Latin to Devanagari Transliteration using Seq2Seq

This project tackles the problem of converting romanized Hindi text (like we type on WhatsApp) back into proper Devanagari script. For example, converting "namaste" → "नमस्ते" or "bharat" → "भारत".

Why This Project?

Ever notice how we all type Hindi in English letters when chatting? This model tries to automatically convert that back to proper Hindi script. I used the Aksharantar dataset from AI4Bharat which has thousands of such word pairs.

The Challenge I Faced

GPU Limitation: My laptop has an NVIDIA MX250 with only 2GB VRAM, which kept crashing when I tried training larger models. So I had to be smart about:

- Using smaller batch sizes (32 instead of 128)
- Testing with a subset of data first
- Keeping the model architecture reasonable (256 hidden units max)
- Running longer on CPU for final training when GPU memory ran out

Despite these limitations, the theoretical framework and code architecture remain solid and scalable.

How It Works

The model uses an encoder-decoder architecture where:

Encoder reads the Latin text character by character and builds an understanding of what the word is. Think of it like reading "ghar" and internally processing "this is the word for house".

Decoder then takes that understanding and generates the Devanagari characters one by one: घ, then र, forming "घर".

The cool part is the encoder passes its final "understanding" (hidden state) to the decoder as a starting point.

Model Architecture

I kept it flexible so you can experiment:

```python
```

```
model_params = {
    'char_embedding': 128,      # how we represent each character
    'hidden_size': 256,         # memory capacity of the network
    'encoder_layers': 1,        # stacked encoder layers
    'decoder_layers': 1,        # stacked decoder layers
    'rnn_type': 'LSTM',         # can switch to GRU or vanilla RNN
    'dropout': 0.3              # regularization
}
```

All of these can be changed in the config without touching the core model code.

Math Behind The Model

Let me break down what's happening computationally:

How Many Operations?

For LSTM (the default choice):

When processing one sequence of length n:

- **Encoder work**: For each character, LSTM does 4 gate operations (input, forget, output, cell). Each gate needs to multiply current input (size m) with weights and previous hidden state (size h) with weights. So per character: $4\times(h\times m + h\times h)$ operations. For full sequence: $n \times 4(hm + h^2)$
- **Decoder work**: Same as encoder, another $n \times 4(hm + h^2)$
- **Making predictions**: Convert hidden state to vocabulary probabilities: $n \times h\times V$ operations where V is vocab size

Total: Roughly $n \times [8hm + 8h^2 + hV]$ operations per sequence

With my settings (m=128, h=256, V≈100, n≈15 avg):

- Around $15 \times [8\times128\times256 + 8\times256^2 + 256\times100]$
- That's about $15 \times [262{,}144 + 524{,}288 + 25{,}600] \approx 12$ million operations per word
- For a batch of 32 words, that's 384 million operations!

No wonder my GPU struggled.

How Many Parameters to Train?

Let's count what the model needs to learn:

1. **Character embeddings**: We need a vector for each character. V characters × m dimensions = 100 × 128 = 12,800 parameters
2. **Encoder LSTM**: Has 4 gates (input, forget, cell, output)

- Each gate connects to: current input (h×m weights) + previous hidden state (h×h weights) + bias (h values)
- Total for encoder: $4(h \times m + h^2 + h) = 4(256 \times 128 + 256^2 + 256) = 395{,}264$ parameters

3. **Decoder LSTM**: Same structure as encoder = 395,264 parameters
4. **Output layer**: Maps hidden state to vocab, needs h×V weights + V biases = 256×100 + 100 = 25,700 parameters

Grand total: 12,800 + 395,264 + 395,264 + 25,700 = **829,028 parameters**

That's nearly a million numbers the model needs to learn! Each needs to be stored, updated during training, and that's where my memory issues came in.

Implementation Details

The Vocabulary System

I built a simple but effective vocab manager that:

- Tracks character frequency in training data
- Only includes characters that appear at least 2 times (removes typos)
- Reserves special tokens: <PAD> for padding, <SOS> to start sequences, <EOS> to end them, <UNK> for unknown chars

Teacher Forcing Trick

During training, sometimes (50% of the time in my case) I feed the correct target character to the decoder instead of its own prediction. This helps it learn faster and more stably. Think of it like learning to ride a bike with training wheels initially.

Without this, the decoder makes a wrong prediction early, then the next prediction is based on that wrong one, and errors compound quickly.

Handling Variable Lengths

Words have different lengths. "ghar" is 4 letters but "ajanabee" is 8. I use padding to make them equal length in a batch, but tell the model to ignore those padded positions using PyTorch's pack_padded_sequence. This speeds up training significantly.

Code Structure

```
project/
├── transliteration_model.py   # Main model classes
├── data_utils.py              # Dataset and vocab handling
├── train.py                   # Training loop
```

```
├── evaluate.py          # Testing and metrics
├── config.py            # All hyperparameters
└── requirements.txt     # Dependencies
```

Everything is modular so you can modify one part without breaking others.

What I Learned

1. **LSTM vs GRU vs RNN**: LSTM has the most parameters but handles long sequences best. GRU is a good middle ground. Vanilla RNN is fastest but forgets long-term patterns.
2. **Hidden size matters**: I tried 128, 256, and 512. Below 256, accuracy suffered. Above 256, my GPU crashed. 256 was the sweet spot.
3. **Learning rate scheduling**: Starting with 0.001 and dropping by half when validation loss plateaus helped a lot. Otherwise the model would oscillate near the end.
4. **Batch size trade-off**: Larger batches (128) trained faster but needed too much memory. Smaller batches (16) fit in memory but took forever. Settled on 32 as a compromise.

How to Run This

Setup

```bash
git clone https://github.com/navyasgr/Seq2Seq-Aksharantar-IITM-navya.git
cd Seq2Seq-Aksharantar-IITM-navya
pip install -r requirements.txt
```

Training

```bash
# With GPU (if you have enough memory)
python train.py --device cuda --epochs 50

# With CPU (slower but reliable)
python train.py --device cpu --epochs 50

# With limited GPU memory
python train.py --device cuda --batch_size 16 --hidden_dim 128
```

Testing

```bash
```

```
python evaluate.py --model_path checkpoints/best_model.pth
```

Interactive Demo

```python
from inference import transliterate

print(transliterate("namaste"))  # नमस्ते
print(transliterate("dhanyavaad"))  # धन्यवाद
```

Results (On Limited Hardware)

Given my hardware constraints, I tested on a smaller subset:

- **Training samples**: 8,000 word pairs
- **Validation**: 1,000 pairs
- **Test**: 1,000 pairs

After 30 epochs (took about 3 hours on CPU):

- Validation loss: 0.89
- Test accuracy: 81.3%
- Character-level accuracy: 94.7%

The model works well for common words but sometimes struggles with:

- Very long words (10+ characters)
- Words with doubled consonants
- Rare character combinations

Future Improvements

If I had better hardware, I'd try:

1. **Attention mechanism** - Let decoder look at all encoder outputs, not just final state
2. **Beam search** - Consider multiple possibilities instead of just picking the most likely character
3. **Bidirectional encoder** - Read the input forward and backward
4. **Larger hidden dimensions** - 512 or 1024 might capture more patterns
5. **Full dataset** - Train on all 100k+ pairs instead of a subset

Files Included

- `transliteration_model.py` - Core model architecture
- `data_utils.py` - Data loading and preprocessing
- `train.py` - Training script with logging
- `evaluate.py` - Evaluation metrics
- `inference.py` - Simple inference interface
- `config.py` - Hyperparameter configuration
- `vocab_builder.py` - Vocabulary construction
- `requirements.txt` - Python packages needed

References & Learning Resources

I learned a lot from:

- The original seq2seq paper by Sutskever et al. (really clever idea!)
- CS224N Stanford lectures on RNNs
- PyTorch documentation on pack_padded_sequence (confusing at first but super useful)
- Papers on attention mechanisms (Bahdanau 2015)

Acknowledgments

- AI4Bharat for the Aksharantar dataset
- IIT Madras for the assignment structure
- Online communities who answered my CUDA memory questions

Note: This was developed on limited hardware (MX250 2GB). The architecture and code are production-ready and scalable - just needs proper GPU infrastructure for full dataset training.

Contact: Issues and questions welcome via GitHub issues.