Competition: Hindi to English Machine Translation System

Ankita Dey
20111013
{ankitadey20}@iitk.ac.in
Indian Institute of Technology Kanpur (IIT Kanpur)

04-05-2021

Abstract

Machine Translation is the automatic translation of text or speech from one language to another. While Statistical and Rule based Machine Translation were popular in the past, the latest entrant to this task is Neural Machine Translator. The competition requires submission of Neural Machine Translator to convert Hindi text to English text. Sequence to Sequence architecture is perhaps the most common way to do the task and has been used in this project. The final submission got a rank of 55 in the leaderboard with METEOR Score of 0.150 and BLEU Score of 0.0152.

1 Competition Result

Codalab Username: A_20111013
Final leaderboard rank on the test set: 55
METEOR Score wrt to the best rank: 0.150
BLEU Score wrt to the best rank: 0.0152
Link to the CoLab/Kaggle notebook: link

2 Problem Description

The task is to implement a Neural Machine Translation (NMT) model that will translate Hindi text to English. Encoder-Decoder architecture is to be used to implement the model. There is no limitations on what neural architecture to use to build the Encoder and Decoder. However, there are several constraints to make the competition more interesting.

Firstly, only Pytorch can be used to build the model and no other libraries including the ones built on top of Pytorch can be used. Secondly, no pre-trained word embeddings can be used. Thirdly, for pre-processing, the only allowed libraries are Spacy, NLTK and IndicNLP. Apart from these only NumPy or SciPy is allowed. This means popular libraries like Pandas, Torchtext and Hugging Face (for transformers) aren't allowed. In fact, transformers have to be implemented from scratch if it is implemented. Basically model implementation has be done using Pytorch only. Also, implementation has to be done on Python3. GPU support can be obtained from Google colab.

3 Data Analysis

The training dataset has been provided in csv format. It consists of 3 columns, first one is the serial number or row number; second is the Hindi sentences (under the heading 'hindi') and third is the English translation of those Hindi sentences (under the heading 'english').

There are 102322 sentence pair in total. Without any pre-processing, the median length of the Hindi and English sentences are 36 and 35, respectively. Length of 90% of the Hindi and English sentences are within 112 and 117, respectively. However the longest Hindi and English sentences are whooping 2088 and 1880 in length respectively!

There are at least 55 Hindi sentences which are blank or just have special characters or punctuation and at least 987 Hindi sentences which are completely made up of English alphabets or numbers. In this project, these sentences have been considered as noise.

Proportionately, in each of the test dataset, each consisting of the 5000 Hindi statements, around 50-60 sentences are completely composed of English alphabets or number and there are single digit blank sentences or sentences with only special characters. However in the final test dataset, noisy data was much less with median length of 36, highest length of 597 and only 15 sentences completely made up of English alphabets and numbers, out of 24102 total sentences.

4 Model Description

The model used in this competition is a very basic sequence to sequence model with LSTM Encoder and Decoder. No special techniques like attention or beam search have been used. Sequence to sequence architecture, as the name suggests, converts one sequence to another; in this case, converts a sequence of indices representing Hindi words to a sequence of indices representing English words.

4.1 Why LSTM?

The translation of a sentence to another requires knowledge of previous words. Hence at the least, Recurrent Neural Network, RNN, is required to do the job since they have loops allowing information to pass on to the next step. However RNN suffers from the problem of vanishing gradient due to which it isn't able to capture long term dependency. Therefore LSTM, which is a special kind of RNN specifically created to learn long term dependencies through additional cells, forget, input and output gates, is used for Encoder and Decoder.

In LSTM, information can flow almost unchanged through something called the cell state, due to little linear interaction. Information flowing through cell state is regulated using structures called gates. Firstly, Forget gate (made up of a sigmoid layer that gives a number between 0 and 1 with 1 meaning keep the cell state and 0 meaning forget the cell state) regulates how much of the existing memory in cell state to forget. Secondly, Input gate regulates how much of the new cell state to keep. It multiplies output of a sigmoid layer that decides which values to update, with the output of a tanh layer, that creates a vector of new candidate values that could be added to the cell state. Finally, the Output gate regulates how much of the cell state should be exposed to the next layers of the network. For this a sigmoid layer decides parts of the cell state to output. Then, the cell state is put through tanh (to push the values to be between 1 and 1) and it is multiplied by the output of the sigmoid gate, so that only filtered parts of cell states goes to the output. [1] is a good step by step explanation of LSTM without going into too much mathematical details.

GRU [1], a variation of LSTM having only 2 gates, Reset and Update, was also used to create Encoder and Decoder in the Sequence to Sequence architecture in the first phase. But due to poor results, that model was replaced with LSTM Encoder Decoder in the first phase itself.

4.2 Overview of working of model

With respect to a single sentence for better understanding; each word of the sentence (appended by a start and end token at front and back respectively) is given to the Encoder LSTM one word at a time. Actually a batch containing one word of different sentences is given to Encoder at each time step but more on that under Experiment section. At each time step, the Encoder produces a context vector which is given as input to the Encoder for the next time step (it is basically a loop structure) along with the next word of the sentence. Once the end of the sentence is reached (actually the end

[1] `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

of the batch has been reached), the context vector obtained in the final time step is then given to Decoder as input along with ‘Start of the sentence’ token. Now similar to Encoder, the context vector obtained by the Decoder at each time step is given as input to the Decoder at the next time step, along with the predicted word in the previous time step. At each step predicted word is chosen in a greedy manner, that is, the word having maximum probability among all the words in the vocabulary. The prediction is stopped when ‘End of the sentence’ token is predicted or when a maximum length of output is reached. The predictions constitute the output sequence (translated English sentence in this case).

4.3 Model objective (loss) functions

The Model objective is to minimize the loss function, Cross Entropy loss in this case, so that the model output is closer to the true output. During model training, the model weights are iteratively adjusted to minimize the Cross-Entropy loss, also called log loss. The process of adjusting the weights is what defines model training. Each predicted class probability is compared to the actual class desired output and a score/loss is calculated that penalizes the probability based on how far it is from the actual expected value. The penalty is logarithmic in nature yielding a large score for large differences close to 1 and small score for small differences tending to 0 [2].

5 Experiments

5.1 Data Pre-processing

Firstly all English text is lowered in both English text and Hindi text (Hindi sentences also have English texts). Secondly an optional pre-processing is removing all the punctuation in the text. Thirdly, trailing white spaces and blank sentences or sentences having only punctuation and special characters are removed. Fourthly, sentences over 100 characters are removed (100 is chosen as almost 90% of the sentences have length within 100 after pre-processesing). Normalization of Hindi text is also done (using indicnlp) to correct spelling mistakes in an attempt to maintain consistency.

5.2 Creating Vocabulary

The pre-processed data is tokenized (using indicnlp) and each token is assigned an unique index in the source and target language, that is Hindi and English respectively. Tokens which are present at least twice are used to create the Hindi and English vocabulary. Two dictionaries for each language are created; one which has tokens as key and has corresponding index as value and another which has vice versa, that is index as key and corresponding token as value. Thus altogether four dictionaries are created. The vocabulary size finally comes to 14964 for English and 16360 for Hindi, using these methods and the total number of training data is comes to 89194.

5.3 Creating Training Data and Batches

To create training data, firstly, convert the sentence from list of tokens (obtained by tokenization of sentences) to list of indices. Special index 0 (representing unknown) is put for tokens which are not present in vocabulary (because they are seen only once in the training data). Next, sort them primarily according to the length of Hindi sentences, and secondarily according to length of English sentences (so that less padding is required later during creation of batches). Finally, append initial and end tokens to all Hindi and English sentences. Thus, we obtain training data in the form of 2D list having sentences represented by indices.

5.4 Creating Batches

The entire training data won’t be passed to the Encoder at once. So, training data is split into batches with 32 sentences in one batch. Each batch is sent to the model at once. Each column in the batch is an individual sentence with extra padding given at the end of shorter sentences so that each batch

[2]`https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e`

has length equal to length of longest sentence in that batch (sorting by length of sentences was done earlier to reduce the need of padding). Each batch is converted to tensor because the model made using pytorch.nn takes tensor as input and gives tensor as output. At each time step, one row of the batch is passed to the LSTM Encoder, in the next time step the next row of the batch is passed to the LSTM Encoder, and so on. In simple words, single token of multiple sentences are being passed to Encoder at the same time.

5.5 Parameters of Encoder and Decoder

The Encoder and Decoder input size must be equal to the vocabulary size of the source and target language (i.e. Hindi and English vocabulary size) respectively. This would be the size of one hot Encoder in which each vector represents a unique token given by the index where it is 1 (rest all indices are 0).

Each token of a vocabulary is converted to a dense representation of words, that captures semantic information of the words in the vocabulary of the training set, called word embedding for which we have to specify the embedding size. In this project embedding size of 300 has been used. Common embedding size used was seen and 300 was the largest, so fixed upon 300 size. No pre-trained embedding is used as per rule.

Basically hidden state and cell states are two output of LSTM cell which together constitute the context vector. This context vector (fixed size) captures the essence of the source sequences and is given to Decoder along with target language (English) vocabulary to get translated sentences. Size of hidden state is chosen as a power of 2 value. Here it is specified as 1024. Here again 1024 was decided upon because it was found to be the most common hidden state size used in Sequence to Sequence model.

Number of LSTM layers used for Encoder and Decoder is specified. If n LSTM layers are used, then n context vectors are obtained. 2 layers [2] are used in this model.

Dropout is a regularization technique of ignoring random neurons during training to prevent overfitting. The probability of ignoring each neuron is specified as 0.5.

5.6 Encoder and Decoder

All the neural network functionality (including the various losses) are created using pytorch nn package[3]. The nn package defines a set of Modules, which are roughly equivalent to neural network layers[4]. It inputs and outputs tensors (which is why the batches given as input were converted to tensor earlier).

Now, a little bit about the code with respect to nn.Module; nn.Module[5] is subclassed to create custom[6] Encoder and Decoder and forward method is defined for both. In forward method, Tensor of input data is accepted, and Tensor of output data is returned. Modules defined in the constructor, as well as arbitrary operators on Tensors, can be used in forward method. Using the constructor, when a Encoder or Decoder LSTM object is created, the attributes of the Encoder LSTM and Decoder LSTM are initialized, which are then used in forward method.

nn.Embedding[7] takes a list of indices as input, and outputs the corresponding word embeddings. Embedding matrix is then given to the LSTM (nn.LSTM[8] is used) which additionally uses previous time step's hidden state and cell state to produce hidden state and cell state that will be used by the LSTM in next time step. Dimension of all the tensors are commented in code. (tensor.shape can be

[3]`https://pytorch.org/docs/stable/nn.html`
[4]`https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html`
[5]`https://pytorch.org/docs/stable/generated/torch.nn.Module.html`
[6]`https://pytorch.org/tutorials/beginner/pytorch_with_examples.html#pytorch-custom-nn-modules`
[7]`https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html`
[8]`https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html`

used to get the dimension of the tensor).

The same arguments are passed to Decoder LSTM as passed to Encoder LSTM except for an extra argument, output size. The output size of Decoder LSTM is same as input size of Decoder LSTM, that is, target (English) vocabulary size. The Encoder and Decoder are identical and code for Encoder and Decoder is the same, except that the prediction vector is also returned from the Decoder. It has the probability of each token in the vocabulary being the correct prediction for all the tokens of the batch.

5.7 Sequence to Sequence Model

Now that Encoder and Decoder has been created, they will be used to create the Sequence to Sequence Model. While training a pair of source and target sentences, the batch created from training data earlier, will be passed to the model. The model will pass the source batch to Encoder, obtain context vector, pass the context vector obtained at each step along with previous prediction (initially, pass start of string token) to Decoder. At each time step, the prediction with highest probability is considered as correct and appended to the output tensor, that would finally contain translation of each sentence of the batch. This outputs tensor is returned by the model. This is a greedy method of obtaining best prediction.

5.8 Loss and Optimiser

Pytorch implementation[9] of Cross-Entropy loss is used in this model. The reason for using Cross Entropy Loss is that it is one of the most popular loss in Neural Network. In effect it is same as Negative log likelihood.

Optimizer determines how long it would take to get good results using the model. Pytorch implementation of Adam optimizer is used in this model. The learning rate has been specified to its default value of 0.001[10].

A loss function determines how far off the predicted sentences is from the actual target sentences. To minimize the loss, gradient descent is used with Stochastic gradient descent or SGD being a very common method. To understand Adam, first a quick overview of SGD. SGD approximates gradient using a single training example (picked uniformly randomly) at a time. This results in much faster convergence than taking the gradient of the entire training set. However, SGD suffers from oscillation near local minima when the slope of one dimension is much higher than the other because of its constant learning rate [[3]]. This slows down convergence. Momentum [4] is a method that helps accelerate SGD in the relevant direction and dampens these oscillations.

Adagrad [5] is an algorithm that automatically tunes the learning rate at each time step for every parameter based on the past gradients that have been computed for that parameter. Root mean square prop or RMSprop is another adaptive learning algorithm that tries to improve AdaGrad[11]. Instead of inefficiently and aggressively storing previous squared gradients like AdaGrad, the sum of gradients is recursively defined as a decaying average of all past squared gradients in RMSprop [3]. The authors of [6] describe Adam as combining the advantages of these two methods and that Adam can efficiently solve practical deep learning problems. In addition to storing an exponentially decaying average of past squared gradients like RMSprop, Adam also keeps an exponentially decaying average of past gradients, similar to momentum[3]. In [3], Adam has been said to be the best optimizer overall. This (paired with not having to choose a learning rate) is the reason Adam has been used in this project.

[9]`https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html`
[10]`https://pytorch.org/docs/master/generated/torch.optim.Adam.html`
[11]https://www.geeksforgeeks.org/intuition-of-adam-optimizer/

5.9 Training

Since training data is too large, the entire data can't be passed to the model at once and the data will be passed in batches. The source and target from batches created earlier will be passed to the model one batch at a time. The number of epoch says how many times the entire training data updates the model. Generally 30 or more epoch gives good results. The loss value for every epoch is calculated. The gradients for weights biases is calculated using back-propagation. The gradient value is clipped to avoid exploding gradient problem is it exceeds 1. After every epoch, it is checked if the current state of the model gives minimum loss till now. If yes, the current state of the model is saved. Finally the model is saved at best loss obtained till now and in testing phase the model state with minimum loss is used. Note that the minimum loss doesn't always give best results in training data due to possible over-fitting.

5.10 Testing

The testing phase is very similar to training phase except that now we don't have to worry about loss or optimizations. The model state with smallest loss is used during testing phase. Firstly, the test Hindi dataset is pre-processed by normalizing it (using indicnlp) and optionally, removing punctuation like it was done for training data. Of course no sentences can be removed even if it's made up entirely of English alphabets or numbers or special symbols or even if it is blank!

Next, input Hindi sentence is tokenized and, start and end tokens appended. This tensor, having index of tokens of current sentence, is then passed to Encoder. Context vector (hidden, cell states) obtained from Encoder is passed to Decoder along with start of string token in the initial step. The most recently obtained translated token and context vector of previous time step is passed to Decoder at every step. When end of string token is predicted by Decoder or maximum sentence length is reached, the process is stopped. In the mean time, index of English translation from Decoder (the translated token obtained in each time step from Decoder) has to be appended to a list. Conversion of these indices to English tokens using index to string dictionaries created earlier would finally give the English translation of Hindi test sentence.

6 Results

Phase	BLEU Score	METEOR
1	0.0022	0.163
2	0.0142	0.158
3	0.0047	0.108
Final	0.0152	0.150

There was minor differences and fine tuning in the model of each phase but the basic model was Sequence to sequence model with LSTM for Encoder and Decoder in all. For example, in the final phase, instead of using the model state after all the epoches, the model state with minimum loss is used for test phase. This helped improve the result.

7 Error Analysis

Between GRU (model not submitted, only output submitted in phase 1) and LSTM, LSTM performed much better. The reason is although GRU is less complex and faster to train, LSTM is said to work better in longer sequence.

Unlike some of the models in the competition which has much difference between BLEU and METEOR score rank, this model didn't have much difference in the two rank in any of the phases. Both metrics are used to evaluate machine transaltion and the higher the score, the better the translation is to human judgement.

Due to teacher's force ratio (a technique in which the original translated word is given as input to

Decoder instead of the predicted output of previous step) of 0.5, the sentences predicted were grammatically correct but often incoherent. Another error that can can be observed very easily is the repeated words in the output for longer sequences. Plus, in a lot of sentences the first words or first few words of the translation is correct followed by incorrect translation. The reason for that is, most likely, not using attention mechanism.

Also, it is seen that sometimes the output words given by the model is close to meaning to the original translation but not the correct one. As a comparison to human beings, the error in wording is often like the ones non native speakers or people trying to learn a new language makes, for example 'woman' instead of 'girl' or 'tempted' instead of 'tried', which although isn't evident to non native speakers, is clearly evident to native speakers that it is not the most correct usage.

8 Conclusion

For an absolute beginner at Machine Learning coding, this competition is immensely helpful to learn Machine learning specific coding especially in the area of Neural Network. Hands on implementation helps bridge gap between theoretical concepts and their actual implementation. It showed that basic fine tuning of model often helps improve score as well as speed up training. Many applications of beginner's concepts can be seen on hands on implementation, like more epoches doesn't necessarily mean better results due to over-fitting.

Since the model is a very basic one, there are many scopes of improvement. Attention mechanism can be applied to learn longer sequence. Beam search can be used for superior predictions. In fact transformers can be used to improve performance by far.

References

[1] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," 2014.

[2] V. Balakrishnakumar, "A comprehensive guide to neural machine translation using seq2seq modelling using pytorch," in *[Blog] Towards Data Science*, Available at: <https://towardsdatascience.com/a-comprehensive-guide-to-neural-machine-translation-using-seq2sequence-modelling-using-pytorch-41c9b84ba350>. [Accessed: 27-04-2021].

[3] S. Ruder, "An overview of gradient descent optimization algorithms," 2017.

[4] N. Qian, "On the momentum term in gradient descent learning algorithms," vol. 12, pp. 145–151, 1999.

[5] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," vol. 12, pp. 2121–2159, 2011.

[6] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.