EE958 Competition: Machine Translation System for India

Vasudev Gupta
241562575
vasudevg24@iitk.ac.in
Indian Institute of Technology Kanpur (IIT Kanpur)

Abstract

This project addresses English-to-Hindi and English-to-Bengali translation for the NLP Capstone competition. I trained Transformer encoder–decoder models with word-level tokenization (30k vocab), using cross-entropy loss, cosine learning rate scheduling, gradient clipping, and checkpointing. The best submission scored chrF++ 0.40, ROUGE 0.42, and BLEU 0.14 on the test leaderboard, placing Rank #2 overall. These results show that carefully tuned Transformer setups can perform strongly on low-resource Indic machine translation. Future improvements could come from subword tokenization (BPE), data augmentation, larger corpora, and exploring decoder-only architectures such as GPT-style models.

1 Competition Result

Codalab Username: g_241562575
Final leaderboard rank on the test set: 2
chrF++ Score wrt to the final rank: 0.40
ROUGE Score wrt to the final rank: 0.42
BLEU Score wrt to the final rank: 0.14

2 Problem Description

The project tackles automatic machine translation (MT) from English into two low-resource Indic languages: Hindi and Bengali. MT aims to convert text from a source language into a fluent and semantically accurate target language. The competition provided parallel sentence pairs for English–Hindi and English–Bengali, with systems evaluated on held-out validation and test sets using BLEU, chrF++, and ROUGE.

While large commercial models exist, this task emphasized building systems from scratch with only the provided data. This setup makes the challenge harder but also more instructive: it highlights the core objectives, design trade-offs, and difficulties of training translation models without relying on massive pretraining.

3 Data Analysis

3.1 Dataset Overview

The competition datasets cover English→Bengali and English→Hindi, each split into train, validation, and test sets. Target translations for validation and test are withheld for scoring, so summary statistics are available only on training data.

3.2 Corpus Statistics

Tables 1 and 2 summarize basic length statistics (in words, approximately) for source and target sides.

Table 1: English→Bengali corpus statistics (lengths in words).

Split	count	source_mean	source_min	source_max	target_mean	target_min / max
Train	68,849	16.82	1.0	100.0	14.27	1.0 / 84.0
Validation	9,836	16.88	1.0	175.0	–	–
Test	19,672	16.93	0.0	99.0	–	–

Table 2: English→Hindi corpus statistics (lengths in words).

Split	count	source_mean	source_min	source_max	target_mean	target_min / max
Train	80,797	17.07	1.0	257.0	18.93	1.0 / 216.0
Validation	11,543	17.08	1.0	128.0	–	–
Test	23,085	17.08	1.0	155.0	–	–

3.3 Length Distributions

Figure 1 shows KDE plots of sentence-length distributions (train/val/test) for both translation tasks. The source-side distributions are well aligned across splits (means $\approx$ 17 words), indicating consistent difficulty between train and evaluation data.

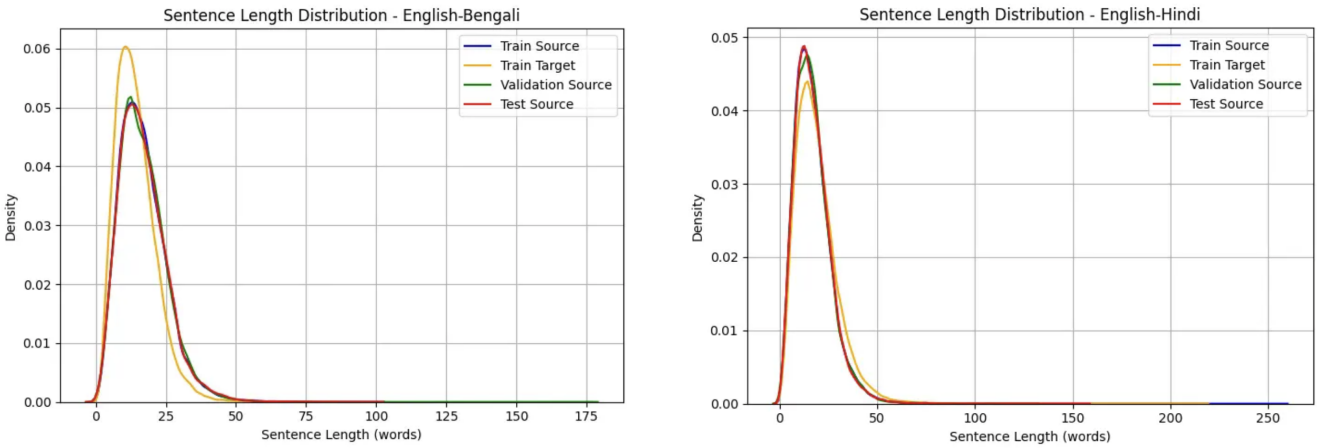

Figure 1: Sentence length distributions (train/val/test) for English→Bengali (left) and English→Hindi (right).

3.4 Key Observations

- Source lengths are stable across splits ($\sim$17 words), indicating little covariate shift.
- Hindi targets are longer on average (18.9 vs. 14.3 for Bengali), reflecting richer morphology.
- Both corpora contain long-tail sentences (up to 257 tokens), so training capped sequences at 256 tokens for efficiency.
- Validation/test target stats are hidden, but aligned source distributions reduce evaluation mismatch.

4 Model Description

4.1 Model Evolution

The approach was based on the Transformer encoder–decoder architecture [1]. I first implemented it from scratch in PyTorch, which was slow and unstable, then switched to PyTorch's optimized `nn.Transformer` [2, 3] for faster, more stable training. I also tried subword tokenization with BPE [4] and SentencePiece [5], though my best submissions used word-level vocabularies. Decoder-only GPT-style models were partly explored [6, 7], but not fully completed in time.

4.2 Final Model

Both English-to-Hindi and English-to-Bengali systems used a Transformer encoder–decoder with word-level tokenization (30k vocab, max length 256). The main hyperparameters were:

- **English-to-Hindi:** $d_{model} = 128$, $n_{heads} = 8$, $n_{layers} = 6$, $d_{ff} = 512$, dropout = 0.1, batch size = 32, 30 epochs. Learning rate peaked at 9×10^{-4} with cosine decay, final LR 9×10^{-5}, and 3.5% warmup.
- **English-to-Bengali:** Same architecture, but $n_{heads} = 4$ and peak LR 6×10^{-4} (final 6×10^{-5}).

Training progress was tracked in Weights & Biases (wandb), and checkpoints were saved regularly.

4.3 Objective Function

Training used token-level cross-entropy loss, implemented with PyTorch [2, 3]. For each target token y_t,

$$\mathcal{L}(\theta) = -\sum_{t=1}^{T} m_t \log p_\theta(y_t \mid y_{<t}, x),$$

where x is the source sentence and m_t masks out PAD tokens.

4.4 Inference Strategy

Inference used greedy decoding: at each step, the token with highest probability was chosen until an end-of-sequence token or maximum length was reached. While beam search may yield higher scores, greedy decoding was sufficient for leaderboard submissions.

5 Experiments

5.1 Data Pre-Processing

5.1.1 Data Format

The raw dataset was provided in JSON. For efficient streaming during training, I converted it into JSONL, where each line contains one source–target pair. This avoided loading the entire dataset into memory.

5.1.2 Tokenization

English text was lowercased, whitespace-normalized, and split on punctuation. A 30k word-level vocabulary was built with four special tokens: `<PAD>`, `<UNK>`, `<SOS>`, `<EOS>`. Hindi and Bengali text used regex patterns from the Indic NLP Library [8] to handle script-specific punctuation and numerals.

A simplified version of my tokenizer logic:

```
def text_to_sequence(text, word2idx, max_len):
    tokens = tokenize(text)
    seq = [word2idx["<SOS>"]]
    for tok in tokens:
        if len(seq) >= max_len - 1: break
        seq.append(word2idx.get(tok, word2idx["<UNK>"]))
    seq.append(word2idx["<EOS>"])
    return pad(seq, max_len, word2idx["<PAD>"])
```

5.1.3 Data Loader

I wrote a minimal custom loader (`DataLoaderLite`) to cycle through batches sequentially, rather than PyTorch's heavier `DataLoader`. This made training loops transparent and easy to debug.

```
class DataLoaderLite:
    def __init__(self, X, y, batch_size):
        self.X, self.y, self.bs = X, y, batch_size
        self.i, self.epoch = 0, 0

    def next_batch(self):
        start, end = self.i, self.i + self.bs
        if end > len(self.X):  # restart epoch
            self.epoch += 1
            start, end, self.i = 0, self.bs, self.bs
        else:
            self.i = end
        return self.X[start:end], self.y[start:end]
```

5.2 Training Procedure

5.2.1 Optimizer and Schedule

Models were trained with AdamW [9] (extending Adam [10]) and gradient clipping. The learning rate followed a cosine decay schedule with warm restarts [11], with 3.5% warmup.

```
if step < warmup_steps:
    lr = max_lr * (step+1)/warmup_steps
else:
    decay = (step - warmup_steps) / (MAX - warmup_steps)
    coeff = 0.5 * (1 + cos(pi * decay))
    lr = min_lr + coeff * (max_lr - min_lr)
```

Peak LR was task-specific: 9×10^{-4} for Hindi, 6×10^{-4} for Bengali. Figure 2 shows the curve over 30k steps.

5.2.2 Main Experiment Configurations

Three main experiments were run, progressively refining implementation and hyperparameters. Table 3 summarizes the setups.

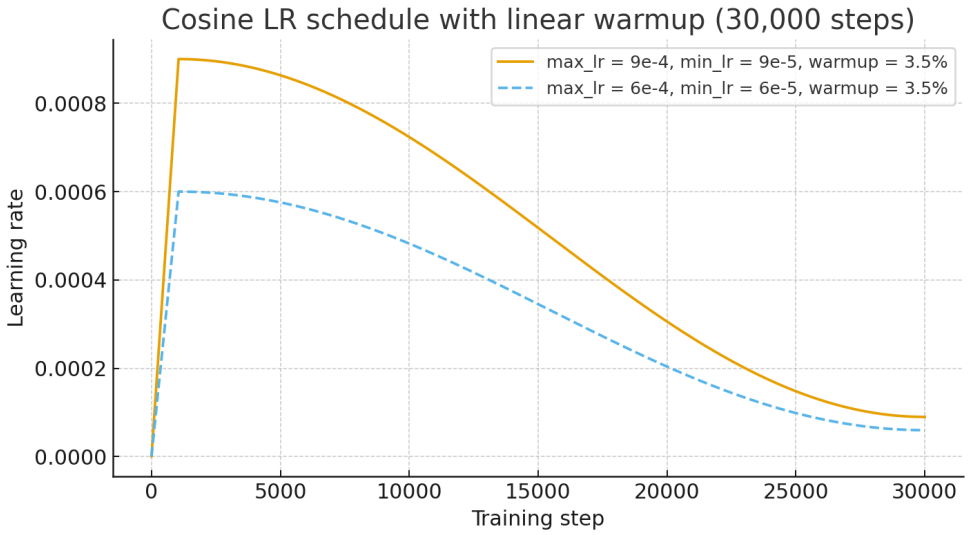

Figure 2: Cosine learning rate schedule with 3.5% warmup over 30,000 steps.

Table 3: Main Experiment Configurations

Configuration	**Experiment 1 (exp3)**	**Experiment 2 (exp4)**	**Experiment 3 (exp4.1)**
Implementation	Custom Transformer	PyTorch nn.Transformer	PyTorch nn.Transformer
d_{model}	128	128	128
n_{layers}	6	6	6
d_{ff}	512	512	512
n_{heads} (Hindi)	4	4	8
n_{heads} (Bengali)	4	4	4
Epochs	10	10	30
Batch Size	32	32	32
Vocab Size	30k	30k	30k
Peak LR (Hindi)	6×10^{-4}	6×10^{-4}	9×10^{-4}
Peak LR (Bengali)	6×10^{-4}	6×10^{-4}	6×10^{-4}
Schedule	Cosine	Cosine	Cosine

5.2.3 Training Dynamics

Figures 3–12 illustrate training curves for both language pairs, including loss, validation, learning rate, gradient norms, and GPU dynamics (Multiprocessing clock speed). The custom Transformer showed unstable GPU usage (7, 12), highlighting inefficiencies compared to the PyTorch implementation. Gradient norms were expected to remain stable but instead grew over time, which was a concern during training. Loss curves (train and validation) improved steadily, though Experiment 3 eventually began to overfit after extended training.

Training progress was tracked in Weights & Biases (wandb) [12], and checkpoints were saved for reproducibility.

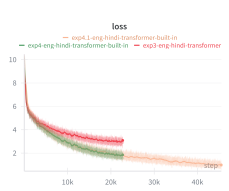

Figure 3: Training loss (Hindi)

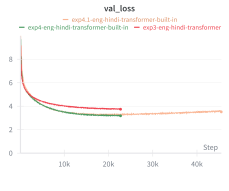

Figure 4: Validation loss (Hindi)

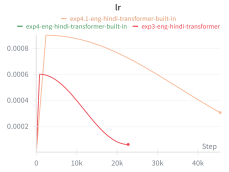

Figure 5: Learning rate (Hindi)

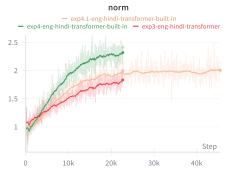

Figure 6: Grad norm (Hindi)

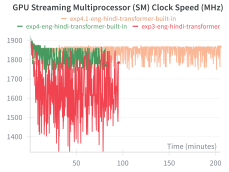

Figure 7: GPU usage (Hindi)

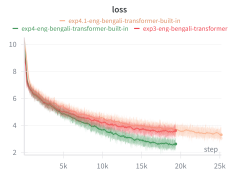

Figure 8: Training loss (Bengali)

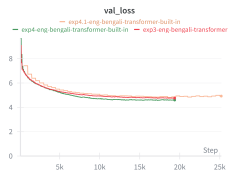

Figure 9: Validation loss (Bengali)

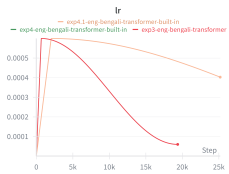

Figure 10: Learning rate (Bengali)

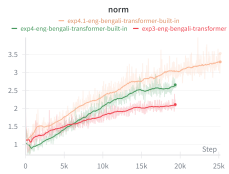

Figure 11: Grad norm (Bengali)

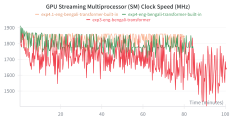

Figure 12: GPU usage (Bengali)

6 Results

6.1 Development Set Performance

Table 4 shows results during the train phase. Transitioning from a custom Transformer to PyTorch's `nn.Transformer` gave a clear boost across all metrics, and extended training (Experiment 3) achieved the best scores.

Table 4: Development Set Results

Experiment	Rank	BLEU	ROUGE	chrF++	Total
Exp 1 (custom)	–	0.063	0.334	0.275	0.672
Exp 2 (nn.Transf.)	–	0.114	0.401	0.374	0.889
Exp 3 (30 epochs)	#1	**0.126**	**0.410**	**0.396**	**0.932**

6.2 Test Set Performance

On the final leaderboard (Table 5), Experiment 3 generalized well, securing Rank #2 overall with a total score of 0.957.

6.3 Performance Analysis

Experiment 3 gave the strongest results: BLEU improved from 0.126 (dev) to 0.140 (test), showing good generalization. Key factors behind this improvement were:

- Switching from a custom to PyTorch's optimized `nn.Transformer`, which improved stability and efficiency.
- Longer training (30 epochs) with tuned learning rates and more attention heads for Hindi (8 vs. 4).
- Cosine LR scheduling with warmup, which stabilized convergence.

Table 5: Test Set Results

Experiment	Rank	BLEU	ROUGE	chrF++	Total
Exp 1 (custom)	–	0.022	0.198	0.128	0.348
Exp 2 (nn.Transf.)	–	0.125	0.410	0.380	0.915
Exp 3 (30 epochs)	#2	**0.140**	**0.421**	**0.396**	**0.957**

Overall, the final model balanced efficiency with performance, ranking #2 on the leaderboard while avoiding severe overfitting.

7 Error Analysis

1. **Implementation Stability:** The custom Transformer was prone to numerical instability and poor GPU utilization, which limited training efficiency. In contrast, PyTorch's optimized `nn.Transformer` provided stable training and a large improvement in scores.

2. **Training Convergence:** Extended training (30 epochs) improved results, but gradient norms tended to grow rather than stabilize, raising concerns about long-term stability. Validation loss curves also showed overfitting in Experiment 3, indicating the need for stronger regularization or early stopping.

3. **Translation Quality:** Despite overall progress, BLEU scores remained low. Errors were most visible in word order, morphology (especially Hindi), and handling of proper nouns. The model sometimes produced fluent but semantically off-target translations, reflecting limitations of word-level tokenization.

4. **Key Takeaway:** Implementation quality and training stability had greater impact on performance than architectural tweaks. Addressing overfitting and improving tokenization remain open challenges.

8 Conclusion

This project achieved Rank #2 in the competition with BLEU 0.14, chrF++ 0.40, and ROUGE 0.42. The strongest gains came from using PyTorch's `nn.Transformer`, extending training, and tuning hyperparameters for each task.

At the same time, the experiments showed that Indic MT remains difficult: word-level vocabularies struggled with morphology and rare words, and models tended to overfit despite careful scheduling.

Future directions:

- Future improvements could include completing the partially implemented BPE/SentencePiece tokenizers [4, 5], data augmentation, incorporating additional corpora, and experimenting with decoder-only GPT-style models [6, 7].

- Scaling model size and adding regularization methods (e.g., dropout scheduling, label smoothing) to address overfitting.

Overall, the results demonstrate that with careful engineering, even modest Transformer models can perform competitively in low-resource Indic translation. Future work along these lines could push performance further while deepening understanding of architecture–data trade-offs.

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 5998–6008, 2017.

[2] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, and et al., "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[3] PyTorch Team, "Pytorch documentation." `https://pytorch.org/docs/stable`, 2024.

[4] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of ACL*, vol. 1, pp. 1715–1725, 2016.

[5] T. Kudo and J. Richardson, "Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," in *Proceedings of EMNLP: System Demonstrations*, pp. 66–71, 2018.

[6] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners." OpenAI Technical Report, 2019.

[7] T. B. Brown, B. Mann, and N. e. a. Ryder, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[8] A. Kunchukuttan, "Indic nlp library." `https://github.com/anoopkunchukuttan/indic_nlp_library`, 2020.

[9] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *International Conference on Learning Representations (ICLR)*, 2019.

[10] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations (ICLR)*, 2015.

[11] I. Loshchilov and F. Hutter, "Sgdr: Stochastic gradient descent with warm restarts," *arXiv preprint arXiv:1608.03983*, 2016.

[12] L. Biewald, "Weights & biases." `https://wandb.ai`, 2020.