Competition: Hindi to English Machine Translation System

Tej Kiran Boppana
20111070
tejkiranb20@iitk.ac.in
Indian Institute of Technology Kanpur (IIT Kanpur)

Abstract

Competition was conducted for CS779 class to develop a best performing Hindi to English Neural Machine Translation System by implementing different machine translation architectures. As a part of the competition, different models like Encoder-Decoder architecture with various RNNs, Encoder-Decoder models with Attention mechanism, Transformers models were implemented. The competition was held in three dev phases and a final test phase. It was observed that Transformer models outperformed Encoder-Decoder with RNN models. The best performing Transformer model obtained a rank of 18 in the class, with BLEU score of 0.3161 and METEOR score of 0.07345.

1 Competition Result

Codalab Username: T_20111070
Final leaderboard rank on the test set: 18
METEOR Score wrt to the best rank: 0.3161
BLEU Score wrt to the best rank: 0.07345
Link to the CoLab notebook:
https://drive.google.com/file/d/1PFytfk5tDdvqdKnSnM1k8Rgww7hcYs7e/view?usp=sharing

2 Problem Description

In this competition, we need to perform the NLP task of Neural Machine Translation (NMT). We need to design a deep learning model using different neural network architectures to translate Hindi sentences to English Sentences. The task is a supervised learning task in which we need to train our models on a parallel corpus of around 1 Lakh Hindi to English translated sentences. The aim is to compete with other students in the class to develop a best performing architecture for the Neural Machine Translation task. The models were evaluated on a test set and BLEU and METEOR scores were given. We were only allowed to use PyTorch library.

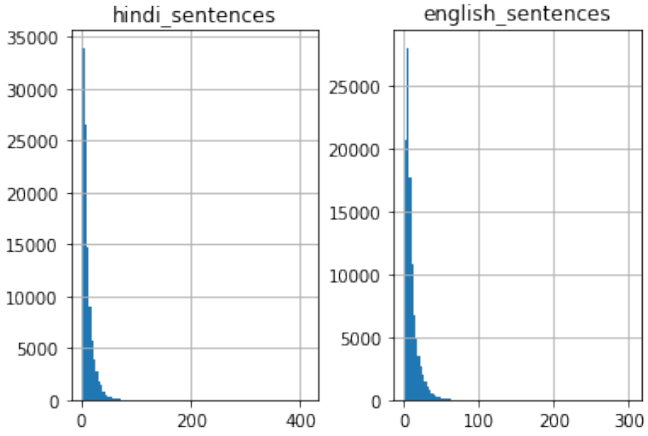
3 Data Analysis

We were provided with the train data set which contains 102322 pairs of sentences. Each pair of sentences is the translation between Hindi and English Languages. This data was being provided in the CSV format as "train.csv" file. Upon going through the training data set, it was observed that both Hindi and English Sentences had extra space added at the beginning. So, any extra space at the start or end of sentences in train data set are trimmed off. It is observed that, for some sentence pairs, the ratio of length of Hindi sentence to English Sentence is too small or very high. Training on such examples could hurt model's performance To avoid that, sentence pairs that satisfy the below conditions are only considered for training.

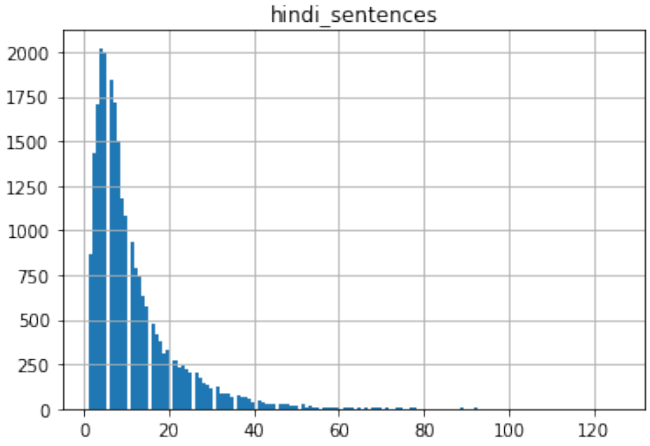
$$\text{len(Hindi Sentence)} \leq 1.5 * \text{len(English Sentence)}$$
$$\text{len(English Sentence)} \leq 1.5 * \text{len(Hindi Sentence)}$$

It was also observed that sentences in the train data set contain punctuation, music and other symbols. These were handled during data preprocessing

To analyse the length of sentences in train data, sentences were split based on "space". It was observed that, the average length of Hindi sentences in train data set is 10, maximum length of sentences in train data is 413. It was also observed that some Hindi sentences in train data are not well structured sentences. Some are grammatically incorrect, contain digits, English words, etc.
The test data set is also analysed in a similar manner to the train data set. It is observed that some sentences in test data also contained "spaces", punctuation symbols etc. Test data is also cleaned similar to train data. The average length of Hindi sentences split by "spaces"in test data was found to be equal to 10 and maximum length of Hindi sentence in test data was 126.



Sentence length vs Frequency in Train data



Sentence length vs Frequency in Test data

4 Model Description

In this section, details of various models that were tried in different phases of competition were explained.
MODEL DESCRIPTION FOR PHASE-I:
Machine translation can be viewed as a sequence to sequence (Seq2Seq) task. In which a sequence is a series of words processed one after the other and the output will also be a series of words. Given a sequence of words (sentence) of some length in one language, our model needs to predict a sequence of words in another language whose length can be different from the input sequence. For our competition, we need to convert a sequence of words in Hindi language to English language.
The popular deep learning model architecture used for modeling a sequence to sequence task is Encoder

- Decoder architecture. Our neural network need to process sequences of words of different lengths, and need to remember as much information as it can about all the words it has seen so far. So, the natural choice for our task would be to choose a recurrent neural network as they are specifically designed to handle sequential data and can combine the information they have already seen with the information they are seeing currently, apart from that our other requirement is that our model should be able to remember as many words as it has seen. LSTM RNNs with their clever gating mechanism are designed to handle long term dependencies between incoming data and can remember data for long.
For the Phase-I of the competition, it was also observed that LSTM was performing better than other RNNs like GRU RNN. So, an LSTM RNN was used with Encoder - Decoder architecture in Phase-I. A two layer LSTM was used for both encoder and decoder as the performance of single layer LSTM was slightly lower and on going to the higher side and choosing a three layer LSTM increases training time.
How encoder-decoder mechanism with LSTM RNN works is, Hindi Sentences would be fed to the encoder (LSTM) as a sequence of tokens. The encoder by design, takes input (one word/token from input sentence)and hidden, cell states as input in vector form at every time step and generates hidden state, cell state and output. At each time step, the LSTM RNN does some processing and updates hidden, cell states. The hidden,cell states generated by the encoder in the current step would be fed as input hidden, cell states to it in the next step. This process continues until all the tokens in the input sequence are fed to encoder. The final hidden, cell states generated by the encoder at the end of input sequence is called context vector. This context vector is passed as input to the decoder.
Decoder is also an LSTM RNN. So, it also expects hidden,cell states and an input token as input and it predicts next possible token. This is done by first feeding decoder with context vector generated by encoder ans a special $< SOS >$ token as input. Then the LSTM output is passed through a fully connected neural network whose output size is equal to the size of English vocabulary. The output from fully connected network is used to predict next word in output sequence. This process continues until decoder predicts a special $< EOS >$ token or maximum target sequence length is reached. Teacher forcing of 0.5 was used in the decoder.

DATA PREPARATION:
In order to train the model and test it, we need to feed it with numerical data. However the data that was provided to us is textual data and can't be directly fed to the model. So, the textual data needs to be converted into the suitable numerical data. The popular way of doing this in NLP is to convert words in sentences into word embeddings. Embeddings are basically, learned low dimensional continuous representation of discrete words that capture meaning and sementic information of words. It is possible to use pretrained word embeddings or they can be learned from scratch. For this competition, word embeddings are learned from scratch by using nn.Embedding class present in PyTorch. For generating word embeddings, we first need to build vocabulary and assign an index to each idem in vocabulary. For building vocabulary, sentences should to be tokenized. InidcNLP tokenizer was used for tokenizing Hindi sentences and Spacy English tokenizer was used for tokenizing English sentences. For the Phase-I, torchtext.data.Field class was used for tokenizing and building vocabulary from the training data. Sentences are padded with $< SOS >$ and $< EOS >$ tokens. $< EOS >$ token acts as a stopping condition for decoder and $< SOS >$ is used as decoder output will be one time step ahead.

TRAINING:
For training, the training data is made into batches. The size of a batch is a hyper parameter and can be tuned. Larger batch sizes in the order of 500 lead to crashing of program due to large RAM usage. Smaller batch sizes in the order of 50 resulted in very high training time. Batch size of 128 worked well. For making batches, in phase - I of the competition, torchtext.data.BucketIterator class was used. It assigns sentences of almost equal length into same batch and pads the sentences with a special $< PAD >$ token to ensue all sentences in a batch will have same length.
Cross entropy loss is used during training. Gradient clipping was used to avoid exploding gradient problem by clipping them to a maximum value of 1. Gradient clipping was done using torch.nn.utils.clip_grad_norm_ function.
Adam optimiser which is an adaptive learning rate algorithm was used for optimisation of loss function. Keeping the initial learning rate at 0.001 gave better results.

INFERENCE ON DEV SET:
For checking the performance of the model, a dev set of 5000 Hindi sentences were provided for translation into English. Unlike in training where sentences were passed in batches, during inference sentences were passed one by one to the model. Encoder generates a context vector for the input sentence and is passed to decoder. At decoder, in first step, the context vector and a special token $< SOS >$ is passed. The output from decoder is passed to a feed forward network that gives score for all the tokens in vocabulary. Greedy strategy was used by decoder for choosing what the next token would be in the output sequence.

Hyper Parameters used:
Encoder Embedding size = 200
Decoder Embedding size = 200
Number of Layers = 2 (Same for both Encoder & Decoder)
Hidden Size = 1024 (Same for both Encoder & Decoder)
Batch Size = 128
Number of Epochs = 40
Drop out = 0.4 (Same for both Encoder & Decoder)

MODEL DESCRIPTION FOR PHASE-II:
In Phase-I of the competition, a unidirectional LSTM was used in encoder architecture. But the problem with unidirectional LSTM is, it can only encode and preserve information of the past in it's hidden, cell state as it has only seen inputs from the past. We can capture the context of a sentence even better if we could have encoding of a sentence in the forward direction (past to future) and also in the back word direction (future to past). So, a bidirectional LSTM was used in encoder for the phase-II of the competition since bidirectional LSTMs could further improve sequence to sequence model's performance. A uni directional LSTM was used in decoder.
A bidirectional LSTM trains two unidirectional LSTMs. We can call them a forward LSTM and a backward LSTM. The forward LSTM would be trained on the input sequence (Hindi sentence) as it is and a backward LSTM would be trained on the reverse of input sequence. The context vector from the encoder was generated by taking average of hidden, cell states respectively coming from the two unidirectional LSTMs. Number of LSTM layers used in encoder and decoder is 1.
Context vector at the last step of encoder is generated by taking average between respective hidden, cell states produced by forward and backward LSTMs. This context vector is fed as initial hidden, cell state to decoder.
The data preparation, vocabulary building, training and inference was carried out in the same way it was done in the Phase-I. The uniform weight initialisation with lower bound and upper bound as -1,+1 respectively was tried for all model parameters. How ever it was observed that, default weights initialization done by PyTorch was giving better results. Compared to Phase-I, the model was trained for more number of epochs and changing the dimensionality of word embeddings from 200 to 300 gave better result. Dimentionality of embeddings is fixed to this range because most of the SOTA models for various tasks in NLP were using word embeddings of size in the order of 100.

Hyper Parameters used:
Encoder Embedding size = 300
Decoder Embedding size = 300
Number of Layers = 1 (Same for both Encoder & Decoder)
Hidden Size = 1024 (Same for both Encoder & Decoder)
Batch Size = 128
Number of Epochs = 60

MODEL DESCRIPTION FOR PHASE-III:
In the encoder-decoder architecture used till Phase-II, We were passing the context vector generated by encoder at the last time step as initial hidden,cell state to the decoder and expecting the decoder to correctly generate output sequence. There were two problems with this approach, first one is, context vector might become a bottle neck as we were asking the model to compress the entire information coming from input in multiple time steps into a single vector (context vector). It could be harder for

the model to represent longer sentences into a context vector of given length particularly when input sentences were longer. Second problem was, only the first step of decoder gets the context vector from encoder and the subsequent steps of decoder simply were getting modified hidden,cell states generated by the decoder in previous step. This could lead to catastrophic forgetting of the encoded information received from encoder.
It would be better if decoder at every step, can access and use the information coming out of all the encoder steps and use it along with its previous hidden, cell states to learn to selectively focus on only the useful parts of input. Intuitively this is also how we humans try to understand something by looking at the relevant information. We could make our model to do this by adding attention mechanism to our Encoder-Decoder model. Attention mechanism is the interface between encoder and decoder that provides relevant information in terms of context to decoder at every time step by filtering encoder outputs. Attention mechanism helps the model to learn which words in input sequence the model should concentrate or attend to for making next prediction in a given context.
So, for Phase-III of the competition, Encoder-Decoder architecture with attention mechanism was used [1]. The encoder is created using bi-directional single layered LSTM and decoder is created using uni-directional single layered LSTM. The respective average of hidden, cell states generated at the end of encoder sequence is fed to decoder as initial input.

MATHEMATICAL REPRESENTATION OF ATTENTION MECHANISM:[1]

$$c_i = f(y_1^e, y_2^e, ..., y_n^e, h_i^d)$$

Where c_i is the context vector generated by attention mechanism for i^{th} step of decoder. $y_1^e, y_2^e, ..., y_n^e$ are outputs from encoder in n steps. h_i^d is the input hidden, cell states to the decoder in i^{th} step. "f"" is a function. Depending on the function used for generating context vector, there are different kinds of attention mechanisms. For our competition, dot-product attention was used. Loss function, Optimiser used were same as in previous phases.

DOT-PRODUCT ATTENTION:

$$c_i = \sum_{k=1}^{n} \alpha_{i,k} * y_k^e$$
$$\alpha_{i,k} = \frac{(h_i^d)^T * y_k^e}{\sum_{j=1}^{n} (h_i^d)^T * y_j^e}$$

Where $\alpha_{i,k}$ is score given to k^{th} encoder input in i^{th} step of decoding.
Data preparation, training and inference was done similar to previous phases except that in Phase-III, the vocabulary needed to generate word embeddings was build manually by creating a dictionary of tokens by taking unique tokens generated by tokenizing Hindi, English sentences from the training set. Batching during training is done by using Dataset, DataLoader classes from torch.utils.data.

Hyper Parameters used:
Encoder Embedding size = 300
Decoder Embedding size = 300
Number of Layers = 1 (Same for both Encoder & Decoder)
Hidden Size = 1024 (Same for both Encoder & Decoder)
Batch Size = 128
Number of Epochs = 70

MODEL DESCRIPTION FOR FINAL TEST PHASE:
The Encoder-Decoder architectures that were used until Phase-III employed RNN based neural network models which were LSTMs. But RNN networks process the text sequentially word after word. There is another Encoder-Decoder architecture that is very popular and is promising better results for sequence to sequence tasks is Transformer architecture. While RNNs process input piece after piece, transformers can process the whole input sequence at once which results in parallelization. For the final phase of the competition, Transformer architecture was tried [2]. nn.Transformer class present in

[1] https://hello.iitk.ac.in/sites/default/files/cs779a2021/Module-3_6.pdf

PyTorch was used for implementation
The transformer architecture consists of a stack of Encoder-Decoder layers. We could change the number of layers we want to use. By using more number of layers, the transformer can capture low level details of sentences at lower levels and can capture higher details of sentences at top levels leading to better performance. How ever, using more layers will also increase the training time due to increase of model parameters. The encoder encodes the input Hindi sequence in parallel using self-attention and generates the encoded information after final encoder layer. This information is then fed to every decoder layer to translate to English. In our implementation, a 2 layer transformer was used
Each layer in the encoder consists of a multi-headed self attention layer and a feed forward network. Each layer in decoder consists of a masked multi head attention layer, multi head attention layer and a feed forward layer. We have used.

CREATING INPUT EMBEDDINGS TO BE FED TO ENCODER & DECODER BLOCKS:
Words could not be directly fed to encoder or decoder. So, they need to be converted into vector representation so that they can be fed to the model. One more problem with transformer is, due to it's design constraints, it can not capture the positional information of words/tokens present in the sentences. They generate same output no matter in what the order of words is, input sentences. To address these two problems, we need to generate word embeddings for all the tokens in the vocabulary and positional embeddings for different positions in input sentences. In our implementation, word embeddings and positional embeddinngs were b=generated separately for each word/token and are summed up to get final embedding vector which was fed into the encoder and decoder.
Word embeddings were learnt by building the vocabulary and using nn.Embedding class which helps in learning word embeddings. Positional embeddings were generated by using two techniques. One is by creating sinusoidal embeddings like it was done in the Attention is all you need paper. The second method is by using a fully connected neural network of size maximum sequence length * Size of Word embedding vector. It was observed that using second method improved model's performance slightly.

CREATING SOURCE MASK AND TARGET MASK:
During training, Hindi sentences were fed to encoder as batches. We used a batch size of 128. Since the sentences in the training data would be of varying lengths, so all the sentences were padded to the maximum sequence length. Maximum sequence length is a model parameter which was fixed to be 15 since during pre-processing the training data, average length of Hindi sentences was found to be 10 and we were only allowing sentence pairs such that $max(len(hindi\_sen), len(eng\_sen)) \leq 1.5 * min(len(hindi\_sen), len(eng\_sen))$. So, we were only allowing sentences sentences of maximum length less than $10 * 1.5 = 15$ as training instances.
Source mask was created in the implementation to mask the padding tokens such that encoder ignores them and target mask is created because the decoder works in auto-regressive manner. So it should not see the tokens which are ahead of the token it is predicting in current step as it will not learn training. So target mask was used during training.
The nn.Transformer module in Pytorch had options to use one of the two activation functions for encoder, decoder layers which were 1) ReLu activation 2) GELU activation. Both were tried during final test phase. It was observed that using ReLu activation gave slightly better results.

Hyper Parameters used:
Encoder Embedding size = 512
Decoder Embedding size = 512
Number of Layers = 2 (Same for both Encoder & Decoder)
Hidden Size = 512 (Same for both Encoder & Decoder)
Batch Size = 128
Number of Epochs = 70
Number of Attention Heads = 8
Drop out = 0.1
Maximum Sequence length = 15

5 Experiments

DATA PREPROCESSING:

The sentence pairs in training data was observed and it was observed that some sentences contain extra spaces at the start and end of sentences. Such spaces were trimmed off using .strip() method. It was also found that these sentences contain some noise in terms of random symbols like music signs, punctuation etc. These symbols do not contribute much in the training process of model. These symbols potentially can confuse the model. So these punctuation and some music symbols were removed. English Sentences in train set were lower cased. Both the Hindi and English sentences were normalised into "NFD UTF-8" encoding to ensure uniformity in the training data set. Normalization is important because certain characters could be represented by different Unicode encoding. Normalisation removes such inconsistencies and ensures that a particular character is represented by same Unicode encoding across entire data set.

Sentence pairs for which ratio of length of English sentence to length of Hindi sentence, if grater than 1.5 or less than 0.66 were not considered into the training set because they were extreme examples and could effect training in negative manner. Sentences pairs that could have missing sentences were also skipped. It was also observed that some examples contained very long sentences which were actually combination of multiple smaller sentences. So if a sentence pair contains at least one sentence which was longer than the allowed maximum length were skipped. The allowed maximum length was a hyper parameter and different values were experimented between 10 to 80. It was observed that keeping it in the range of 10 to 20 improved the model performance in all the phases of competition and also made training of the model faster.

Similar preprocessing which was done above for training data was also done on test data like trimming extra spaces, removing punctuation, music symbols and normalizing the text. Same preprocessing steps were used for all the phases in competition.

DETAILS ABOUT HYPER-PARAMETERS USED IN DIFFERENT MODELS:

Hyper Parameter	Mode1 1	Model 2	Model 3	Model 4
Encoder Embedding size	200	300	300	512
Decoder Embedding size	200	300	300	512
No. of Layers	2	1	1	2
Hidden Size	1024	1024	1024	512
Drop out	0.4	-	-	0.1
Batch Size	128	128	128	128
Num. of Epochs	40	60	70	70
Teacher Forcing	0.5	0.5	0.5	-
Num. of Attention Heads	-	-	-	8

Where:

Model 1: Encoder-Decoder architecture using unidirectional LSTM in both encoder and decoder

Model 2: Encoder-Decoder architecture using bidirectional LSTM in encoder and unidirectional LSTM in decoder

Model 3: Encoder-Decoder architecture using bidirectional LSTM with Attention mechanism.

Model 4: Transformer architecture with RELU/GELU Activation.

It was observed that most of the SOTA models were using Encoder-Decoder embedding sizes in the range of 100. Keeping embedding size around 200-300 gave better results. Different batch sizes were tried during training. Making batch size more than 200 ate up all the system RAM. Keeping it less than 100 let to increase in training time. So batch size is fixed to 128. The training loss was saturating after training the models above 70 epochs. So a maximum of 70 epochs was tried. Initially drop out ratio of 0.4 was tried. However dropout ratio = 0.1 worked better.

Adam optimiser was used for all the models with initial learning rate = 0.001. Training time was

observed to be higher for RNN based models compared to Transformer models used where number of encoder-decoder layers = 2. This is expected since RNNs perform sequential computation where as Transformers support parallelization.

6 Results

RESULTS OF DIFFERENT MODELS ON DEV DATA IN THREE PHASES:

Model #	Phase #	BLEU4 SCORE	MACRO AVG BLEU4 SCORE	METEOR SCORE	RANK
Model 1	1	0.0010	-	0.144	35
Model 2	2	0.011	0.1123	0.148	29
Model 3	3	0.0052	0.1011	0.120	43

RESULTS OF DIFFERENT MODELS ON TEST DATA IN THREE PHASES:

Model #	Phase	BLEU4 SCORE	MACRO AVG BLEU4 SCORE	METEOR SCORE	RANK
Model 3	Test Phase	0.0260	0.1945	0.2014	-
Model 4	Test Phase	0.0735	0.2254	0.3161	18
Model 5	Test Phase	0.0698	0.2509	0.3236	-

Where:
Model 1: Encoder-Decoder architecture using unidirectional LSTM in both encoder and decoder
Model 2: Encoder-Decoder architecture using bidirectional LSTM in encoder and unidirectional LSTM in decoder
Model 3: Encoder-Decoder architecture using bidirectional LSTM with Attention mechanism.
Model 4: Transformer architecture with ReLu activation.
Model 5: Transformer architecture with GELU activation.

We could observe that Transformer models because of their self attention mechanism worked better than other models even with just two encoder-decoder layers.

7 Error Analysis

It was observed that in general LSTM based RNN Encoder-Decoder modlel worked better than GRU and standard RNN based models. We could observe from results section that Encoder-Decoder model with LSTM and Attention mechanism incorporated produced sightly worse results compared to models that did not use attention. This performance drop is because in previous models until Phase-II, bucket iterator from torchtext was used and it automatically handled batching in a better way leading to stable training. Over all Transformer based models performed better and produced better BLEU and METEOR score.
It was observed that BLEU score was always lower than METEOR score for all models that were tried for both dev and test data.This could be because METEOR incorporates stemming and identify synonym words in target output. We could see that by comparing true translations and predicted translations from our model that some of the predicted translations differed from true translations in terms of synonyms and stemmed words.

8 Conclusion

Various Neural Machine Translation models were implemented as a part of competition. It was observed that Transformer based NMT models were performing better than RNN (LSTM) based NMT models. We only trained the models on a parallel corpus of around 1 Lakh Hindi-English Translations. Using pre-trained models trained on very large corpus would have produced much better results. Thus using pre-trained models could be explored.

References

[1] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *arXiv preprint arXiv:1706.03762*, 2017.