video, bước tiếp theo là xem cách các video được phân bổ theo nhãn. Đây là yếu tố ảnh hưởng trực tiếp tới lựa chọn hàm mất mát và chiến lược lấy mẫu trong quá trình huấn luyện.



Hình 6: Thống kê tổng số video theo từng lớp hành động trong tập huấn luyện.

Hình 6 cho thấy số lượng video giữa các lớp chênh lệch rất lớn. Một vài lớp ở bên trái biểu đồ có tới gần 100 video, trong khi nhiều lớp ở phía bên phải chỉ có vài chục, thậm chí dưới 10 video. Đường cong giảm dần nhanh và kéo dài cho tới cuối trực nhãn, thể hiện rõ dạng phân phối “đuôi dài”.

Để nhìn rõ hơn phần đuôi này, ta phóng to các lớp có ít mẫu nhất như trong Hình 7.

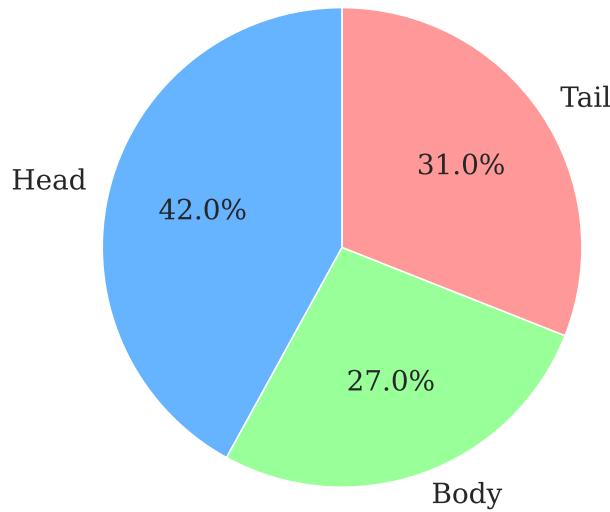


Hình 7: Các lớp có số lượng video thấp nhất trong tập huấn luyện.

Có thể thấy không ít lớp chỉ sở hữu khoảng 4-8 video. Với số lượng mẫu ít như vậy, nếu huấn luyện theo cách thông thường, mô hình rất dễ “bỏ quên” những lớp này và tập trung vào các lớp xuất hiện nhiều hơn.

Để định lượng mức độ mất cân bằng, ta chia các lớp thành ba nhóm dựa trên số lượng video mỗi lớp: nhóm Head gồm các lớp có từ 50 video trở lên, nhóm Body gồm các lớp có từ 20-49 video/lớp, và nhóm Tail gồm các lớp hiếm với dưới 20 video/lớp. Sau đó, ta so tỉ lệ từng nhóm như Hình 8.

Biểu đồ tỷ lệ mất cân bằng lớp



Hình 8: Tỷ lệ giữa nhóm lớp Head (nhiều mẫu), Body và Tail (ít mẫu).

Kết quả cho thấy nhóm Head chỉ chiếm 42% tổng số lớp nhưng nắm phần lớn số lượng video, nhóm Body chiếm 27%, còn nhóm Tail chiếm tới 31% số lớp nhưng mỗi lớp lại có rất ít mẫu. Nói cách khác, gần một phần ba số nhãn trong bài toán rơi vào vùng “hiếm dữ liệu”.

Đây là một thách thức quan trọng đối với việc tối ưu mô hình, đặc biệt khi thước đo mục tiêu là Macro-F1 - vốn coi trọng đều tất cả các lớp. Nếu không có biện pháp xử lý phù hợp, mô hình sẽ thiên lệch về các lớp Head, đạt độ chính xác cao trên những lớp phổ biến nhưng lại dự đoán rất kém đối với những lớp Tail ít dữ liệu.

## IV. Phân tích baseline

Sau khi đã hiểu tương đối rõ đặc điểm của dữ liệu qua phần phân tích dữ liệu, bước tiếp theo là phân tích mô hình baseline được cung cấp trong đề bài trước khi đề xuất các hướng cải thiện. Hiểu được baseline một cách vững chắc là bước đầu tiên để đánh giá hiệu quả của các phương pháp cải tiến sau này.

Baseline hiện tại sử dụng kiến trúc CRNN (Convolutional Recurrent Neural Network), kết hợp sức mạnh trích xuất đặc trưng hình ảnh của CNN và khả năng mô hình hóa chuỗi thời gian của RNN (cụ thể hơn là mạng LSTM).

### IV.1. Xử lý dữ liệu đầu vào

Dữ liệu video như được nói ở phần EDA trên có đặc thù là có số lượng frame không cố định (từ 16 - 30 frames). Để đưa vào mô hình deep learning, ta cần chuẩn hóa chúng về một định dạng thống nhất.

Quy trình xử lý dữ liệu trong baseline được thực hiện qua các bước sau:

1. Đọc video: Sử dụng thư viện OpenCV để đọc từng frame hình ảnh từ file video.
2. Lấy mẫu: Vì video có độ dài khác nhau, ta cần cố định số lượng frame đầu vào (trong baseline là  $T = 16$ ).
  - Nếu video dài hơn 16 frame: Lấy mẫu đều (uniform sampling) để giữ lại thông tin toàn cục.
  - Nếu video ngắn hơn 16 frame: Lặp lại frame cuối cùng (padding) cho đủ độ dài.
3. Chuẩn hóa: Các frame được đưa về phân phối chuẩn của ImageNet (mean và std) để tận dụng tốt các trọng số pretrained. Các bạn có thể coi cách các thư viện và paper mẫu chuẩn hóa tập dataset ImageNet ở [đây](#).

Dưới đây là đoạn code minh họa việc lấy mẫu frame trong lớp “VideoDataset”:

**Hàm lấy mẫu frame**

```

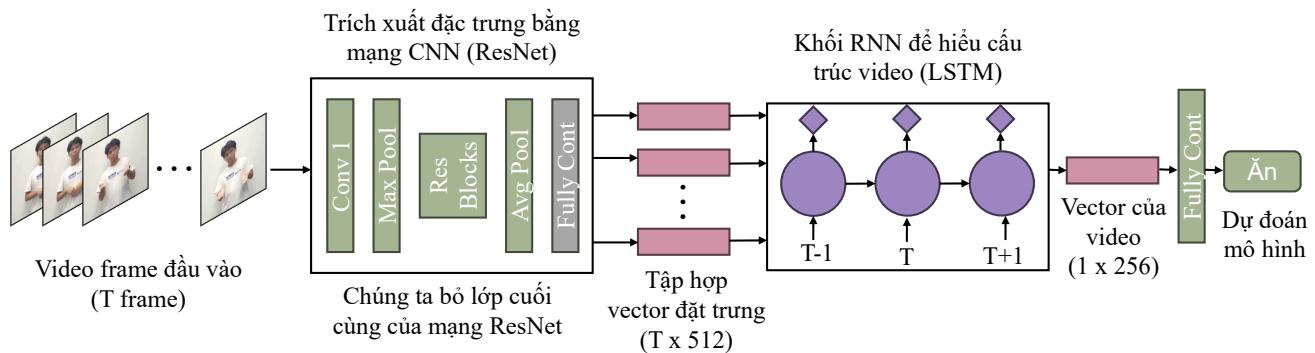
1 def _downsample_frames(self, frames):
2     num_frames = frames.shape[0]
3     if num_frames == self.target_frames:
4         return frames
5     elif num_frames < self.target_frames:
6         # Padding nếu thiếu frame
7         pad = self.target_frames - num_frames
8         return torch.cat([frames, frames[-1:].repeat(pad, 1, 1, 1)], dim=0)
9     else:
10        # Lấy mẫu đều nếu thừa frame
11        idx = torch.linspace(0, num_frames - 1, self.target_frames).long()
12        return frames[idx]

```

Tuy nhiên, hàm `_downsample_frames` hiện tại đang xử lý các video ngắn (số frame  $< 16$ ) bằng cách lặp lại frame cuối cùng, Việc này tạo ra một chuỗi hình ảnh “tĩnh” (đóng băng) ở cuối video. Mạng LSTM có thể hiểu nhầm rằng đặc trưng của hành động bao gồm cả việc “đứng yên” này, gây nhiễu khi phân loại các hành động có tính chuyển động liên tục.

## IV.2. Kiến trúc mô hình CRNN

Một trong các mô hình đơn giản nhất để giải quyết bài toán này là mô hình CRNN (Convolutional Recurrent Neural Network). Mô hình được thiết kế theo dạng encoder-decoder với cách hiểu thời gian thông qua cơ chế của mạng LSTM.



Hình 9: Sơ đồ kiến trúc CRNN: ResNet18 trích xuất đặc trưng từng frame, sau đó LSTM tổng hợp thông tin theo thời gian.

Cấu trúc chi tiết bao gồm:

1. Spatial Encoder (CNN): Sử dụng “ResNet18” đã được huấn luyện trước trên ImageNet 1K.
  - Ta loại bỏ lớp Fully Connected cuối cùng để lấy vector đặc trưng.
  - Đầu ra của mỗi frame là một vector có kích thước 512.
2. Temporal Encoder (RNN): Sử dụng mạng “LSTM”.
  - Đầu vào: Chuỗi các vector đặc trưng từ CNN (kích thước  $Batch \times 16 \times 512$ ).
  - LSTM giúp mô hình học được sự thay đổi của hành động theo thời gian (ví dụ: giơ tay lên rồi hạ tay xuống).
  - Hidden size được thiết lập là 256.
3. Classifier: Một lớp tuyến tính (Linear layer) đơn giản để ánh xạ từ trạng thái ẩn cuối cùng của LSTM sang 100 lớp hành động cần phân loại.

### Định nghĩa mô hình CRNN

```

1 class CRNN(nn.Module):
2     def __init__(self, num_classes=100, hidden_size=256):
3         super(CRNN, self).__init__()
4         # Sử dụng ResNet18 làm backbone
5         resnet = models.resnet18(weights=models.ResNet18_Weights.IMGNET1K_V1)
6         self.cnn = nn.Sequential(*list(resnet.children())[:-2])
7
8         # LSTM để xử lý chuỗi thời gian
9         self.rnn = nn.LSTM(512, hidden_size, batch_first=True, dropout=0.3)
10        self.fc = nn.Linear(hidden_size, num_classes)
11
12    def forward(self, x):
13        B, T, C, H, W = x.size()
14        # Gộp Batch và Time để đưa qua CNN
15        x = x.view(B * T, C, H, W)
16        features = self.cnn(x)
17

```

```

18     # Reshape lại để đưa vào RNN
19     seq = features.view(B, T, 512)
20     rnn_out, _ = self.rnn(seq)
21
22     # Lấy output tại bước thời gian cuối cùng
23     return self.fc(rnn_out[:, -1, :])

```

## IV.3. Chiến lược huấn luyện

### Hàm huấn luyện model

```

1 def train_model(model, train_loader, val_loader,
2                 num_epochs=10, lr=5e-4, device='cuda', save_path='best_model.pth'):
3     model = model.to(device)
4     criterion = nn.CrossEntropyLoss()
5     optimizer = AdamW(model.parameters(), lr=lr, weight_decay=1e-4)
6     scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min',
7                  factor=0.5, patience=3)
8     return model

```

Quá trình huấn luyện sử dụng hàm mất mát “CrossEntropyLoss” tiêu chuẩn cho bài toán phân loại đa lớp. Một số điểm đáng chú ý trong cấu hình huấn luyện như sau:

- Optimizer: Sử dụng “AdamW” với learning rate khởi tạo là  $1e-4$  và weight decay  $1e-4$  để tránh overfitting.
- Scheduler: Sử dụng “ReduceLROnPlateau”. Nếu loss trên tập validation không giảm sau 3 epoch, learning rate sẽ giảm đi một nửa. Điều này giúp mô hình hội tụ tốt hơn ở các epoch sau trong việc huấn luyện mô hình.
- Metric: Đánh giá dựa trên F1-score (Macro average) để đảm bảo sự cân bằng giữa độ chính xác (Precision) và độ phủ (Recall) trên tất cả các lớp hành động.

Hàm CrossEntropyLoss mặc định đối xử bình đẳng với tất cả các lớp. Trong bối cảnh F1-score (Macro average) là trọng số đánh giá, việc dự đoán sai một lớp hiếm (minority class) sẽ bị phạt nặng tương đương với lớp phổ biến. Mô hình hiện tại có xu hướng ưu tiên các lớp đa số để tối ưu hóa Accuracy tổng thể, làm giảm F1-score.

## V. Ứng dụng các cải tiến cho dữ liệu

Từ các kết quả EDA ở phần phân tích dữ liệu ta có thể nhận thấy dữ liệu video ngôn ngữ ký hiệu có ba đặc điểm nổi bật như sau:

- **Chiều thời gian:** Số khung hình của đa số video gần như tập trung tại 30 frames, chỉ có một số ít ngoại lệ ngắn hơn. Nếu đưa toàn bộ 30 frames vào mô hình vừa dư thừa thông tin, vừa tốn tài nguyên, trong khi các video ngắn lại không đủ độ dài.

- **Chiều không gian:** Video có sự biến thiên về vị trí người biểu diễn ngôn ngữ ký hiệu trong khung hình, điều kiện ánh sáng, màu sắc, tốc độ biểu diễn,... như đã thấy trong các ví dụ Hình 3. Nếu không xử lý tốt, mô hình dễ overfit vào những điều kiện quay cụ thể.
- **Phân bố nhãn:** Biểu đồ phân bố nhãn cho thấy dữ liệu mang dạng phân phối đuôi dài, với nhóm Head có rất nhiều video, trong khi nhóm Tail chỉ có vài mẫu mỗi lớp. Điều này gây thiên lệch khi huấn luyện, đặc biệt khi tiêu chí chấm điểm là Macro-F1 (coi trọng đều các lớp).

Những đặc điểm này dẫn tới ba vấn đề: chuỗi thời gian không đồng nhất, mô hình dễ overfit vào điều kiện quay cụ thể, và lớp hiếm bị “bỏ quên” khi huấn luyện. Dựa trên đó, ta thiết kế lại data pipeline với ba nhóm cải tiến: (i) chuẩn hoá chuỗi video, (ii) tăng cường dữ liệu nhất quán theo thời gian, và (iii) lấy mẫu cân bằng cho tập huấn luyện.

## V.1. Chuẩn hoá chuỗi video

Từ phân tích chiều thời gian và độ phân giải, ta quyết định: (1) cố định số frame đầu vào là  $T = 16$  để vừa bao phủ toàn bộ diễn biến cử chỉ, vừa giảm dư thừa so với 30 frames gốc; (2) đưa mọi khung hình về kích thước chuẩn  $224 \times 224$  và chuẩn hoá theo mean/std của ImageNet để tương thích với backbone pretrained trên ImageNet.

Lớp `VideoDataset` chịu trách nhiệm: (i) duyệt cấu trúc thư mục `train/`, (ii) ánh xạ tên lớp sang chỉ số bằng `label_mapping.pkl`, (iii) đọc video, áp dụng augmentation (nếu train), (iv) downsample và chuẩn hoá.

Khởi tạo lớp VideoDataset

```

1 class VideoDataset(Dataset):
2
3     # ... Xem thêm tại phần Phụ lục
4
5     def __init__(self, root_dir, target_frames=16, transform=None):
6         self.root_dir = root_dir
7         self.target_frames = target_frames
8         self.transform = transform
9
10        self.label_mapping = self._load_label_mapping()
11
12        self._downsample_frames()
13
14        self._normalize_frames()
15
16        self._check_frames_shape()
17
18        self._permute_and_to_tensor()
19
20        self._float_and_interpolate()
21
22        self._permute_and_to_uint8()
23
24
25    def _load_label_mapping(self):
26        with open(os.path.join(self.root_dir, 'label_mapping.pkl'), 'rb') as f:
27            return pickle.load(f)
28
29    def _downsample_frames(self, frames):
30        total = frames.shape[0]
31
32        if total >= self.target_frames:
33            indices = torch.linspace(0, total - 1, self.target_frames).long()
34        else:
35            indices = torch.arange(total)
36            pad = self.target_frames - total
37            indices = torch.cat([indices, indices[-1].repeat(pad)])
38
39        frames = frames[indices]
40
41        if frames.shape[1] != 224 or frames.shape[2] != 224:
42            frames = frames.permute(0, 3, 1, 2).float()
43            frames = F.interpolate(frames, size=(224, 224),
44                                   mode='bilinear', align_corners=False)
45            frames = frames.permute(0, 2, 3, 1).to(torch.uint8)
46
47        return frames
48
49    def _normalize_frames(self, frames):
50        frames = frames / 255.0
51
52        return frames
53
54    def _check_frames_shape(self, frames):
55        if frames.shape[1] != 224 or frames.shape[2] != 224:
56            raise ValueError("Frames must have shape (batch_size, 3, 224, 224).")
57
58    def _permute_and_to_tensor(self, frames):
59        frames = frames.permute(0, 3, 1, 2).float()
60
61        return frames
62
63    def _float_and_interpolate(self, frames):
64        frames = frames.float()
65
66        frames = F.interpolate(frames, size=(224, 224),
67                               mode='bilinear', align_corners=False)
68
69        return frames
70
71    def _permute_and_to_uint8(self, frames):
72        frames = frames.permute(0, 2, 3, 1).to(torch.uint8)
73
74        return frames
75
76    def __len__(self):
77        return len(self._frames)
78
79    def __getitem__(self, index):
80        frame = self._frames[index]
81
82        if self.transform is not None:
83            frame = self.transform(frame)
84
85        return frame
86
87    def _check_root_dir(self, root_dir):
88        if not os.path.exists(root_dir):
89            raise ValueError(f"Root directory '{root_dir}' does not exist.")
90
91        if not os.path.isdir(root_dir):
92            raise ValueError(f"Root directory '{root_dir}' is not a directory.")
```

```

25     """Normalize về ImageNet mean/std"""
26     frames = frames.float() / 255.0
27     frames = frames.permute(0, 3, 1, 2) # (T, H, W, C) -> (T, C, H, W)
28
29     mean = torch.tensor(self.mean).view(1, 3, 1, 1)
30     std = torch.tensor(self.std).view(1, 3, 1, 1)
31     frames = (frames - mean) / std
32
33     return frames

```

Ở đây, hàm `_downsample_frames` tạo ra các chỉ số cách đều trên trục thời gian, giúp “nén” toàn bộ diễn biến hành động thành đúng  $T = 16$  frame đại diện nhưng vẫn giữ được thông tin từ đầu đến cuối video (uniform sampling).

```

1 indices = torch.linspace(0, total - 1, self.target\_frames).long()

```

Hàm `_normalize` nhằm chuẩn hóa phân theo phân phối của ImageNet, `_normalize` trước tiên đưa giá trị pixel về khoảng [0, 1], sau đó chuyển thứ tự chiều từ  $(T, H, W, C)$  sang  $(T, C, H, W)$  để phù hợp chuẩn PyTorch. Tiếp theo, từng kênh màu được chuẩn hoá bằng bộ `mean/std` của ImageNet:

$$\text{frames} \leftarrow \frac{\text{frames} - \text{mean}}{\text{std}}.$$

Nhờ vậy, phân phối pixel của dữ liệu ngôn ngữ ký hiệu được đưa về cùng không gian thống kê với tập ImageNet, giúp tận dụng hiệu quả hơn các trọng số pretrained của backbone pretrained trên ImageNet khi trích xuất đặc trưng từ từng frame.

## V.2. Augmentation video nhất quán

Từ phần tổng quan dữ liệu (Hình 3), ta thấy các video khác nhau đáng kể về tốc độ biểu diễn ký hiệu, vị trí người biểu diễn trong khung hình và điều kiện ánh sáng. Để mô hình học được các đặc trưng bền vững trước những thay đổi này, ta cần tăng cường dữ liệu. Tuy nhiên, nếu áp dụng augmentation độc lập trên từng frame, chuyển động trong video sẽ bị méo (rung, giật), khiến mô hình khó nắm bắt các đặc trưng.

Vì vậy, ta xây dựng lớp `VideoAugmentation` với nguyên tắc: các tham số biến đổi được sinh một lần và áp dụng đồng nhất cho toàn bộ chuỗi frame, bảo toàn tính liên tục theo thời gian.

### Khởi tạo lớp `VideoAugmentation` với các phương pháp `Speed`, `Crop`, `Color jitter`

```

1 class VideoAugmentation:
2     def __init__(self,
3                  crop_scale=(0.85, 1.0),
4                  brightness=0.2,
5                  contrast=0.2,
6                  saturation=0.2,
7                  speed_range=(0.9, 1.1)):

```

```

8     self.crop_scale = crop_scale
9     self.brightness = brightness
10    self.contrast = contrast
11    self.saturation = saturation
12    self.speed_range = speed_range
13
14
15    # ... Xem thêm tại phần Phụ lục
16
17    def _speed_augment(self, frames):
18        """Thay đổi tốc độ video bằng cách resample frames"""
19        T = frames.shape[0]
20        speed = random.uniform(self.speed_range[0], self.speed_range[1])
21
22        new_T = int(T / speed)
23        if new_T < 4:
24            new_T = 4
25        if new_T == T:
26            return frames
27
28        # Resample frames
29        indices = torch.linspace(0, T - 1, new_T).long()
30        indices = torch.clamp(indices, 0, T - 1)
31        frames = frames[indices]
32
33    return frames

```

Hàm `_speed_augment` điều chỉnh tốc độ “ảo” của video bằng cách thay đổi số lượng frame hưu hiệu thông qua tham số `speed` được rút ngẫu nhiên trong khoảng `speed_range`. Khi `speed < 1`, số frame đầu ra tăng lên, mô phỏng việc người biểu diễn ký chậm hơn; khi `speed > 1`, số frame giảm xuống, tương ứng một động tác được thực hiện nhanh hơn. Việc nội suy chỉ số frame bằng:

#### `_speed_augment:`

```
1 indices = torch.linspace(0, T - 1, new_T)
```

giúp giữ nguyên thứ tự thời gian nhưng nén/giãn trực thời gian một cách mượt mà, phản ánh đúng thực tế rằng cùng một ký hiệu có thể được thực hiện với tốc độ rất khác nhau giữa các video.

#### Khởi tạo lớp VideoAugmentation với các phương pháp Speed, Crop, Color jitter

```

1 class VideoAugmentation:
2     def __init__(self,
3                  crop_scale=(0.85, 1.0),
4                  brightness=0.2,
5                  contrast=0.2,
6                  saturation=0.2,
7                  speed_range=(0.9, 1.1)):
8

```

```

9     self.crop_scale = crop_scale
10    self.brightness = brightness
11    self.contrast = contrast
12    self.saturation = saturation
13    self.speed_range = speed_range
14
15    # ... Xem thêm tại phần Phụ lục
16
17    def _random_resized_crop(self, frames):
18        """Random crop rồi resize về 224x224"""
19        T, H, W, C = frames.shape
20
21        # Random scale và position
22        scale = random.uniform(self.crop_scale[0], self.crop_scale[1])
23        crop_h, crop_w = int(H * scale), int(W * scale)
24        top = random.randint(0, H - crop_h)
25        left = random.randint(0, W - crop_w)
26
27        # Crop tất cả frames
28        frames = frames[:, top:top+crop_h, left:left+crop_w, :]
29        # Resize về 224x224
30        frames = frames.permute(0, 3, 1, 2).float()
31        frames = F.interpolate(frames, size=(224, 224),
32                               mode='bilinear', align_corners=False)
33        # (T, C, H, W) -> (T, H, W, C)
34        frames = frames.permute(0, 2, 3, 1)
35
36    return frames.to(torch.uint8)

```

Trong `_random_resized_crop`, hệ số `scale = random.uniform(crop_scale[0], crop_scale[1])` cùng vị trí `top, left` được lấy ngẫu nhiên một lần, sau đó áp dụng cho toàn bộ tensor `frames`. Cách làm này mô phỏng việc người biểu diễn đứng gần/xa hoặc lệch trái/phải so với camera, đồng thời đảm bảo mọi frame đều bị cắt ở cùng một vùng. Nhờ đó, quỹ đạo chuyển động của tay vẫn liên tục và mạch lạc theo thời gian, thay vì “nhảy” vị trí nếu ta crop từng frame một cách độc lập.

### Khởi tạo lớp VideoAugmentation với các phương pháp Speed, Crop, Color jitter

```

1 class VideoAugmentation:
2     def __init__(self,
3                  crop_scale=(0.85, 1.0),
4                  brightness=0.2,
5                  contrast=0.2,
6                  saturation=0.2,
7                  speed_range=(0.9, 1.1)):
8
9         self.crop_scale = crop_scale
10        self.brightness = brightness
11        self.contrast = contrast
12        self.saturation = saturation
13        self.speed_range = speed_range

```

```

14
15 # ... Xem thêm tại phần Phụ lục
16 def _color_jitter(self, frames):
17     """Color jitter"""
18     # Random parameters
19     brightness_factor = 1.0 + random.uniform(-self.brightness, self.brightness)
20     contrast_factor = 1.0 + random.uniform(-self.contrast, self.contrast)
21     saturation_factor = 1.0 + random.uniform(-self.saturation, self.saturation)
22
23     frames = frames.float()
24
25     # Brightness
26     frames = frames * brightness_factor
27     # Contrast
28     mean = frames.mean(dim=(1, 2), keepdim=True)
29     frames = (frames - mean) * contrast_factor + mean
30     # Saturation
31     gray = frames.mean(dim=-1, keepdim=True)
32     frames = gray + (frames - gray) * saturation_factor
33     # Clamp to valid range
34     frames = torch.clamp(frames, 0, 255)
35
36     return frames.to(torch.uint8)

```

Hàm `_color_jitter` tạo ra các biến thiên về độ sáng, độ tương phản và độ bão hoà thông qua ba hệ số `brightness_factor`, `contrast_factor`, `saturation_factor` được khởi tạo một lần cho toàn bộ video. Điều này cho phép mô phỏng sự khác biệt về điều kiện chiếu sáng, tông màu da hay độ sáng phòng mà ta quan sát được trong Hình 3, nhưng vẫn tránh được hiện tượng “nhấp nháy” khó chịu nếu mỗi frame bị chỉnh màu khác nhau. Kết quả là mô hình được khuyến khích tập trung vào hình thái cùi chỉ thay vì phụ thuộc vào một cấu hình ánh sáng cố định.

Khi ghép vào `VideoDataset`, lớp `VideoAugmentation` chỉ được kích hoạt khi `training=True`. Điều này đảm bảo rằng tập huấn luyện được làm giàu bằng các biến thiên thực tế (tốc độ, vị trí, ánh sáng), trong khi tập validation và test vẫn giữ nguyên phân phối gốc của dữ liệu, giúp việc đánh giá mô hình phản ánh đúng chất lượng tổng quát hoá trên dữ liệu thật.

### V.3. Lấy mẫu cân bằng cho tập huấn luyện

Cuối cùng, phân tích ở Hình 6-8 cho thấy gần một phần ba số lớp thuộc nhóm Tail với rất ít video. Nếu giữ nguyên sampling ngẫu nhiên, các lớp này hiếm khi xuất hiện trong mini-batch, trong khi Macro-F1 lại coi trọng đều mọi lớp.

Để khắc phục, ta sử dụng `WeightedRandomSampler` với trọng số tỉ lệ nghịch với tần suất xuất hiện của từng lớp, qua đó tăng xác suất chọn các mẫu thuộc lớp hiếm trong quá trình huấn luyện - nói cách khác, lớp nào có ít mẫu trong tập train thì sẽ được “ưu tiên bốc” thường xuyên hơn khi tạo batch.

### Balanced Sampler cho tập train

```

1 def create_balanced_sampler(dataset):
2     if hasattr(dataset, 'dataset'):
3         all_labels = [dataset.dataset.label_idx[i] for i in dataset.indices]
4     else:
5         all_labels = dataset.label_idx
6
7     class_counts = np.bincount(all_labels)
8     class_weights = 1.0 / class_counts
9     sample_weights = [class_weights[label] for label in all_labels]
10    sample_weights = torch.FloatTensor(sample_weights)
11
12    sampler = WeightedRandomSampler(
13        weights=sample_weights,
14        num_samples=len(sample_weights),
15        replacement=True
16    )
17
18    print(f"Balanced Sampler: class counts min={class_counts.min()},"
19          f" max={class_counts.max()}")
20    return sampler

```

Đoạn code trên hiện thực hoá đúng ý tưởng “càng hiếm càng được ưu tiên”:

- `all_labels` là danh sách nhãn (dạng chỉ số) của toàn bộ mẫu trong tập train hiện tại. Nếu `dataset` là một `Subset`, ta truy ngược về `dataset.dataset` và lấy theo `indices` để không làm sai lệch tần suất.
- `class_counts = np.bincount(all_labels)` cho ta số lượng mẫu của từng lớp  $c$ . Đây chính là thống kê mà ta đã hình dung thông qua các biểu đồ phân bố nhãn ở phần EDA.
- `class_weights = 1.0 / class_counts` chuyển tần suất thành trọng số nghịch đảo: lớp có ít mẫu (`class_counts` nhỏ) sẽ nhận trọng số lớn, lớp có nhiều mẫu thì ngược lại.
- `sample_weights` gán trọng số lớp tương ứng cho từng mẫu cụ thể; `WeightedRandomSampler` sau đó sử dụng các trọng số này để quyết định xác suất một mẫu được chọn vào batch.
- Tham số `replacement=True` cho phép cùng một mẫu được chọn nhiều lần trong một epoch nếu nó thuộc lớp rất hiếm, qua đó “bù” lại số lượng dữ liệu ít ỏi của lớp đó.

Sampler này được gắn trực tiếp vào `DataLoader` dành cho huấn luyện, thay thế cho cơ chế `shuffle=True` ngẫu nhiên thuần túy:

### Khởi tạo DataLoader với balanced sampler

```

1 # ... Xem thêm tại phần Phụ lục
2
3 train_size = int(0.8 * len(train_dataset_base))
4 val_size = len(train_dataset_base) - train_size
5
6 indices = list(range(len(train_dataset_base)))

```

```

7 np.random.seed(42)
8 np.random.shuffle(indices)
9 train_indices = indices[:train_size]
10 val_indices = indices[train_size:]
11
12 train_dataset = torch.utils.data.Subset(train_dataset_base, train_indices)
13 val_dataset = torch.utils.data.Subset(val_dataset_base, val_indices)
14
15 balanced_sampler = create_balanced_sampler(train_dataset)
16
17 train_loader = DataLoader(
18     train_dataset,
19     batch_size=16,
20     sampler=balanced_sampler,
21     collate_fn=collate_fn,
22     num_workers=4
23 )
24
25 val_loader = DataLoader(
26     val_dataset,
27     batch_size=16,
28     shuffle=False,
29     collate_fn=collate_fn,
30     num_workers=4
31 )

```

Nhờ cơ chế này, các lớp Tail tuy có rất ít video trong toàn bộ tập train nhưng lại xuất hiện với tần suất hợp lý hơn trong từng mini-batch. Điều đó giúp gradient từ các lớp hiếm không bị “chìm” so với các lớp Head/Body, và mô hình buộc phải học ranh giới quyết định cho cả những lớp ít dữ liệu. Cách lấy mẫu cân bằng này vì thế phù hợp với tiêu chí Macro-F1: mỗi lớp - dù phổ biến hay hiếm - đều có cơ hội đóng góp công bằng vào hiệu năng cuối cùng của mô hình.

## VI. Ứng dụng các cải tiến cho mô hình

Mặc dù mô hình CRNN (ResNet + LSTM) là một baseline phổ biến và đơn giản, nó tồn tại một số hạn chế:

- **Backbone ResNet18:** Ra đời từ 2015, khả năng trích xuất đặc trưng không mạnh mẽ bằng các kiến trúc hiện đại.
- **LSTM:** Xử lý tuần tự khiến việc huấn luyện chậm và gặp khó khăn trong việc nắm bắt các phụ thuộc xa (long-range dependencies) trong chuỗi video dài. Ngoài ra, việc chỉ lấy trạng thái ẩn cuối cùng (last hidden state) dễ bỏ sót thông tin nếu hành động quan trọng diễn ra ở giữa video.

Để khắc phục, chúng tôi đề xuất kiến trúc **ConvNeXt-Transformer** với ba cải tiến chính: (1) Backbone ConvNeXt-Tiny mạnh mẽ hơn, (2) Transformer Encoder thay thế LSTM để mô hình hóa thời gian, và (3) Cơ chế Attention Pooling để tổng hợp thông tin.

## VI.1. Backbone: ConvNeXt-Tiny

Thay vì ResNet18, chúng ta sẽ sử dụng **ConvNeXt-Tiny** để thay thế. Đây là kiến trúc CNN hiện đại được thiết kế lại dựa trên các nguyên lý của Vision Transformer (như patchify stem, depthwise convolution, layer normalization, v.v.).

### 💡 Tại sao chúng ta tận dụng backbone?

Một thách thức lớn trong bài toán này là số lượng dữ liệu huấn luyện khá hạn chế so với độ phức tạp của không gian đặc trưng video. Nếu huấn luyện một mạng nơ-ron sâu từ đầu (training from scratch), mô hình sẽ phải học hàng triệu tham số chỉ từ vài nghìn video mẫu, dẫn đến nguy cơ “overfitting” rất cao và khả năng tổng quát hóa kém.

Thay vào đó, chiến lược tối ưu là sử dụng “Transfer Learning” (Tạm dịch: học chuyển giao):

- **Tận dụng tri thức sẵn có:** Backbone ConvNeXt-Tiny đã được huấn luyện trên tập dữ liệu khổng lồ ImageNet (1.2 triệu ảnh), do đó nó đã học được các bộ lọc (filters) để nhận diện cạnh, hình dạng, và kết cấu vật thể một cách xuất sắc.
- **Tối thiểu hóa tham số cần học:** Thay vì học lại cách “nhìn” hình ảnh từ con số 0, mô hình chỉ cần tinh chỉnh (fine-tune) một lượng nhỏ trọng số để thích nghi với miền dữ liệu ngôn ngữ ký hiệu.
- **Hội tụ nhanh hơn:** Việc khởi tạo trọng số từ pre-trained model giúp loss giảm nhanh hơn và ổn định hơn so với khởi tạo ngẫu nhiên.

Do đó, kiến trúc ConvNeXt-Transformer ở đây được thiết kế để “đứng trên vai người khổng lồ”, tập trung năng lực tính toán vào việc học mối quan hệ thời gian thay vì tốn tài nguyên để trích xuất đặc trưng thị giác cơ bản mà các paper tốt nhất hiện có đang làm.

Về mặt hiệu năng thực tế, mặc dù mang tên “Tiny”, ConvNeXt vẫn sở hữu khả năng trích xuất đặc trưng vượt trội, đạt độ chính xác cao hơn cả ResNet50 trên tập ImageNet trong khi vẫn duy trì chi phí tính toán thấp hơn. Quan trọng hơn, đầu ra của backbone này cung cấp vector đặc trưng có kích thước  $D = 768$ , giàu thông tin ngữ nghĩa hơn rất nhiều so với kích thước 512 của ResNet18, tạo tiền đề vững chắc cho các lớp xử lý thời gian phía sau hoạt động hiệu quả.

## VI.2. Mô hình hóa thời gian: Transformer Encoder

Tương tự như cách chúng ta đổi qua mô hình ConvNeXt ở backbone, để mô hình được hiểu cấu trúc video tốt hơn ta cũng cần đổi cách mô hình hóa thời gian bằng **Transformer Encoder**. Cơ chế Self-Attention cho phép mô hình nhìn thấy toàn bộ chuỗi frame cùng lúc, từ đó học được mối quan hệ giữa các frame bất kể khoảng cách thời gian. Vì Transformer không có khái niệm về thứ tự, chúng ta phải bổ sung **Positional Encoding** vào chuỗi đặc trưng trước khi đưa vào encoder. Các làm được miêu tả như code sau:

### Positional Encoding và Transformer

```

1 class PositionalEncoding(nn.Module):
2     """Positional encoding cho temporal sequence"""
3     def __init__(self, d_model, max_len=64, dropout=0.1):

```

```

4     super().__init__()
5     # ... (Khởi tạo ma trận sin/cos) ...
6     self.register_buffer('pe', pe)
7
8     def forward(self, x):
9         # x: (B, T, d_model)
10        x = x + self.pe[:, :x.size(1), :]
11        return self.dropout(x)
12
13 # Trong ConvNeXtTransformer:
14 # 2. Positional Encoding
15 self.pos_encoder = PositionalEncoding(d_model=768, max_len=64, dropout=0.1)
16
17 # 3. Transformer Encoder
18 encoder_layer = nn.TransformerEncoderLayer(
19     d_model=768, nhead=8, dim_feedforward=768*4,
20     dropout=0.3, activation='gelu', batch_first=True, norm_first=True
21 )
22 self.transformer = nn.TransformerEncoder(encoder_layer, num_layers=2)

```

### VI.3. Tổng hợp thông tin: Attention Pooling

Baseline CRNN sử dụng trạng thái ẩn tại bước thời gian cuối cùng ( $h_T$ ) để phân loại. Điều này rất rủi ro đối với video ngôn ngữ ký hiệu, vì frame cuối cùng thường là lúc người biểu diễn đã hạ tay xuống (trạng thái nghỉ), không chứa thông tin đặc trưng của từ vựng.

Giải quyết việc này ta sẽ sử dụng cơ chế **Attention Pooling**. Mạng sẽ học một trọng số  $a_t$  cho mỗi frame  $t$ , thể hiện mức độ quan trọng của frame đó. Vector đại diện cho cả video  $V$  là tổng có trọng số của các frame:

$$a_t = \text{softmax}(W_2 \tanh(W_1 x_t))$$

$$V = \sum_{t=1}^T a_t x_t$$

#### Cơ chế Attention Pooling

```

1 class AttentionPooling(nn.Module):
2     def __init__(self, dim):
3         super().__init__()
4         self.attention = nn.Sequential(
5             nn.Linear(dim, dim // 4),
6             nn.Tanh(),
7             nn.Linear(dim // 4, 1)
8         )
9
10    def forward(self, x):
11        # x: (B, T, dim)
12        attn_weights = self.attention(x) # (B, T, 1)

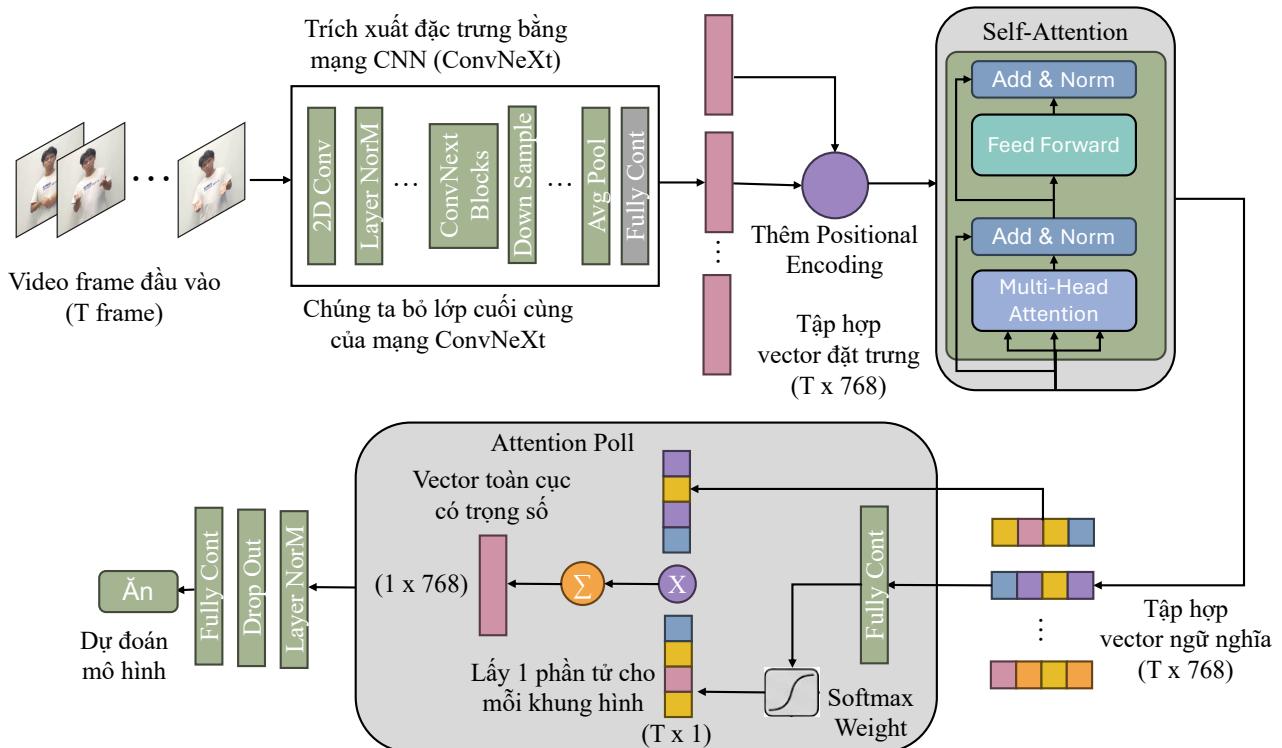
```

```

13     attn_weights = F.softmax(attn_weights, dim=1)
14     pooled = torch.sum(attn_weights * x, dim=1) # (B, dim)
15     return pooled

```

Nhờ cơ chế này, mô hình có thể tự động “tập trung” vào các frame giữa video nói cùi chỉ diễn ra rõ ràng nhất và “lò đi” các frame đầu/cuối không quan trọng. Hơn nữa, với cấu trúc Transformer, việc sử dụng Attention Pooling sẽ lưu trữ được những thông tin quan trọng nhất cho các layer sau.



Hình 10: Sơ đồ kiến trúc mạng cải tiến: ConvNeXt trích xuất đặc trưng từng frame, sau đó cơ chế Attention giúp các vector đặc trưng nói trên có thêm ngữ nghĩa. Tiếp theo, chúng ta Pooling toàn bộ chuỗi thời gian thành 1 vector duy nhất và dùng FC để thực hiện phân loại.

## VII. Cải tiến chiến lược huấn luyện

Bên cạnh kiến trúc, chiến lược huấn luyện cũng được tinh chỉnh để phù hợp với dữ liệu và mô hình mới. Mà ở đây, chúng ta cần lưu ý nhất là hai hiện tượng: dữ liệu mất cân bằng và quá ít (dễ dẫn tới overfit).

Trong bài toán phân loại với nhiều lớp (100 lớp) và dữ liệu có độ nhiễu nhất định (do góc quay, ánh sáng), việc sử dụng one-hot encoding cứng (1 cho đúng, 0 cho sai) dễ dẫn đến overfitting. Mô hình trở nên quá tự tin vào dự đoán của mình. Áp dụng **Label Smoothing** với hệ số  $\epsilon = 0.1$  là cách để giải quyết vấn đề này một cách đơn giản nhất. Hàm loss sẽ không ép xác suất lớp đúng phải là 1.0 tuyệt đối, mà chỉ cần đạt  $1.0 - \epsilon$ . Điều này giúp mô hình tổng hóa tốt hơn trên

tập test.

### Hàm loss với Label Smoothing

```
1 criterion = nn.CrossEntropyLoss(label_smoothing=0.1)
```

Và cũng để giải quyết vấn đề làm sao để mô hình không bị quá khớp với dữ liệu, chúng ta cũng sẽ thay thế thuật toán tối ưu **Adam** bằng **AdamW**. Adam Weight Decay hoạt động bằng cách thêm một “án phạt” vào hàm mất mát. Thay vì chỉ tối thiểu hóa sai số dự đoán, mô hình buộc phải tối thiểu hóa cả độ lớn của các trọng số. Khi trọng số của mô hình được giữ nhỏ, chúng ta sẽ giảm được độ phức tạp của mô hình qua đó giảm được hiện tượng overfitting. Tham số weight decay được sử dụng là  $1e - 4$ . Tiếp theo, chiến lược điều chỉnh learning rate **ReduceLROnPlateau** được giữ nguyên.

### Cấu hình huấn luyện

```
1 optimizer = AdamW(model.parameters(), lr=1e-4, weight_decay=1e-4)
2 scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
3     optimizer, 'min', factor=0.5, patience=3
4 )
```

#### VII.0.1. Kết quả xếp hạng

#	Thí sinh	Ngày	ID	Total Score	Public CV Score	Private CV Score
1	Không phải TQKhang	2025-11-23 03:56	6590	160.37	83.49	76.88
2	Thai Phu An	2025-11-24 00:47	6654	159.76	81.84	77.92
3	WrongAnswer	2025-11-23 18:44	6618	157.99	82.96	75.03
4	Konichichi	2025-11-20 04:04	6175	146.35	73.16	73.18
5	hieupm123	2025-11-21 09:00	6323	143.52	72.12	71.39
6	perry	2025-11-27 05:44	6882	139.26	69.67	69.58
7	Nguyễn Minh Hiếu	2025-12-01 01:49	7227	137.32	69.03	68.29
8	aiteam	2025-11-30 23:07	7211	137.15	69.07	68.08

Hình 11: Kết quả xếp hạng sau khi áp dụng các bước cải tiến trên trong cuộc thi.

Tổng hợp lại, sự kết hợp giữa pipeline dữ liệu (Augmentation + Balanced Sampling) và kiến trúc mô hình (ConvNeXt + Transformer + Attention Pooling) đã đưa về kết quả khả quan trên hệ thống chấm điểm trực tuyến của ban tổ chức Sau khi huấn luyện xong, mô hình được dùng để sinh dự đoán cho hai thư mục video ngôn ngữ ký hiệu public\_test và private\_test theo định

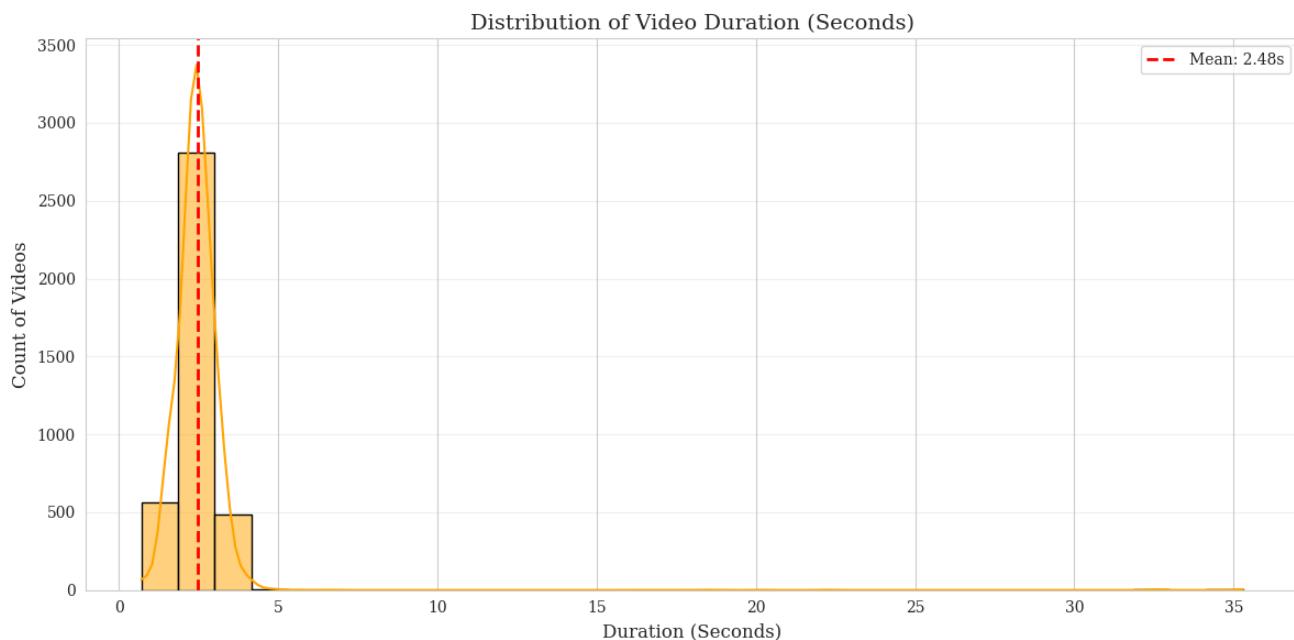
dang được yêu cầu, đưa lên hệ thống và thu được điểm tổng 137.15, xếp hạng 9 như thể hiện trên bảng xếp hạng. Kết quả này cho thấy thiết kế kiến trúc và quy trình tiền xử lý dữ liệu huấn luyện bị mất cân bằng là phù hợp, đủ sức cạnh tranh với các phương pháp khác trong cuộc thi nhận diện ngôn ngữ ký hiệu.

## VIII. Các phương pháp mở rộng thêm

### VIII.1. Phân tích thêm dữ liệu

Sau khi đã có một pipeline chạy ổn định và một mô hình ConvNeXt–Transformer cho kết quả khá tốt, một hướng mở rộng tự nhiên là quay lại bước EDA để quan sát dữ liệu kỹ hơn: thời lượng video, các lỗi kỹ thuật trong khung hình và những mẫu trùng lặp. Từ đó có thể đưa ra các quyết định xử lý dữ liệu phù hợp hơn trước khi thử các kiến trúc hoặc thuật toán mới.

#### Thời lượng video



Hình 12: Phân bố thời lượng video trong tập huấn luyện.

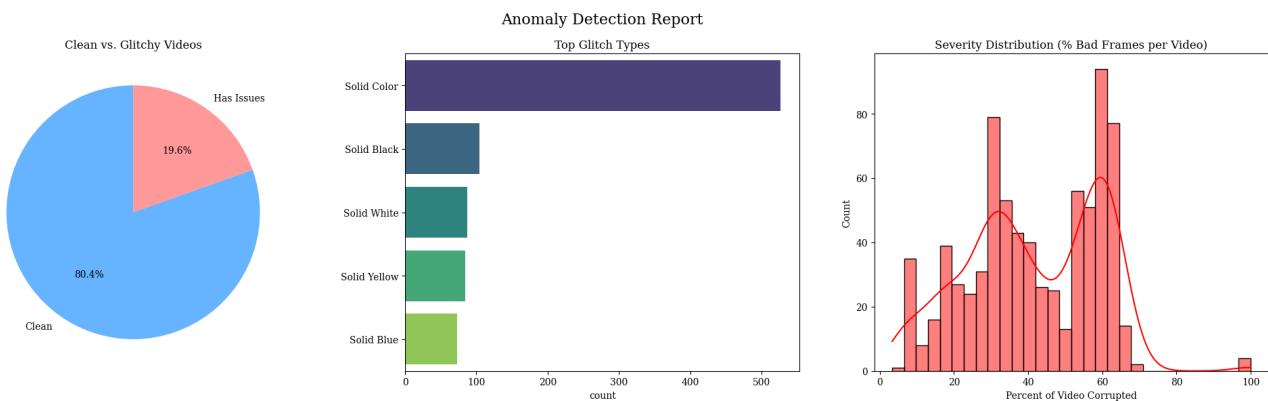
Hình 12 (biểu đồ phân bố thời lượng video) cho thấy phần lớn video chỉ kéo dài quanh mức 2–3 giây, với giá trị trung bình xấp xỉ 2.5 giây. Dải phân bố rất hẹp: gần như mọi video đều nằm trong vùng này, chỉ có một vài điểm lẻ kéo dài tới hơn 30 giây.

Từ đó có thể cân nhắc hai lựa chọn:

- Giữ nguyên chiến lược chuẩn hoá  $T = 16$  frame, vì đa số video có thời lượng tương đương nên việc nén chúng về cùng một số frame vẫn hợp lý.

- Với những video dài bất thường (hơn 20–30 giây) có thể xem xét cắt chỉ giữ lại đoạn giữa nơi diễn ra hành động, hoặc loại khỏi tập huấn luyện nếu nội dung thực tế khó xác định.

## Video lỗi



Hình 13: Phân bố mức độ hỏng theo phần trăm số khung hình lỗi trên mỗi video.

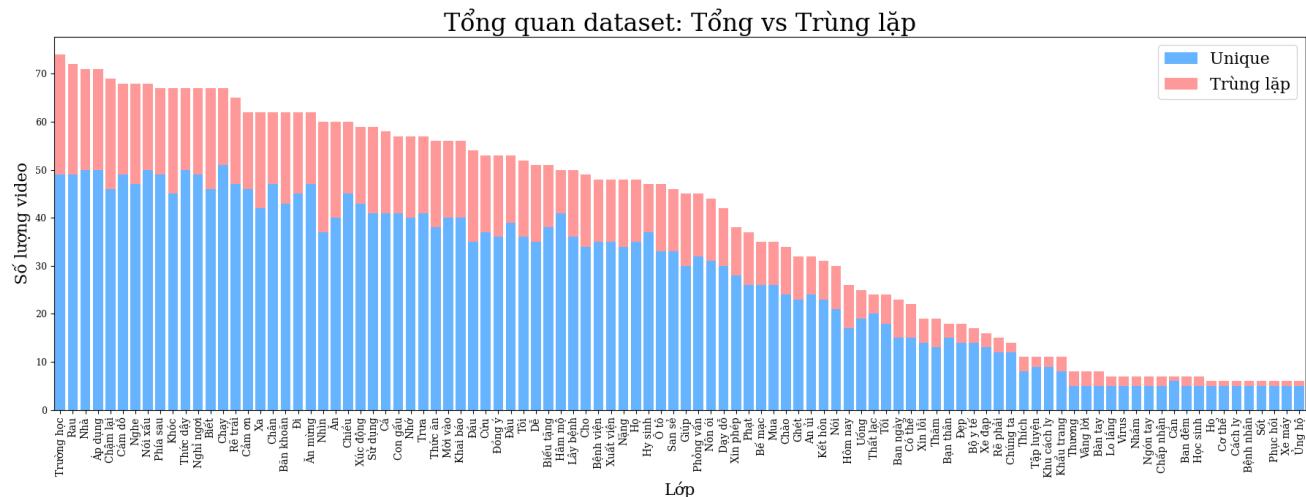
Một hướng EDA khác là quét lại bộ dữ liệu gốc để tìm các khung hình bất thường: toàn màu đen, toàn trắng, chỉ một màu, hoặc bị lỗi mã hoá khiến hình ảnh biến dạng. Kết quả được tóm tắt trong Hình 13.

- Khoảng 80% video sạch, không phát hiện vấn đề rõ rệt.
- Gần 20% video có lỗi, trong đó dạng phổ biến nhất là các khung hình solid color kéo dài (đen, trắng, vàng, xanh).
- Biểu đồ phân bố tỷ lệ khung hình hỏng trên mỗi video cho thấy hai cụm chính: một nhóm quanh mức 30–40%, và một nhóm nặng hơn quanh 60%.

Nếu bỏ qua hoàn toàn các lỗi này, mô hình sẽ học trên những frame không chứa thông tin ngôn ngữ ký hiệu, vô tình coi nhiều là dữ liệu thật. Dựa trên kết quả đó, có thể thử:

- Loại bỏ các frame hỏng trước khi downsample: khi phát hiện frame toàn màu hoặc gần như không có cấu trúc, bỏ chúng khỏi chuỗi trước khi lấy mẫu đều; nếu sau khi loại bỏ quá nhiều frame mà video còn lại quá ngắn thì loại luôn video đó khỏi tập train.
- Đặt ngưỡng cho mỗi video: nếu hơn, ví dụ, 60% frame của một video bị phát hiện là hỏng, đánh dấu video đó là không đáng tin cậy và loại khỏi tập huấn luyện. Số lượng mẫu giảm nhẹ nhưng chất lượng dữ liệu hiệu dụng tăng lên.

## Trùng lặp dữ liệu



Hình 14: Phân bố mức độ trùng lặp video trên bộ dữ liệu gốc.

Hình 14 cho thấy trong nhiều lớp, số lượng video bị phát hiện là trùng lặp chiếm một tỉ lệ đáng kể so với tổng số mẫu. Với một vài lớp ở phía bên trái biểu đồ, gần như một phần ba số video là bản sao của nhau.

Nếu không xử lý, các bản duplicate khiến mô hình tưởng rằng đang học từ rất nhiều ví dụ, trong khi thực tế chỉ xem đi xem lại cùng một chuỗi pixel. Điều này càng bất lợi khi tổng dữ liệu đã ít và phân bố nhãm mất cân bằng. Từ đó có thể đặt ra một quy tắc đơn giản:

- Giữ lại một bản duy nhất cho mỗi cụm duplicate, coi đó như một mẫu gốc.
- Với các lớp thuộc nhóm Tail vốn đã ít dữ liệu, có thể giữ lại thêm một số bản copy nhưng áp dụng augmentation mạnh hơn, thay vì giữ nguyên bản giống hệt nhau.

Tóm lại, những biểu đồ EDA bổ sung này cho thấy lượng dữ liệu hiệu dụng thực sự có thể nhỏ hơn tưởng tượng do video lỗi và video trùng. Đây là động lực để thử thêm hai hướng mở rộng: (i) dùng các kiến trúc mô hình tận dụng pretraining mạnh hơn, và (ii) điều chỉnh hàm mất mát để đối phó tốt hơn với phân bố nhãm đuôi dài.

## VIII.2. Đổi backbone sang Vision Transformer (ViT)

Một hướng mở rộng về kiến trúc là thay ConvNeXt bằng Vision Transformer (ViT). Với bài toán nhận diện ký hiệu tay, việc nắm bắt mối quan hệ không gian tinh vi giữa các vùng trong ảnh (bàn tay, khuôn mặt, vai, ngực, ...) rất quan trọng. ViT là ứng viên phù hợp vì chia ảnh thành các patch nhỏ, biến mỗi patch thành một token và xử lý toàn bộ chuỗi token bằng Transformer.

Với mỗi frame, ta thu được một chuỗi patch embedding; mô hình có thể học trực tiếp các quan hệ như “bàn tay tương quan với khuôn mặt ra sao”, “cánh tay nằm ở vị trí nào so với thân người”, mà không bị giới hạn bởi receptive field cục bộ.

Việc thay ConvNeXt bằng ViT trong kiến trúc hiện tại khá tự nhiên. Có thể định nghĩa một backbone ViT như sau:

### Thay backbone ConvNeXt bằng ViT

```

1 import timm
2 import torch.nn as nn
3
4 class ViTBackbone(nn.Module):
5     def __init__(self, model_name="vit_base_patch16_224", pretrained=True):
6         super().__init__()
7         self.vit = timm.create_model(
8             model_name,
9             pretrained=pretrained,
10            num_classes=0,      # bỏ classifier head
11            global_pool="avg" # lấy pooled embedding
12        )
13         self.out_dim = self.vit.num_features
14
15     def forward(self, x):
16         # x: (B*T, C, H, W)
17         return self.vit(x)    # (B*T, out_dim)

```

Phần còn lại của mô hình có thể giữ nguyên ý tưởng:

- Reshape đầu ra từ kích thước  $(B*T, D)$  về  $(B, T, D)$ .
- Cộng thêm positional encoding theo thời gian.
- Đưa vào Transformer Encoder hoặc dùng Attention Pooling theo thời gian như đã trình bày.

ViT thường cần nhiều dữ liệu hơn CNN nếu huấn luyện từ đầu, nhưng trong bối cảnh này có thể tận dụng các mô hình ViT được pretrained rất mạnh (ImageNet-21K hoặc các tập lớn hơn). Một vài mẹo thực nghiệm hữu ích:

- Đóng băng một phần các layer đầu trong vài epoch đầu, chỉ fine-tune các lớp cuối.
- Sử dụng learning rate nhỏ cho các tham số pretrained, lớn hơn một chút cho các lớp mới thêm như temporal encoder hoặc classifier.
- Kết hợp augmentation mạnh và các kỹ thuật regularization như label smoothing, weight decay, dropout để giảm nguy cơ overfitting.

Với các cử chỉ khác nhau chỉ ở chi tiết nhỏ của bàn tay, backbone ViT có thể giúp mô hình phân biệt tốt hơn nhờ khả năng chú ý toàn cục trên không gian ảnh.

### VIII.3. Thay đổi hàm Loss

Ngay cả khi đã áp dụng balanced sampler, bài toán vẫn chịu ảnh hưởng mạnh của phân phối đuôi dài: nhiều lớp chỉ có vài chục, thậm chí dưới 10 video. Một hướng mở rộng hợp lý là thay đổi hàm mất mát để mô hình quan tâm hơn đến các lớp hiếm.

## Weighted Cross-Entropy

Ý tưởng là gán cho mỗi lớp một trọng số  $w_c$  tỉ lệ nghịch với số lượng mẫu của lớp đó. Khi tính Cross-Entropy, lỗi của lớp hiếm sẽ được khuếch đại hơn lỗi của lớp phổ biến.

Hàm mất mát có dạng:

$$\mathcal{L}_{\text{WCE}} = - \sum_{c=1}^C w_c y_c \log p_c,$$

trong đó  $y_c$  là nhãn one-hot,  $p_c$  là xác suất mô hình dự đoán cho lớp  $c$ ,  $w_c$  có thể được chọn là

$$w_c = \frac{1}{\text{count}_c + \epsilon},$$

rồi chuẩn hoá lại sao cho  $\sum_c w_c = C$ .

Trong PyTorch, cách cài đặt khá gọn:

### Weighted Cross-Entropy trong PyTorch

```

1 import torch.nn as nn
2 import numpy as np
3 import torch
4
5 # class_counts: mảng numpy kích thước [num_classes]
6 class_weights = 1.0 / (class_counts + 1e-6)
7 class_weights = class_weights / class_weights.mean() # chuẩn hoá
8 class_weights = torch.FloatTensor(class_weights).to(device)
9
10 criterion = nn.CrossEntropyLoss(
11     weight=class_weights,
12     label_smoothing=0.1
13 )

```

Khi kết hợp với balanced sampler, Weighted Cross-Entropy giúp mô hình “nghe” tiếng nói của các lớp hiếm rõ hơn, trong khi vẫn giữ được sự ổn định trên các lớp phổ biến.

## Focal Loss và Weighted Focal Loss

Focal Loss được đề xuất trong bài toán phát hiện vật thể để xử lý mất cân bằng giữa foreground và background. Ý tưởng là giảm bớt ảnh hưởng của các mẫu dễ (đã dự đoán đúng với xác suất cao), tập trung gradient vào các mẫu khó.

Với một mẫu có nhãn đúng  $t$  và xác suất dự đoán  $p_t$ , Focal Loss có dạng:

$$\mathcal{L}_{\text{focal}} = -(1 - p_t)^\gamma \log(p_t),$$

trong đó  $\gamma \geq 0$  là tham số điều khiển mức độ tập trung. Khi  $p_t$  gần 1, hệ số  $(1 - p_t)^\gamma$  rất nhỏ, làm giảm đáng kể loss của mẫu đó; khi  $p_t$  còn thấp, hệ số này gần 1 và loss xấp xỉ Cross-Entropy bình thường.

Để xử lý đồng thời mất cân bằng giữa các lớp, có thể đưa thêm trọng số theo lớp  $\alpha_c$ , thu được Weighted Focal Loss:

$$\mathcal{L}_{WFocal} = -\alpha_t(1 - p_t)^\gamma \log(p_t),$$

trong đó  $\alpha_t$  là trọng số tương ứng với lớp đúng  $t$ .

Một hiện thực đơn giản trong PyTorch:

### Weighted Focal Loss

```

1 import torch.nn.functional as F
2 import torch.nn as nn
3 import torch
4
5 class WeightedFocalLoss(nn.Module):
6     def __init__(self, class_weights=None, gamma=2.0):
7         super().__init__()
8         self.gamma = gamma
9         if class_weights is not None:
10             self.class_weights = class_weights
11         else:
12             self.class_weights = None
13
14     def forward(self, logits, targets):
15         # logits: (B, C), targets: (B,)
16         log_probs = F.log_softmax(logits, dim=-1)
17         probs = torch.exp(log_probs)  # (B, C)
18
19         # lấy p_t và log(p_t) tương ứng với từng mẫu
20         targets = targets.view(-1, 1)
21         log_p_t = log_probs.gather(1, targets).squeeze(1)
22         p_t = probs.gather(1, targets).squeeze(1)
23
24         focal_factor = (1 - p_t) ** self.gamma
25
26         if self.class_weights is not None:
27             alpha_t = self.class_weights[targets.squeeze(1)]
28         else:
29             alpha_t = 1.0
30
31         loss = - alpha_t * focal_factor * log_p_t
32         return loss.mean()

```

Trong bối cảnh bộ dữ liệu ký hiệu tay:

- Các lớp Head thường được mô hình học tốt sau vài epoch; với Focal Loss, các mẫu thuộc những lớp này sẽ dần nhẹ tay hơn trong hàm mất mát, tránh chi phối quá mạnh.
- Các lớp Tail với ít dữ liệu và khó phân biệt hơn sẽ liên tục tạo ra loss lớn hơn, buộc mô hình dành thêm năng lực biểu diễn để tách biệt chúng khỏi các lớp còn lại.

Kết hợp Weighted Focal Loss với balanced sampler và backbone mạnh như ViT hoặc ConvNeXt là một hướng thử nghiệm hứa hẹn cho những bài toán “ít dữ liệu, nhiều lớp, mất cân bằng” như nhận diện ngôn ngữ ký hiệu trong cuộc thi này.

Tổng kết lại, các phân tích EDA mở rộng gợi ý rằng dữ liệu không chỉ ít và mất cân bằng, mà còn chứa một lượng đáng kể video lỗi và trùng lặp. Từ đó, có thể tiếp tục thử nghiệm các bước làm sạch dữ liệu sâu hơn, sử dụng backbone pretrain mạnh như ViT, và điều chỉnh hàm mất mát (Weighted Cross-Entropy, Weighted Focal Loss) để mô hình đổi xử lý công bằng hơn với các lớp hiếm. Nếu xem pipeline ban đầu là phiên bản ổn định, thì những phương pháp mở rộng này là các bước tiếp theo để đẩy mô hình tiến gần hơn tới giới hạn mà bộ dữ liệu hiện có cho phép.

## Phụ lục

1. **Code:** Các file code được đề cập trong bài có thể được tải tại [đây](#).
2. **Dataset:** Bộ dữ liệu có thể được tải tại [đây](#).
3. **Contest:** Thể lệ về cuộc thi có thể đọc tại [đây](#).
4. **Q&A:** Bạn có thể đặt thêm câu hỏi về nội dung bài đọc trong group Facebook hỏi đáp tại [đây](#). Tất cả câu hỏi sẽ được trả lời trong vòng 3-4 tiếng.

### AIO\_QAs-Verified

🔒 Private group · 1.4K members



Hình 15: Hình ảnh group facebook AIO Q&A.

# Olympia AI Training

CV: Sign Language Recognition  
NLP: Chinese – Vietnamese Translation

Code-Data

MSc. Nguyen Quoc Thai  
Nguyen Phuc Thinh – Tran Hoang Duy  
Ho Quang Hien – Xin Quy Hung

# Objectives

## Introduction CV Task

- ❖ Task Statement
- ❖ Dataset EDA
- ❖ Baseline Analysis
- ❖ Improvement

## Introduction NLP Task

- ❖ Task Statement
- ❖ Dataset EDA
- ❖ Baseline Analysis
- ❖ Improvement

## Model Implementation

- ❖ Data Imbalance Handle
- ❖ Transformer Encoder
- ❖ Attention Pooling
- ❖ Result

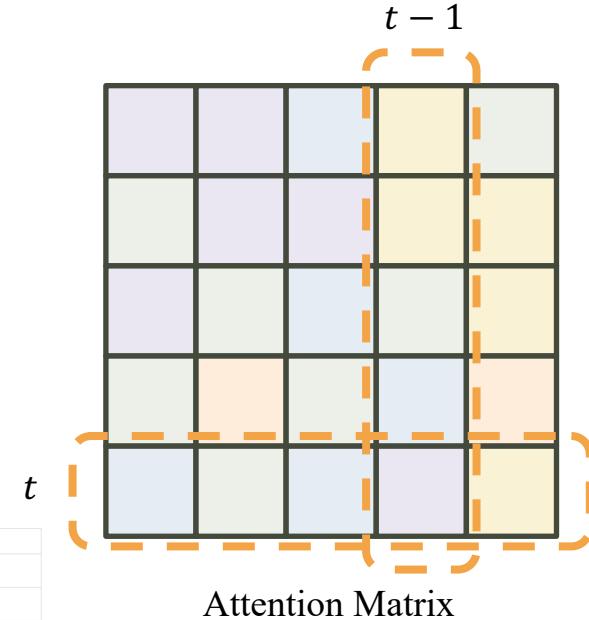
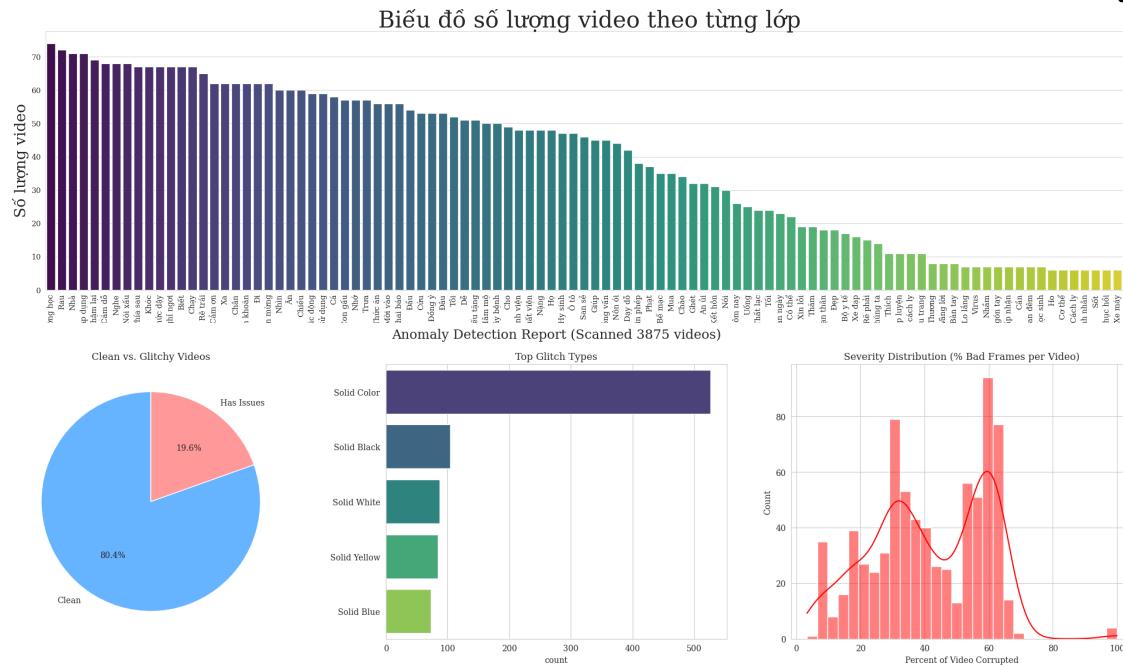
## Model Implementation

- ❖ Pre-process Data
- ❖ Transformer Encoder-Decoder
- ❖ Pretraining Model
- ❖ Contrastive Training
- ❖ Result

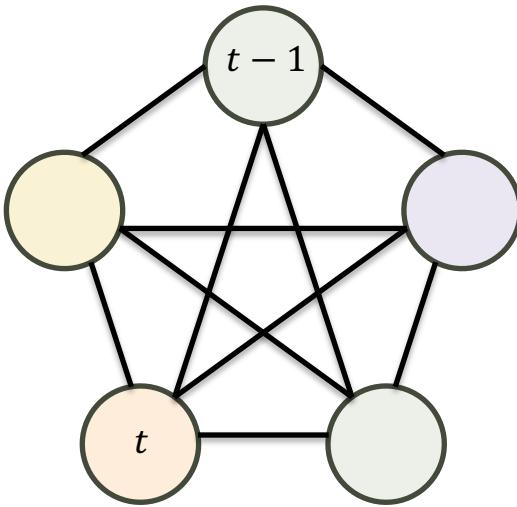
# CV Objectives

## Introduction

- ❖ Task Statement
- ❖ Dataset EDA
- ❖ Baseline Analysis
- ❖ Improvement



Attention Matrix



Latent Graph

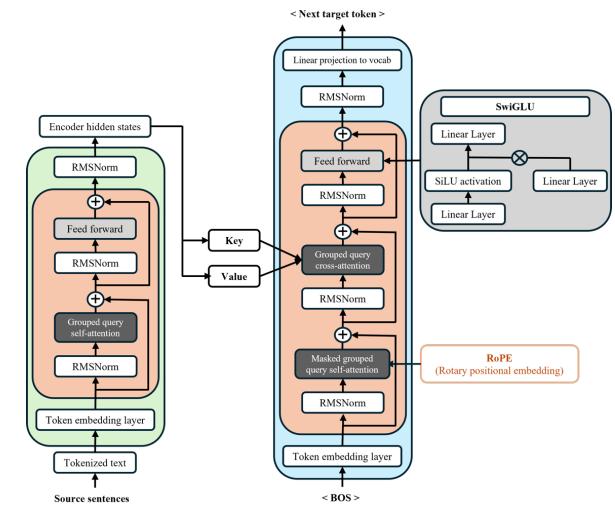
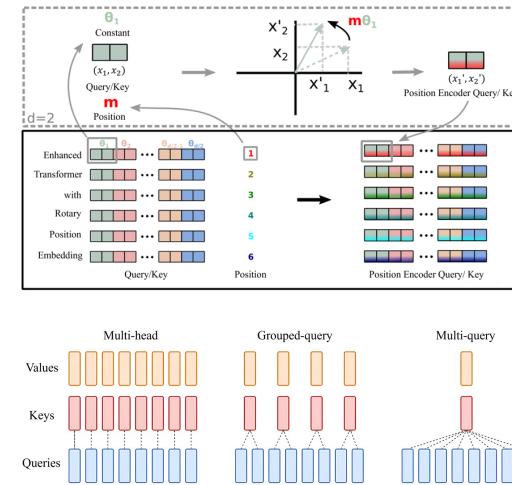
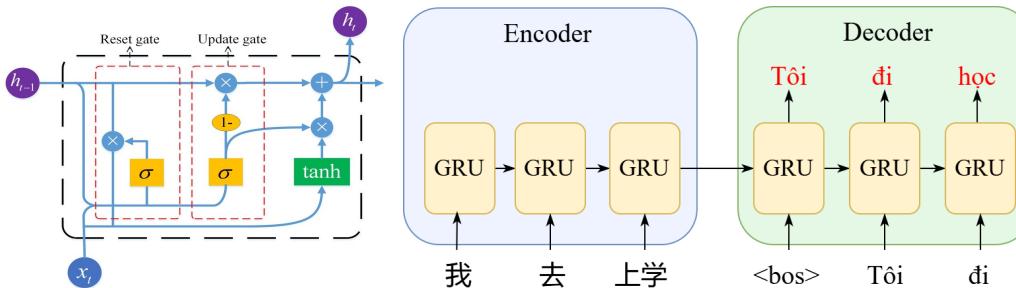
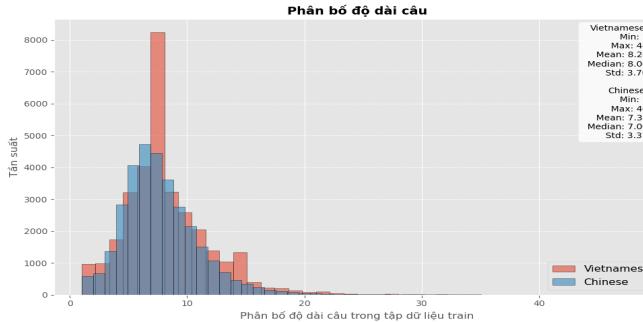
## Model Implementation

- ❖ Data Imbalance Handle
- ❖ Transformer Encoder
- ❖ Attention Pooling
- ❖ Result

# NLP Objectives

## Introduction NLP Task

- ❖ Task Statement
- ❖ Dataset EDA
- ❖ Baseline Analysis
- ❖ Improvement



## Model Implementation

- ❖ Pre-process Data
- ❖ Transformer Encoder-Decoder
- ❖ Pretraining Model
- ❖ Contrastive Training
- ❖ Result

# Outline

SECTION 1

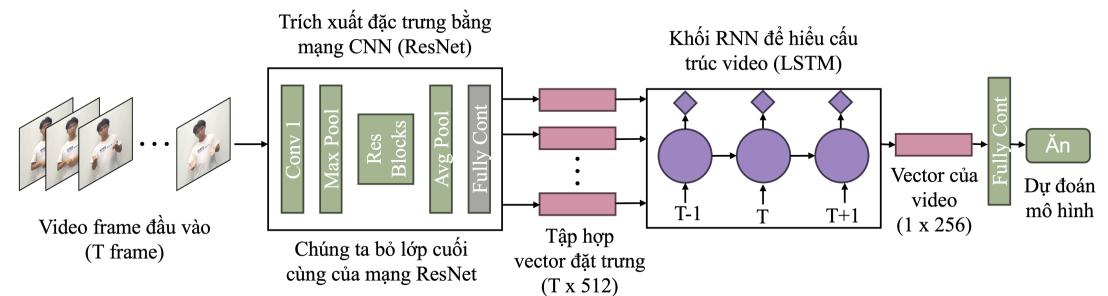
## Task Statement

SECTION 2

## Dataset EDA

SECTION 3

## Baseline Analysis



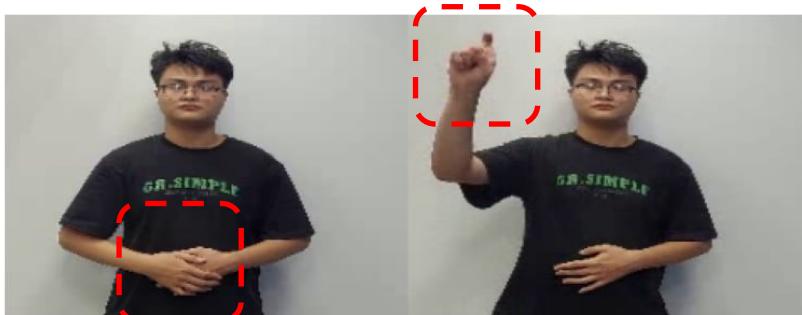
# Task Statement



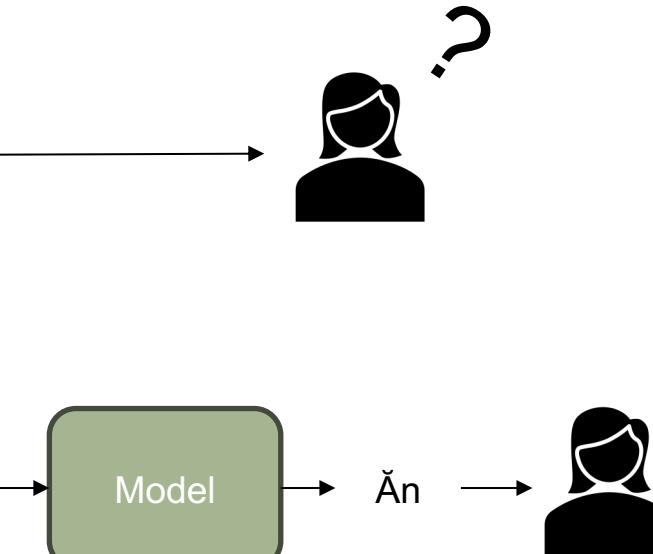
## Sign Language Recognition System



Using AI to bridge communication gaps for the deaf community.



Basic classification model for video



Classify videos of hand gestures into their corresponding Vietnamese words or phrases.

Build an automated system to support communication and information accessibility.

# Task Statement



## Sign Language Recognition System



A set of videos  $X = \{I_1, I_2, \dots, I_n\}$ .

Each input is a sequence of images (frames) capturing hand and arm gestures.

Advances in Computer Vision and Deep Learning allow for accurate pose estimation without wearable devices.



$$P(\text{Ngủ} | X) = 0.05$$

$$P(\text{Uống} | X) = 0.10$$

$$P(\text{Chơi} | X) = 0.05$$



$$P(\text{Ăn} | X) = 0.80$$

The training set contains 100 different action classes.

Predicted label corresponding to the word or phrase being signed.

# Task Statement



## Input Data Characteristics

Lighting conditions and shooting angles vary significantly.

The dataset features various signers differing in gender, body shape, skin tone, and clothing.

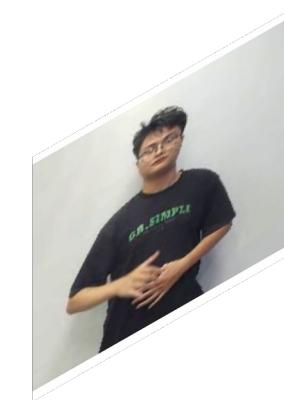
Spatial Features



Frame 1s



Temporal Features



Frame 30s

Data contains both Spatial (image dimensions) and Temporal (sequence length) information.

The model must learn invariant features rather than memorizing specific individuals.

# Task Statement



## Macro-F1 Score

Why Macro-F1?

It calculates the average F1 score across all classes.

It balances Precision and Recall for every class.

Unlike Accuracy, Macro-F1 treats all classes equally.

Forces to address the data imbalance problem effectively.

Heavily penalized for the fails to recognize the rare "Tail" classes.



# Outline

SECTION 1

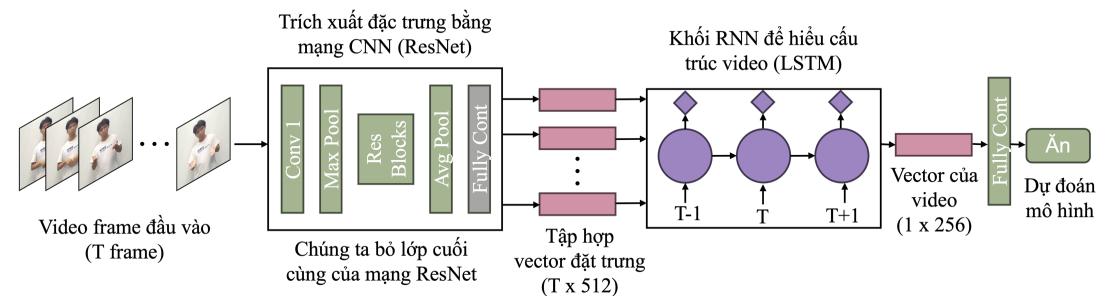
## Task Statement

SECTION 2

## Dataset EDA

SECTION 3

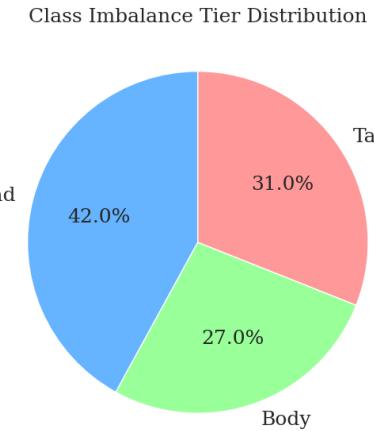
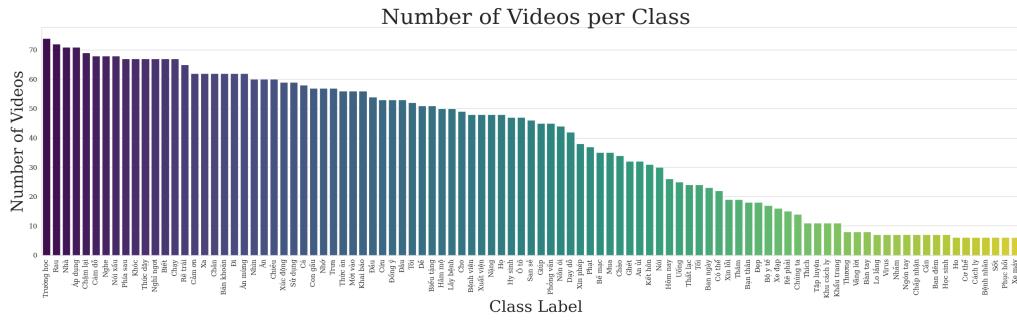
## Baseline Analysis



# Dataset EDA



# Imbalance Dataset



HEAD ( $\geq 50$  video/class)  
BODY (20-49 video/class)  
TAIL (< 20 video/class)

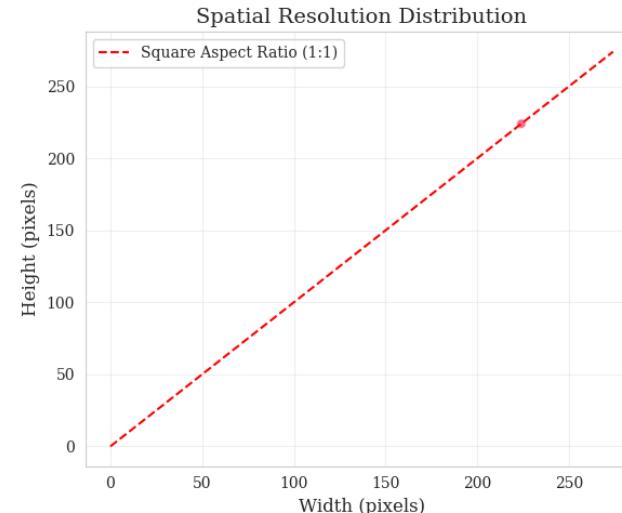
The dataset follows a typical long-tail distribution. A few "Head" classes dominate, while many "Tail" classes have very few samples.



# Dataset EDA

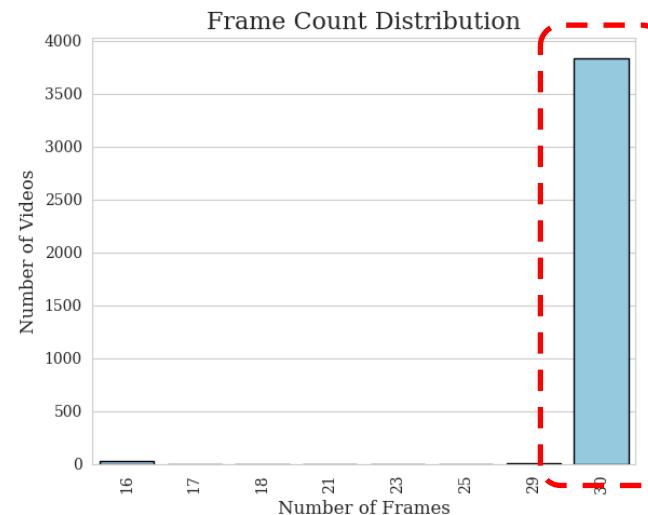


## Spatial & Temporal EDA



All videos have a near-square aspect ratio.

Resize all frames to 224 x 224 to match the ConvNeXt pre-trained input requirement without significant distortion.

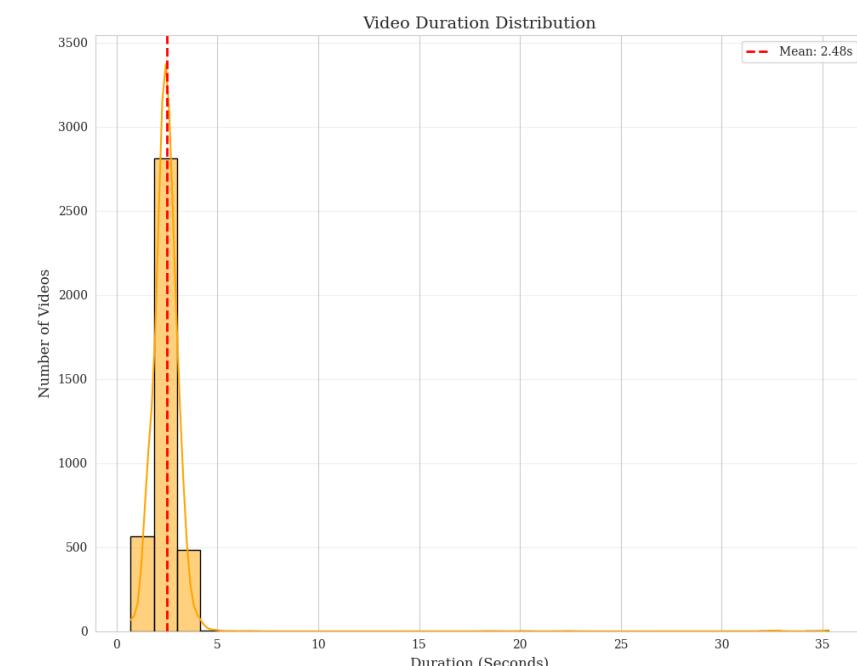


Strong peak at 30 frames (approx. 1 second duration).

We fix the input sequence length to T=16 to not overfit model. All the frame is not require to predict.

Short, concise gestures make the task suitable for memory-intensive models like Transformers.

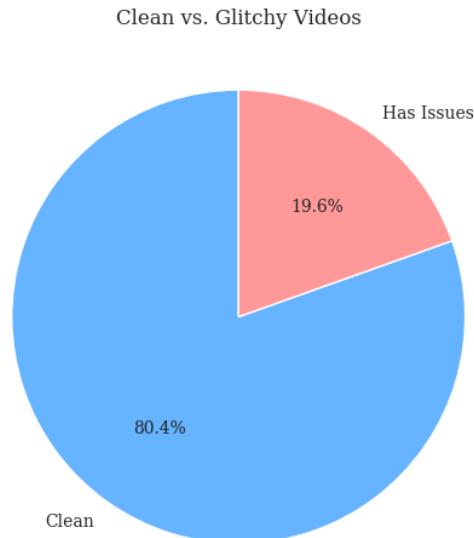
There is **outlier** for video frame that we can remove



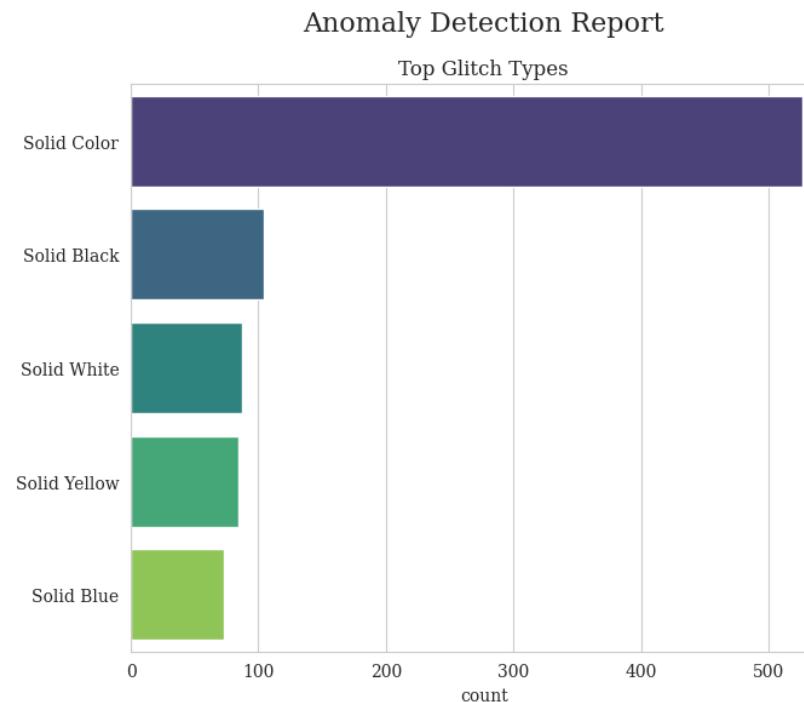
# Dataset EDA



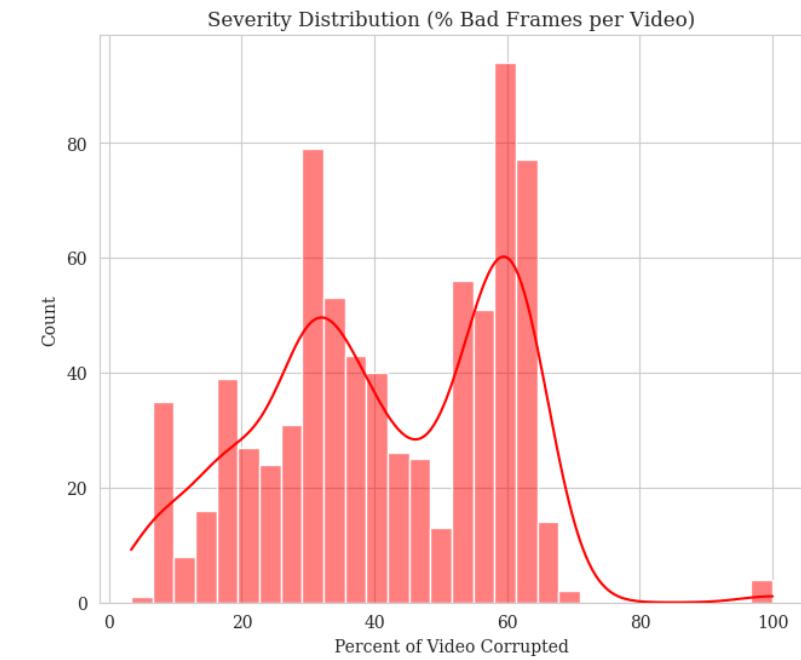
## Others EDA



Approximately 20% of the dataset contains "Glitches" or issues.



Solid color frames (Black/Blue screens) or corrupted frames.



Filter out fully corrupted videos.  
Apply robust preprocessing to handle noise.

# Outline

SECTION 1

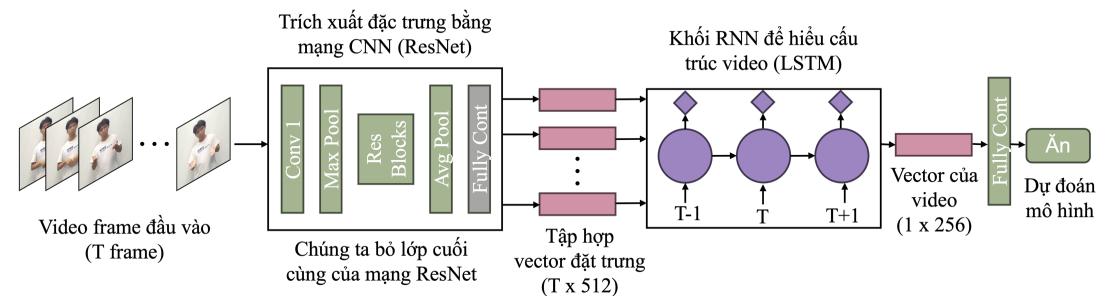
## Task Statement

SECTION 2

## Dataset EDA

SECTION 3

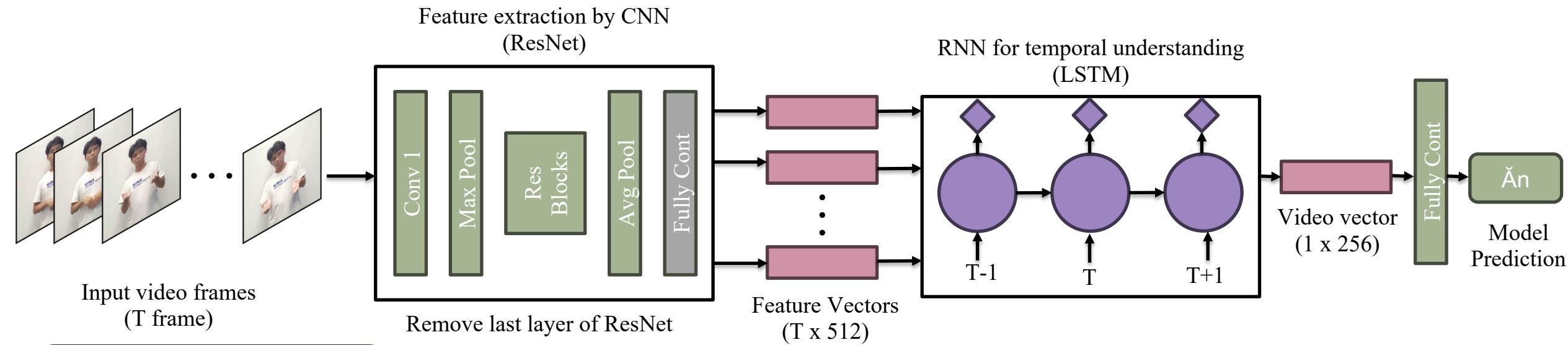
## Baseline Analysis



# Baseline Analysis



## Baseline Architecture (CRNN)



Why using backbone

- Fine-tuning requires fewer parameters than learning from scratch.
- Small set of data, easily to overfitting if we not careful.

Is LSTM the best?

- Struggles to capture long-range dependencies in complex or lengthy gestures.
- Relying solely on the last hidden state, misses details occurring in the middle of the video

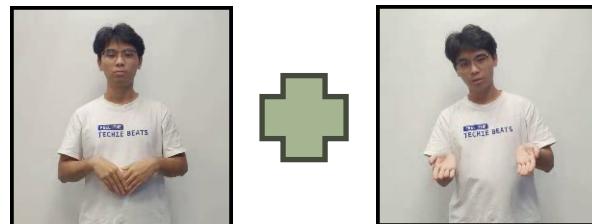
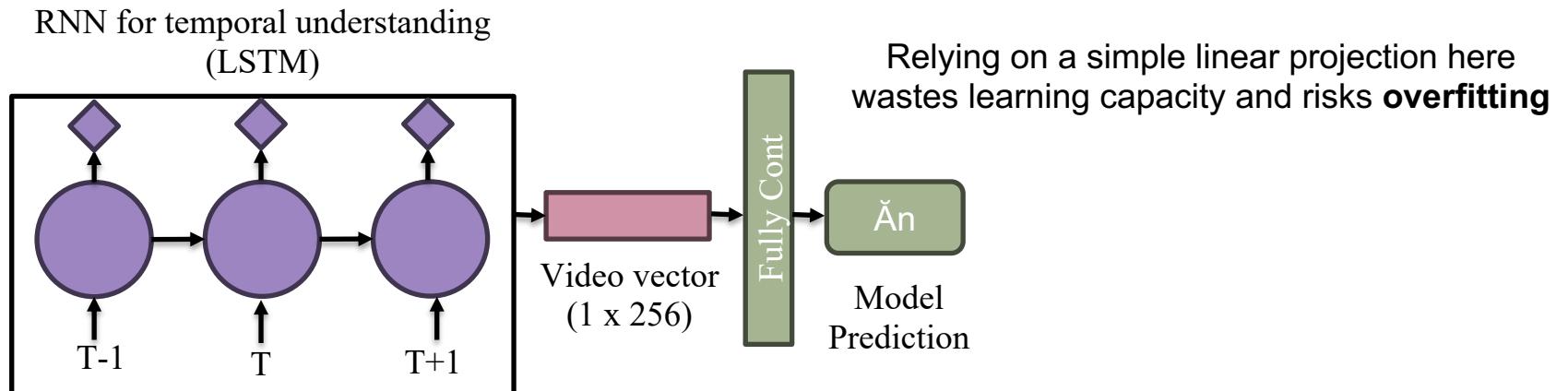
# Baseline Analysis



## Baseline Architecture (CRNN)

Nearly all the parameter is in the Fully Connected Layer. We have to do some thing!

Compressing a dynamic sequence of 16-30 frames into a single  $1 \times 256$  vector is too reductive.



If the action has multiple phases, a single vector often fails to capture the temporal evolution required to distinguish similar signs.

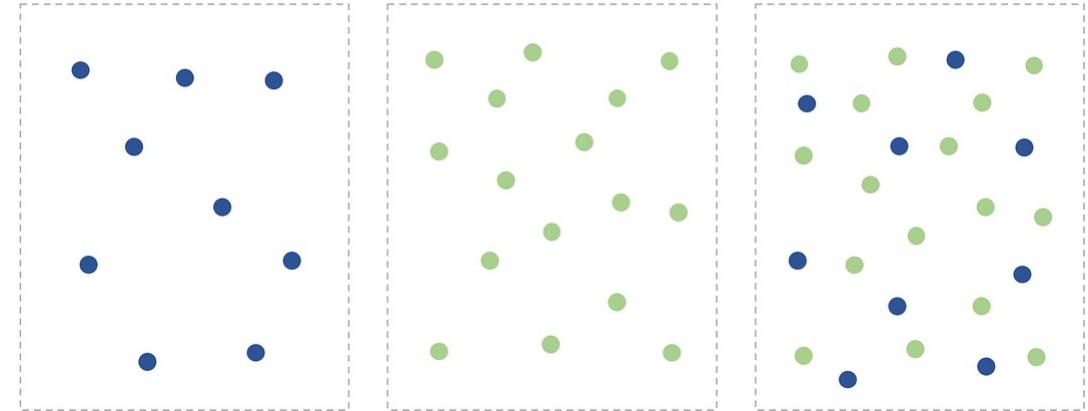


In sign language, the final frame is often the “resting state” (hands dropping down), which contains **zero semantic meaning**.

# Outline

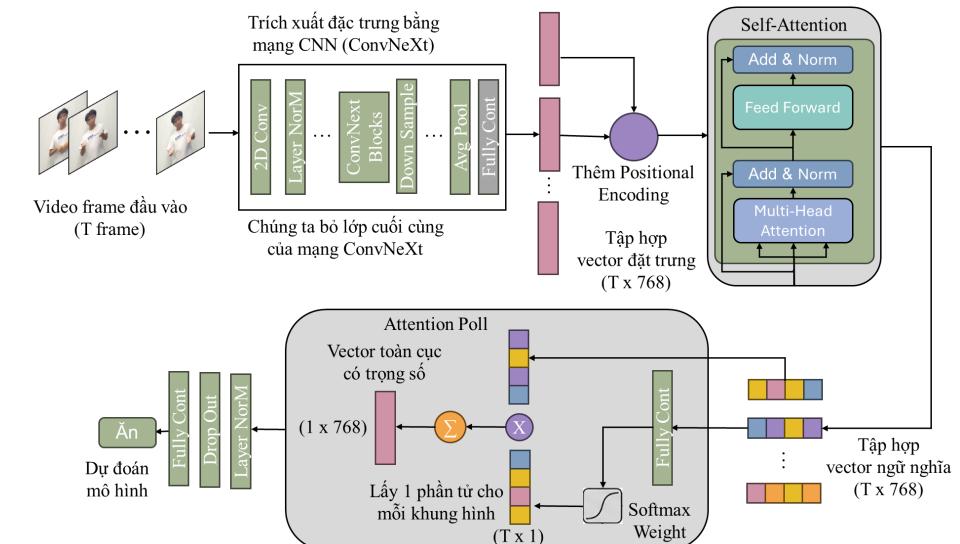
SECTION 4

## Data Improvement

Raw Samples      Modified Samples      Augmented Dataset  
To use for Training

SECTION 5

## Model Improvement



# Data Improvement



## Enhanced Data Pipeline

Standardize input to fixed length T=16 frames.

Uniform Sampling: For long videos ( $>16$ ), take frames evenly to preserve global context.  
(avoids losing info)

How about the last frame?

Padding: For short videos ( $<16$ ), repeat the last frame (handled later by attention mechanisms).

```
1 def _downsample_frames(self, frames):
2     """Lấy target_frames từ video"""
3     total = frames.shape[0]
4     if total >= self.target_frames:
5         indices = torch.linspace(0, total - 1, self.target_frames).long()
6     else:
7         indices = torch.arange(total)
8         pad = self.target_frames - total
9         indices = torch.cat([indices, indices[-1].repeat(pad)])
10
11    frames = frames[indices]
12
13    # Resize về 224x224 nếu chưa
14    if frames.shape[1] != 224 or frames.shape[2] != 224:
15        frames = frames.permute(0, 3, 1, 2).float()
16        frames = F.interpolate(
17            frames, size=(224, 224), mode='bilinear', align_corners=False)
18        frames = frames.permute(0, 2, 3, 1).to(torch.uint8)
19
20    return frames
```

# Data Improvement

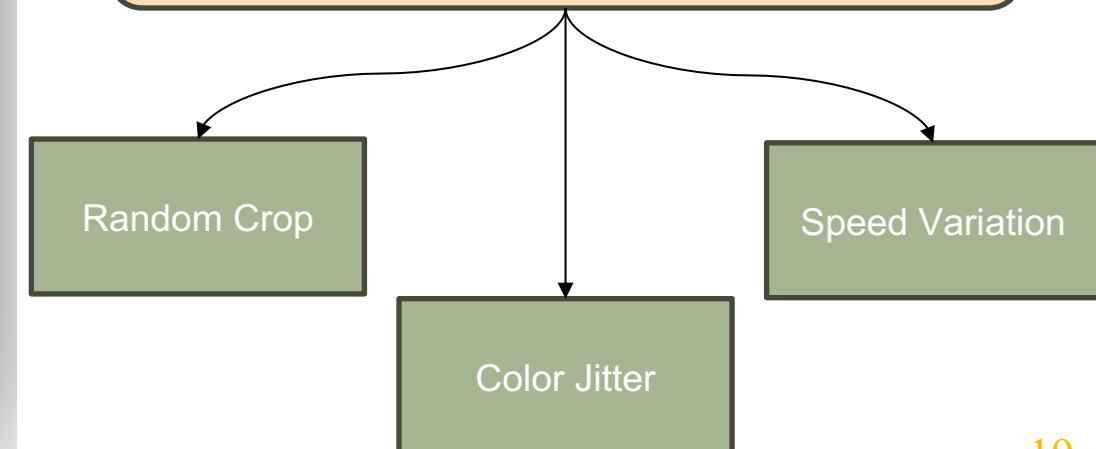


## Data Augmentation

```
1 def _color_jitter(self, frames):
2     """Color jitter - CONSISTENT cho tất cả frames"""
3     # Random parameters (CÙNG cho tất cả frames)
4     brightness_factor = 1.0 + random.uniform(-self.brightness, self.brightness)
5     contrast_factor = 1.0 + random.uniform(-self.contrast, self.contrast)
6     saturation_factor = 1.0 + random.uniform(-self.saturation, self.saturation)
7
8     frames = frames.float()
9
10    # Brightness
11    frames = frames * brightness_factor
12
13    # Contrast
14    mean = frames.mean(dim=(1, 2), keepdim=True)
15    frames = (frames - mean) * contrast_factor + mean
16
17    # Saturation
18    gray = frames.mean(dim=-1, keepdim=True)
19    frames = gray + (frames - gray) * saturation_factor
20
21    # Clamp to valid range
22    frames = torch.clamp(frames, 0, 255)
23
24    return frames.to(torch.uint8)
```

Applying random augmentations per frame makes the video "jittery" and destroys motion features.

Apply the same transformation parameters to all frames in a video clip.



# Data Improvement



## Data Augmentation

```
1 def _speed_augment(self, frames):
2     """Thay đổi tốc độ video bằng cách resample frames"""
3     T = frames.shape[0]
4     speed = random.uniform(self.speed_range[0], self.speed_range[1])
5
6     new_T = int(T / speed)
7     if new_T < 4:
8         new_T = 4
9     if new_T == T:
10        return frames
11
12     # Resample frames
13     indices = torch.linspace(0, T - 1, new_T).long()
14     indices = torch.clamp(indices, 0, T - 1)
15     frames = frames[indices]
16
17     return frames
```

Simulate real-world variations where different people sign at different speeds (fast vs. slow).



Do at home:  
Can we modify the model to do this for us instead?

Make the model robust to changes in signer position and camera distance (zoom).

```
1 def _random_resized_crop(self, frames):
2     """Random crop rồi resize về 224x224 - CONSISTENT"""
3     T, H, W, C = frames.shape
4
5     # Random scale và position (CÙNG cho tất cả frames)
6     scale = random.uniform(self.crop_scale[0], self.crop_scale[1])
7     crop_h, crop_w = int(H * scale), int(W * scale)
8
9     top = random.randint(0, H - crop_h)
10    left = random.randint(0, W - crop_w)
11
12    # Crop tất cả frames GIỐNG NHAU
13    frames = frames[:, top:top+crop_h, left:left+crop_w, :]
14
15    # Resize về 224x224
16    # (T, H, W, C) -> (T, C, H, W) for interpolate
17    frames = frames.permute(0, 3, 1, 2).float()
18    frames = F.interpolate(
19        frames, size=(224, 224), mode='bilinear', align_corners=False)
20    # (T, C, H, W) -> (T, H, W, C)
21    frames = frames.permute(0, 2, 3, 1)
22
23    return frames.to(torch.uint8)
```

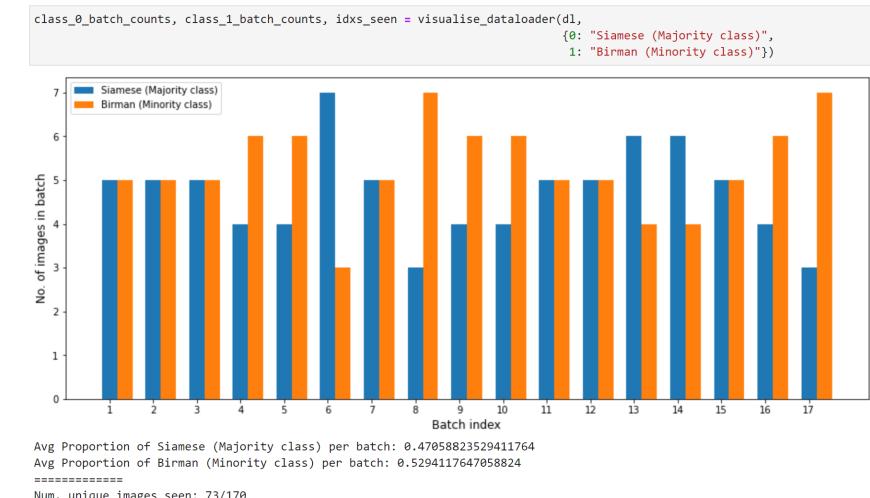
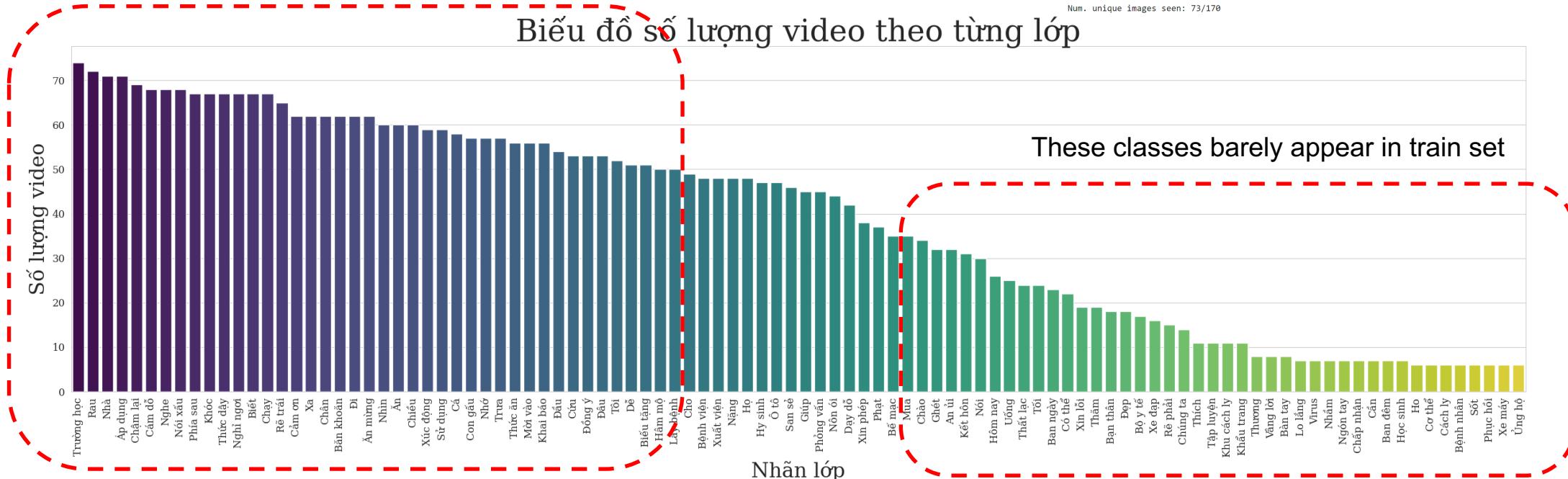
# Data Improvement



## Balanced Sampling

Use a WeightedRandomSampler with inverse frequency weights ( $w_c = 1/N_c$ ). This forces the model to see minority classes more often during training.

Model learns a lot from these classes



# Checklist

## EDA

- Class Imbalance (Long-tail?)
- Resolution & Aspect Ratio
- Outliers / Duplicates



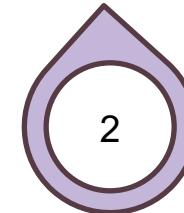
## Modeling

- Baseline: ResNet18/34/50.
- SOTA: ConvNeXt (Tiny/Base), Swin Transformer, or ViT.



## Post-processing

- Ensemble?
- Test Time Augmentation?
- Check model overfit or underfit!



Transformer Encoder + Attention Pooling



- Resize vs. Padding
- Normalization
- Simple Augmentation: Random Crop, Horizontal Flip
- Advance Augmentation: CutMix, Mixup



- Loss Function: Label Smoothing
- Sampling: Balanced Sampler
- Optimizer: AdamW

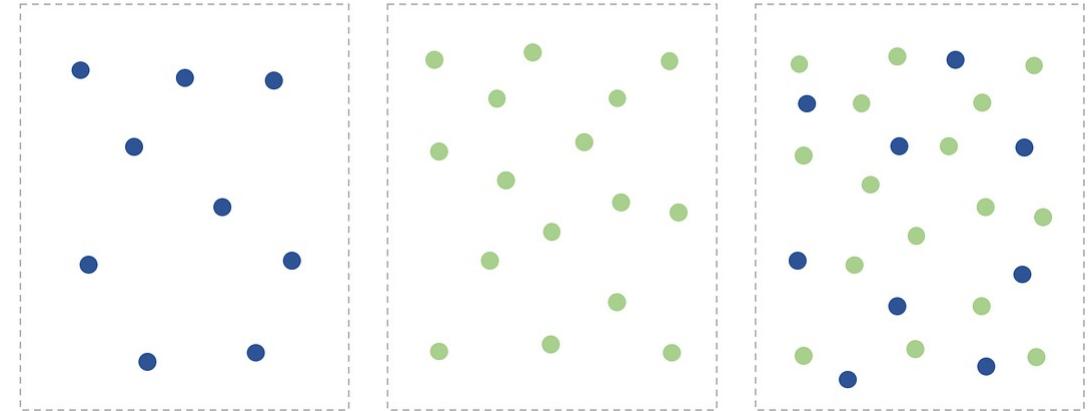
Data Pipeline

Training

# Outline

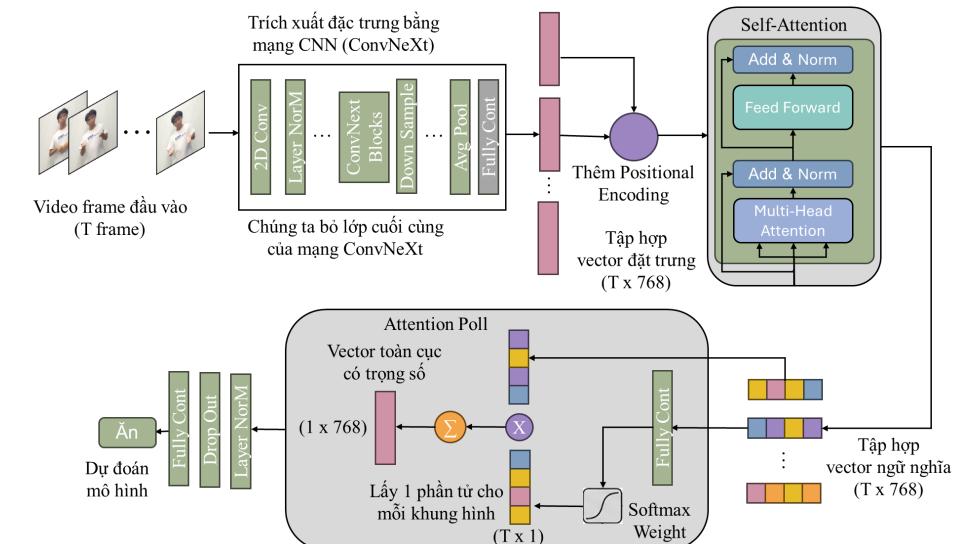
SECTION 4

## Data Improvement

Raw Samples      Modified Samples      Augmented Dataset  
To use for Training

SECTION 5

## Model Improvement



# Model Improvement



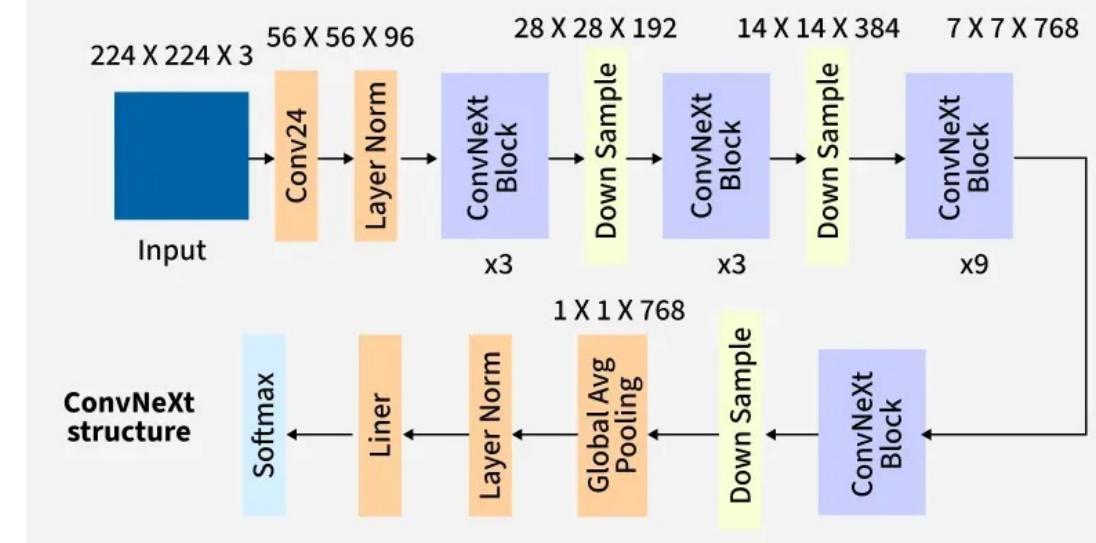
## Backbone Upgrade - ConvNeXt-Tiny

### Why ConvNeXt-Tiny?

- Replaces the outdated ResNet18.
- A modern CNN architecture inspired by Vision Transformers (ViT).
- Transfer Learning: Pre-trained on ImageNet-1K (1.2M images), enabling faster convergence on small datasets.

Generates a richer feature vector ( $D=768$ ) compared to ResNet18 ( $D=512$ ).

Provides high-quality semantic visual features for the temporal model to process.

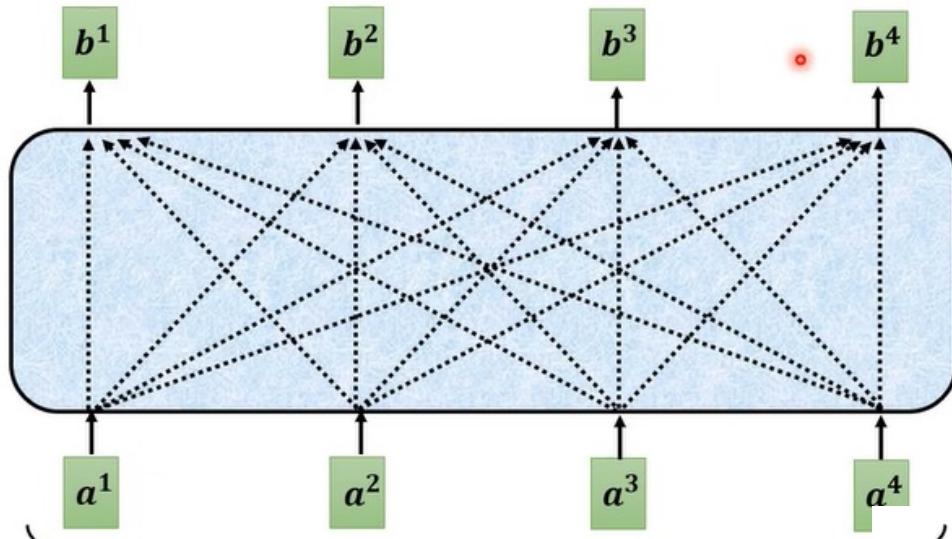


Try its your self:  
Does other types of backbone like ViT or SWIN can work too?

# Model Improvement

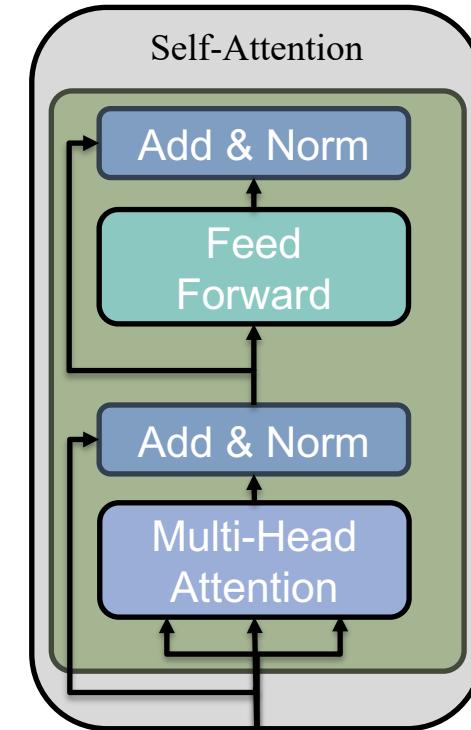


## Temporal Modeling - Transformer Encoder



### Replacing LSTM

- LSTM processes frames sequentially (slow, forgets long-term context).
- Transformer Encoder processes the whole sequence in parallel.



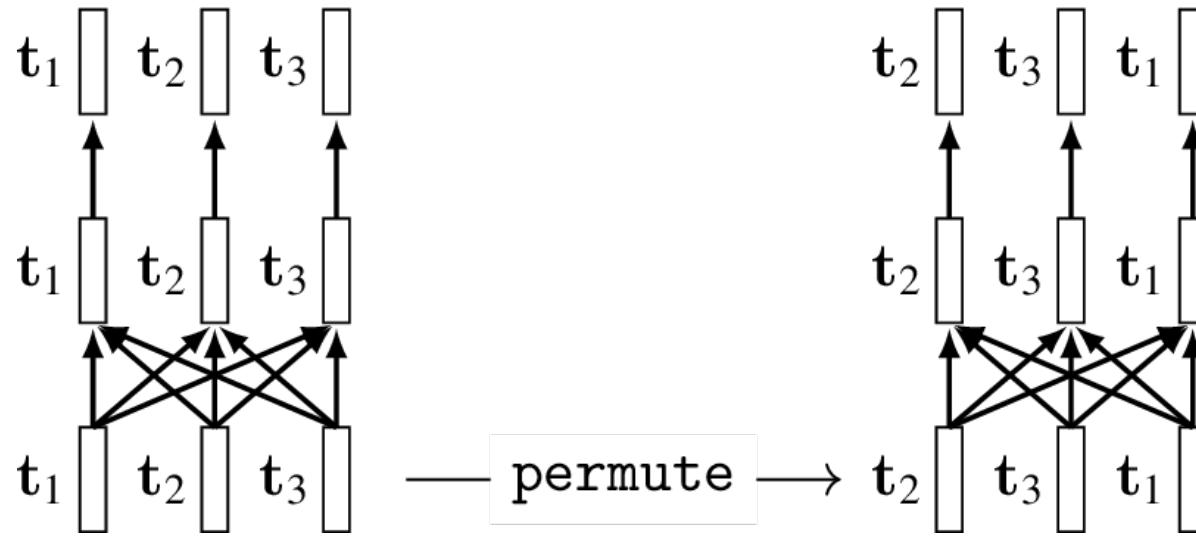
### Self-Attention Mechanism

- Allows the model to look at all frames simultaneously.
- Captures long-range dependencies (relationships between the start and end of a gesture) much better than RNNs.

# Model Improvement



## Temporal Modeling - Transformer Encoder



Transformers are permutation equivariant. For notational simplicity, we omit layer indices on the token variables here.

We need to define way for Transformer  
to know which frame is prior to the next

```
1  class PositionalEncoding(nn.Module):
2      """Positional encoding cho temporal sequence"""
3      def __init__(self, d_model, max_len=64, dropout=0.1):
4          super().__init__()
5          self.dropout = nn.Dropout(p=dropout)
6
7          pe = torch.zeros(max_len, d_model)
8          position = torch.arange(
9              0, max_len, dtype=torch.float).unsqueeze(1)
10         div_term = torch.exp(
11             torch.arange(0, d_model, 2).float() *
12             (-math.log(10000.0) / d_model)
13         )
14
15         pe[:, 0::2] = torch.sin(position * div_term)
16         pe[:, 1::2] = torch.cos(position * div_term)
17         pe = pe.unsqueeze(0) # (1, max_len, d_model)
18
19         self.register_buffer('pe', pe)
20
21     def forward(self, x):
22         # x: (B, T, d_model)
23         x = x + self.pe[:, :x.size(1), :]
24         return self.dropout(x)
```

# Model Improvement



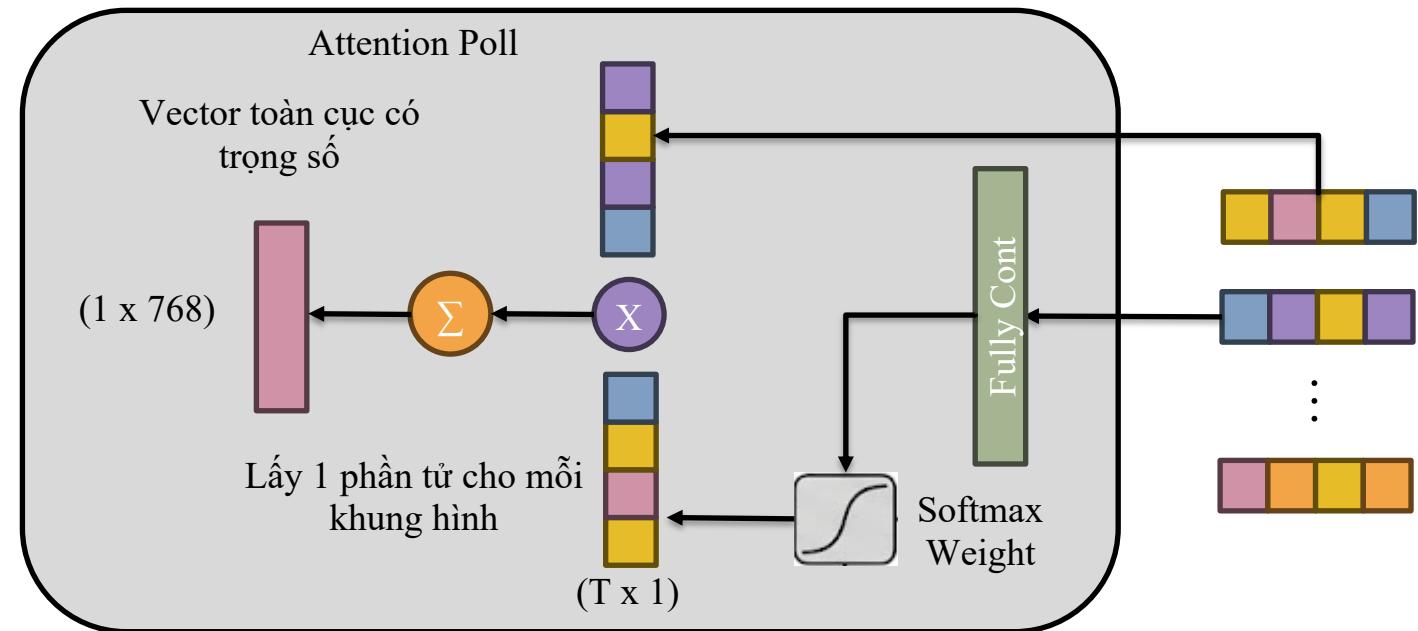
## Smart Aggregation - Attention Pooling

```

1  class AttentionPooling(nn.Module):
2      """Attention pooling thay cho last hidden của LSTM"""
3      def __init__(self, dim):
4          super().__init__()
5          self.attention = nn.Sequential(
6              nn.Linear(dim, dim // 4),
7              nn.Tanh(),
8              nn.Linear(dim // 4, 1)
9          )
10
11     def forward(self, x):
12         # x: (B, T, dim)
13         attn_weights = self.attention(x) # (B, T, 1)
14         attn_weights = F.softmax(attn_weights, dim=1)
15         pooled = torch.sum(attn_weights * x, dim=1) # (B, dim)
16         return pooled

```

Baseline uses the last hidden state, assuming the gesture ends at the last frame. In reality, the meaningful gesture often happens in the middle.



The model learns a weight  $a_t$  for each frame  $t$ .

$$V = \sum_{t=1}^T a_t x_t.$$

The model automatically learns to “focus” on keyframes containing the core gesture and “ignore” static frames (like padding or resting hands). 27

# Model Improvement



## Optimized Training Strategy

Regularization with Label Smoothing



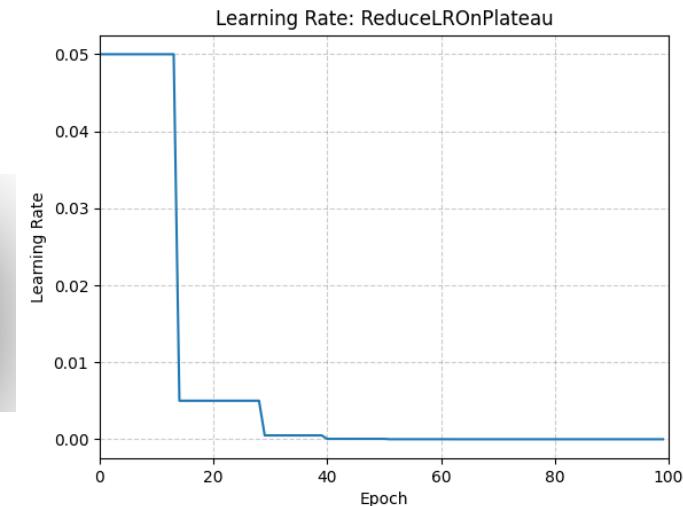
Prevents the model from being "over-confident" (forcing probability to 1.0). Helps generalization on noisy data.

Advanced Optimizer (AdamW)

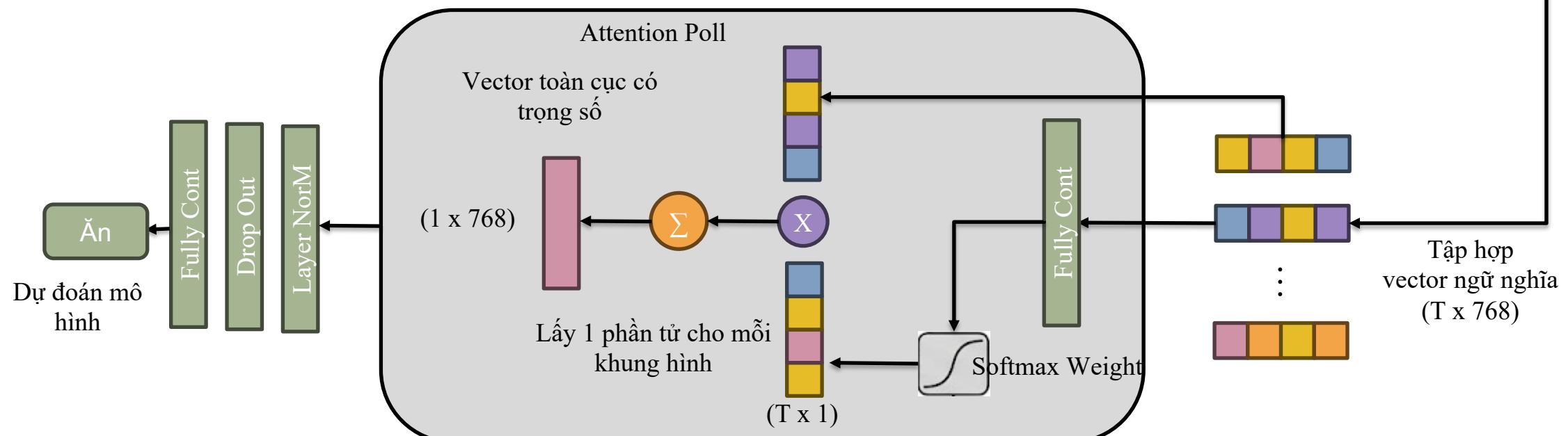
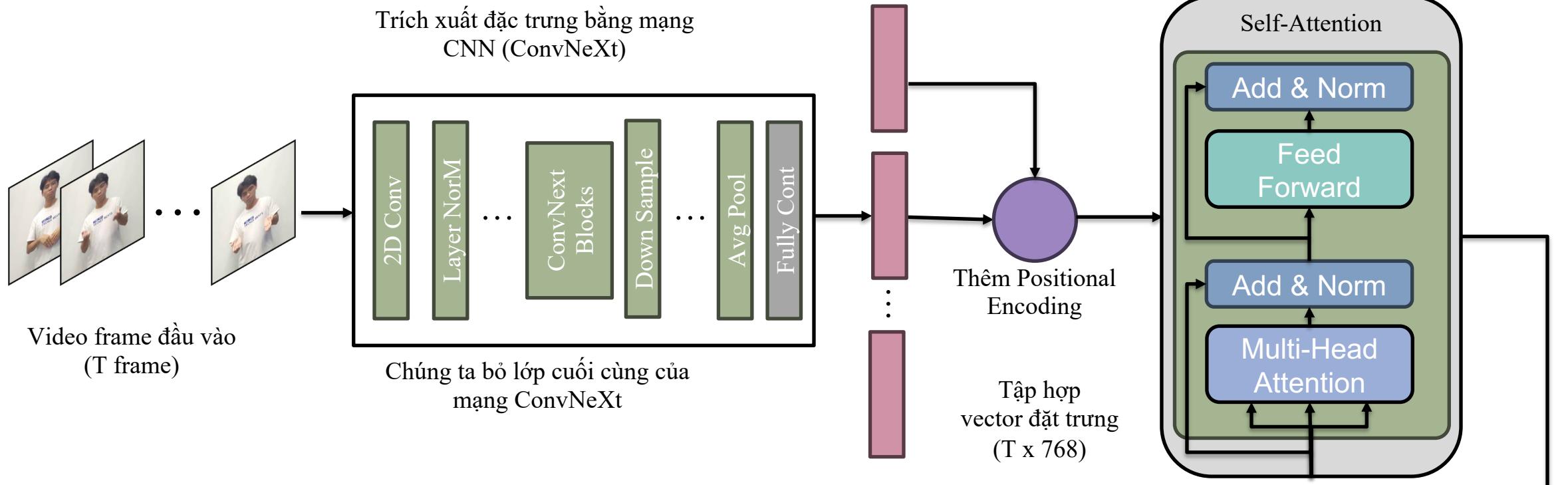
Adds Weight Decay (1e-4): Penalizes large weights to reduce model complexity and prevent overfitting on the small dataset.

```
1 criterion = nn.CrossEntropyLoss(label_smoothing=0.1)
2 optimizer = AdamW(model.parameters(), lr=lr, weight_decay=1e-4)
3 scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
4     optimizer, 'min', factor=0.5, patience=3
5 )
```

Learning Rate Scheduler



Reduces LR by factor 0.5 if validation loss stagnates for 3 epochs.

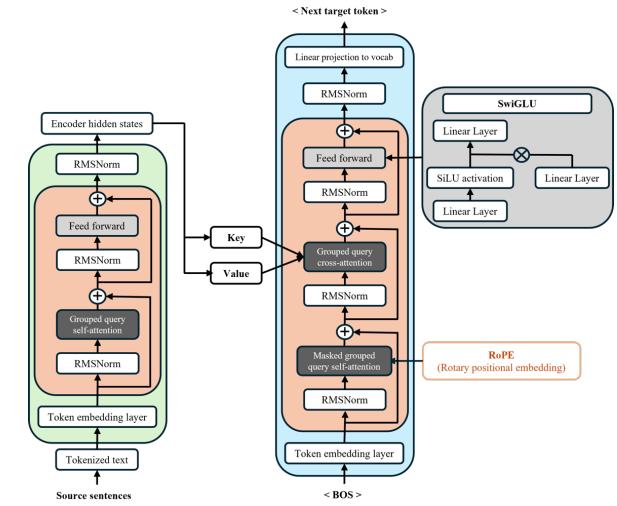
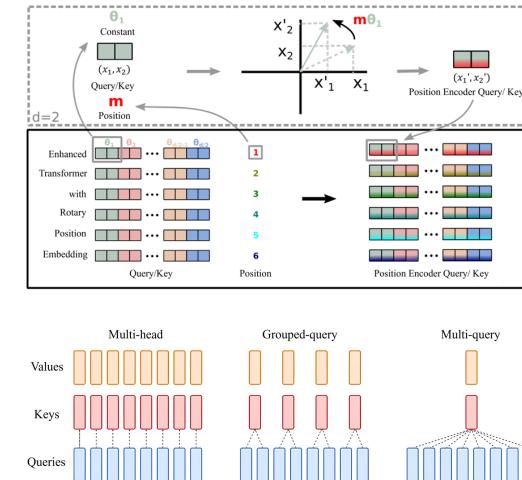
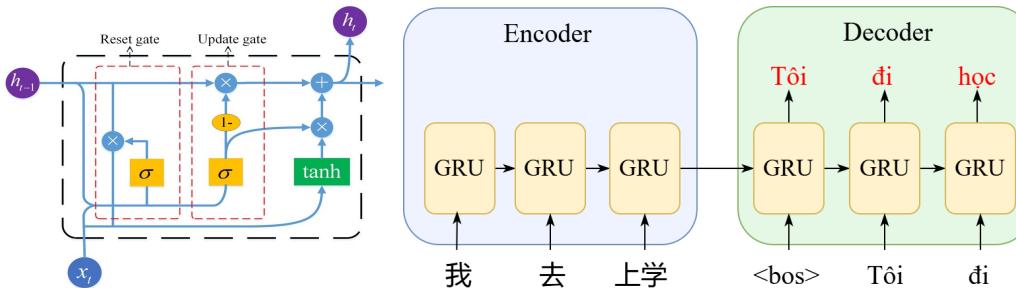
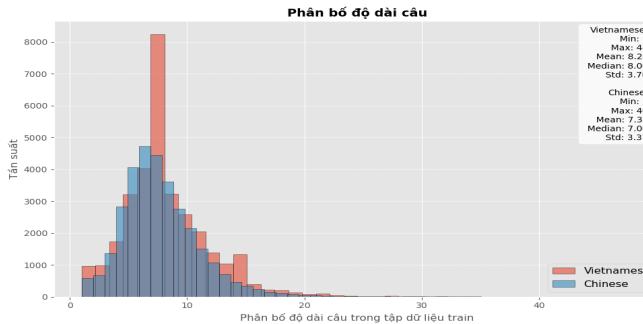


#	Thí sinh	Ngày	ID	Total Score	Public CV Score	Private CV Score
1	Không phải TQKhang	2025-11-23 03:56	6590	160.37	83.49	76.88
2	Thai Phu An	2025-11-24 00:47	6654	159.76	81.84	77.92
3	WrongAnswer	2025-11-23 18:44	6618	157.99	82.96	75.03
4	Konichichi	2025-11-20 04:04	6175	146.35	73.16	73.18
5	hieupm123	2025-11-21 09:00	6323	143.52	72.12	71.39
6	perry	2025-11-27 05:44	6882	139.26	69.67	69.58
7	Nguyễn Minh Hiếu	2025-12-01 01:49	7227	137.32	69.03	68.29
8	aiteam	2025-11-30 23:07	7211	137.15	69.07	68.08

# NLP Objectives

## Introduction NLP Task

- ❖ Task Statement
- ❖ Dataset EDA
- ❖ Baseline Analysis
- ❖ Improvement



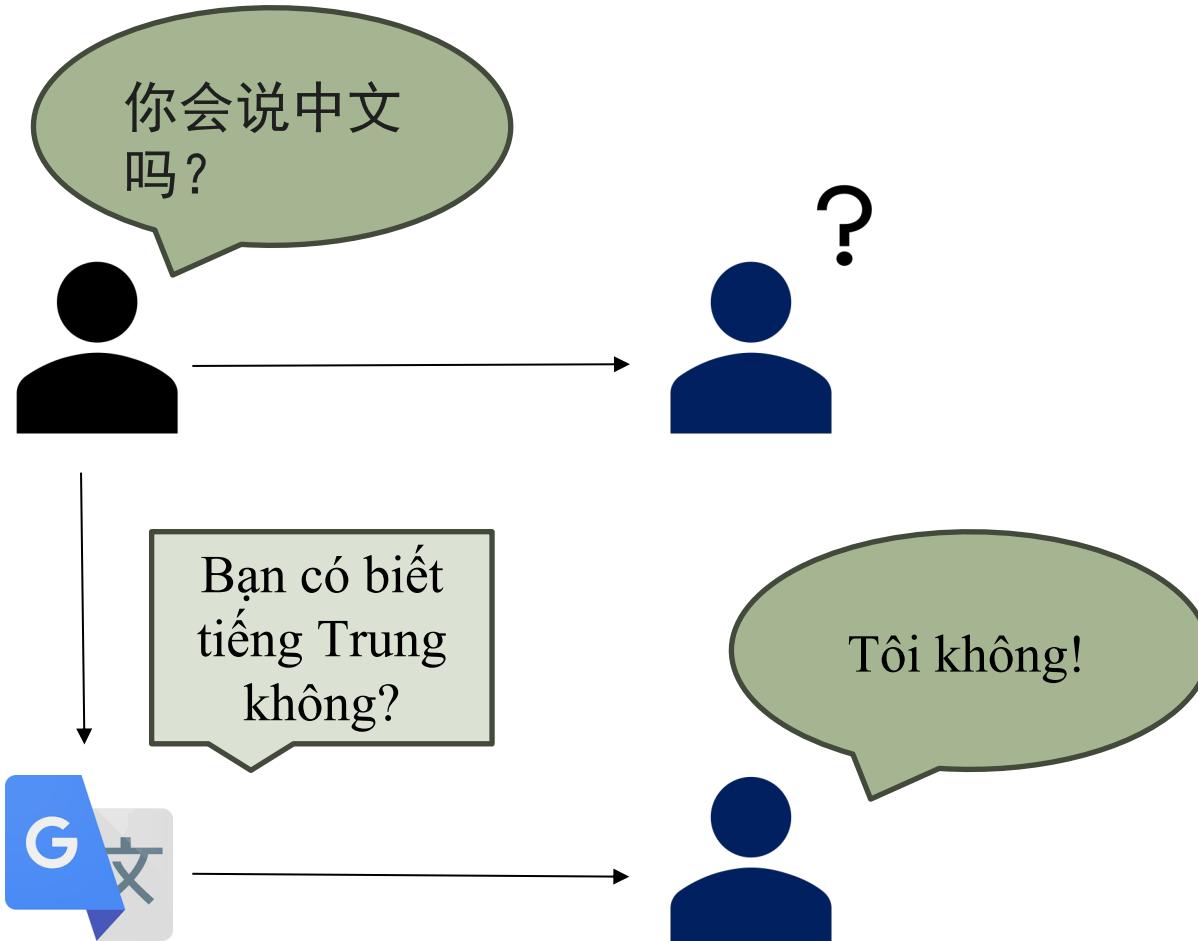
## Model Implementation

- ❖ Pre-process Data
- ❖ Transformer Encoder-Decoder
- ❖ Pretraining Model
- ❖ Contrastive Training
- ❖ Result

# Task Statement



## Translation system



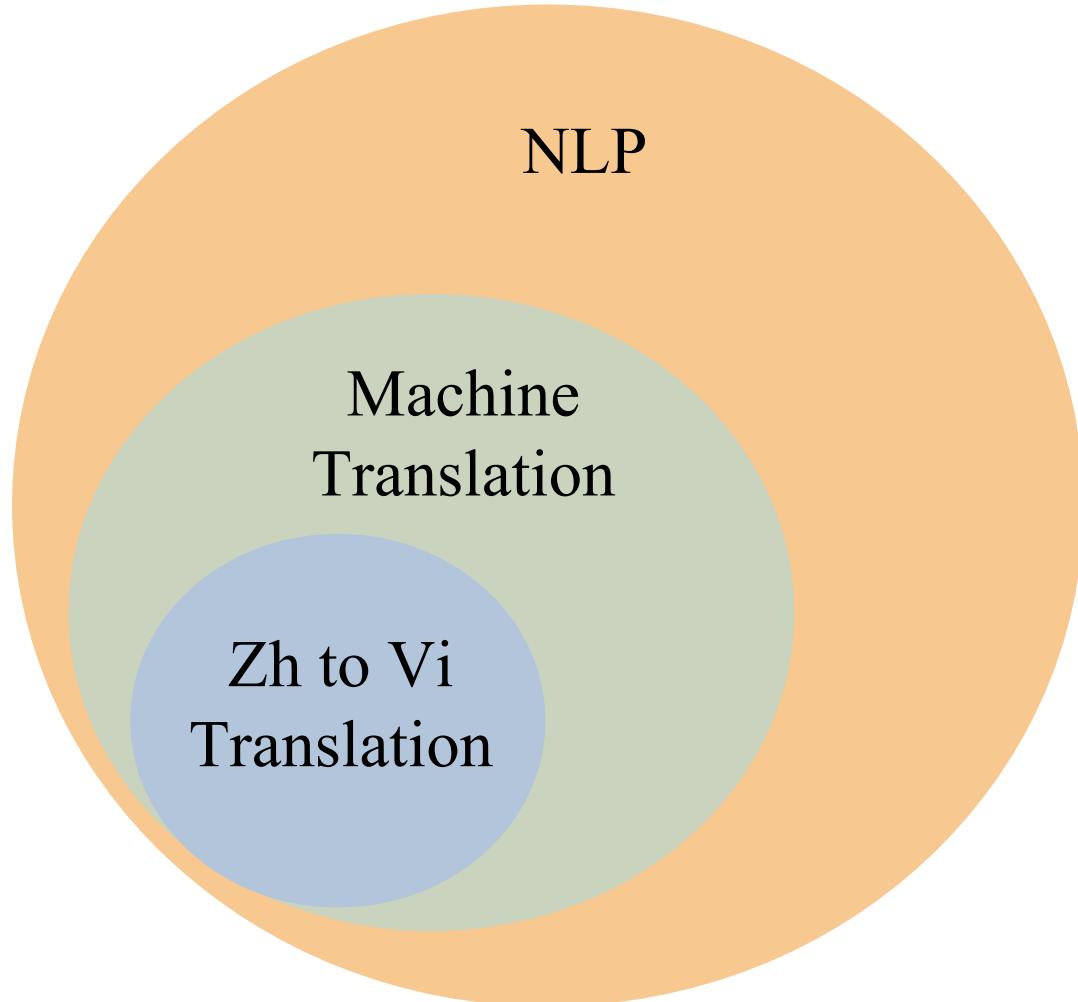
AI can break the language barrier!

Translation systems (Google Translate) or LLMs (ChatGPT,..) can translate text from one language to another with high accuracy.

# Task Statement



## About Machine Translation



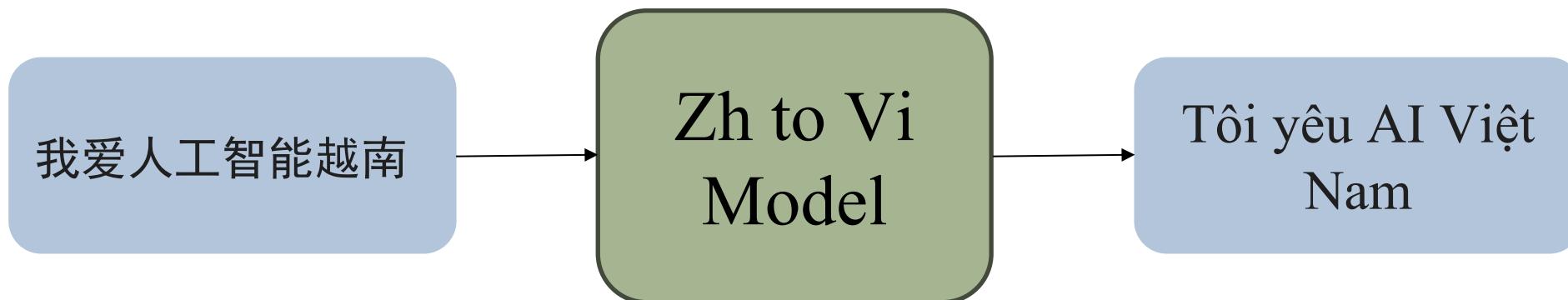
Machine Translation (MT) is a task in the Natural Language Processing (NLP) domain

Meanwhile, Chinese to Vietnamese translation is a specific task in MT

# Task Statement



## NLP Task



The task is to build a model that can translate Chinese to Vietnamese

# Task Analysis



## Rules of the NLP Task

**Given:**

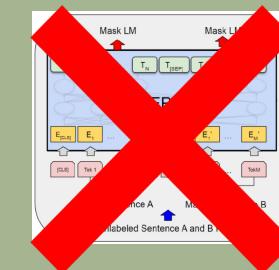
- **Train set:** train.zh, train.vi
- **Test set:** public\_test.zh,  
private\_test.zh
- **Baseline.ipynb**

**Task:** Train a Zh to Vi model from scratch

**Evaluation:** SacreBLEU score

**Additional rules:**

- Using pre-trained models (BERT,...) is **not allowed**
- Using additional dataset is **not allowed**



# Task Analysis



## Evaluation Metric

$$\text{SacreBLEU} = BP \cdot \exp \left( \sum_{n=1}^N w_n \log p_n \right)$$

$p_n$ : The n-gram precision at order  $n$ , computed as

$$p_n = \frac{\sum_{\text{ngram} \in \text{hyp}_n} \min(\text{count}_{\text{hyp}}(\text{ngram}), \max_{\text{ref}} \text{count}_{\text{ref}}(\text{ngram}))}{\sum_{\text{ngram} \in \text{hyp}_n} \text{count}_{\text{hyp}}(\text{ngram})}.$$

$w_n$ : Weights, typically  $\frac{1}{4}$  for  $n = 1, 2, 3, 4$ .

BP: The brevity penalty, defined as

$$BP = \begin{cases} 1, & \text{if } c > r \\ \exp \left( 1 - \frac{r}{c} \right), & \text{if } c \leq r \end{cases}$$

$\text{count}_{\text{hyp}}(\text{ngram})$ : Number of occurrences of the n-gram in the hypothesis (machine translation output).

$\text{count}_{\text{ref}}(\text{ngram})$ : Number of occurrences of the n-gram in the reference translation.

$c$ : Total length of the hypothesis translation.

$r$ : Total length of the reference translation.

# Task Analysis



## Evaluation Metric

他买了三本书

*Source*

Anh ấy đã mua ba cuốn sách

*Reference translation*

Hắn ta sắm 3 quyển truyện

#1 *Worst SacreBLEU score*

Ba cuốn sách được mua bởi anh ấy

#2 *Better but still bad*

Anh ấy từng mua ba cuốn sách

#3 *Better than #1 and #2*

Anh ấy đã mua ba cuốn sách

#4 *Best SacreBLEU score*

Correct translation can still lead to bad  
SacreBLEU score due to using different words

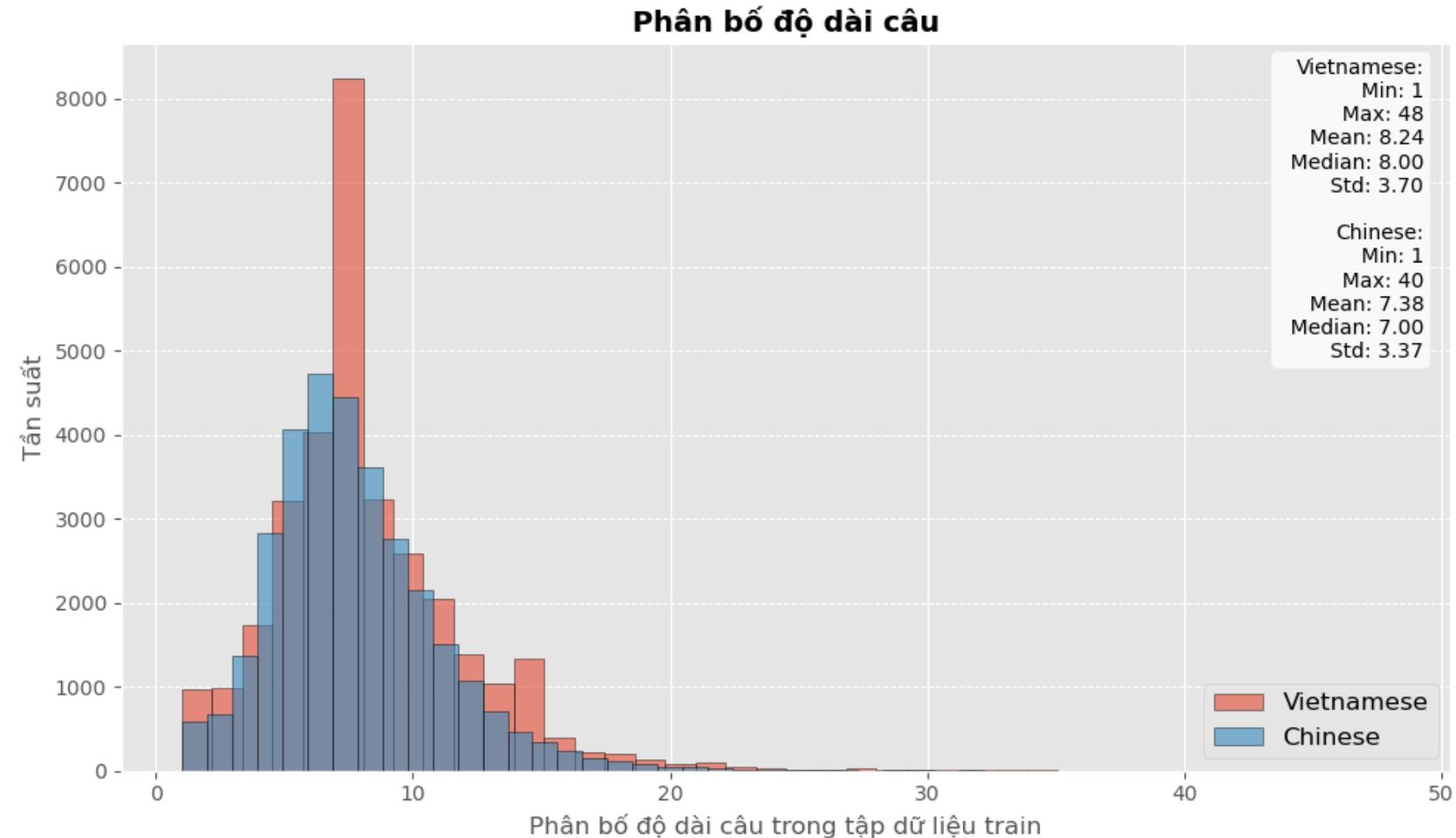
# Task Analysis



## EDA

There are 32,061 samples in the train set:  
Not too small but not too big either!

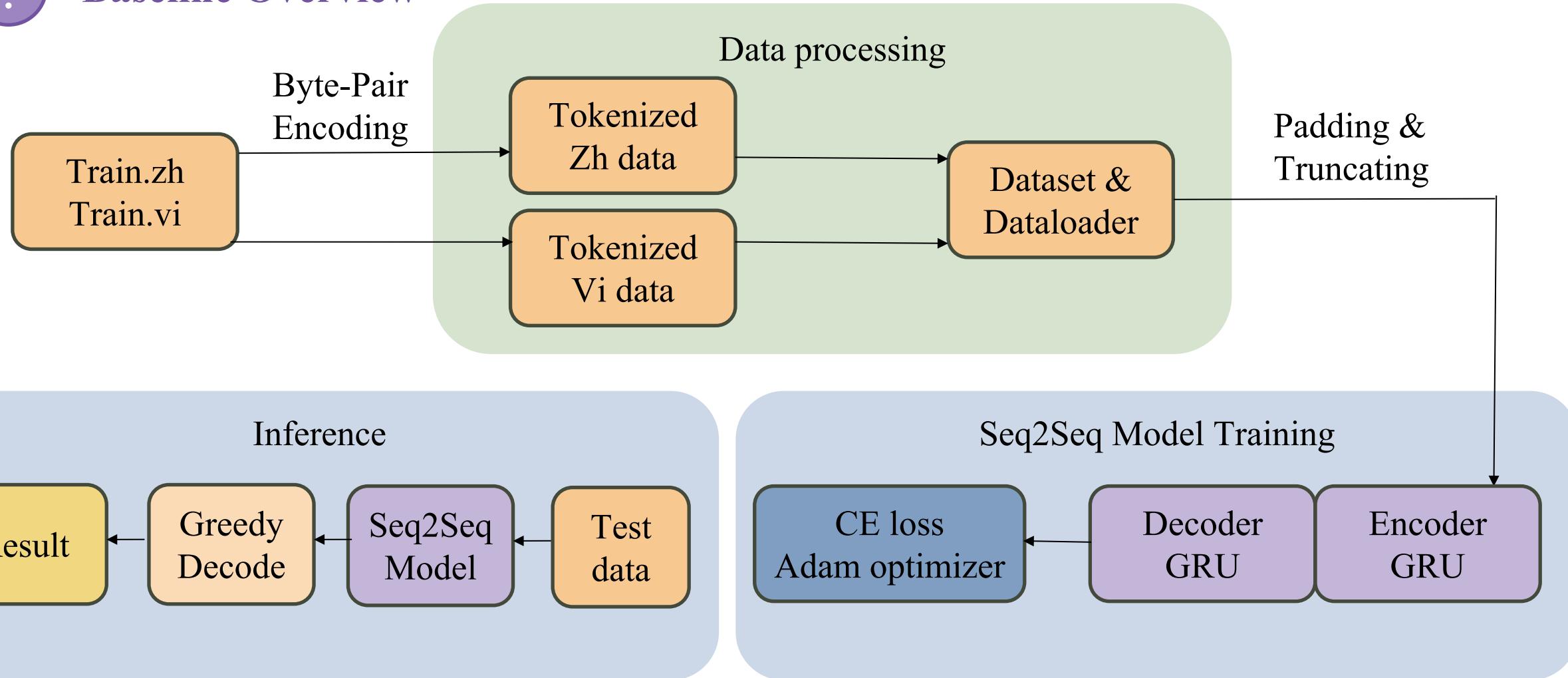
Sentence length varies



# Baseline Analysis



## Baseline Overview



# Baseline analysis



## Import libraries



google/  
**sentencepiece**

Unsupervised text tokenizer for Neural Network-based text generation.



```
import os
import random
import pandas as pd
import sentencepiece as spm
import sacrebleu
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import zipfile
from tqdm import tqdm

print(f"PyTorch version: {torch.__version__}")
print(f"CUDA available: {torch.cuda.is_available()}")
```

# Baseline analysis



## Configuration and Hyperparameters

```
# Data paths
TRAIN_SRC = "dataset/train/train.zh"
TRAIN_TGT = "dataset/train/train.vi"
TEST_SRC = "dataset/public_test/public_test.zh"
SAVE_DIR = "./checkpoints"
SPM_ZH_PREFIX = os.path.join(SAVE_DIR, "spm_zh")
SPM_VI_PREFIX = os.path.join(SAVE_DIR, "spm_vi")

# Model hyperparameters
VOCAB_SIZE = 3000
EMB_SIZE = 64
HID_SIZE = 128
BATCH_SIZE = 64
EPOCHS = 10
LR = 0.01
MAX_LEN = 80
SEED = 42
```

Relatively small!

# Baseline analysis



## Dataloading and Preprocessing

```
def read_lines(path):
    """Read lines from a text file."""
    with open(path, "r", encoding="utf-8") as f:
        return [l.strip() for l in f if l.strip()]

# Load training and test data
train_src = read_lines(TRAIN_SRC)
train_tgt = read_lines(TRAIN_TGT)
test_src = read_lines(TEST_SRC)

print(f"Training samples: {len(train_src)}")
print(f"Test samples: {len(test_src)}")
print(f"\nExample Chinese sentence: {train_src[0]}")
print(f"Example Vietnamese sentence: {train_tgt[0]}")
```

---

Training samples: 32061  
Test samples: 1781

Example Chinese sentence: 我会给您拿一些。  
Example Vietnamese sentence: Tôi sẽ mang cho bạn một ít .

# Baseline analysis



## Sentencepiece tokenization

```
def train_spm(input_file, model_prefix, vocab_size=VOCAB_SIZE):
    """Train a SentencePiece BPE model."""
    args = (
        f"--input={input_file} --model_prefix={model_prefix}"
        f"--vocab_size={vocab_size} "
        f"--model_type=bpe --character_coverage=1.0 "
        f"--pad_id=0 --unk_id=1 --bos_id=2 --eos_id=3"
    )
    spm.SentencePieceTrainer.Train(args)
    print(f"Trained SentencePiece model: {model_prefix}.model")

def load_sp(model_path):
    """Load a trained SentencePiece model."""
    sp = spm.SentencePieceProcessor()
    sp.Load(model_path)
    return sp
```

Use BPE: Good choice for Zh to Vi transaltion!

```
# Train Chinese tokenizer
tmp_zh = os.path.join(SAVE_DIR, "tmp_zh.txt")
if not os.path.exists(SPM_ZH_PREFIX + ".model"):
    with open(tmp_zh, "w", encoding="utf-8") as f:
        for s in train_src:
            f.write(s + "\n")
train_spm(tmp_zh, SPM_ZH_PREFIX)

# Train Vietnamese tokenizer
tmp_vi = os.path.join(SAVE_DIR, "tmp_vi.txt")
if not os.path.exists(SPM_VI_PREFIX + ".model"):
    with open(tmp_vi, "w", encoding="utf-8") as f:
        for s in train_tgt:
            f.write(s + "\n")
train_spm(tmp_vi, SPM_VI_PREFIX)

# Test tokenization
test_sent = train_src[0]
tokens = sp_zh.EncodeAsIds(test_sent)
print("\nExample tokenization:")
print(f"Original: {test_sent}")
print(f"Token IDs: {tokens[:20]}...")
```

Chinese vocab size: 3000  
Vietnamese vocab size: 3000

Example tokenization:  
Original: 我会给您拿一些。  
Token IDs: [5, 47, 28, 56, 109, 162, 4]...

# Baseline analysis



## Dataset and Dataloader

```
class TranslationDataset(Dataset):
    """Dataset for Chinese-Vietnamese translation pairs."""

    def __init__(self, src, tgt, sp_src, sp_tgt, max_len=MAX_LEN):
        self.src = src
        self.tgt = tgt
        self.sp_src = sp_src
        self.sp_tgt = sp_tgt
        self.max_len = max_len

    def __len__(self):
        return len(self.src)

    def __getitem__(self, idx):
        # Add BOS (2) and EOS (3) tokens
        src_ids=[2]+self.sp_src.EncodeAsIds(self.src[idx])[:self.max_len-2]+[3]
        tgt_ids=[2]+self.sp_tgt.EncodeAsIds(self.tgt[idx])[:self.max_len-2]+[3]
        return torch.tensor(src_ids), torch.tensor(tgt_ids)
```

```
def collate_fn(batch):
    """Collate function to pad sequences to the same length."""
    srcs, tgts = zip(*batch)
    max_src = max(len(s) for s in srcs)
    max_tgt = max(len(t) for t in tgts)

    # Pad with 0 (PAD token)
    src_pad = torch.zeros(len(batch), max_src, dtype=torch.long)
    tgt_pad = torch.zeros(len(batch), max_tgt, dtype=torch.long)

    for i, (s, t) in enumerate(zip(srcs, tgts)):
        src_pad[i, :len(s)] = s
        tgt_pad[i, :len(t)] = t

    return src_pad, tgt_pad
```

Padding and truncating  
are required

# Baseline analysis



## Dataset and Dataloader

```
# Create training dataset and dataloader
# Split data: 90% train, 10% validation
total_samples = len(train_src)
train_size = int(0.9 * total_samples)

train_src_split = train_src[:train_size]
train_tgt_split = train_tgt[:train_size]
valid_src = train_src[train_size:]
valid_tgt = train_tgt[train_size:]

dataset = TranslationDataset(train_src_split, train_tgt_split, sp_zh, sp_vi)
dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True,
                       collate_fn=collate_fn, num_workers=4, pin_memory=True)

valid_dataset = TranslationDataset(valid_src, valid_tgt, sp_zh, sp_vi)
valid_loader = DataLoader(valid_dataset, batch_size=32, shuffle=False,
                         collate_fn=collate_fn, num_workers=2, pin_memory=True)
```

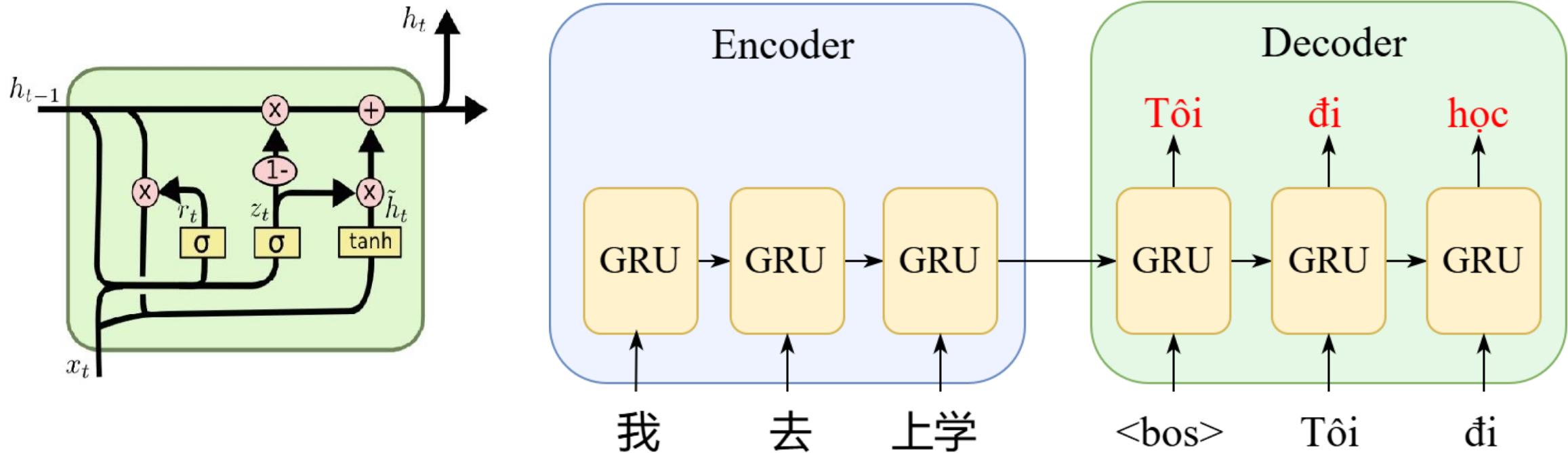
The data processing part of the baseline is simple.  
No data augmentation was used

Not good for medium-sized dataset!

# Baseline analysis



## Model Architecture



The baseline used Seq2Seq model with GRU blocks

# Baseline analysis



## Model Architecture

```
class EncoderRNN(nn.Module):
    """GRU-based encoder."""

    def __init__(self, vocab_size, emb_size, hidden_size):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, emb_size, padding_idx=0)
        self.dropout = nn.Dropout(0.3)
        self.rnn = nn.GRU(emb_size, hidden_size, batch_first=True)

    def forward(self, src):
        emb = self.dropout(self.embedding(src))
        _, hidden = self.rnn(emb)
        return hidden
```

```
class DecoderRNN(nn.Module):
    """GRU-based decoder with teacher forcing."""

    def __init__(self, vocab_size, emb_size, hidden_size):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, emb_size, padding_idx=0)
        self.dropout = nn.Dropout(0.3)
        self.rnn = nn.GRU(emb_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, input_step, hidden):
        emb = self.dropout(self.embedding(input_step))
        output, hidden = self.rnn(emb, hidden)
        pred = self.fc(output.squeeze(1))
        return pred, hidden
```

The baseline used Seq2Seq model with GRU blocks

# Baseline analysis



## Model Architecture

```
class Seq2Seq(nn.Module):
    """Sequence-to-sequence model."""
    def __init__(self, encoder, decoder):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, src, tgt, teacher_forcing_ratio=0.3):
        batch_size = src.size(0)
        tgt_len = tgt.size(1)
        vocab_size = self.decoder.fc.out_features
        # Store outputs
        outputs = torch.zeros(batch_size, tgt_len, vocab_size).to(src.device)
        # Encode source sentence
        hidden = self.encoder(src)
        # Start with BOS token
        input_step = tgt[:, 0].unsqueeze(1)
        # Decode step by step
        for t in range(1, tgt_len):
            pred, hidden = self.decoder(input_step, hidden)
            outputs[:, t] = pred
            # Teacher forcing
            teacher_force = random.random() < teacher_forcing_ratio
            input_step = tgt[:, t].unsqueeze(1) if teacher_force else pred.argmax(1).unsqueeze(1)
        return outputs
```

```
Model has 919,992 trainable parameters

Model architecture:
Seq2Seq(
    (encoder): EncoderRNN(
        (embedding): Embedding(3000, 64, padding_idx=0)
        (dropout): Dropout(p=0.3, inplace=False)
        (rnn): GRU(64, 128, batch_first=True)
    )
    (decoder): DecoderRNN(
        (embedding): Embedding(3000, 64, padding_idx=0)
        (dropout): Dropout(p=0.3, inplace=False)
        (rnn): GRU(64, 128, batch_first=True)
        (fc): Linear(in_features=128, out_features=3000, bias=True)
    )
)
```

The model is quite small and simple.  
Many better model options to improve  
baseline! (Transformer,...)

# Baseline Analysis



## Training functions

```
def train_epoch(model, dataloader, criterion, optimizer):
    """Train for one epoch."""
    model.train()
    total_loss = 0

    for src, tgt in dataloader:
        src, tgt = src.to(DEVICE), tgt.to(DEVICE)

        optimizer.zero_grad()
        output = model(src, tgt)

        # Calculate loss (ignore first BOS token)
        loss = criterion(
            output[:, 1:].reshape(-1, output.size(-1)),
            tgt[:, 1:].reshape(-1)
        )

        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        optimizer.step()

        total_loss += loss.item()

    return total_loss / len(dataloader)
```

```
@torch.no_grad()
def evaluate_bleu(model, dataloader, sp_tgt):
    """Evaluate model using SacreBLEU metric."""
    model.eval()
    hyps, refs = [], []
    pbar = tqdm(dataloader, desc="Evaluating", leave=False)
    for src, tgt in pbar:
        src = src.to(DEVICE)
        # Encode
        hidden = model.encoder(src)
        # Decode (greedy)
        input_step = torch.full((src.size(0), 1), 2, dtype=torch.long, device=DEVICE)
        decoded = [[] for _ in range(src.size(0))]
        for _ in range(MAX_LEN):
            pred, hidden = model.decoder(input_step, hidden)
            next_token = pred.argmax(1).unsqueeze(1)
            for i in range(src.size(0)):
                decoded[i].append(next_token[i].item())
            input_step = next_token
        # Convert to text
        for i in range(src.size(0)):
            ids = decoded[i]
            if 3 in ids: # Stop at EOS
                ids = ids[:ids.index(3)]
            hyps.append(sp_tgt.DecodeIds(ids))
            ref_ids = tgt[i].tolist()[1:-1] # Remove BOS and EOS
            refs.append(sp_tgt.DecodeIds([x for x in ref_ids if x not in [0, 1]]))
    bleu = sacrebleu.corpus_bleu(hyps, [refs])
    return bleu.score
```

# Baseline Analysis



## Training loop

```
# Loss function and optimizer
criterion = nn.CrossEntropyLoss(ignore_index=0) # Ignore padding
optimizer = optim.Adam(model.parameters(), lr=LR)

print("Starting training...\n")

# Training loop with best model saving
best_bleu = 0.0
best_model_path = os.path.join(SAVE_DIR, "best_model.pt")

for epoch in range(1, EPOCHS + 1):
    loss = train_epoch(model, dataloader, criterion, optimizer)
    bleu = evaluate_bleu(model, valid_loader, sp_vt)

    # Save best model
    if bleu > best_bleu:
        best_bleu = bleu
        torch.save({
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'bleu': bleu,
            'loss': loss
        }, best_model_path)
        print(f"Epoch {epoch:02d} | Loss={loss:.3f} | SacreBLEU={bleu:.2f} Best model saved!")
    else:
        print(f"Epoch {epoch:02d} | Loss={loss:.3f} | SacreBLEU={bleu:.2f}")
```

Use Cross Entropy Loss  
and Adam Optimizer

Basic training pipeline. Good but  
we can do better!

# Baseline Analysis



## Decoding stage

```
@torch.no_grad()
def translate_test(model, test_src, sp_src, sp_tgt, out_path):
    """Translate test set and save to CSV."""
    model.eval()
    outputs = []
    for s in tqdm(test_src, desc="Translating"):
        # Tokenize source sentence
        src_ids = [2] + sp_src.EncodeAsIds(s)[:MAX_LEN-2] + [3]
        src_tensor = torch.tensor(src_ids, dtype=torch.long, device=DEVICE).unsqueeze(0)
        # Encode
        hidden = model.encoder(src_tensor)
        # Decode
        input_step = torch.full((1, 1), 2, dtype=torch.long, device=DEVICE)
        decoded = []
        for _ in range(MAX_LEN):
            pred, hidden = model.decoder(input_step, hidden)
            next_token = pred.argmax(1)
            token = next_token.item()
            if token == 3: # EOS
                break
            decoded.append(token)
            input_step = next_token.unsqueeze(1)
        # Decode to Vietnamese text
        vi_sent = sp_tgt.DecodeIds(decoded)
        outputs.append(vi_sent)
    # Save to CSV
    df = pd.DataFrame({"tieng_trung": test_src, "tieng_viet": outputs})
    df.to_csv(out_path, index=False, encoding="utf-8-sig")
    return out_path
```

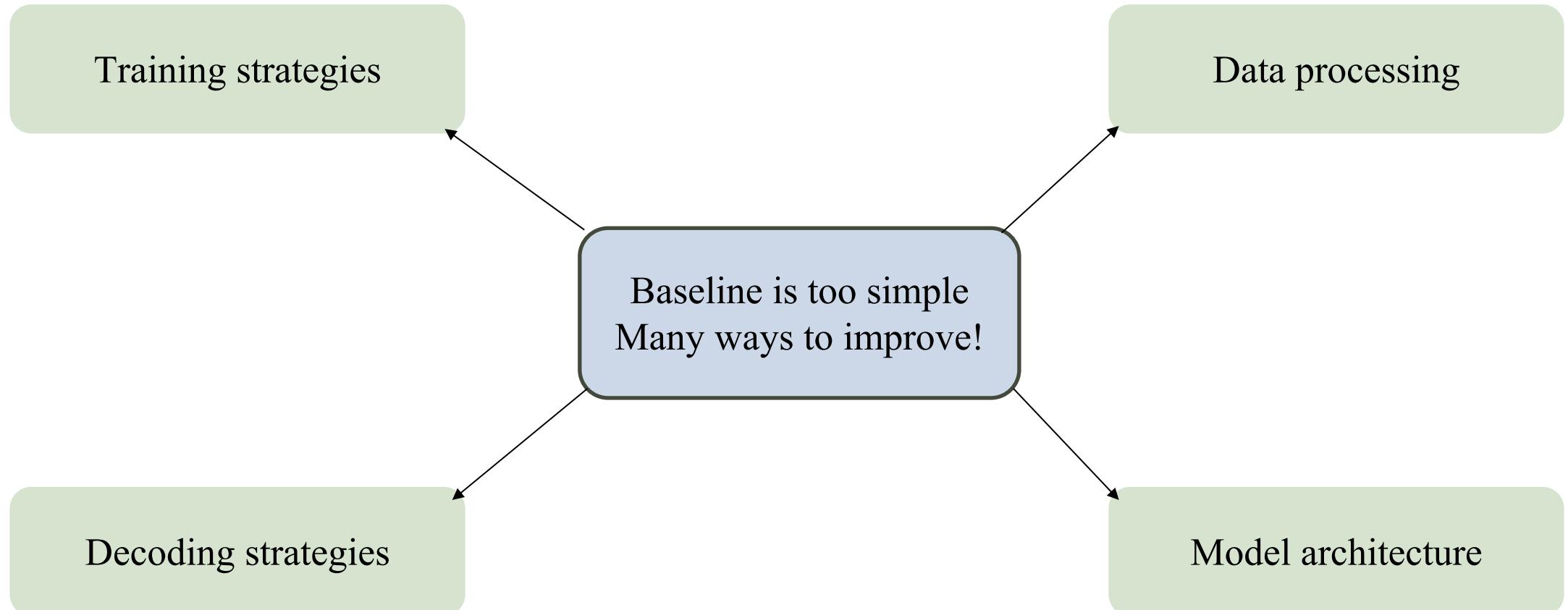
Use Greedy Decode

Many better options: Beam search

# Baseline Analysis



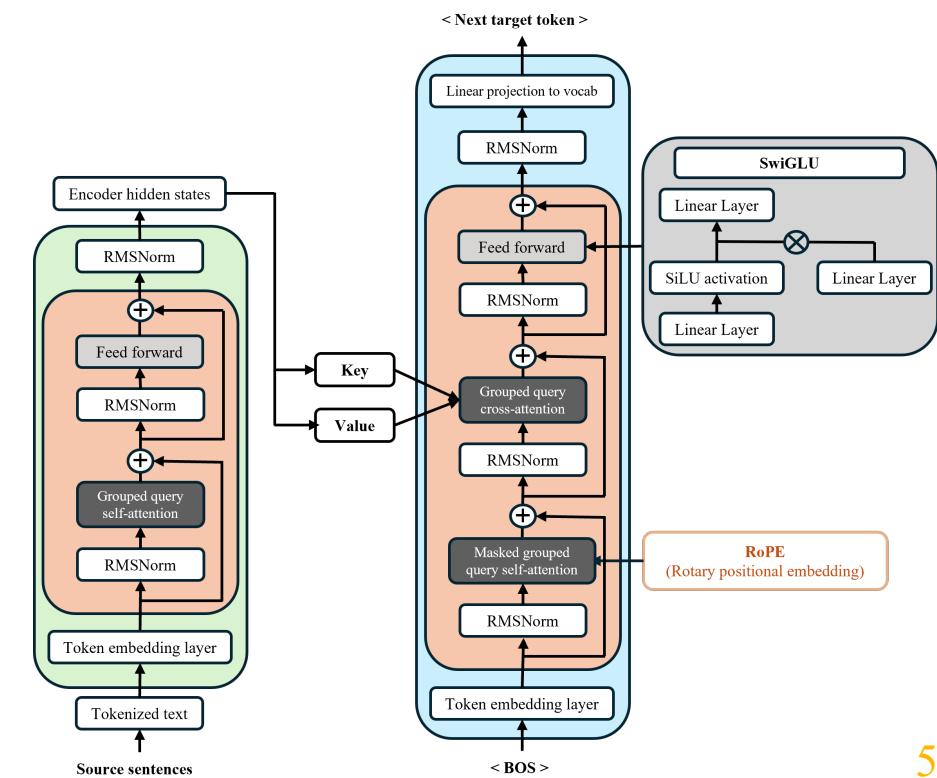
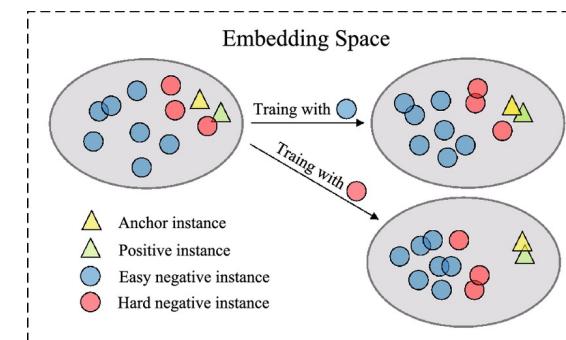
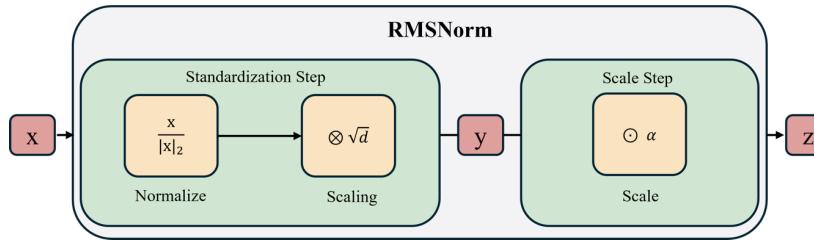
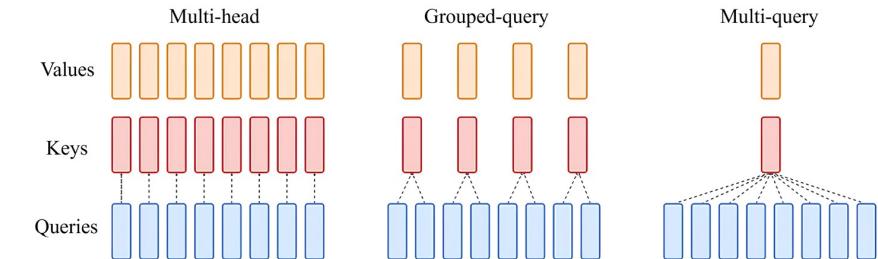
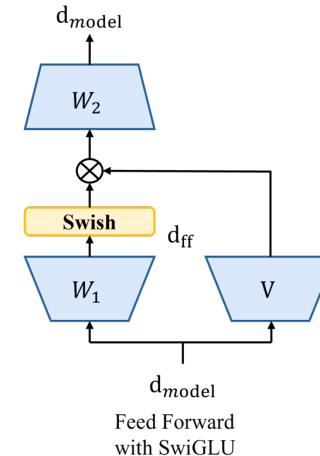
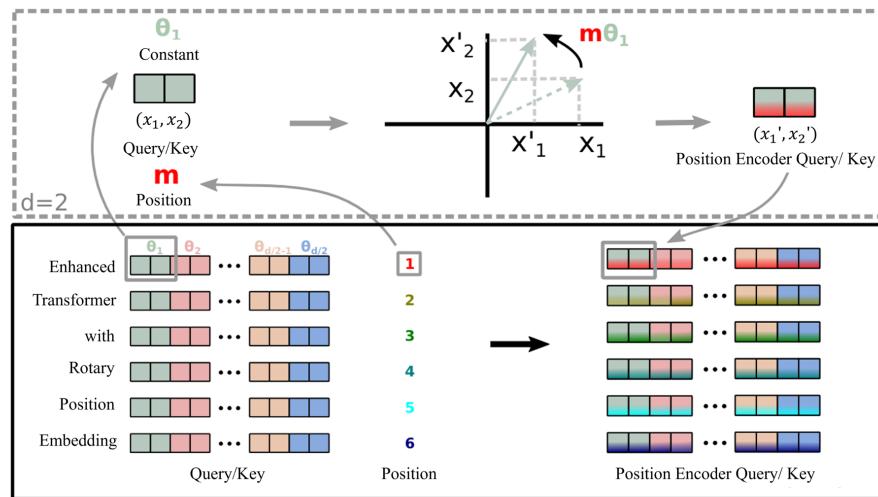
## Summary



# Model Implementation



## Overview



# Model Implementation



## Check list



### DATA EXPLORATION

- Action: Check size, alignment, sentence length and token stats on parallel CSVs.
- Lib: pandas, numpy, matplotlib, re



### SOLUTION DESIGN

- Action: Decide cleaning pipeline, bi-directional zh↔vi setup, tokenizer strategy, model and loss.
- Lib: numpy, pandas



### MODEL DESIGN

- Action: Choose encoder-decoder Transformer or decoder only, set embedding, positional encoding, heads/layers, FFN and dropout.
- Lib: torch, transformers, pytorch\_lightning



### TRAINING & EVALUATION

- Action: Build Dataset/DataLoader, run first end-to-end training, monitor loss and dev SacreBLEU for stability.
- Lib: torch, pytorch\_lightning, evaluate, tensorboard, tqdm



### IMPROVEMENTS

- Action: Try new configs, add bi-directional training, back-translation, contrastive/domain-adaptive training to boost SacreBLEU.
- Lib: torch, pytorch\_lightning, datasets, evaluate



### DEADLINE MANAGEMENT

- Action: Fix a cutoff, freeze best checkpoint, generate final translation CSV and verify format/rules.

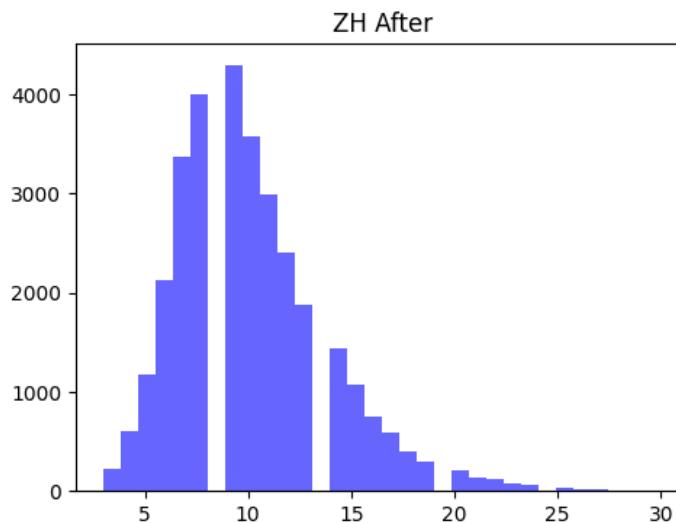


# Model Implementation

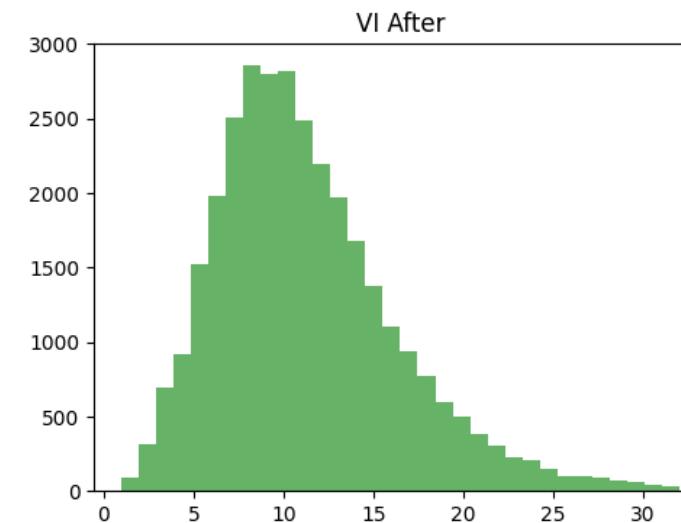
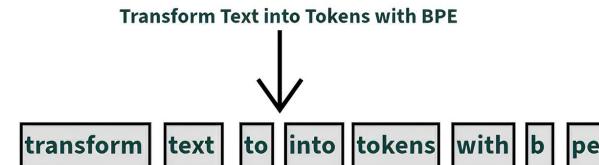


## Pre-process Data

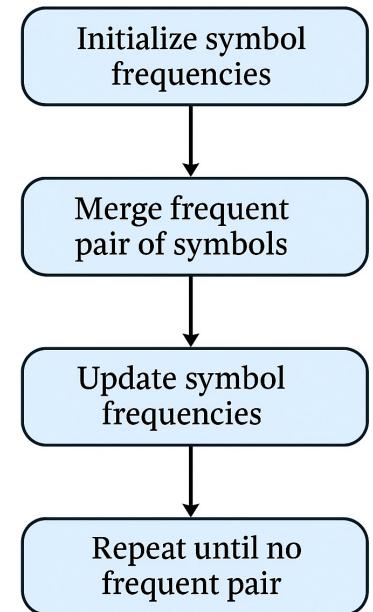
- ❖ Merge all sentences and train a shared SentencePiece BPE tokenizer (vocab size 8000)
- ❖ Tokenize each sentence and filter out pairs where the source or target exceeds 32 tokens.



## Tokenization in Transformer Byte Pair Encoding



## Byte-Pair Encoding Algorithm

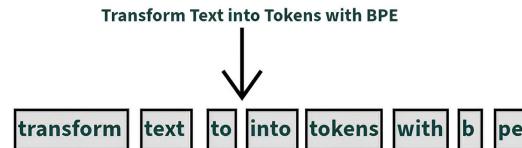


# Model Implementation

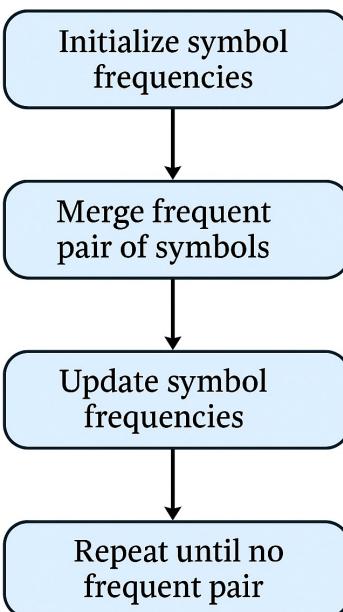


## Pre-process Data

### Tokenization in Transformer Byte Pair Encoding



### Byte-Pair Encoding Algorithm



```
import sentencepiece as spm

TOKENIZER_DIR = os.path.join(BASE_DIR, "tokenizer_train32")
TOKENIZER_PREFIX = os.path.join(TOKENIZER_DIR, "spm_zh_vi_joint")
VOCAB_SIZE = 8000
USER_SYMBOLS = ["<2zh>", "<2vi>"]

temp_corpus = os.path.join(TOKENIZER_DIR, "temp_corpus.txt")
with open(temp_corpus, "w", encoding="utf-8") as fout:
    for zh, vi in zip(zh_lines, vi_lines):
        fout.write(zh + "\n")
        fout.write(vi + "\n")

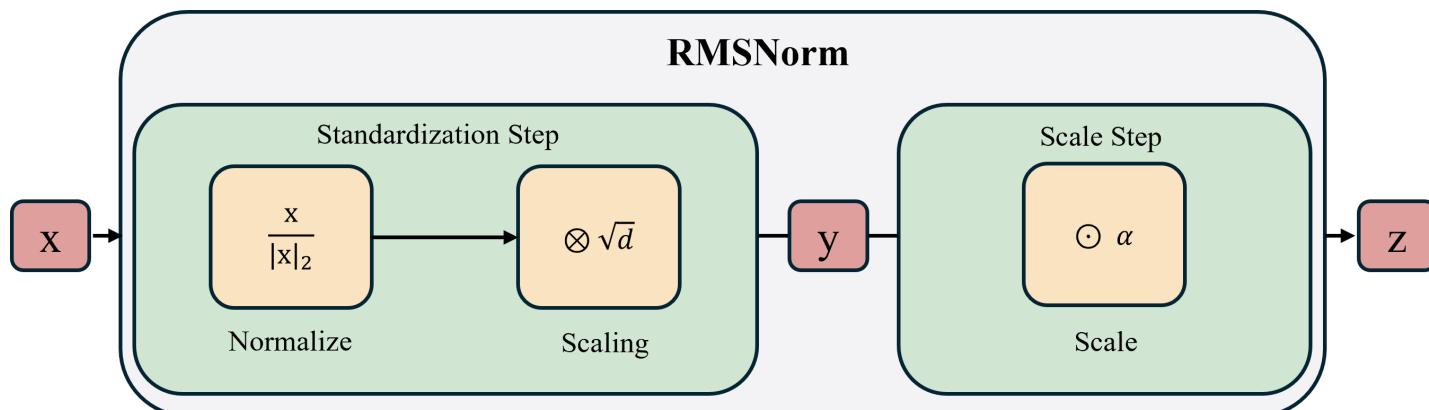
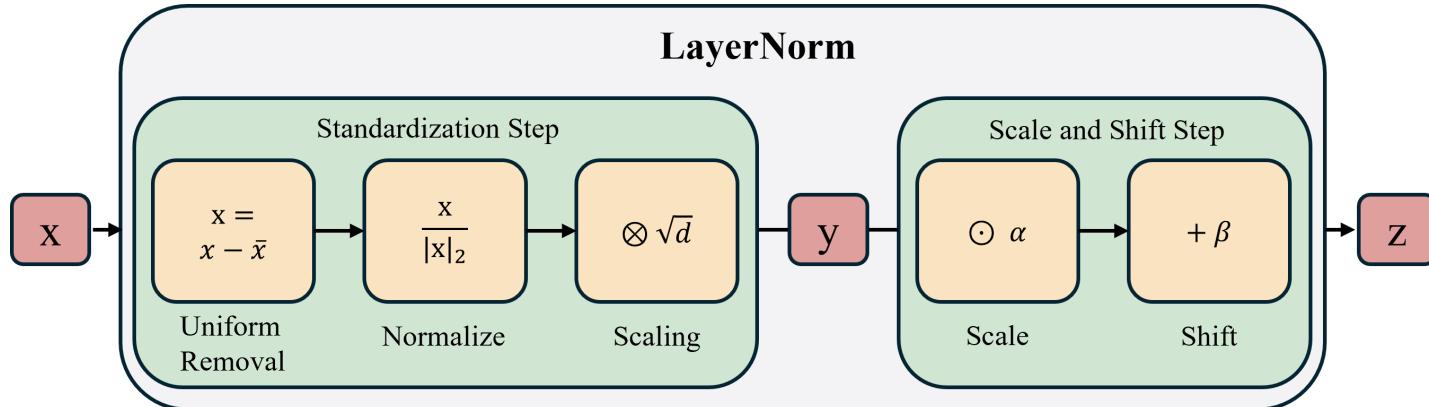
spm_args = (
    f"--input={temp_corpus} ",
    f"--model_prefix={TOKENIZER_PREFIX} ",
    f"--vocab_size={VOCAB_SIZE} ",
    f"--model_type=bpe ",
    f"--character_coverage=1.0 ",
    "--pad_id=0 --unk_id=1 --bos_id=2 --eos_id=3 ",
    f"--user_defined_symbols={','.join(USER_SYMBOLS)}"
)
spm.SentencePieceTrainer.Train(' '.join(spm_args))

print(f"Tokenizer saved to {TOKENIZER_PREFIX}.model")
Tokenizer saved to ./tokenizer_train32/spm_zh_vi_joint.model
```

# Model Implementation



## Transformer Encoder-Decoder

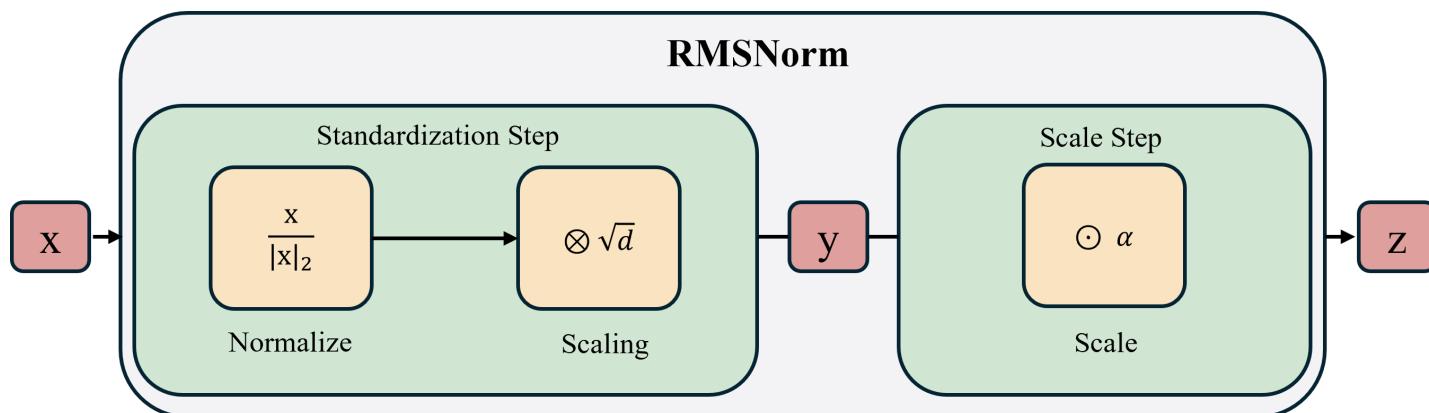
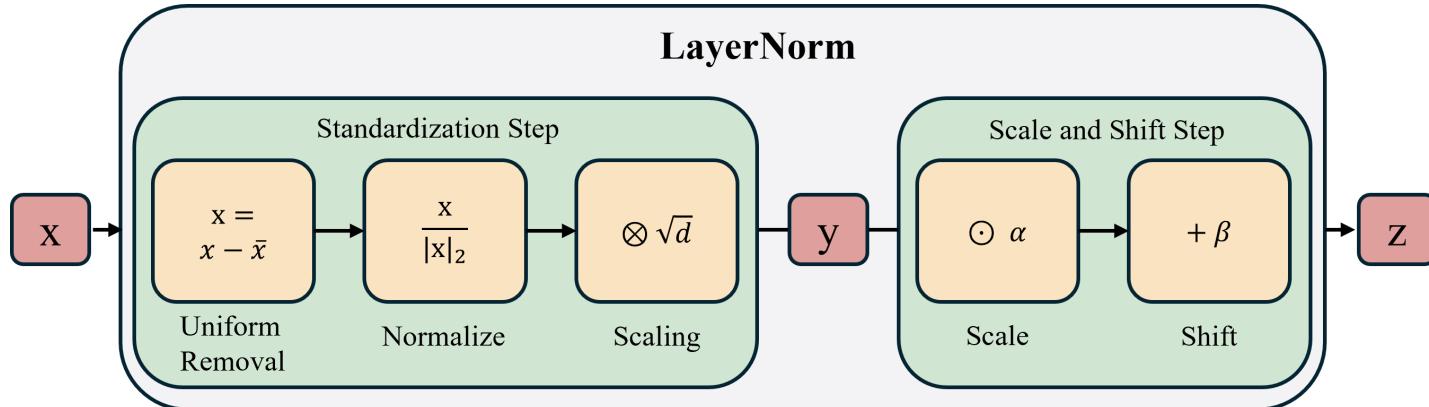


- ❖ LayerNorm subtracts the mean, normalizes the vector, and applies scale and shift.
- ❖ It fully standardizes features for stable training.
- ❖ RMSNorm skips mean subtraction and normalizes only by RMS.
- ❖ It uses just a scale parameter, making it simpler and faster.

# Model Implementation



## Transformer Encoder-Decoder



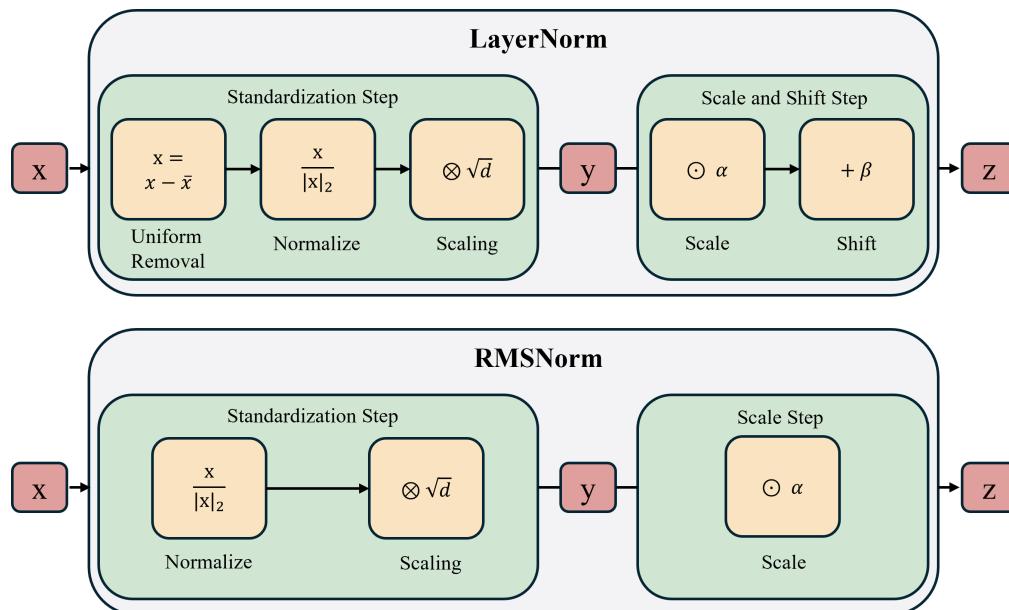
- ❖ LayerNorm is complete but adds extra computation and can destabilize deep models.
- ❖ Its mean-centering step is costly at large scale.
  
- ❖ RMSNorm is cheaper and more numerically stable for large transformers.
- ❖ That's why modern LLMs (LLaMA, Qwen, Mistral) prefer RMSNorm.

# Model Implementation



## Transformer Encoder-Decoder

Code implementation:



```
class RMSNorm(nn.Module):
    def __init__(self, d_model: int, eps: float = 1e-8):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(d_model))

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        rms = torch.sqrt(
            torch.mean(x ** 2, dim=-1, keepdim=True) + self.eps
        )
        return self.weight * x / rms
```

INPUT:

$x$  shape: `torch.Size([4, 16, 768])`

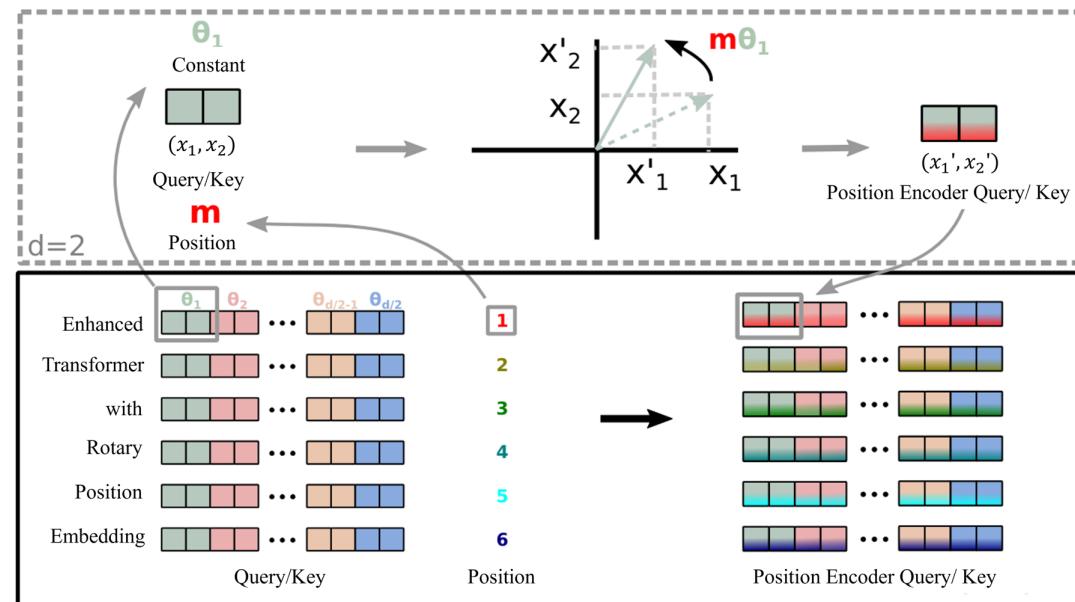
OUTPUT:

normed shape: `torch.Size([4, 16, 768])` # Same shape  
RMS before: 24.3692  
RMS after: 0.8750

# Model Implementation



## Transformer Encoder-Decoder

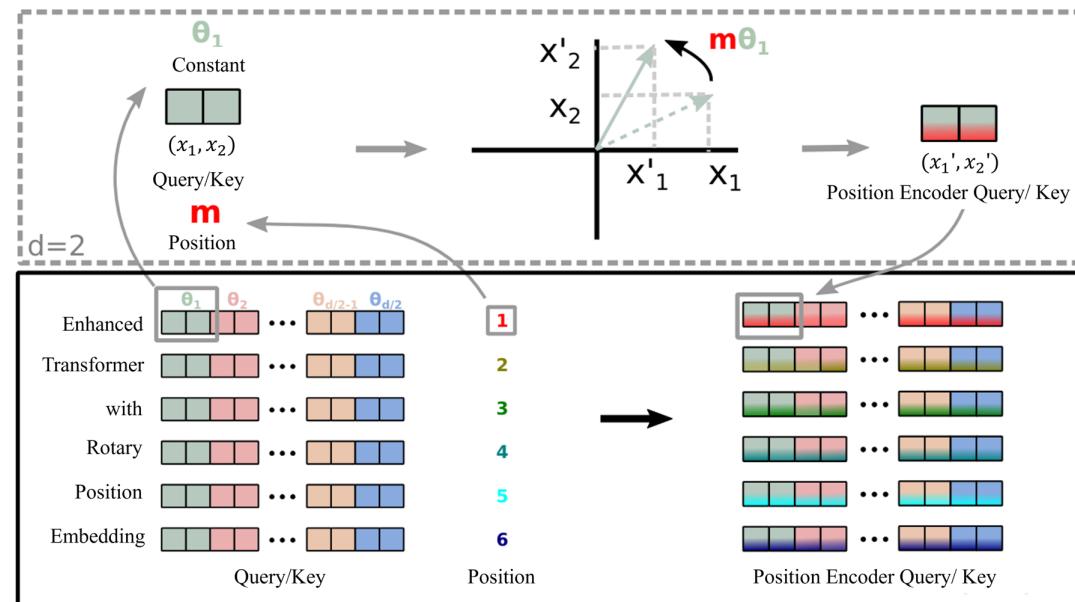


- ❖ Absolute PE: Add a learned position vector to each token.
- ❖ Sinusoidal PE: Fixed sin–cos patterns that encode position.
- ❖ RoPE: Rotate Q and K by an angle based on token index → position is baked into the dot product.

# Model Implementation



## Transformer Encoder-Decoder



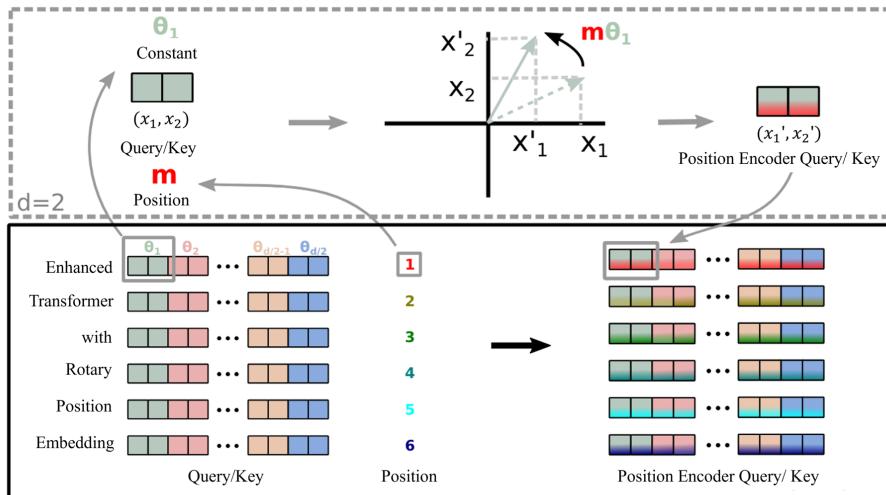
- ❖ Pros: Relative position awareness, better long-context generalization, almost no extra parameters.
- ❖ Cons: Implementation a bit more complex than simple additive PE.
- ❖ Result: Modern LLMs adopt RoPE because it gives strong performance and stable training for long sequences.

# Model Implementation



## Transformer Encoder-Decoder

Code implementation:



```

class RoPE(nn.Module):
    def __init__(self, d_model: int, base: float = 10000.0):
        super().__init__()
        inv_freq = 1.0 / (base ** (torch.arange(0, d_model, 2).float() / d_model))
        self.register_buffer("inv_freq", inv_freq)
        self._cos = None
        self._sin = None
        self._seq_len_cached = 0

    def _maybe_update_cache(self, seq_len: int, device, dtype):
        if seq_len > self._seq_len_cached or self._cos is None or self._cos.device != device:
            self._seq_len_cached = seq_len
            positions = torch.arange(seq_len, device=device, dtype=dtype)
            freqs = torch.outer(positions, self.inv_freq.to(device))
            self._cos = freqs.cos()
            self._sin = freqs.sin()

    def forward(self, x: torch.Tensor, seq_len: Optional[int] = None):
        seq_len = seq_len or x.size(-2)
        self._maybe_update_cache(seq_len, x.device, x.dtype)
        return self._cos[:seq_len], self._sin[:seq_len]

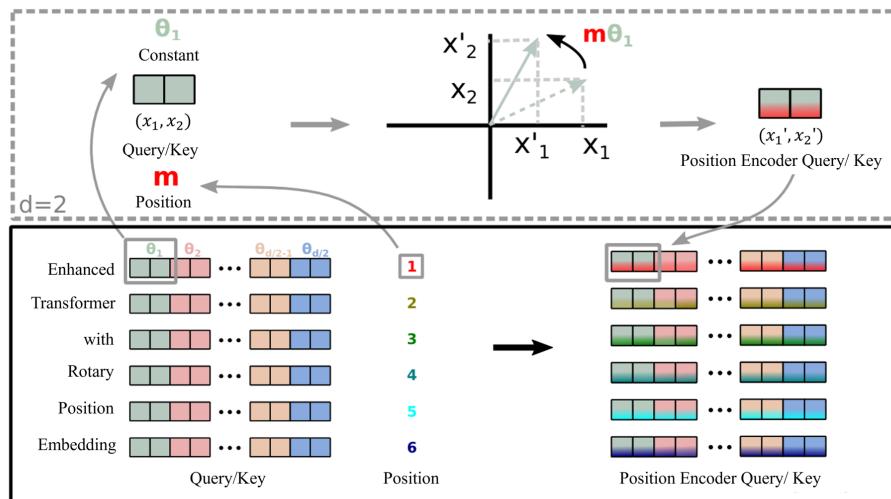
```

# Model Implementation



## Transformer Encoder-Decoder

Code implementation:



INPUT:

```
Q shape: torch.Size([4, 12, 16, 64]) # (batch, heads, seq, d_k)
```

OUTPUT:

```
cos shape: torch.Size([16, 32]) # (seq, d_k/2)
sin shape: torch.Size([16, 32]) # (seq, d_k/2)
Q_rope shape: torch.Size([4, 12, 16, 64]) # Same as input
RoPE base: 10000.0
```

```
def apply_rope(x: torch.Tensor, cos: torch.Tensor, sin: torch.Tensor) -> torch.Tensor:
    """Áp dụng RoPE lên tensor (batch, heads, seq, d_k)"""
    x1 = x[..., 0::2]
    x2 = x[..., 1::2]
    cos = cos.unsqueeze(0).unsqueeze(0)
    sin = sin.unsqueeze(0).unsqueeze(0)
    rot1 = x1 * cos - x2 * sin
    rot2 = x1 * sin + x2 * cos
    return torch.stack([rot1, rot2], dim=-1).flatten(-2)

# Demo
d_k = d_model // n_heads # 64
rope = RoPE(d_k, base=rope_base).to(device)

# Giả sử Q sau linear projection
Q = torch.randn(batch_size, n_heads, src_len, d_k).to(device)

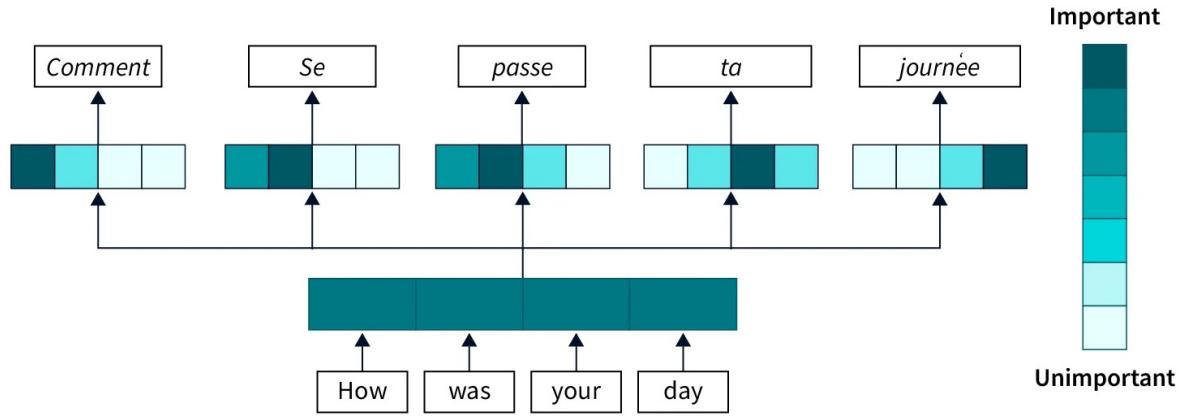
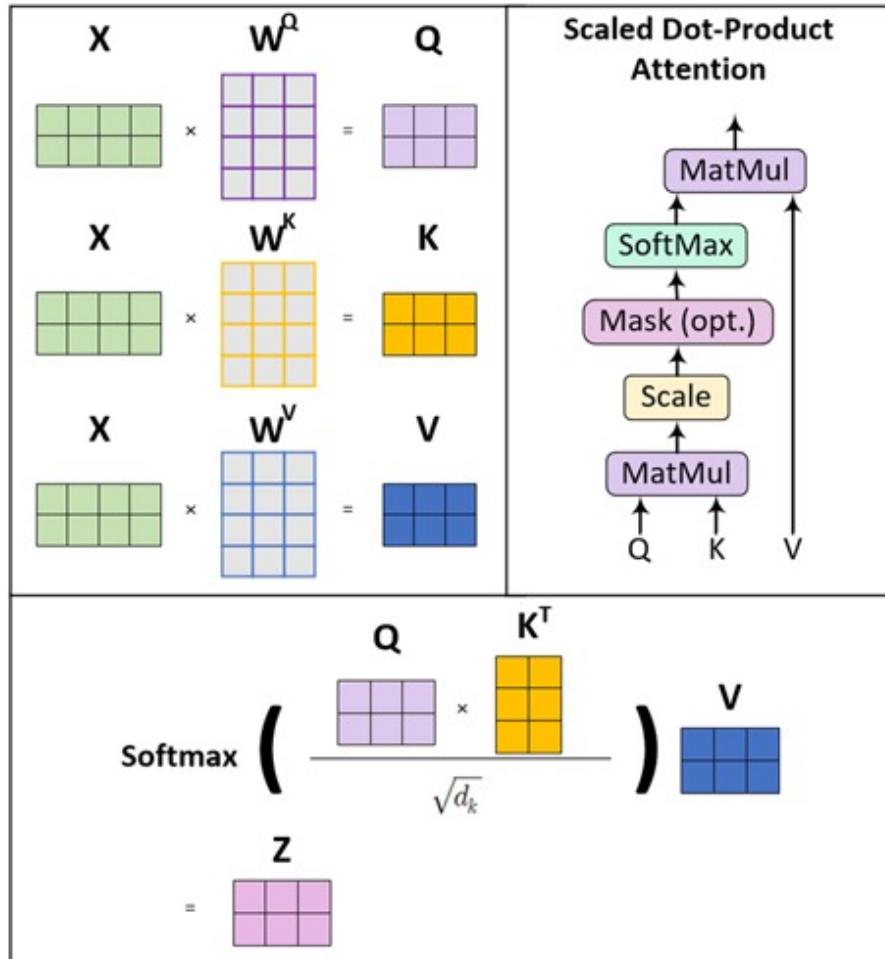
print("INPUT:")
print(f" Q shape: {Q.shape} # (batch, heads, seq, d_k)")

cos, sin = rope(Q, src_len)
Q_rope = apply_rope(Q, cos, sin)
```

# Model Implementation



## Transformer Encoder-Decoder

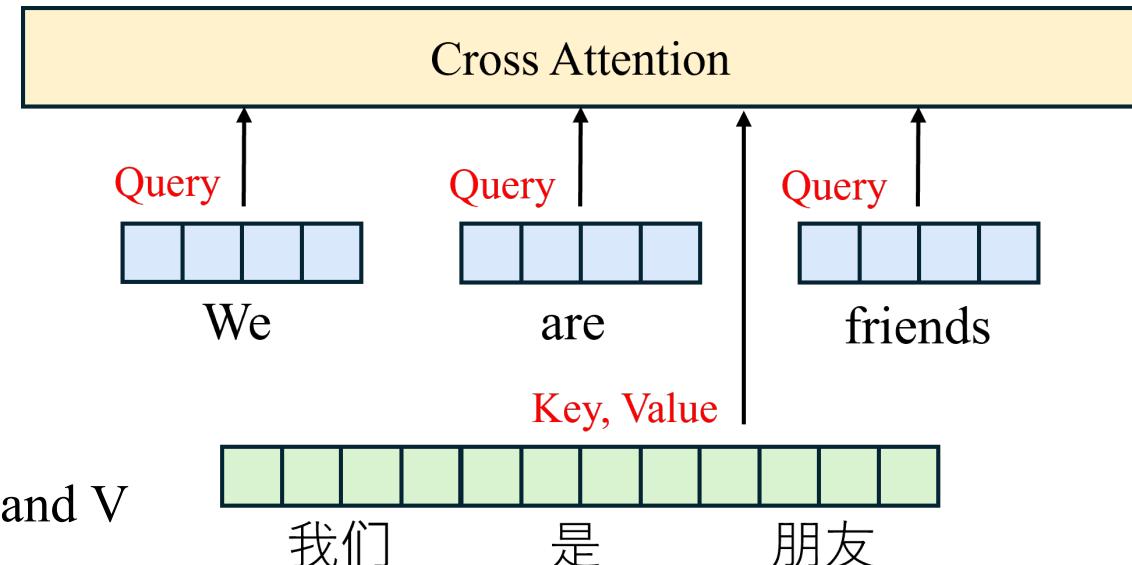
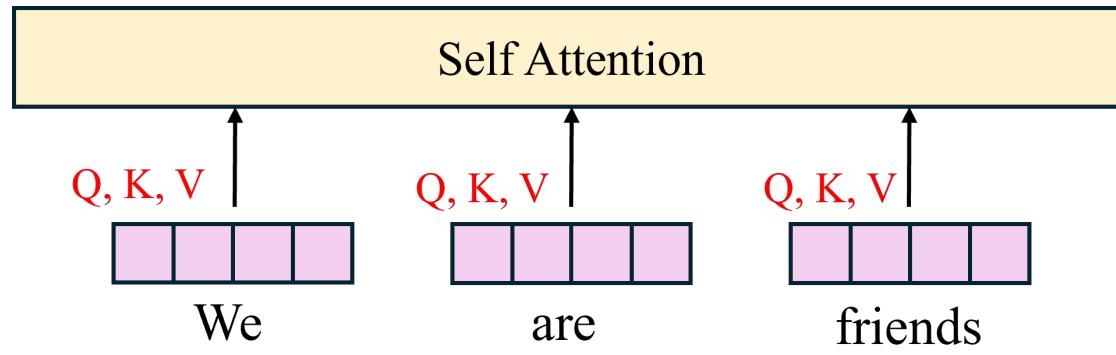


- ❖ Attention computes relevance between tokens.
- ❖ Important tokens get higher weights.
- ❖ Multi-heads capture different types of relationships.
- ❖ Outputs are context-aware representations.

# Model Implementation



## Transformer Encoder-Decoder

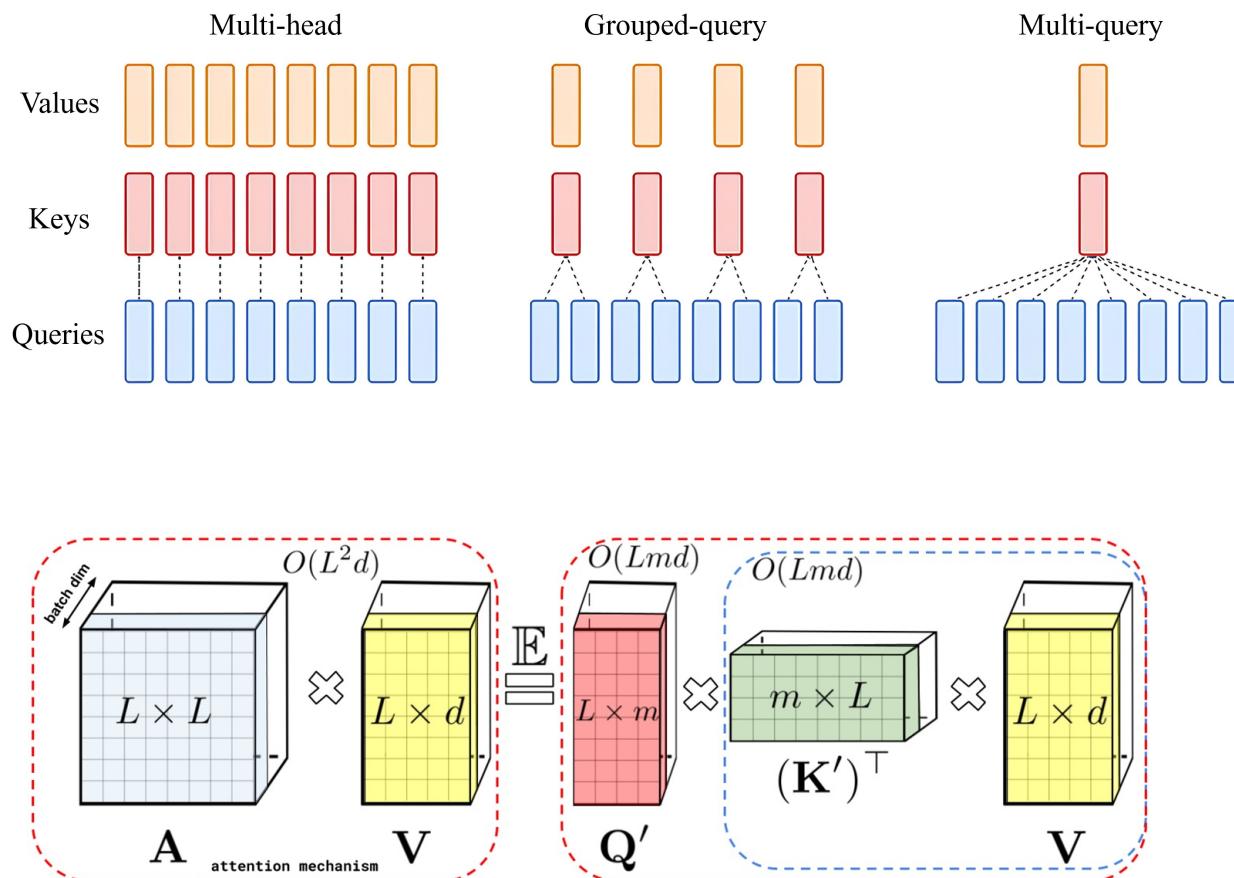


- ❖ In self-attention, each token generates its own Q, K and V from the same sequence.
- ❖ Every token attends to all other tokens in that sequence.
  
- ❖ In cross-attention, Queries come from the decoder tokens (target sentence).
- ❖ Keys and Values come from the encoder outputs (source sentence).

# Model Implementation



## Transformer Encoder-Decoder



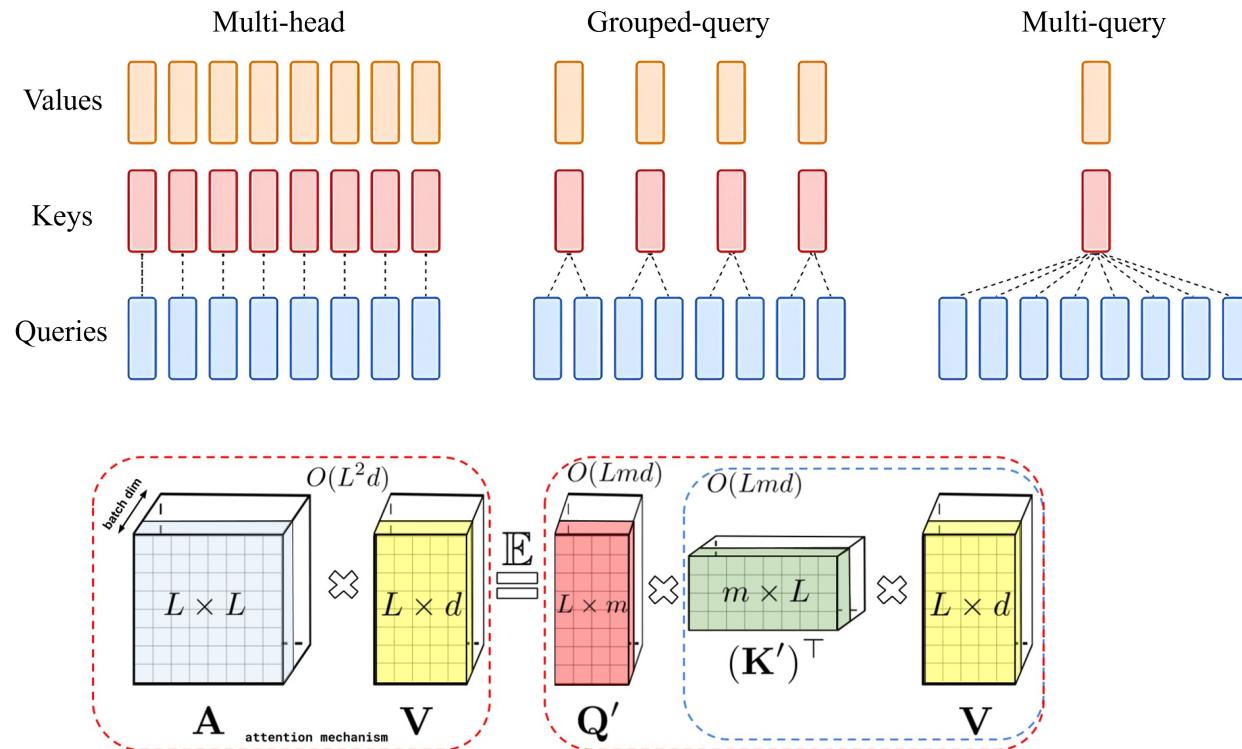
- ❖ Multi-head attention is flexible but expensive in memory and KV cache size.
- ❖ Multi-query attention is very efficient but may lose representation diversity.
- ❖ Grouped-query attention provides a middle ground: lower memory like MQA, but with richer attention patterns closer to MHA.
- ❖ Modern LLMs often choose GQA because it balances quality and efficiency, especially for fast inference.

# Model Implementation



## Transformer Encoder-Decoder

Code implementation:



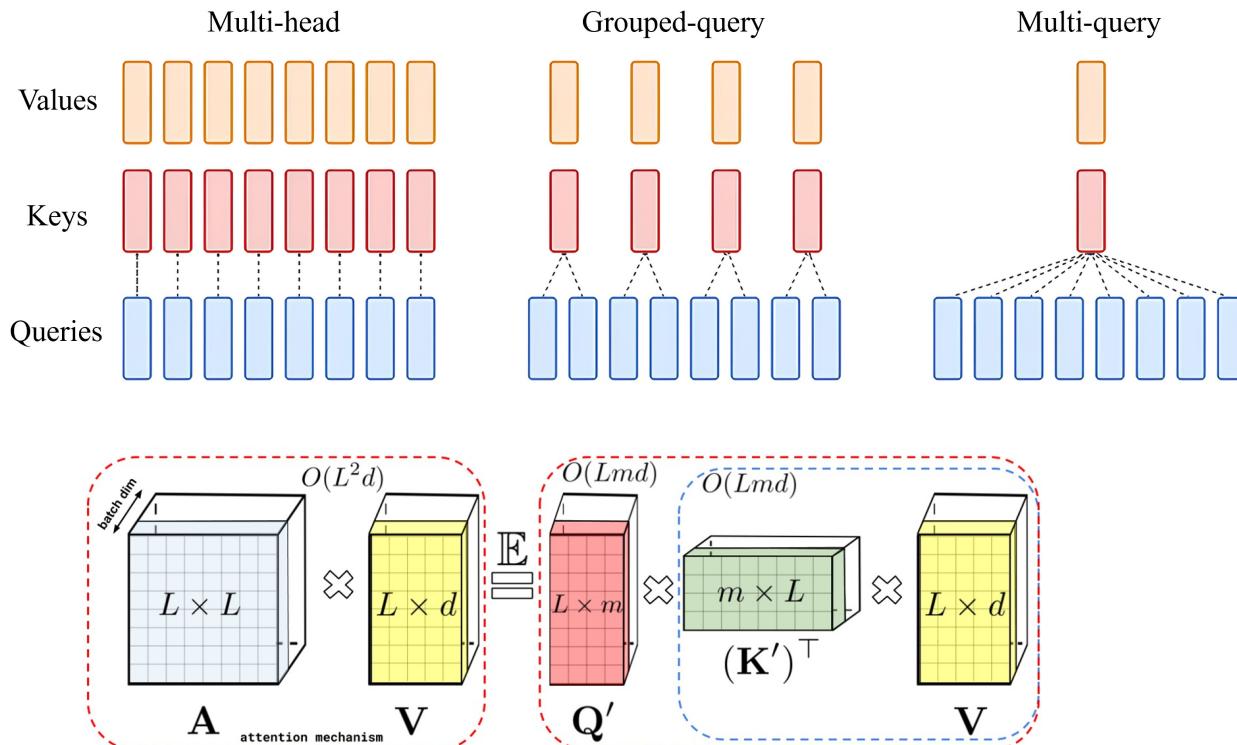
```
class GroupedQueryAttentionRoPE(nn.Module):
    def __init__(self,
                 d_model: int, n_heads: int, n_kv_heads: int,
                 dropout: float, rope_base: float
                 ):
        super().__init__()
        assert n_heads % n_kv_heads == 0
        self.n_heads = n_heads
        self.n_kv_heads = n_kv_heads
        self.n_groups = n_heads // n_kv_heads
        self.d_k = d_model // n_heads
        self.W_q = nn.Linear(d_model, n_heads * self.d_k, bias=False)
        self.W_k = nn.Linear(d_model, n_kv_heads * self.d_k, bias=False)
        self.W_v = nn.Linear(d_model, n_kv_heads * self.d_k, bias=False)
        self.W_o = nn.Linear(d_model, d_model, bias=False)
        self.dropout = nn.Dropout(dropout)
        self.scale = math.sqrt(self.d_k)
        self.rope = RoPE(self.d_k, base=rope_base)
```

# Model Implementation



## Transformer Encoder-Decoder

Code implementation:



```

def forward(self, q, k, v, key_padding_mask=None, attn_mask=None):
    B, T_q = q.size(0), q.size(1)
    T_k = k.size(1)
    Q = self.W_q(q).view(B, T_q, self.n_heads, self.d_k).transpose(1, 2)
    K = self.W_k(k).view(B, T_k, self.n_kv_heads, self.d_k).transpose(1, 2)
    V = self.W_v(v).view(B, T_k, self.n_kv_heads, self.d_k).transpose(1, 2)

    # Apply RoPE
    cos_q, sin_q = self.rope(Q, T_q)
    cos_k, sin_k = self.rope(K, T_k)
    Q = apply_rope(Q, cos_q, sin_q)
    K = apply_rope(K, cos_k, sin_k)

    # Repeat KV heads
    K = K.repeat_interleave(self.n_groups, dim=1)
    V = V.repeat_interleave(self.n_groups, dim=1)

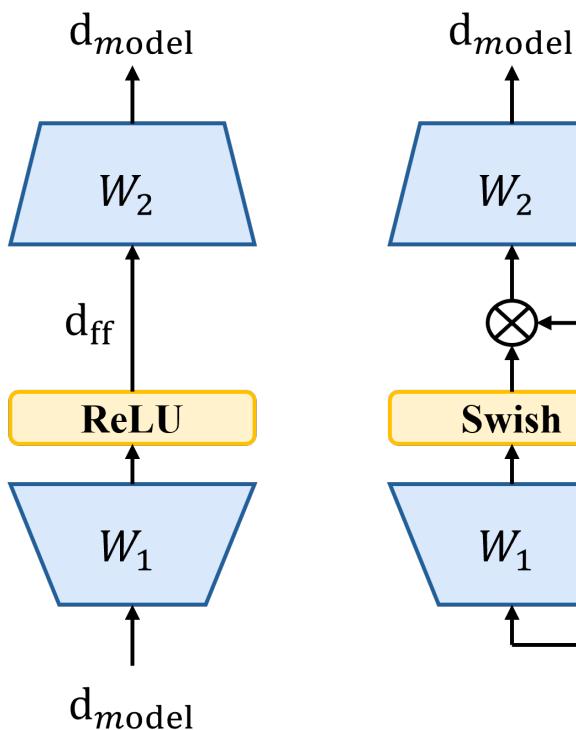
    scores = torch.matmul(Q, K.transpose(-2, -1)) / self.scale
    if key_padding_mask is not None:
        scores = scores.masked_fill(
            key_padding_mask.unsqueeze(1).unsqueeze(2), float("-inf"))
    )
    if attn_mask is not None:
        scores = scores.masked_fill(
            attn_mask.unsqueeze(0).unsqueeze(0), float("-inf"))
    )
    attn = F.softmax(scores, dim=-1)
    attn = self.dropout(attn)
    out = torch.matmul(attn, v)
    out = out.transpose(1, 2).contiguous().view(B, T_q, -1)
    return self.W_o(out)

```

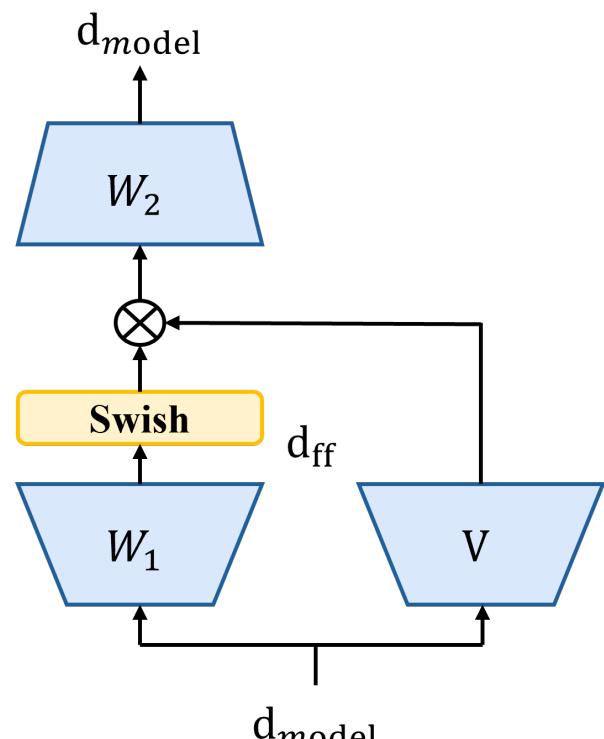
# Model Implementation



## Transformer Encoder-Decoder



Original Feed  
Forward layer



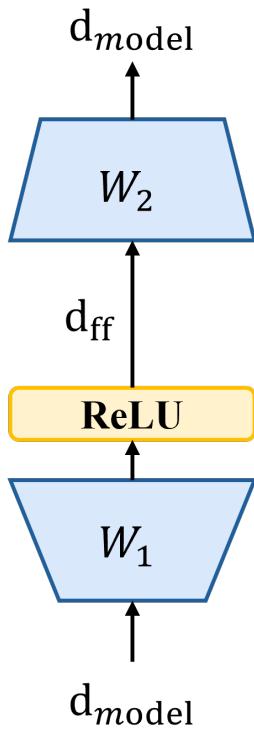
Feed Forward  
with SwiGLU

- ❖ ReLU-based feed-forward layers are simple and efficient, but they zero out all negative activations.
- ❖ This can lead to "dead" neurons and piecewise-linear representations that are sometimes too rigid for subtle semantic patterns.
- ❖ SwiGLU replaces ReLU with a smooth Swish nonlinearity and a multiplicative gate.

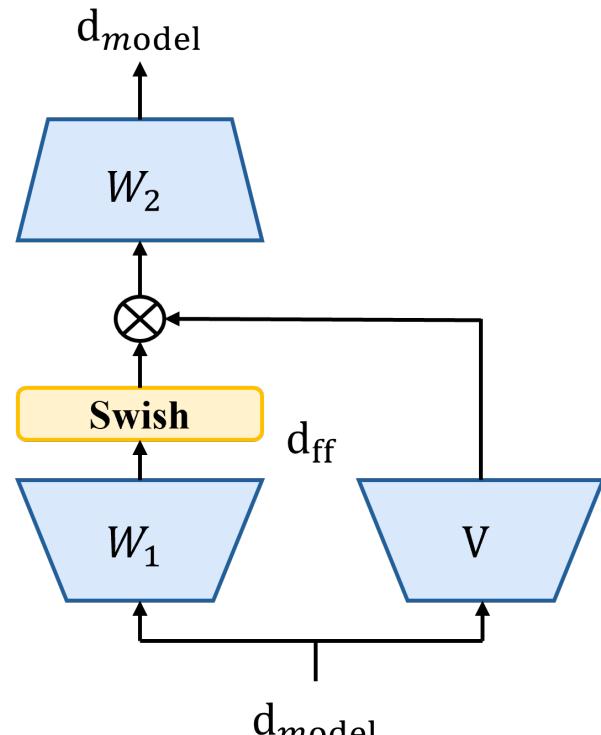
# Model Implementation



## Transformer Encoder-Decoder



Original Feed  
Forward layer



Feed Forward  
with SwiGLU

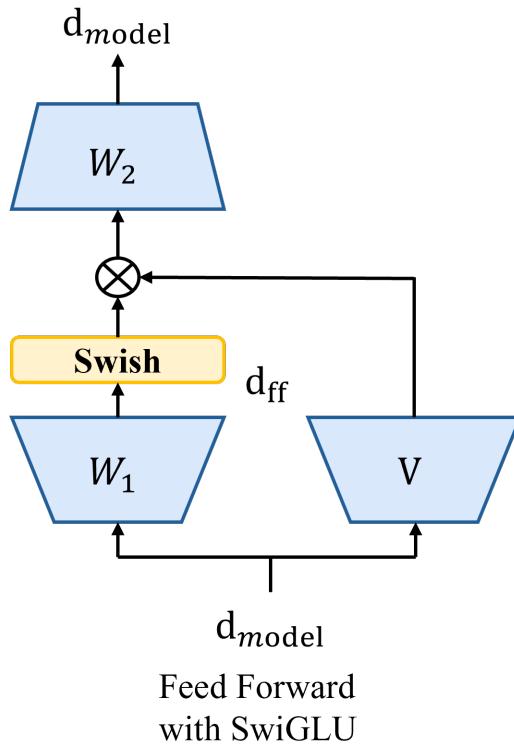
- ❖ It preserves more information, provides smoother gradients, and yields richer feature transformations, which empirically benefits large language models.
- ❖ SwiGLU gives smoother gradients and stronger non-linearity than ReLU.
- ❖ It improves model expressiveness and perplexity with similar or slightly higher cost.

# Model Implementation



## Transformer Encoder-Decoder

Code implementation:



```
class FFN_SwiGLU(nn.Module):
    def __init__(self, d_model: int, d_ff: int, dropout: float):
        super().__init__()
        self.d_ff = d_ff
        self.linear1 = nn.Linear(d_model, 2 * d_ff, bias=False)
        self.linear2 = nn.Linear(d_ff, d_model, bias=False)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        h = self.linear1(x)
        g, v = h[..., : self.d_ff], h[..., self.d_ff :]
        s = g * torch.sigmoid(g) # SwiGLU
        out = self.linear2(s * v)
        return self.dropout(out)
```

INPUT:

`x shape: torch.Size([4, 16, 768]) # (batch, seq, d_model=768)`

FFN: 768 → 6144 → 3072 → 768

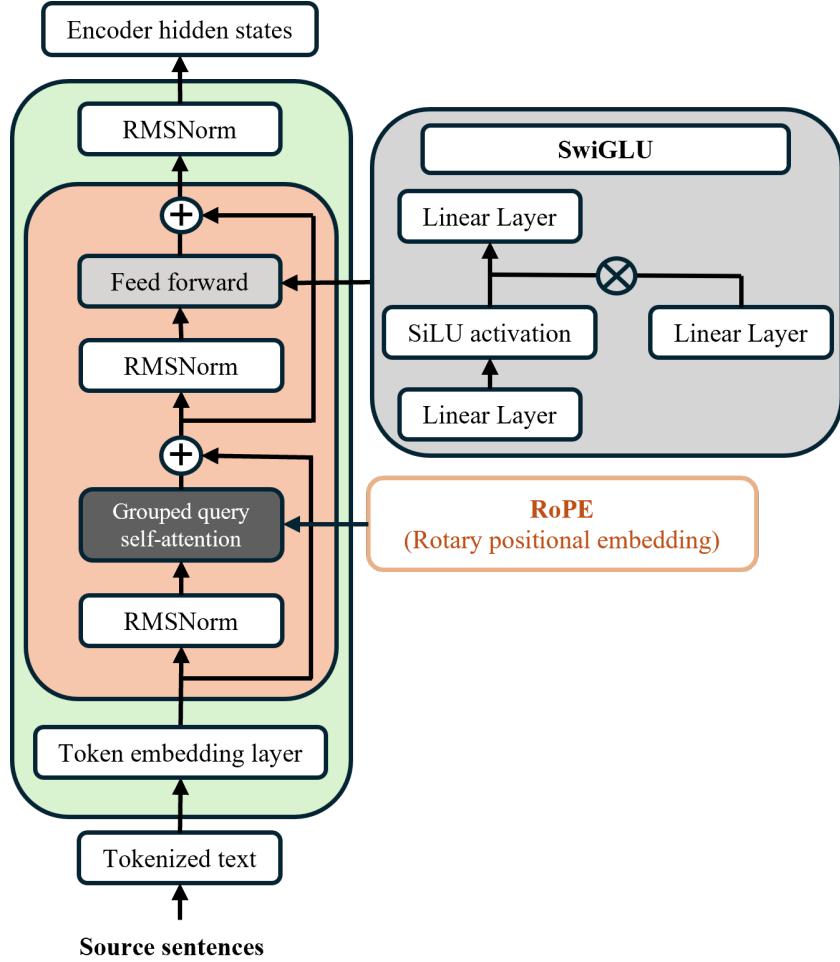
OUTPUT:

`ffn_out shape: torch.Size([4, 16, 768]) # (batch, seq, d_model)`

# Model Implementation



## Transformer Encoder-Decoder



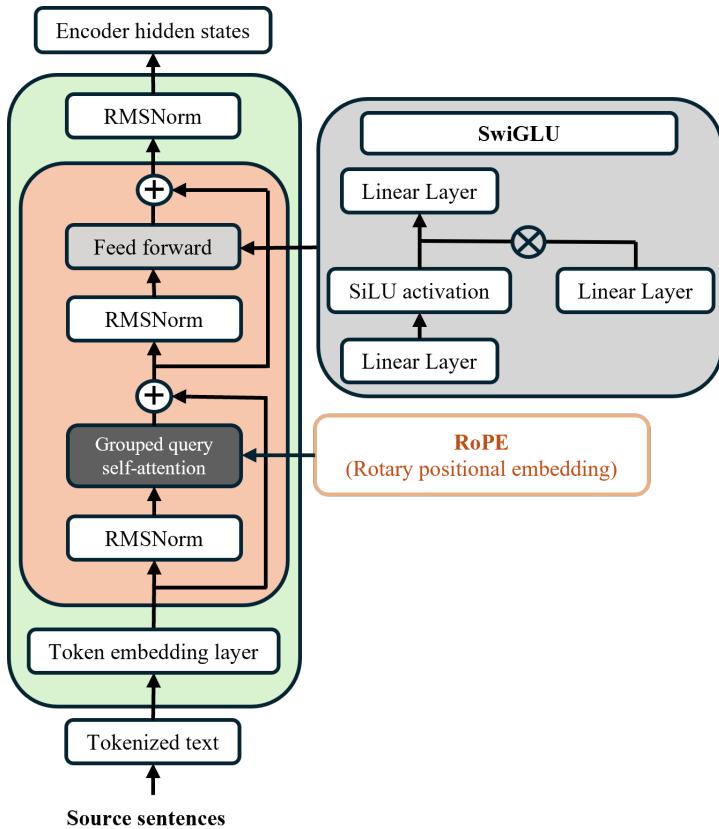
- ❖ Encoder stacks RMSNorm, grouped-query self-attention with RoPE, and SwiGLU feed-forward on top of the token embeddings.
- ❖ It takes the source sentence as input and produces context-aware hidden vectors for each token.
- ❖ These vectors encode the sentence meaning and are passed to the decoder.

# Model Implementation



## Transformer Encoder-Decoder

Code implementation:



```
class EncoderLayer(nn.Module):
    def __init__(self, d_model, n_heads, n_kv_heads, d_ff, dropout, rope_base):
        super().__init__()
        self.ln1 = RMSNorm(d_model)
        self.self_attn = GroupedQueryAttentionRoPE(
            d_model, n_heads, n_kv_heads, dropout, rope_base
        )
        self.ln2 = RMSNorm(d_model)
        self.ffn = FFN_SwiGLU(d_model, d_ff, dropout)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, src_pad_mask=None):
        x1 = self.ln1(x)
        attn = self.self_attn(x1, x1, x1, key_padding_mask=src_pad_mask)
        x = x + self.dropout(attn)
        x2 = self.ln2(x)
        return x + self.ffn(x2)
```

INPUT:

```
x shape: torch.Size([4, 16, 768])
src_pad_mask shape: torch.Size([4, 16])
```

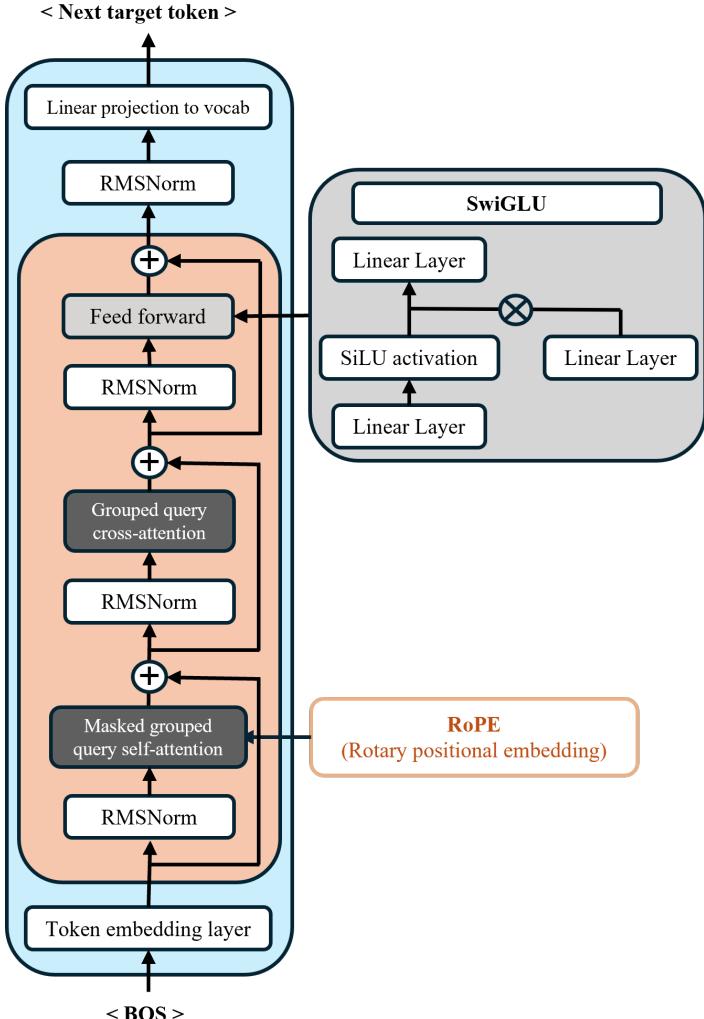
OUTPUT:

```
enc_out shape: torch.Size([4, 16, 768]) # Same shape
→ Stack 8 layers như này để tạo Encoder đầy đủ
```

# Model Implementation



## Transformer Encoder-Decoder



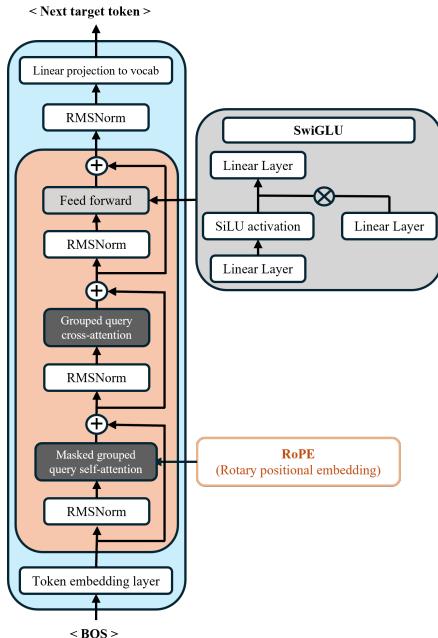
- ❖ The decoder takes BOS-prefixed target tokens, embeds them, and passes them through stacked blocks with RMSNorm, masked grouped-query self-attention (with RoPE), cross-attention, and a SwiGLU feed-forward layer.
- ❖ Masked self-attention lets each position look only at previous target tokens, while cross-attention uses encoder outputs as Keys and Values.
- ❖ The final RMSNorm + linear projection to vocab turns the decoder states into logits for predicting the next token.

# Model Implementation



## Transformer Encoder-Decoder

Code implementation:



INPUT:

```
tgt_emb shape: torch.Size([4, 12, 768]) # (batch, tgt_len, d_model)
enc_out shape: torch.Size([4, 16, 768]) # (batch, src_len, d_model)
tgt_causal_mask shape: torch.Size([12, 12]) # (tgt_len, tgt_len)
```

OUTPUT:

```
dec_out shape: torch.Size([4, 12, 768]) # (batch, tgt_len, d_model)
```

→ Stack 8 layers như này để tạo Decoder đầy đủ

```
class DecoderLayer(nn.Module):
    def __init__(self, d_model, n_heads, n_kv_heads, d_ff, dropout, rope_base):
        super().__init__()
        self.ln1 = RMSNorm(d_model)
        self.self_attn = GroupedQueryAttentionRoPE(
            d_model, n_heads, n_kv_heads, dropout, rope_base)
        self.ln2 = RMSNorm(d_model)
        self.cross_attn = GroupedQueryAttentionRoPE(
            d_model, n_heads, n_kv_heads, dropout, rope_base)
        self.ln3 = RMSNorm(d_model)
        self.ffn = FFN_SwiGLU(d_model, d_ff, dropout)
        self.dropout = nn.Dropout(dropout)

    def forward(self, y, enc_out, tgt_pad_mask=None, tgt_causal_mask=None, src_pad_mask=None):
        # Self-attention (masked)
        y1 = self.ln1(y)
        self_attn = self.self_attn(
            y1, y1, y1, key_padding_mask=tgt_pad_mask, attn_mask=tgt_causal_mask)
        y = y + self.dropout(self_attn)

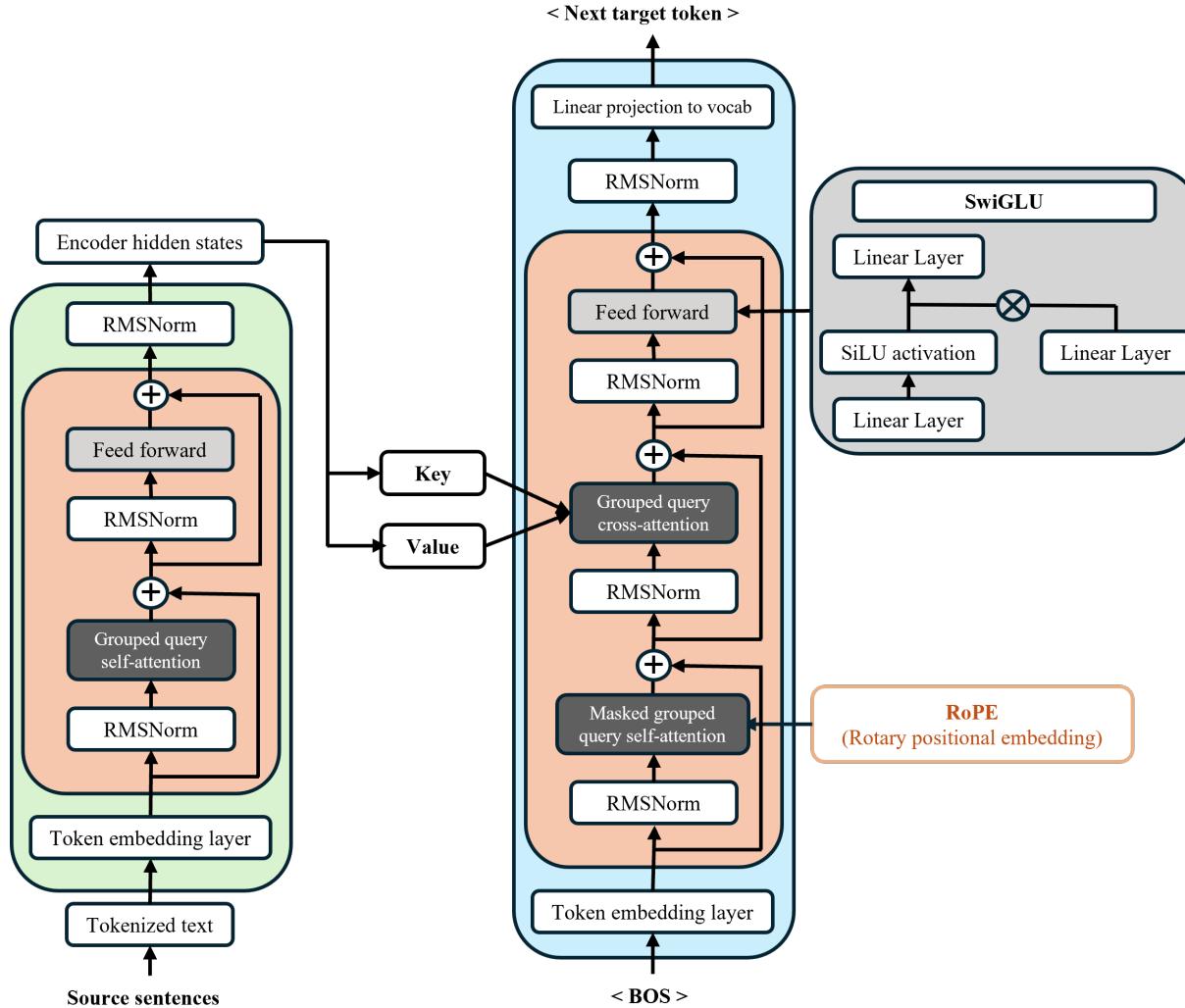
        # Cross-attention
        y2 = self.ln2(y)
        cross_attn = self.cross_attn(y2, enc_out, enc_out, key_padding_mask=src_pad_mask)
        y = y + self.dropout(cross_attn)

        # FFN
        y3 = self.ln3(y)
        return y + self.ffn(y3)
```

# Model Implementation



## Transformer Encoder-Decoder

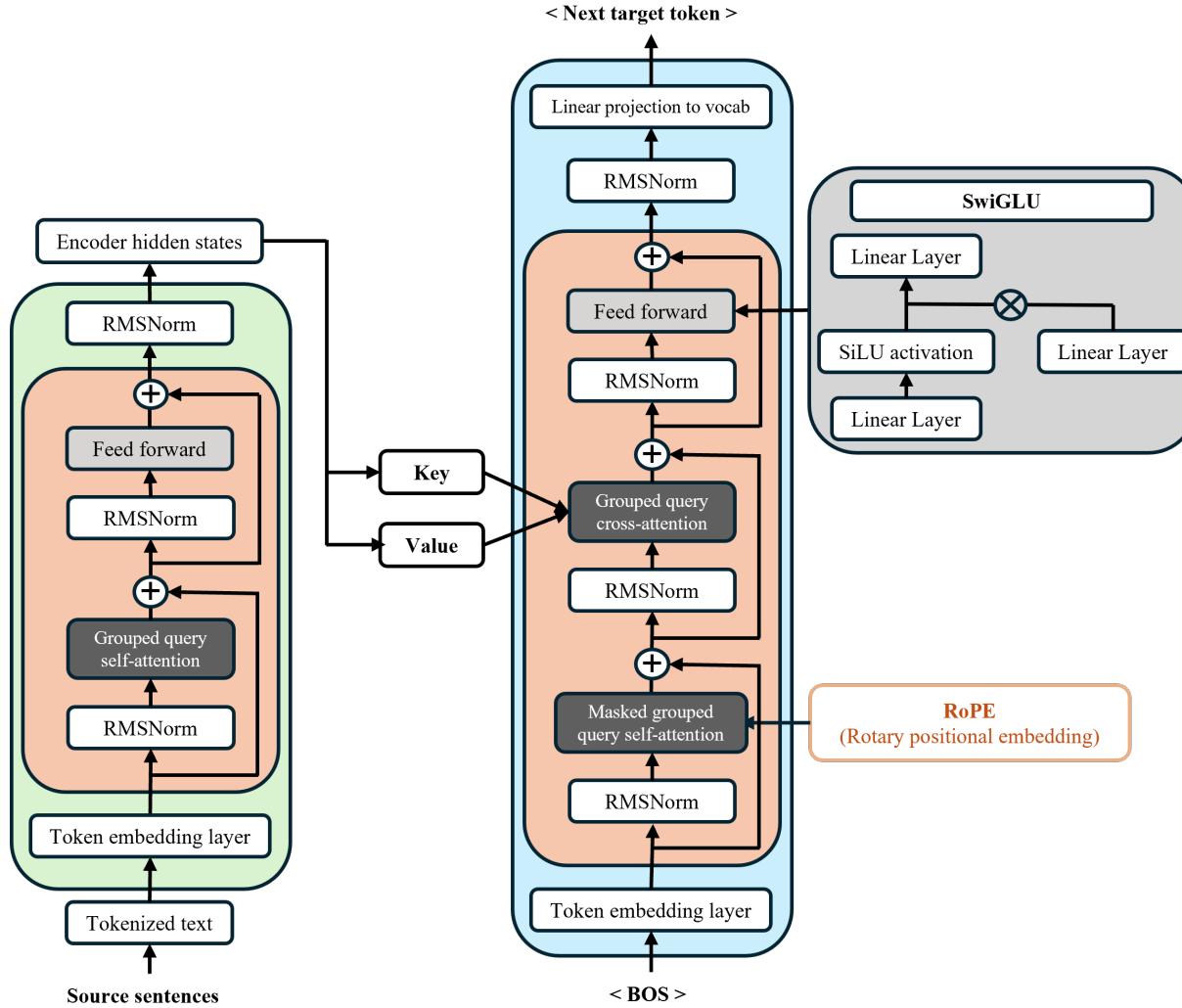


- ❖ Encoder reads the source sentence and produces contextual hidden states.
- ❖ Decoder reads the BOS + previous target tokens and generates the next token.
- ❖ Both sides use stacked blocks of Attention, Feed Forward, and Residual connections.
- ❖ Cross-attention lets the decoder attend to encoder outputs via Key–Value pairs.

# Model Implementation



## Transformer Encoder-Decoder



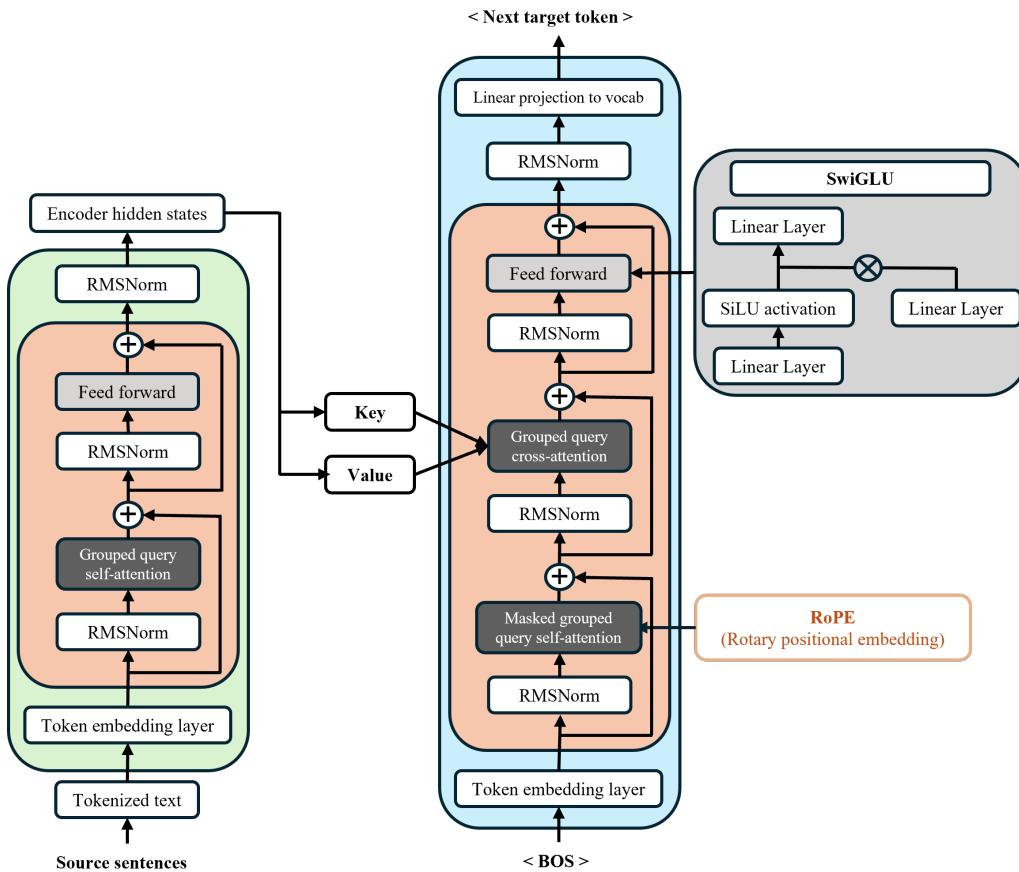
- ❖ RMSNorm is used instead of LayerNorm for stability and efficiency.
- ❖ Self-attention and cross-attention use Grouped-Query Attention for faster inference.
- ❖ RoPE injects positional information directly into Q and K.
- ❖ The Feed Forward block uses SwiGLU to make token representations more expressive.

# Model Implementation



## Transformer Encoder-Decoder

Code implementation:



```
class TransformerModel(nn.Module):
    def __init__(self, vocab_size, d_model, n_heads, n_kv_heads, d_ff, dropout,
                 rope_base, num_enc_layers=8, num_dec_layers=8):
        super().__init__()
        self.d_model = d_model
        self.embedding = nn.Embedding(vocab_size, d_model, padding_idx=0)
        self.emb_dropout = nn.Dropout(dropout)

        self.encoder_layers = nn.ModuleList([
            EncoderLayer(d_model, n_heads, n_kv_heads, d_ff, dropout, rope_base)
            for _ in range(num_enc_layers)
        ])
        self.encoder_final_ln = RMSNorm(d_model)

        self.decoder_layers = nn.ModuleList([
            DecoderLayer(d_model, n_heads, n_kv_heads, d_ff, dropout, rope_base)
            for _ in range(num_dec_layers)
        ])
        self.decoder_final_ln = RMSNorm(d_model)

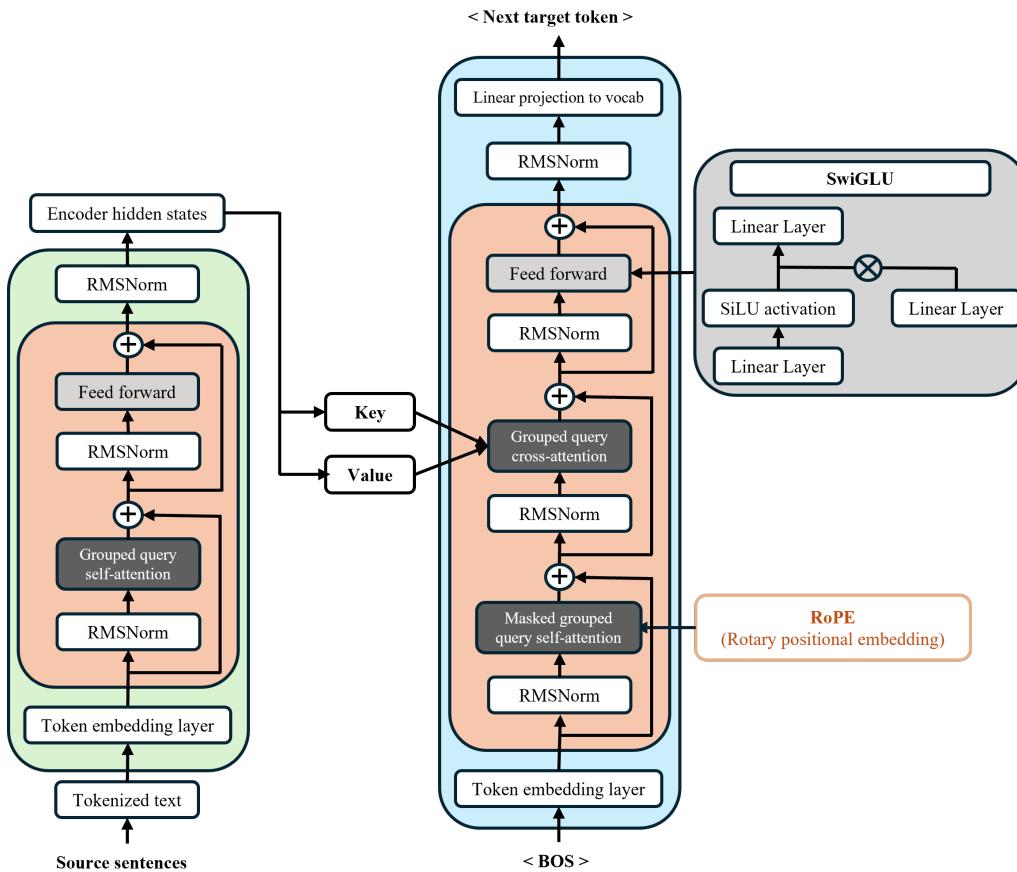
        self.output_bias = nn.Parameter(torch.zeros(vocab_size))
        self.emb_scale = math.sqrt(d_model)
```

# Model Implementation



## Transformer Encoder-Decoder

Code implementation:



```
def forward(self, src_ids: torch.Tensor, tgt_ids: torch.Tensor) -> torch.Tensor:
    # Masks
    src_pad = (src_ids == 0)
    tgt_pad = (tgt_ids == 0)
    tgt_in = tgt_ids[:, :-1]
    tgt_pad_in = tgt_pad[:, :-1]
    T = tgt_in.size(1)
    tgt_causal = torch.triu(torch.ones(T, T, dtype=torch.bool, device=src_ids.device), diagonal=1)

    # Encoder
    src_emb = self.emb_dropout(self.embedding(src_ids) * self.emb_scale)
    enc_out = src_emb
    for layer in self.encoder_layers:
        enc_out = layer(enc_out, src_pad)
    enc_out = self.encoder_final_ln(enc_out)

    # Decoder
    tgt_emb = self.emb_dropout(self.embedding(tgt_in) * self.emb_scale)
    dec_out = tgt_emb
    for layer in self.decoder_layers:
        dec_out = layer(dec_out, enc_out, tgt_pad_in, tgt_causal, src_pad)
    dec_out = self.decoder_final_ln(dec_out)

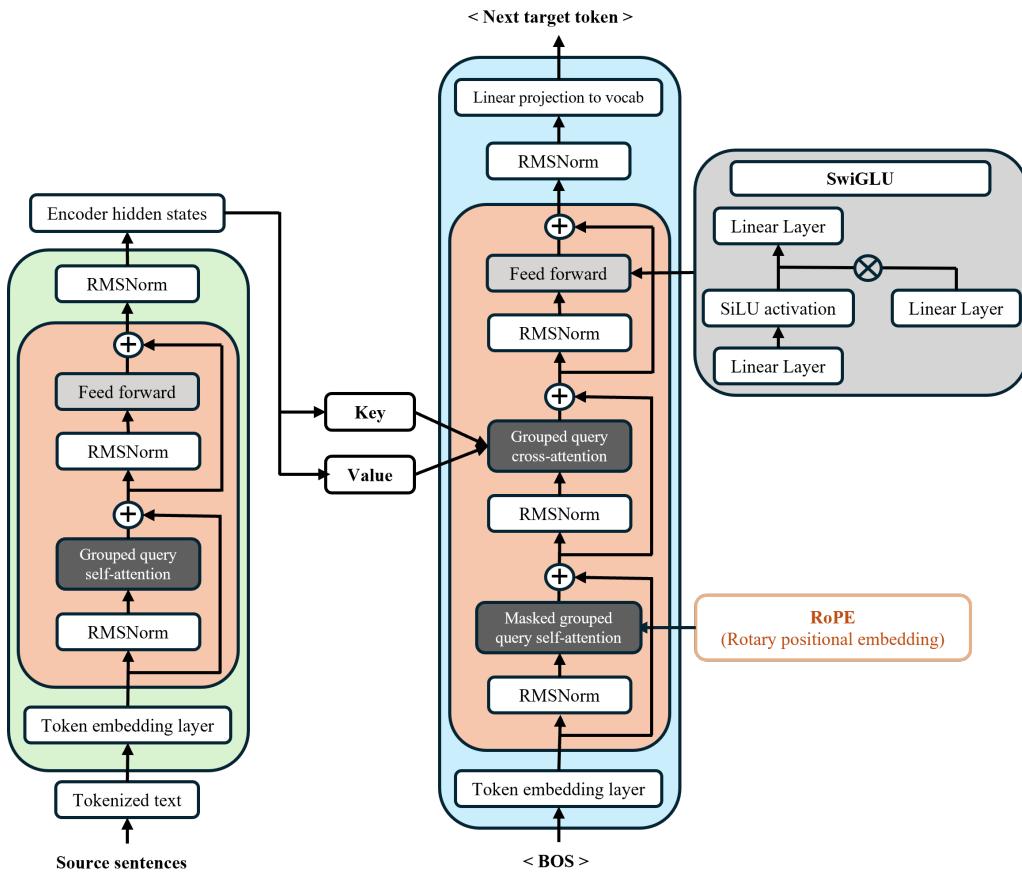
    # Output (tied embedding)
    return F.linear(dec_out, self.embedding.weight, self.output_bias)
```

# Model Implementation



## Transformer Encoder-Decoder

Code implementation:



### INPUT:

```
src_ids shape: torch.Size([4, 16]) # (batch, src_len)  
tgt_ids shape: torch.Size([4, 12]) # (batch, tgt_len)
```

### Model structure:

- Embedding: 8000 → 768
- Encoder: 8 layers
- Decoder: 8 layers
- Output: tied embedding weights

### OUTPUT:

```
logits shape: torch.Size([4, 11, 8000]) # (batch, tgt_len-1, vocab_size)  
→ Predict next token for each position
```

Total parameters: 157,179,200

# Model Implementation



## Pretraining Model

=====  
LUỒNG DỮ LIỆU TRANSFORMER ZH&VI  
=====

### 1. INPUT

```
src_ids: (batch, src_len) → token IDs nguồn  
tgt_ids: (batch, tgt_len) → token IDs đích
```

### 2. ENCODER

```
└─ Embedding(src_ids) * √768 → (batch, src_len, 768)  
└─ Dropout(0.01)  
└─ 8x EncoderLayer:  
    └─ RMSNorm  
    └─ Self-Attention (GQA: 12 Q heads / 4 KV heads + RoPE)  
    └─ Residual  
    └─ RMSNorm  
    └─ FFN (SwiGLU: 768 → 6144 → 3072 → 768)  
    └─ Residual  
└─ RMSNorm → enc_out: (batch, src_len, 768)
```

### 3. DECODER

```
└─ Embedding(tgt_ids[:-1]) * √768 → (batch, tgt_len-1, 768)  
└─ Dropout(0.01)  
└─ 8x DecoderLayer:  
    └─ RMSNorm  
    └─ Masked Self-Attention (GQA + RoPE + causal mask)  
    └─ Residual  
    └─ RMSNorm  
    └─ Cross-Attention (Q từ decoder, K/V từ enc_out)  
    └─ Residual  
    └─ RMSNorm  
    └─ FFN (SwiGLU)  
    └─ Residual  
└─ RMSNorm → dec_out: (batch, tgt_len-1, 768)
```

### 4. OUTPUT

```
└─ Linear(dec_out, embedding.weight.T) + bias  
→ logits: (batch, tgt_len-1, 8000)
```

### THÔNG SỐ QUAN TRỌNG

```
Vocab size: 8,000  
d_model: 768  
d_ff: 3072  
Attention heads: 12 Q heads / 4 KV heads  
Head dimension: 64  
Encoder layers: 8  
Decoder layers: 8  
RoPE base: 10000.0  
Max length: 32  
Dropout: 0.01
```

# Model Implementation



## Pretraining Model

- ❖ We use label-smoothed cross-entropy instead of the standard cross-entropy loss.
- ❖ Label smoothing does not give the correct token probability 1.0, but slightly lowers it.
- ❖ The remaining probability is spread over other tokens, which reduces overconfidence and helps the model generalize better.

```
class LabelSmoothedCrossEntropyLoss(nn.Module):  
    def __init__(self, smoothing: float = 0.1, ignore_index: int = 0):  
        super().__init__()  
        self.smoothing = smoothing  
        self.ignore_index = ignore_index  
  
    def forward(self, logits: torch.Tensor, targets: torch.Tensor) -> torch.Tensor:  
        vocab_size = logits.size(-1)  
        log_probs = F.log_softmax(logits, dim=-1)  
        mask = targets != self.ignore_index  
        with torch.no_grad():  
            true_dist = torch.full_like(  
                log_probs, self.smoothing / (vocab_size - 1))  
            true_dist.scatter_(1, targets.unsqueeze(1), 1.0 - self.smoothing)  
            true_dist[targets == self.ignore_index] = 0.0  
        loss = -(true_dist * log_probs).sum(dim=-1)  
        loss = loss.masked_fill(~mask, 0.0)  
        return loss.sum() / mask.sum().clamp(min=1)
```

# Model Implementation



## Pretraining Model

- ❖ Loss: LabelSmoothedCrossEntropyLoss with ignore\_index=0.
- ❖ Optimizer: Adam with tuned betas, small eps, and weight\_decay.
- ❖ Learning rate: warmup + inverse-sqrt via WarmupInverseSqrtScheduler.
- ❖ package\_tokenizer(config.spm\_prefix) exports the trained SentencePiece tokenizer.

```
model = TransformerModel(config, vocab_size).to(config.device)
criterion = LabelSmoothedCrossEntropyLoss(config.label_smoothing, ignore_index=0)
optimizer = optim.AdamW(
    model.parameters(),
    lr=config.lr_base,
    betas=(0.9, 0.98),
    eps=1e-9,
    weight_decay=config.weight_decay,
)
scheduler = WarmupInverseSqrtScheduler(optimizer, config.warmup_steps, config.lr_base)
tokenizer_payload = package_tokenizer(config.spm_prefix)
```

# Model Implementation



## Pretraining Model

Epoch 1/40

vi→zh coverage: 10598/15140 (~70.0% per epoch)  
Train Loss: 6.2664  
Valid Loss: 2.4952  
Valid BLEU: 1.50  
Learning Rate: 0.000101

Epoch 2/40

vi→zh coverage: 10598/15140 (~70.0% per epoch)  
Train Loss: 4.1094  
Valid Loss: 1.8673  
Valid BLEU: 7.86  
Learning Rate: 0.000199

Epoch 15/40

vi→zh coverage: 10598/15140 (~70.0% per epoch)  
Train Loss: 0.3244  
Valid Loss: 0.8093  
Valid BLEU: 59.15  
Learning Rate: 0.000073

Epoch 40/40

vi→zh coverage: 10598/15140 (~70.0% per epoch)  
Train Loss: 0.1665  
Valid Loss: 0.8474  
Valid BLEU: 63.35  
Learning Rate: 0.000044

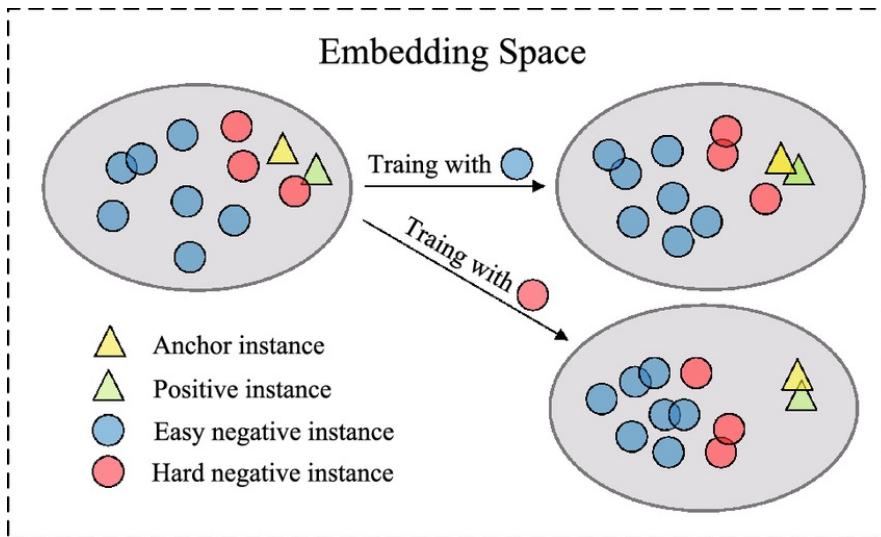
- ❖ Each epoch uses only about 70% of the vi→zh pairs, with a rotating window so that all vi→zh samples are covered across epochs.
- ❖ Zh→vi pairs are always fully seen, so the training schedule is biased toward the zh→vi direction.
- ❖ In experiments, this asymmetric schedule gives better zh→vi translation quality than using 100% of both zh→vi and vi→zh pairs every epoch.
- ❖ Train loss and valid loss steadily decrease over time. Validation BLEU improves from 1.5 → 63.35, showing much better translation quality.

# Model Implementation



## Contrastive Training

Contrastive training:



Idea:

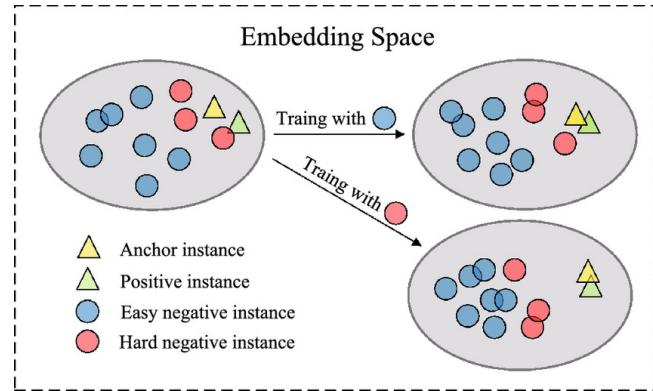
- ❖ Projection head maps embeddings into a normalized contrastive space.
- ❖ Contrastive loss pulls true translation pairs closer and pushes unrelated sentences apart.
- ❖ This enforces a bilingual semantic space that improves translation quality.

# Model Implementation



## Contrastive Training

Code implementation:



- ❖ `contrastive_loss` computes a similarity matrix and makes each embedding match its true pair.
- ❖ The function uses cross-entropy so correct pairs score highest among all candidates.
- ❖ `compute_crosslingual_loss` encodes four sentence variants to form cross-lingual positives and negatives.

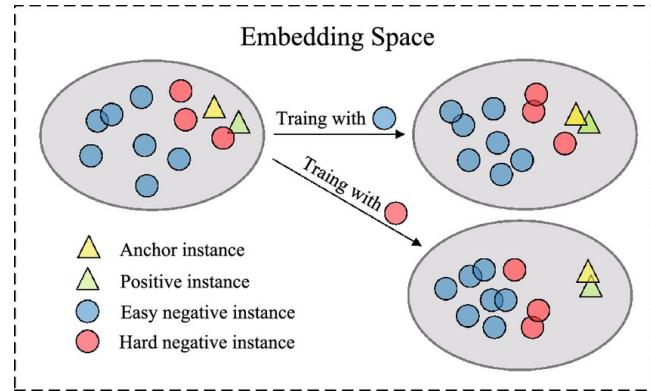
```
def contrastive_loss(z_a, z_b, tau):  
    sim = torch.matmul(z_a, z_b.transpose(0, 1)) / tau  
    labels = torch.arange(sim.size(0), device=sim.device)  
    loss_i = F.cross_entropy(sim, labels)  
    loss_j = F.cross_entropy(sim.transpose(0, 1), labels)  
    return 0.5 * (loss_i + loss_j)  
  
def compute_crosslingual_loss(model, projection, ids_zh_vi, ids_vi_vi,  
                               ids_vi_zh, ids_zh_zh, cl_config, pad_id, special_ids):  
    if ids_zh_vi.size(0) < 2:  
        return torch.zeros(1, device=config.device)  
  
    # Encode all contexts  
    enc_zh_vi = encode_context(model, ids_zh_vi, pad_id)  
    enc_vi_vi = encode_context(model, ids_vi_vi, pad_id)  
    enc_vi_zh = encode_context(model, ids_vi_zh, pad_id)  
    enc_zh_zh = encode_context(model, ids_zh_zh, pad_id)  
  
    # Mean pool and project  
    z_zh_vi = projection(mean_pool(enc_zh_vi, ids_zh_vi, pad_id, special_ids))  
    z_vi_vi = projection(mean_pool(enc_vi_vi, ids_vi_vi, pad_id, special_ids))  
    z_vi_zh = projection(mean_pool(enc_vi_zh, ids_vi_zh, pad_id, special_ids))  
    z_zh_zh = projection(mean_pool(enc_zh_zh, ids_zh_zh, pad_id, special_ids))  
  
    # Contrastive losses  
    cl_vi_space = contrastive_loss(z_zh_vi, z_vi_vi, cl_config.contrastive_tau)  
    cl_zh_space = contrastive_loss(z_vi_zh, z_zh_zh, cl_config.contrastive_tau)  
  
    return 0.5 * (cl_vi_space + cl_zh_space)
```

# Model Implementation



## Contrastive Training

Code implementation:



- ❖ Mean-pooling removes special tokens and extracts a clean sentence-level embedding.
- ❖ Contrastive loss is computed in both Vietnamese and Chinese spaces to align their semantics.
- ❖ The final loss averages both directions, ensuring stable and symmetric cross-lingual training.

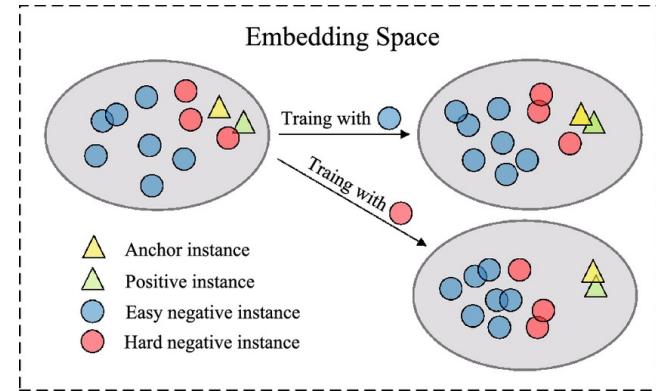
```
def contrastive_loss(z_a, z_b, tau):  
    sim = torch.matmul(z_a, z_b.transpose(0, 1)) / tau  
    labels = torch.arange(sim.size(0), device=sim.device)  
    loss_i = F.cross_entropy(sim, labels)  
    loss_j = F.cross_entropy(sim.transpose(0, 1), labels)  
    return 0.5 * (loss_i + loss_j)  
  
def compute_crosslingual_loss(model, projection, ids_zh_vi, ids_vi_vi,  
                               ids_vi_zh, ids_zh_zh, cl_config, pad_id, special_ids):  
    if ids_zh_vi.size(0) < 2:  
        return torch.zeros(1, device=config.device)  
  
    # Encode all contexts  
    enc_zh_vi = encode_context(model, ids_zh_vi, pad_id)  
    enc_vi_vi = encode_context(model, ids_vi_vi, pad_id)  
    enc_vi_zh = encode_context(model, ids_vi_zh, pad_id)  
    enc_zh_zh = encode_context(model, ids_zh_zh, pad_id)  
  
    # Mean pool and project  
    z_zh_vi = projection(mean_pool(enc_zh_vi, ids_zh_vi, pad_id, special_ids))  
    z_vi_vi = projection(mean_pool(enc_vi_vi, ids_vi_vi, pad_id, special_ids))  
    z_vi_zh = projection(mean_pool(enc_vi_zh, ids_vi_zh, pad_id, special_ids))  
    z_zh_zh = projection(mean_pool(enc_zh_zh, ids_zh_zh, pad_id, special_ids))  
  
    # Contrastive losses  
    cl_vi_space = contrastive_loss(z_zh_vi, z_vi_vi, cl_config.contrastive_tau)  
    cl_zh_space = contrastive_loss(z_vi_zh, z_zh_zh, cl_config.contrastive_tau)  
  
    return 0.5 * (cl_vi_space + cl_zh_space)
```

# Model Implementation



## Contrastive Training

Code implementation:



Contrastive Epoch 1/20

CE Loss: 0.2613  
CL Loss: 1.8903  
Total Loss: 0.3245  
Learning Rate: 0.000045

Contrastive Epoch 3/20

CE Loss: 0.1611  
CL Loss: 0.0718  
Total Loss: 0.1683  
Learning Rate: 0.000026

Contrastive Epoch 2/20

CE Loss: 0.1731  
CL Loss: 0.1547  
Total Loss: 0.1886  
Learning Rate: 0.000032

Contrastive Epoch 20/20

CE Loss: 0.1497  
CL Loss: 0.0114  
Total Loss: 0.1508  
Learning Rate: 0.000010

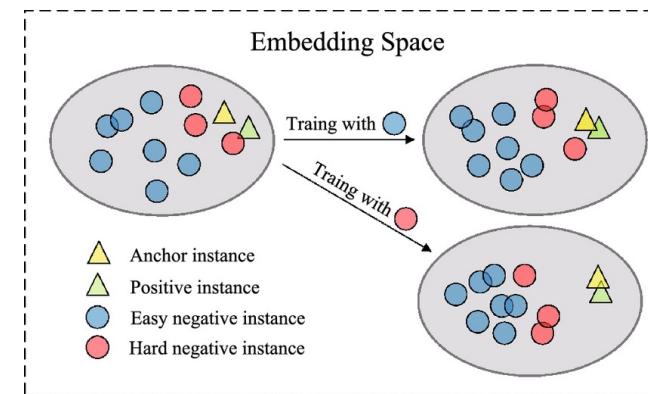
- ❖ Contrastive training pushes parallel sentences closer while separating unrelated ones in embedding space.
- ❖ CE loss guides translation quality, while CL loss shapes a shared cross-lingual semantic space.
- ❖ Over epochs, CL loss drops rapidly, showing the model is learning stronger bilingual alignment.

# Model Implementation

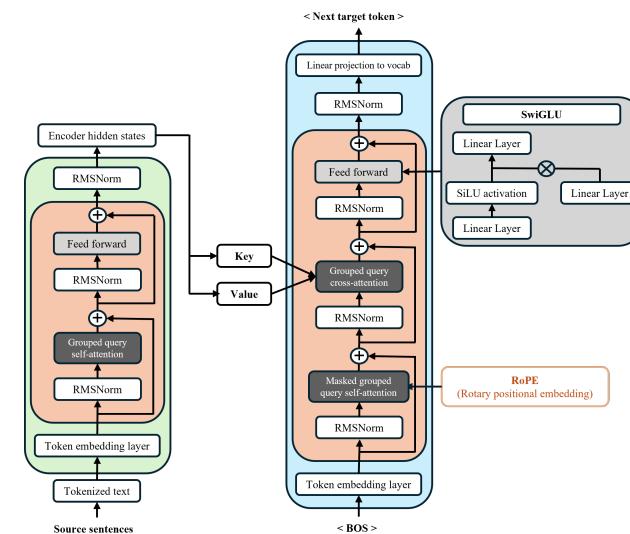


## Contrastive Training

OLP-AI'25						
Task:				NLP		
#	Thí sinh	Ngày	ID	Total Score	Public NLP Score	Private NLP Score
1	Không phải TQKhang	2025-11-19 14:59	6067	<b>68.8</b>	35.06	33.74
2	anhht	2025-11-27 17:02	6965	<b>68.55</b>	34.62	33.94
3	tvan	2025-11-22 03:35	6471	<b>66.95</b>	33.5	33.45
4	Hao Buoi Cong	2025-11-20 14:33	6234	<b>65.9</b>	33.26	32.65
5	quanghien	2025-11-28 11:50	7019	<b>65.71</b>	33.41	32.29



- ❖ Generate `public_test.csv` and `private_test.csv` as required by the BTC format.
- ❖ The outputs were submitted to the evaluation system for scoring.
- ❖ The final submission reached Top 5 on the leaderboard.



# QUIZ TIME

# Summary

## Introduction CV Task

- ❖ Task Statement
- ❖ Dataset EDA
- ❖ Baseline Analysis
- ❖ Improvement

## Introduction NLP Task

- ❖ Task Statement
- ❖ Dataset EDA
- ❖ Baseline Analysis
- ❖ Improvement

## Model Implementation

- ❖ Data Imbalance Handle
- ❖ Transformer Encoder
- ❖ Attention Pooling
- ❖ Result

## Model Implementation

- ❖ Pre-process Data
- ❖ Transformer Encoder-Decoder
- ❖ Pretraining Model
- ❖ Contrastive Training
- ❖ Result

# Thanks!

Any questions?