CSE472: Machine Learning

Sessional

An Encoder-Decoder Model with Attention for Sequence-to-Sequence Prediction

Collector: 1305017

Coder: 1305057

Trainer: 1305024

Writer: 1305031

Leader: 1305039

Project Overview:

The translation of text by a computer without human involvement is called machine translation. Machine translation utilizes statistical translation models whose parameters stem from the analysis of bilingual corpora. Building statistical translation models is a quick process, but the technology relies heavily on existing multilingual corpora.

Neural machine translation (NMT) is an approach to machine translation that uses a large artificial neural network to predict the likelihood of a sequence of words, typically modeling entire sentences in a single integrated model. It learns to translate by analyzing large amounts of data for each language pair.

Deep neural machine translation is an extension of neural machine translation. Both use a large neural network with the difference that deep neural machine translation processes multiple neural network layers instead of just one.

The encoder-decoder architecture for recurrent neural networks is proving to be powerful on a host of sequence-to-sequence prediction problems in the field of natural language processing such as machine translation and caption generation.

Attention is a mechanism that addresses a limitation of the encoder-decoder architecture on long sequences, and that in general speeds up the learning.

Problem Statement:

For the project of machine learning sessional (CSE 472), our group has chosen to develop an encoder-decoder model with attention for sequence-to-sequence prediction. The model is to be developed using Python with support from the neural network libraries: Keras, TensorFlow, nltk and gensim. The model will be a bilingual machine translation model. It will translate from Bangla sentence to English sentence. The attention-based sequence-to-sequence encoder-decoder model will take Bangla and English language sentences as input, train on Bangla and English vocabulary to predict translations from Bangla sentence to English sentence.

Performance Metrics:

The performance metric used for our project is BLEU. BLEU (bilingual evaluation understudy) is an algorithm for evaluating the quality of text which has been machine-translated from one natural language to another. Quality is considered to be the correspondence between a machine's output and that of a human: "the closer a machine translation is to a professional human translation, the better it is" – this is the central idea behind BLEU.

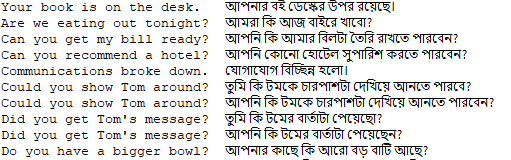
Scores are calculated for individual translated segments—generally sentences—by comparing them with reference translations. Intelligibility (a measure of comprehension ) or grammatical correctness are not taken into account.

BLEU’s output is always a number between 0 and 1. This value indicates how similar the candidate text is to the reference texts, with values closer to 1 representing more similar texts. Few human translations will attain a score of 1, since this would indicate that the candidate is identical to one of the reference translations. For this reason, it is not necessary to attain a score of 1. Because there are more opportunities to match, adding additional reference translations will increase the BLEU score.

Google Translate’s progress after going neural, the translation accuracy improved from .3694 to .4263. human quality is about .4636.

Data Overview**:**

We used dataset from <http://www.manythings.org/anki/> for our project. We used <http://www.manythings.org/anki/ben-eng.zip> dataset. There are 4379 lines of data in this file. Each line contains a pair of tab-delimited line. The first part is in English followed by its corresponding line in Bangla. There are small sentences in the beginning followed by bigger sentences afterwards.



Data Preprocessing**:**

We had to preprocess the Bangla and English part of the data separately. We had to use ‘utf-8’ encoding while reading the file. After that we preprocess the Bangla and English part of the data separately. For English part, we striped the sentence of whitespace, converted the words to lower case, removed all the punctuation from the words, removed non-printable characters and removed any alpha-numeric words or words that has number in them.

For the Bangla part we the task was quite similar. We striped the sentence of whitespace, removed all punctuation including "।" and removed any word that had Bangla digits in them. Then we saved the lines.

We used pickle file to save the data. We split the dataset into train and test part. Then we saved the total, train and test data in corresponding pickle file.

Model Description:

Our model is an encoder-decoder model with attention for sequence-to-sequence prediction for Bangla to English translation. Long short-term memory, or LSTM, is a network that operates on a sequence and uses its own output as input for subsequent steps.

A Sequence to Sequence network, or seq2seq network, or Encoder Decoder network, is a model consisting of two LSTMs called the encoder and decoder. The encoder reads an input sequence and outputs a single vector, and the decoder reads that vector to produce an output sequence.

Unlike sequence prediction with a single RNN, where every input corresponds to an output, the seq2seq model frees us from sequence length and order, which makes it ideal for translation between two languages.

Consider the sentence “Je ne suis pas le chat noir” → “I am not the black cat”. Most of the words in the input sentence have a direct translation in the output sentence, but are in slightly different orders, e.g. “chat noir” and “black cat”. Because of the “ne/pas” construction there is also one more word in the input sentence. It would be difficult to produce a correct translation directly from the sequence of input words.

The encoder of a seq2seq network is an LSTM that outputs some value for every word from the input sentence. For every input word the encoder outputs a vector and a hidden state and uses the hidden state for the next input word. Finally with a seq2seq model the encoder creates a single vector which, in the ideal case, encodes the “meaning” of the input sequence into a single vector — a single point in some N dimensional space of sentences.

The decoder is another LSTM that takes the encoder output vector and outputs a sequence of words to create the translation.

In the simplest seq2seq decoder we use only last output of the encoder. This last output is sometimes called the context vector as it encodes context from the entire sequence. This context vector is used as the initial hidden state of the decoder.

If only the context vector is passed between the encoder and decoder, that single vector carries the burden of encoding the entire sentence.

Attention allows the decoder network to “focus” on a different part of the encoder’s outputs for every step of the decoder’s own outputs. First, we calculate a set of attention weights. These will be multiplied by the encoder output vectors to create a weighted combination. The result should contain information about that specific part of the input sequence, and thus help the decoder choose the right output words.

Calculating the attention weights is done using the decoder’s input and hidden state as inputs. Because there are sentences of all sizes in the training data, to actually create and train this layer we have to choose a maximum sentence length (input length, for encoder outputs) that it can apply to. Sentences of the maximum length will use all the attention weights, while shorter sentences will only use the first few.

Architecture Description:

An overview of the architecture:

1. Word2Vec Layer
2. Encoder Layer
3. Attention Decoder Layer

**Word2Vec Layer:**

Word embeddings are a modern approach for representing text in natural language processing. Word2Vec is a key to the state-of-the-art results achieved by neural network models on natural language processing problems like machine translation.

A word embedding is an approach to provide a dense vector representation of words that capture something about their meaning.

Word embeddings are an improvement over simpler bag-of-word model word encoding schemes like word counts and frequencies that result in large and sparse vectors (mostly 0 values) that describe documents but not the meaning of the words.

Word embeddings work by using an algorithm to train a set of fixed-length dense and continuous-valued vectors based on a large corpus of text. Each word is represented by a point in the embedding space and these points are learned and moved around based on the words that surround the target word.

It is defining a word by the company that it keeps that allows the word embedding to learn something about the meaning of words. The vector space representation of the words provides a projection where words with similar meanings are locally clustered within the space.

The use of word embeddings over other text representations is one of the key methods that has led to breakthrough performance with deep neural networks on problems like machine translation.

Word2vec is one algorithm for learning a word embedding from a text corpus. These models are shallow, two-layer neural networks that are trained to reconstruct linguistic contexts of words. Word2vec takes as its input a large corpus of text and produces a vector space, typically of several hundred dimensions, with each unique word in the corpus being assigned a corresponding vector in the space. Word vectors are positioned in the vector space such that words that share common contexts in the corpus are located in close proximity to one another in the space.

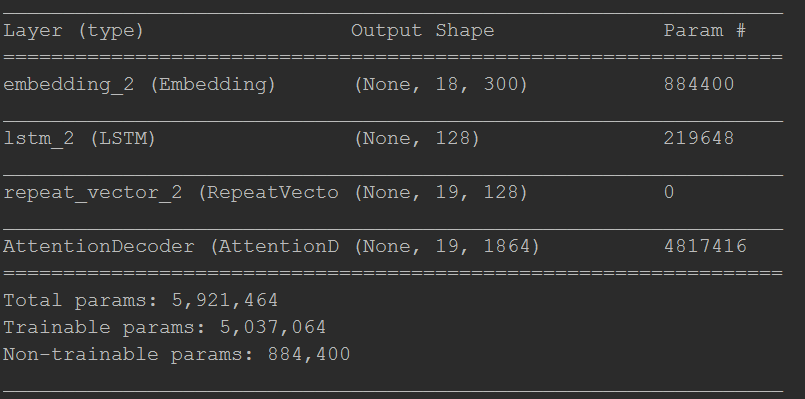


Fig: Layers with number of parameters

**Encoding Layer:**

The encoder is responsible for stepping through the input time steps and encoding the entire sequence into a fixed length vector called a context vector. An LSTM is used with pre-defined number of units to act as an encoder for our machine translation model. Each word from the input sequence is associated to a vector(via a lookup table). Then, we simply run an LSTM over this sequence of vectors and store the last hidden state outputted by the LSTM: this will be our encoder representation.

An intermediary step from encoder to decoder is the use of **RepeatVector**. The reason to use RepeatVector is that the output from the encoder is a single vector, while the decoder expects sequential input, i.e. output\_length timesteps of vectors. By using RepeatVector, we achieve that. The same vector (coming from the encoder) is used as the input for each timestep of the decoder.

**Attention Decoder Layer:**

we have a vector from the encoder that captures the meaning of the input sequence, we’ll use it to generate the target sequence word by word. Feed to another LSTM cell considered as a decoder cell. The LSTM computes the next hidden state. A function with softmax activation is applied to the hidden state and the result of the said function has the size of target vocabulary. The entries in the result point to probabilities of the target words. The index of the entry for the maximum probability is backtracked to predict the word most likely to be the translation. This word and the computed hidden state is fed to the LSTM decoder again to predict the next word.

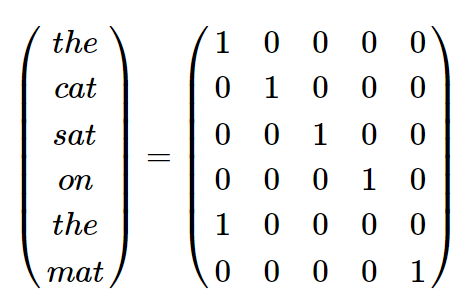
Attention is a mechanism that forces the model to learn to focus (=to attend) on specific parts of the input sequence when decoding, instead of relying only on the hidden vector of the decoder’s LSTM. This is done through determining a context vector to be used as an input to the LSTM cell along with the previous hidden state and previous predicted word. A new context vector is determined in each decoding step.

Algorithm:

**Basic Word2Vec:**

If we want to feed words into machine learning models, unless we are using tree based methods, we need to convert the words into some set of numeric vectors. A straight-forward way of doing this would be to use a “one-hot” method of converting the word into a sparse representation with only one element of the vector set to 1, the rest being zero. This is the same method we use for classification tasks – see this tutorial.

So, for the sentence “the cat sat on the mat” we would have the following vector representation:



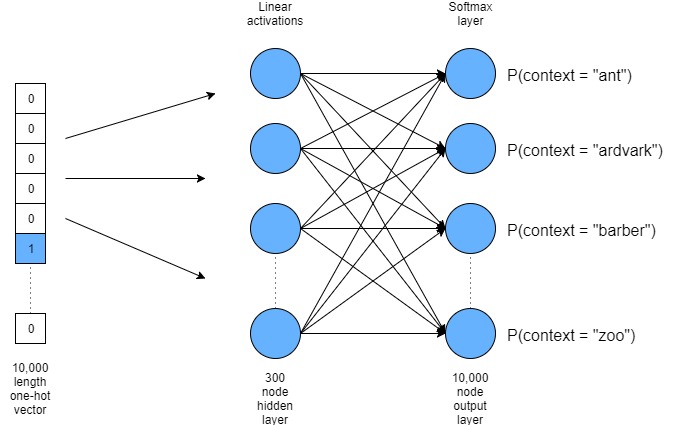
Here we have transformed a six word sentence into a 6×5 matrix, with the 5 being the size of the vocabulary (“the” is repeated). In practical applications, however, we will want machine and deep learning models to learn from gigantic vocabularies i.e. 10,000 words plus. You can begin to see the efficiency issue of using “one hot” representations of the words – the input layer into any neural network attempting to model such a vocabulary would have to be at least 10,000 nodes. Not only that, this method strips away any local context of the words – in other words, it strips away information about words which commonly appear close together in sentences (or between sentences).

For instance, we might expect to see “United” and “States” to appear close together, or “Soviet” and “Union”. Or “food” and “eat”, and so on. This method loses all such information, which, if we are trying to model natural language, is a large omission. Therefore, we need an efficient representation of the text data which also conserves information about local word context. This is where the Word2Vec methodology comes in.

there is two components to the Word2Vec methodology. The first is the mapping of a high dimensional one-hot style representation of words to a lower dimensional vector. This might involve transforming a 10,000 columned matrix into a 300 columned matrix, for instance. This process is called word embedding. The second goal is to do this while still maintaining word context and therefore, to some extent, meaning. The method called Continuous Bag Of Words (CBOW) takes some context words as input and tries to find the single word that has the highest probability of fitting that context.

Consider the diagram below – in this case we’ll assume the sentence “The cat sat on the mat” is part of a much larger text database, with a very large vocabulary – say 10,000 words in length. We want to reduce this to a 300 length embedding.

Consider the diagram below – in this case we’ll assume the sentence “The cat sat on the mat” is part of a much larger text database, with a very large vocabulary – say 10,000 words in length. We want to reduce this to a 300 length embedding.

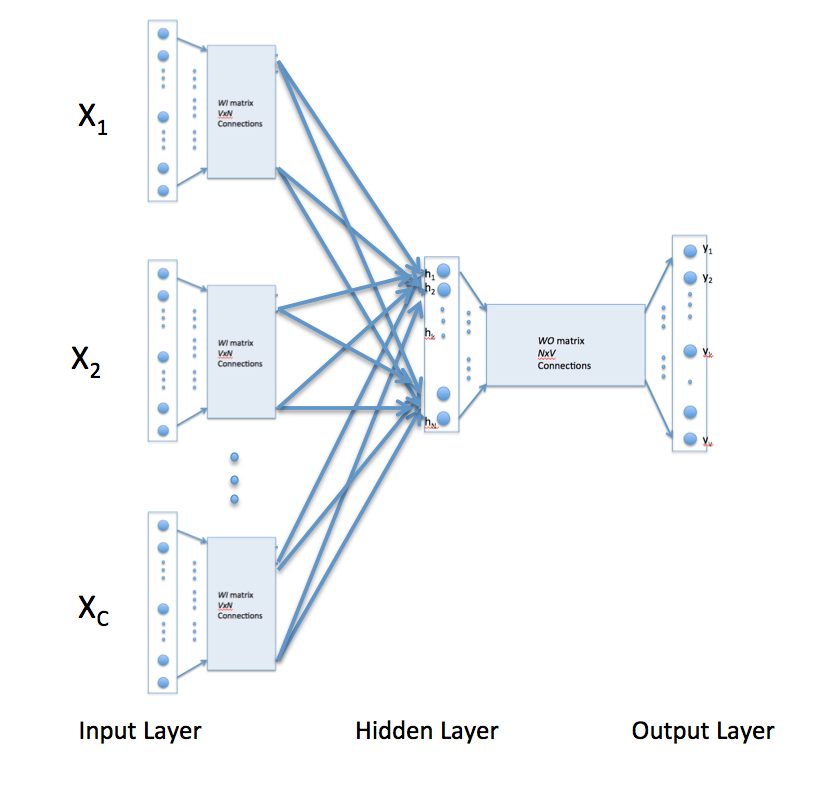


With respect to the diagram above, if we take the word “cat” it will be one of the words in the 10,000 word vocabulary. Therefore we can represent it as a 10,000 length one-hot vector. We then interface this input vector to a 300 node hidden layer . The weights connecting this layer will be our new word vectors – more on this soon. The activations of the nodes in this hidden layer are simply linear summations of the weighted inputs (i.e. no non-linear activation, like a sigmoid or tanh, is applied). These nodes are then fed into a softmax output layer. During training, we want to change the weights of this neural network so that words surrounding “cat” have a higher probability in the softmax output layer. So, for instance, if our text data set has a lot of Dr Seuss books, we would want our network to assign large probabilities to words like “the”, “sat” and “on” (given lots of sentences like “the cat sat on the mat”).

By training this network, we would be creating a 10,000 x 300 weight matrix connecting the 10,000 length input with the 300 node hidden layer. Each row in this matrix corresponds to a word in our 10,000 word vocabulary – so we have effectively reduced 10,000 length one-hot vector representations of our words to 300 length vectors. The weight matrix essentially becomes a look-up or encoding table of our words. Not only that, but these weight values contain context information due to the way we’ve trained our network. Once we’ve trained the network, we abandon the softmax layer and just use the 10,000 x 300 weight matrix as our word embedding lookup table.

**Continuous Bag of Words:**

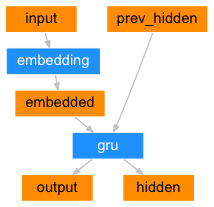
The above description and architecture is meant for learning relationships between pair of words. Within Word2Vec we have used continuous bag of words algorithm. In the continuous bag of words model, context is represented by multiple words for a given target words. The input to the hidden layer connections is replicated C times, the number of context words, and adding a divide by C operation in the hidden layer neurons.



With the above configuration to specify C context words, each word being coded using 1-out-of-V representation means that the hidden layer output is the average of word vectors corresponding to context words at input. The output layer remains the same and the training is done in the manner discussed above.

**The Encoder Algorithm:**

The encoder of a seq2seq network is an LSTM that outputs some value for every word from the input sentence. For every input word the encoder outputs a vector and a hidden state, and uses the hidden state for the next input word.



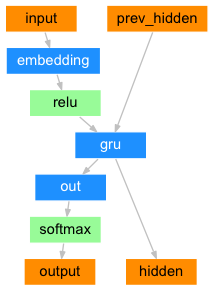
The Decoder Algorithm:

The decoder is another LSTM that takes the encoder output vector(s) and outputs a sequence of words to create the translation.

Simple Decoder Algorithm:

In the simplest seq2seq decoder we use only last output of the encoder. This last output is sometimes called the context vector as it encodes context from the entire sequence. This context vector is used as the initial hidden state of the decoder.

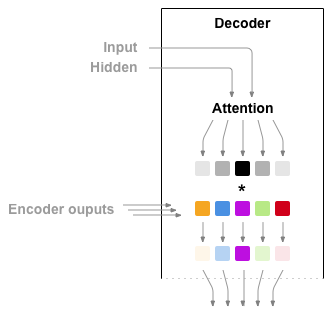
At every step of decoding, the decoder is given an input token and hidden state. The initial input token is the start-of-string <SOS> token, and the first hidden state is the context vector (the encoder’s last hidden state).



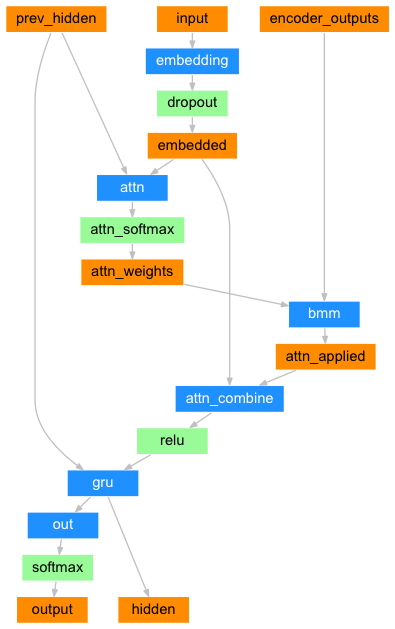
**Attention Decoder Algorithm:**

If only the context vector is passed between the encoder and decoder, that single vector carries the burden of encoding the entire sentence.

Attention allows the decoder network to “focus” on a different part of the encoder’s outputs for every step of the decoder’s own outputs. First we calculate a set of attention weights. These will be multiplied by the encoder output vectors to create a weighted combination. The result should contain information about that specific part of the input sequence, and thus help the decoder choose the right output words.



Calculating the attention weights is done with another feed-forward layer, using the decoder’s input and hidden state as inputs. Because there are sentences of all sizes in the training data, to actually create and train this layer we have to choose a maximum sentence length (input length, for encoder outputs) that it can apply to. Sentences of the maximum length will use all the attention weights, while shorter sentences will only use the first few.



Hyper-parameter Tuning:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Hyperparameter Tuning results** | | | | | | |
| Number of units in LSTM layer | Embedding Dimension | Learning rate | Train  Loss | Validation Loss | Train  Accuracy | Test  Accuracy |
| 64 | 20 | 0.01 | 0.5008222389221192 | 3.7258299827575683 | 0.8981578946113586 | 0.669497613473372 |
| 64 | 20 | 0.001 | 0.5065907406806945 | 3.6674797664989125 | 0.898421049118042 | 0.669497613473372 |
| 64 | 20 | 0.0001 | 0.5003519201278687 | 3.699722888252952 | 0.8988157892227173 | 0.669497613473372 |
| 64 | 30 | 0.01 | 0.5134423875808716 | 3.6310723651539196 | 0.8988157892227173 | 0.669497613473372 |
| 64 | 30 | 0.001 | 0.5087630701065063 | 3.6819287386807527 | 0.8988157868385315 | 0.669497613473372 |
| 64 | 30 | 0.0001 | 0.511182005405426 | 3.665239429473877 | 0.8988157796859741 | 0.669497613473372 |
| 64 | 50 | 0.01 | 0.5058231687545777 | 3.683366385373202 | 0.8978947377204896 | 0.669497613473372 |
| 64 | 50 | 0.001 | 0.5102478408813477 | 3.6966113263910465 | 0.8988157820701599 | 0.669497613473372 |
| 64 | 50 | 0.0001 | 0.4983781397342682 | 3.7176015333695847 | 0.8988157844543457 | 0.669497613473372 |
| 128 | 20 | 0.01 | 0.47396977186203004 | 4.1430848121643065 | 0.899210524559021 | 0.669497613473372 |
| 128 | 20 | 0.001 | 0.4741345202922821 | 4.117505298961293 | 0.9001315879821777 | 0.6696172302419489 |
| 128 | 20 | 0.0001 | 0.4764213538169861 | 4.1218470486727625 | 0.8977631545066833 | 0.6622009526599537 |
| 128 | 30 | 0.01 | 0.4734188580513001 | 4.111656353690408 | 0.8989473748207092 | 0.6696172302419489 |
| 128 | 30 | 0.001 | 0.4729907488822937 | 4.111656353690408 | 0.8989473748207092 | 0.6696172302419489 |
| 128 | 30 | 0.0001 | 0.47355868101119997 | 4.136147325689143 | 0.9007894825935364 | 0.6696172302419489 |
| 128 | 50 | 0.01 | 0.4731629037857056 | 4.134229425950484 | 0.9010526347160339 | 0.6696172302419489 |
| 128 | 50 | 0.001 | 0.4746765422821045 | 4.127498921481046 | 0.8976315712928772 | 0.669497613473372 |
| 128 | 50 | 0.0001 | 0.4729781603813171 | 4.129414280978116 | 0.8985526299476624 | 0.6696172302419489 |
| 256 | 20 | 0.01 | 0.46796571850776675 | 4.321620802445845 | 0.9001315879821777 | 0.6696172302419489 |
| 256 | 20 | 0.001 | 0.4654586148262024 | 4.32614296132868 | 0.8998684215545655 | 0.6696172302419489 |
| 256 | 20 | 0.0001 | 0.46484880685806274 | 4.296801887858997 | 0.8986841988563538 | 0.6696172302419489 |
| 256 | 30 | 0.01 | 0.4666183662414551 | 4.308587273684415 | 0.8997368502616883 | 0.6696172302419489 |
| 256 | 30 | 0.001 | 0.4661624002456665 | 4.308587273684415 | 0.8997368502616883 | 0.6696172302419489 |
| 256 | 30 | 0.0001 | 0.4645297741889954 | 4.299180802431974 | 0.8993421006202698 | 0.6696172302419489 |
| 256 | 50 | 0.01 | 0.4658980190753937 | 4.330687271464955 | 0.899342098236084 | 0.6696172302419489 |
| 256 | 50 | 0.001 | 0.4660956299304962 | 4.3277471889149055 | 0.9007894706726074 | 0.6622009526599537 |
| 256 | 50 | 0.0001 | 0.467168025970459 | 4.330027701637961 | 0.9001315808296204 | 0.669497613473372 |

Results Description:

|  |  |  |
| --- | --- | --- |
| Epoch | Validation Set  BLEU Score | Test Set  BLEU Score |
| 15 | 0.114940 | 0. 126169 |
| 10 | 0.108369 | 0. 117235 |
| 5 | 0.069835 | 0.067741 |

Table 1: BLEU Score for different Epoch size

|  |  |  |
| --- | --- | --- |
| Data Set | Validation Set  BLEU Score | Test Set  BLEU Score |
| Large | 0.273947 | 0.354768 |
| Medium | 0.145291 | 0.121834 |
| Small | 0.127609 | 0.129932 |

Table 2: BLEU Score for different Data set size

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | Loss | Accuracy | BLEU Score |
| With Attention | 1.1611 | 0.8119 | 0.449860 |
| Without Attention | 1.2145 | 0.8110 | 0.423731 |
|  |  |  |  |

Table 2: Comparison between seq2seq encoder-decoder with attention and without attention

Conclusion:

1. Lack of word2vec readymade models in Keras to implement our embedding layer
2. Lack of large Bangla to English dataset
3. Lack of attention decoder function/API in Keras
4. Lack of good metrics for measuring the accuracy of a neural machine translation, though BLEU score gives clearer results than conventional test accuracy
5. Due to lack of good GPU, tuning took a lot of time
6. Very hard to get accuracy on small dataset
7. Inconsistency is seen sometimes in the hyperparameter tuning.
8. As per results, as epoch increases, the accuracy of the model increases.
9. As per results, as dataset size increases, accuracy increases.