Final Project Report

Sanyog Sharma (AI21MTECH12003), Arvind Roshaan (AI21MTECH12004),
Billakurthi Shai Sasi Deep (SM21MTECH12006), HN Srikanth (SM21MTECH12012),
Deevanshu Gupta (SM21MTECH12014)
Group ID 25
Presentation Group ID 18

Abstract

Translation traditionally is a process utilizing human intelligence to capture and instill the essence of the source language sentence into the target language sentence. The map between sentences in source and target language is many to one, with no pre-defined, non-conflicting set of rules. Hence this task is challenging for an average human and usually requires professional expertise. In this context, we aim to learn and implement a deep learning model to perform translation (a.k.a machine translation). The model is called the Transformer introduced in [13]. Earlier to this model there are many network architectures evolved based on complex recurrent or convolutional neural networks that include an encoder and decoder which are connected through an attention mechanism. The transformer is based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. This model achieved 28.4 BLEU on the WMT 2014 English-to-German translation task and 41.8 BLEU on WMT 2014 English-to-French translation task.

1. Introduction [10]

The translation is the process of translating a sentence from one language (source language) to a sentence in another language (target language). The sentences can either be as text or speech. This task is traditionally performed by a professional human translator, who is well versed in the nuances of both the source and the target language.

The objective of translation is to interpret various properties (like content, tone, style, culture, etc.,) of the source sentence and try to fit/ preserve as many of these properties in the target language. Since interpretations are subjective, we have different professional human translators produce different valid translations in the target language for the same sentence from the source language.

Also, professional human translation services typically involve multiple human translators to quality control the translation. The following are a few properties considered while assessing the quality of a translation - Adequacy, Fidelity, and Fluency. Adequacy addresses, Does the translation capture enough content from the source sentence? Fidelity is defined as "the degree of exactness with which something is copied or reproduced". Fluency addresses, How fluent is translation in the target language?

1.1. Machine translation

Machine translation is an active area of research. There have been 3 major approaches, Rule-based Machine Translation (RBMT); Statistical Machine Translation (SMT); Neural Machine Translation (NMT). The current state of the art approach is NMT. NMT models use Deep learning and Representation learning. The structure of models in NMT is a sequence based model that predicts one word at a time. However this sequence prediction is conditioned on the entire source sentence and the previously produced target sequence.

The word sequence modeling was first coined by Recurrent neural networks. Apart from Recurrent neural networks, long short-term memory and gated recurrent neural networks, have well established themselves as state-of-the-art techniques in language modeling and machine translation. Recurrent models typically factor computation along the symbol positions sequentially. They generate a sequence of hidden states h_t , as a function of the previously hidden state h_{t-1} and the input for position t. In different tasks, attention mechanisms have become a necessary tool to achieve long term dependencies regardless of their distance in the input or output sequences.

In this paper, we present the Transformer, a model architecture that removes recurrence. In favor of drawing global dependencies between input and output, Transformers rely completely on attention mechanism.

2. Motivation

Translation can be hypothetically framed in the imitation setup. That is, given the translation of a source sentence from 2 entities (one human and one machine), can a human correctly judge and identify the translation produced by a fellow human? This hypothetical setup motivates us to look for methods to perform machine translation, with the aim to match/ surpass human translation quality.

The translation is a qualitative task that utilizes human intelligence and knowledge. This is unlike quantitative tasks that require human intelligence (like arithmetic calculations), having a pre-defined, non-conflicting set of rules. Hence making the task of machine translation even more challenging, in comparison. Other challenges involve naturally conveying emotions and humor.

Even though the problem of machine translation has been around for decades, we still do not have a model that produces business-ready translations.

2.1. Why Transformers?

The work in [9] involves both LSTM and attention to get dependencies between the words in the given text. The model becomes heavy as it involves both LSTM and attention mechanism. But in Transformer model dependencies are achieved only through attention which makes model light and less complex.

The downside of model presented in [4] is that of handling unknown or rare words in large sentences, due to which performances reduces for the translation.

To our knowledge, the Transformer is the first transduction model to calculate representations of its input and output using only attention rather than sequence-aligned RNNs or convolution.

3. Literature Survey

We survey our main paper [13] along with [4], [7], [9], [14], [5] and [12].

3.1. Attention is all you need

To our knowledge, the Transformer is the first transduction model to calculate representations of its input and output using only attention rather than sequence-aligned RNNs or convolution.

$$\textbf{Attention}(Q, K, V) = \left(\textbf{softmax}(\frac{Q.K^T}{\sqrt{\mathbf{d_k}}})V\right)$$

Self-attention allows the model to look at other positions in the input sequence for clues that can lead to a better encoding for this word, as it processes each word (each position in the input sequence). So we produce a Query vector, a Key vector, and a Value vector for each word. The embedding

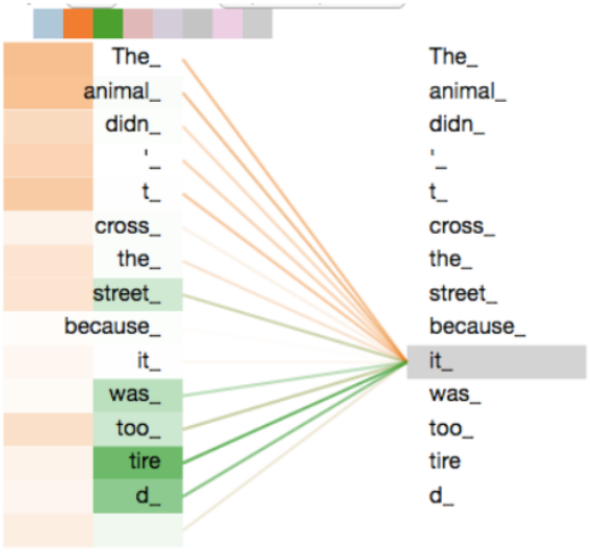

Figure 1. The picture depicts the model's representation of the word "it" bakes in some of the representation of both "animal" and "tired". The image is referred from [3]

is multiplied by three matrices that we trained during the training process to get these vectors. Each word of the input sequence is scored against another word. As we encode a word at a specific position, the score determines how much focus to place on other parts of the input sentence. The score is calculated by taking the dot product of the query vector with the key vector. The next step is to divide the score with the square root of the dimension of the key vectors i.e 8 in this paper (dimension is 64). This gives more stable gradients. Pass the results to the soft max function. This gives all normalized values which are positive and adds up to 1. The next step is to multiply each value vector by the soft max score (in preparation to sum them up). Sum these weighted value vectors which will end up self attention calculation.

Self-attention is used to overcome three major hurdles faced by earlier models. They are Computational complexity, the amount of computation that can be parallelized (measured by minimum no. of sequential operations required), and the path length between long-range dependencies in the network. The shorter this path length between any combination of words the easier it is to learn long-term dependencies.

Self attention works with limited sequential operations i.e O(1), whereas recurrent layers need operations which are in O(n). Apart from time complexity self attention plays well in computational complexity too when compared with recurrent networks. Moreover Self-attention gives meaningful models where recurrent models lack in.

3.2. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation

The neural network model in this paper is called RNN encoder-decoder that consists of two recurrent neural networks. One RNN encodes a sequence of symbols into a fixed length vector representation, and the other decodes the representation into another sequence of symbols.

The two networks are trained jointly to maximise the conditional probability of the target sequence given a source sequence. The encoder reads each symbol of an input sequence 'x' sequentially. As it reads each symbol, the hidden state of the RNN changes according to Eq. (1). After reading the end of the sequence the hidden state of the RNN is a summary $\mathbf{c}$ of the whole input sequence

$$\mathbf{h}_{\langle t \rangle} = f\left(\mathbf{h}_{\langle t-1 \rangle}, x_t\right)$$

where f is a non-linear activation function. f may be as simple as an elementwise logistic sigmoid function and as complex as a long short-term memory (LSTM) unit.

The decoder is trained to generate the output sequence by predicting the next symbol y_t given the hidden state $\mathbf{h}_{\langle t \rangle}$.In decoder the hidden state is conditioned both on y_{t-1} and summary $\mathbf{c}$ of the input sequence.

$$\mathbf{h}_{\langle t \rangle} = f\left(\mathbf{h}_{\langle t-1 \rangle}, y_{t-1}, \mathbf{c}\right),$$

The two components of the proposed RNN Encoder-Decoder are jointly trained to maximize the conditional log-likelihood

$$\max_{\boldsymbol{\theta}} \frac{1}{N} \sum_{n=1}^{N} \log p_{\boldsymbol{\theta}}\left(\mathbf{y}_n \mid \mathbf{x}_n\right)$$

max). where $\boldsymbol{\theta}$ is the set of the model parameters and each $(\mathbf{x}_n, \mathbf{y}_n)$ is an (input sequence, output sequence) pair from the training set.

This model is used in two ways, to generate a target sequence given an input sequence and to score a given pair of input and output sequences where the score is simply a probability $p_{\boldsymbol{\theta}}(\mathbf{y} \mid \mathbf{x})$

A novel hidden unit is proposed that includes a reset gate and an update gate that adaptively control how much each hidden unit remembers or forgets while reading or generating a sequence.

3.3. Neural Machine Translation by Jointly Learning to Align and Translate

The background comparable model RNN Encoder–Decoder (*RNNencdec*), proposed by [7] have done pretty well, but are limited with respect to tracking long-term dependencies and sometimes even lose their ability to translate the end of long sentences correctly. The cause of this limitation is that this architecture encodes

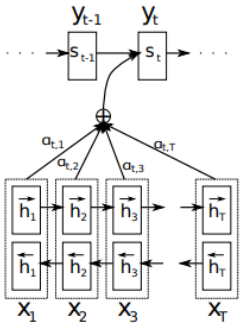

Figure 2. The graphical illustration of the proposed model trying to generate the t-th target word y_t given a source sentence (x_1, x_2,..., x_T).

everything about the variable length input sentence in a single fixed length vector (usually encoder RNN final hidden state).

This paper addresses this problem in two ways: first by using a Bidirectional RNN for input and second by introducing an alignment model on the decoder part.

BiRNN outputs two hidden states for each input location j - forward hidden state $\overrightarrow{h}_j$ contains information from earlier in the sequence and backward hidden state $\overleftarrow{h}_j$ from later in the sequence - these states are concatenated forming an annotation h_j.

Now on decoder, we obtain conditional probability as

$p\left(y_i | y_1, ..., y_{i-1}, x\right) = g\left(y_{i-1}, s_i, c_i\right)$

where s_i is an RNN hidden state for time i, computed as

$s_i = f\left(s_{i-1}, y_{i-1}, c_i\right)$

The context vector c_i computed as a weighted sum of the annotations (h_1,...,h_{T_x})

$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$ where weight α_{ij} of each annotation h_j is computed by $\alpha_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{T_x} exp(e_{ik})}$

where $e_{ij} = a\left(s_{i-1}, h_j\right)$ which is an alignment model which scores how well input around position j and output at position i match.

See Fig. 2 for the graphical illustration of the proposed model.

The training was performed with a couple of models on different length sentences (30 vs 50 words) and with (*RNNsearch*) and without attention (*RNNencdec*). The encoder (BIRNN) and decoder of the *RNNsearch* have 1000 hidden units each same as *RNNencdec*. The word embedding dimension is 620. Once the model is trained, they use a beam search to find a translation that approximately maximizes the conditional probability.

The Corpus used for training was with 348M words. In Fig 3, we can see that the quality of the long sentences (*RNNsearch-50*) was improved and doesnot deteriorate as the sentence length increases. The performance of *RNNencdec* decreases as sentence length increases. The baseline model *RNNencdec-50* produced poor quality long sentences, compared even with the *RNNsearch-30*.

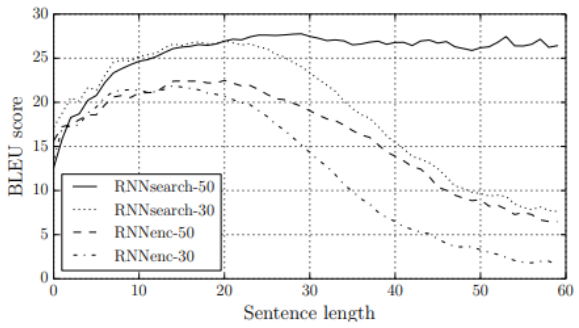

Figure 3. The BLEU scores of the generated translations on the test set with respect to the lengths of the sentence.

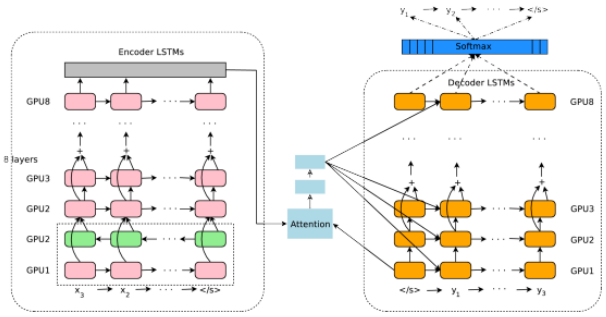

Figure 4. The model architecture of Google's Neural Machine Translation system

3.4. A Structured Self-attentive Sentence Embedding

This paper proposed a new model for meaningful sentence embedding by using self-attention. Instead of using a 1-D vector, it proposed a 2D matrix where each row is a sentence word. They proposed a self-attention mechanism and a particular regularization term for the model. The embedding that is generated is interpretable and can be easily visualized to understand what parts of the sentence are encoded into the embedding. The author evaluates the model on three tasks: author profiling, sentiment classification, and textual entailment.

The paper proposed that carrying the semantics along all the time steps of the recurrent model is complex and unnecessary. The self-attention mechanism allows us to look at different parts of a sentence and provide it into a vector representation. In the proposed sentence embedding model, attention is done on top of an LSTM layer. When there are no extra inputs, attention can be used in this way. Furthermore, it eliminates part of the LSTMs' long-term memorization burden by providing immediate access to hidden representations from past time steps.

The paper approaches sentence embedding in two ways, bidirectional LSTM, and self-attention mechanism. The attention mechanism provides a set of weight vectors for the LSTM hidden states. These weights are multiplied with the LSTM hidden states, and the resulting weighted LSTM hidden states are summed to generate an embedding for the sentence.

One of the most critical parts of this paper is the Penalizing term; the attention mechanism could face redundancy problems. In order to avoid this problem, we would need a penalization term that forces diversity in the attention vectors. We can introduce KL divergence here, but it will result in problems such as getting a lot of zero or very small close to zero values. Also, it does not encourage focusing on different aspects of the same sentence.

$$P = \left\|(A.A^T - I)\right\|_{F^2}$$

where $\|.\|_F$ stands for the Frobenius norm of a matrix The similarity between a_i and a_j is reduced and thus the difference between the attention weights is increased.

3.5. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation

In this paper a model is proposed which consists of a deep LSTM network with 8 encoder and 8 decoder layers using residual connections as well as attention connections from the decoder network to the encoder. The bottom layer of the decoder is connected to the top layer of the encoder to improve parallelism and decrease training time Low precision arithmetic is employed during inference computations to accelerate the translation. Rare words are handled by dividing them into limited set of common sub word units for input and output. See Fig. 4 for the graphical illustration of the proposed model.

3.6. Massive Exploration of Neural Machine Translation Architectures

In this paper analysis of neural machine translation architecture hyperparameters is done and came up with following conclusions. Large embeddings with 2048 dimensions achieve best results compared to small embeddings with 128 dimensions by small margin. LSTM cells outperforms GRU cells Bidirectional encoders with 2 to 4 layers performed best. Deeper encoders were significantly more unstable to train Deep 4 layer decoders slightly outperformed shallower decoders. Residual connections were necessary to train decoders with 8 layers and dense residual connections offer additional robustness Parameterized additive attention yielded the best results A well-tuned beam search with length penalty is crucial.

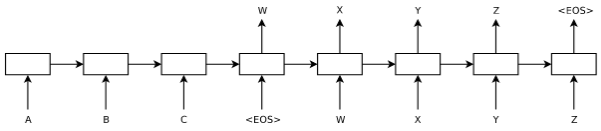

Figure 5. The picture depicts the model's representation of the word "ABC" to word "XYZ". The image is referred from [12]

3.7. Sequence to Sequence Learning with Neural Networks

In their 2014 publication Sequence to Sequence Learning using Neural Networks, Sutskever et al. introduced the seq2seq model to address the Machine Translation problem without requiring any understanding of grammar or syntax and no pre-processing or hand-engineering. The goal is to build a two-part network that includes an encoder and a decoder. At a high level, the encoder is responsible for accepting an input sequence (say, an English sentence) and processing it to generate some compression/representation of that sentence's semantic information. As we'll see, this compression usually entails using a fixed-size vector to represent the entire sequence (which can be any length). The decoder's job is to take this vector representation of the sentence's "meaning" as input and produce an output sequence in the other language (let's assume Hindi). The output sequence should be a genuine Hindi translation of the English sentence if the encoder and decoder complete their duties correctly. For this to be the case, the encoder's compressed output will have to capture the semantic meaning of the English sentence in some weird way. Following model used two RNN for encoder and decoder. Naturally, the weights of these RNNs are different.

This seq2seq concept was incredibly effective, and it's difficult to overstate the significance of this development in NMT. However, it was soon discovered that this approach has limits when dealing with exceedingly long sequences.

The challenge with seq2seq stems from an inherent constraint of LSTMs: compressing/embedding larger and longer sequences becomes increasingly difficult. Seq2seq performed admirably for short sentences. However, when you try to translate sentences that are tens of words long, the quality of the translation begins to deteriorate. As the length of an input sentence grows longer, this becomes a more difficult task to do successfully.

3.8. Neural Machine Translation with Byte Level Subwords [6]

To build vocabulary, we see most of Neural Machine Translations prefer using Byte Pair Encoding (BPE). But as we know BPE works under level of characters which causes rare characters from character-rich languages take up vocabulary slots. Proposed paper uses byte level subwords that can be used to tokenize text into variable-length byte n grams. We can also use additional Bidirectional recurrent layer with GRU to contextualize the embedding before feeding to model. During encoding, each character can be converted to 1 to 4 bytes long sequences under UTF-8 encoding. During decoding, the uniqueness of recovery for characters encoded with multiple bytes is maintained as its trailing bytes will not make a valid UTF-8 encoded character. This method can very well be used with languages which has huge number of characters but only small portion of them are frequently used. Below mentioned is an example of Bytelevel tokenization:

Raw Text	Anarchism is a political philosophy that advocates self-governed societies based on voluntary institutions .
Raw Text (UTF-8, in Hexademical form)	416E617263686973 6D 6973 61 706F6C69746963616C 7068696C6F736F706879 74686174 6164766F6361746573 73656C66 2D 676F7665726E6564 736F63696574696573 6261736564 6F6E 766F6C756E74617279 696E737469747574696F6E73 2E
Tokenized Text (UTF-8, in Hexademical form)	416E61 ##72 ##63686973 ##6D 6973 61 706F6C69746963616C 7068696C6F736F706879 74686174 6164766F63 ##61746573 73656C66 2D 676F7665726E ##6564 736F63696574696573 6261736564 6F6E 766F6C756E74 ##617279 696E737469747574696F6E73 2E
Tokenized Text (Corresponding Text)	Ana ##r ##chis ##m is a political philosophy that advoc ##ates self - govern ##ed societies based on volunt ##ary institutions .

Figure 6. Example for tokenization with byte-elevel subwords

4. Explanation of annotated Transformer code [11]

In the following we have documented our understanding of Annotated Transformer code. In Encoder-Decoder based models, Encoder maps input sequence X to latent space Z from where the decoder generated output sequence Y. Residual connections are used to consider previously generated symbols also, as one of the inputs. All sub-layers in the model, as well as the embedding layers, produce outputs of dimension model = 512 to support these residual connections.

4.1. Encoder-Decoder Architecture

The first block of the code give an overall picture of the Encoder-Decoder stack. Encoder takes source embedding and develops attention among them. On the other side Decoder considers target embeddings and make them ready to be mapped from encoder learning. Finally generator generates predicted output sequence. This predicted sequence is compared with ground-truth sequence, further training is done. This flow can be observed in (Fig.6 and Fig.7)

4.2. Generator, clone and LayerNormalization

In generator block(Fig. 7) each output is mapped to vocabulary through a neural network layer and softmax function is applied to output of the layer.

The clones function (Fig. 8) is used to replicate identical layers in encoder and decoder.

In layernorm(Fig .9), the statistics are calculated across the feature dimension, for each element and instance inde-

```
class EncoderDecoder(nn.Module):
    """
    A standard Encoder-Decoder architecture. Base for this and many
    other models.
    """
    def __init__(self, encoder, decoder, src_embed, tgt_embed, generator):
        super(EncoderDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.generator = generator

    def forward(self, src, tgt, src_mask, tgt_mask):
        "Take in and process masked src and target sequences."
        return self.decode(self.encode(src, src_mask), src_mask,
                           tgt, tgt_mask)

    def encode(self, src, src_mask):
        return self.encoder(self.src_embed(src), src_mask)

    def decode(self, memory, src_mask, tgt, tgt_mask):
        return self.decoder(self.tgt_embed(tgt), memory, src_mask, tgt_mask)
```

Figure 7. Encoder-Decoder Architecture

```
class Generator(nn.Module):
    "Define standard linear + softmax generation step."
    def __init__(self, d_model, vocab):
        super(Generator, self).__init__()
        self.proj = nn.Linear(d_model, vocab)

    def forward(self, x):
        return F.log_softmax(self.proj(x), dim=-1)
```

Figure 8. Generator

```
def clones(module, N):
    "Produce N identical layers."
    return nn.ModuleList([copy.deepcopy(module) for _ in range(N)])
```

Figure 9. Clones

```
class LayerNorm(nn.Module):
    "Construct a layernorm module (See citation for details)."
    def __init__(self, features, eps=1e-6):
        super(LayerNorm, self).__init__()
        self.a_2 = nn.Parameter(torch.ones(features))
        self.b_2 = nn.Parameter(torch.zeros(features))
        self.eps = eps

    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True)
        return self.a_2 * (x - mean) / (std + self.eps) + self.b_2
```

Figure 10. LayerNorm

pendently. In transformers, it is calculated across all features and all elements, for each instance independently.

4.3. Encoder

The Encoder(Fig. 10) has a stack of N=6 identical layers(clone function is used to produce them). Layer normal-

```
class Encoder(nn.Module):
    "Core encoder is a stack of N layers"
    def __init__(self, layer, N):
        super(Encoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, mask):
        "Pass the input (and mask) through each layer in turn."
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)
```

Figure 11. Encoder Stack

```
class EncoderLayer(nn.Module):
    "Encoder is made up of self-attn and feed forward (defined below)"
    def __init__(self, size, self_attn, feed_forward, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 2)
        self.size = size

    def forward(self, x, mask):
        "Follow Figure 1 (left) for connections."
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, mask))
        return self.sublayer[1](x, self.feed_forward)
```

Figure 12. Encoder Layer

ization is done for the output of each layer. Each layer(Fig. 11) in the encoder consists of two sublayers. Multi headed attention is done in first sublayer followed by position wise feed forward network for each position in second sublayer. A residual connection is employed between two sublayers followed by layer Normalization and dropout.

4.4. Decoder

In addition to two sub layers in Encoder, Decoder has one more Multi-head Attention layer which performs multi-head attention on the output of the Encoder stack. Residual connections and layer normalization all are similar to encoders. The main modification done in self-attention of the decoder is masking is applied to each position to prevent it from attending subsequent positions which are clearly depicted in the (Fig .11). This is implemented using inbuilt uppertraingular operation as shown in (Fig .14). The masking combined with the fact that the output embeddings are offset by one place, the predictions for position i can only be based on the known outputs at positions less than i.

4.5. Attention

Attention is Dot-product of Query and transpose of Key. This dot product is divided by no.of dimensions to avoid blowing up the values. Further this is multiplied with value matrix which takes attention forward.This action is observed in attention block(Fig. 16). **Multi-head attention**

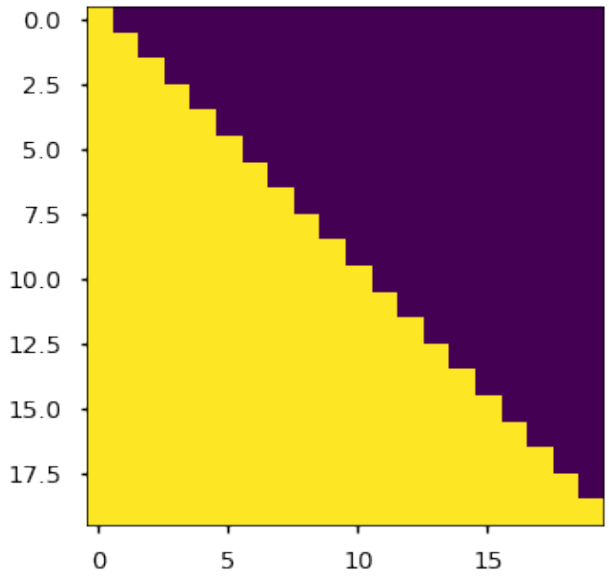

Figure 13. Attention mask

```
class Decoder(nn.Module):
    "Generic N layer decoder with masking."
    def __init__(self, layer, N):
        super(Decoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, memory, src_mask, tgt_mask):
        for layer in self.layers:
            x = layer(x, memory, src_mask, tgt_mask)
        return self.norm(x)
```

Figure 14. Decoder class

```
class DecoderLayer(nn.Module):
    "Decoder is made of self-attn, src-attn, and feed forward (defined below)"
    def __init__(self, size, self_attn, src_attn, feed_forward, dropout):
        super(DecoderLayer, self).__init__()
        self.size = size
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 3)

    def forward(self, x, memory, src_mask, tgt_mask):
        "Follow Figure 1 (right) for connections."
        m = memory
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, tgt_mask))
        x = self.sublayer[1](x, lambda x: self.src_attn(x, m, m, src_mask))
        return self.sublayer[2](x, self.feed_forward)
```

Figure 15. Decoder layer

```
def subsequent_mask(size):
    "Mask out subsequent positions."
    attn_shape = (1, size, size)
    subsequent_mask = np.triu(np.ones(attn_shape), k=1).astype('uint8')
    return torch.from_numpy(subsequent_mask) == 0
```

Figure 16. subsequent mask

```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) \
             / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    p_attn = F.softmax(scores, dim = -1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn
```

Figure 17. attention block

```
class MultiHeadedAttention(nn.Module):
    def __init__(self, h, d_model, dropout=0.1):
        "Take in model size and number of heads."
        super(MultiHeadedAttention, self).__init__()
        assert d_model % h == 0
        # We assume d_v always equals d_k
        self.d_k = d_model // h
        self.h = h
        self.linears = clones(nn.Linear(d_model, d_model), 4)
        self.attn = None
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, query, key, value, mask=None):
        "Implements Figure 2"
        if mask is not None:
            # Same mask applied to all h heads.
            mask = mask.unsqueeze(1)
        nbatches = query.size(0)

        # 1) Do all the linear projections in batch from d_model => h x d_k
        query, key, value = \
            [l(x).view(nbatches, -1, self.h, self.d_k).transpose(1, 2)
             for l, x in zip(self.linears, (query, key, value))]

        # 2) Apply attention on all the projected vectors in batch.
        x, self.attn = attention(query, key, value, mask=mask,
                                 dropout=self.dropout)

        # 3) "Concat" using a view and apply a final linear.
        x = x.transpose(1, 2).contiguous() \
             .view(nbatches, -1, self.h * self.d_k)
        return self.linears[-1](x)
```

Figure 18. Multi head Attention

is also implemented similarly but with respect to different heads in parallel. In this implementation 8 heads are used. Initially model head and number of heads are taken and split is done accordingly. And 4 similar linear flow functions are created out of which three were used to process query, key and value. The fourth one is updated with result in the later stage. Initially masking is done on all heads when this Multi-head attention is implemented on Decoder side. Then they are linearly projected where attention takes place. Finally concatenation is done where original size is regenerated. This flow of Multi-Head attention is shown in code block (Fig. 17)

4.6. Position Wise Feed forward Network

Each of the layers in our encoder and decoder contains a fully connected feed-forward network (Fig .17), which is

```python
class PositionwiseFeedForward(nn.Module):
    "Implements FFN equation."
    def __init__(self, d_model, d_ff, dropout=0.1):
        super(PositionwiseFeedForward, self).__init__()
        self.w_1 = nn.Linear(d_model, d_ff)
        self.w_2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        return self.w_2(self.dropout(F.relu(self.w_1(x))))
```

Figure 19. Position wise Feed forward Network

```python
class Embeddings(nn.Module):
    def __init__(self, d_model, vocab):
        super(Embeddings, self).__init__()
        self.lut = nn.Embedding(vocab, d_model)
        self.d_model = d_model

    def forward(self, x):
        return self.lut(x) * math.sqrt(self.d_model)
```

Figure 20. Embedding

applied to each position separately and identically. This consists of two linear transformations with a ReLU activation in between.The dimensionality of input and output is d_{model} = 512, while the hidden layer has dimensionality of d_{ff}=2048

4.7. Embedding and Positional Encoding

They use learned embeddings(Fig .19) to convert the input tokens and output tokens to vectors of dimension d_{model}. They also use the learned linear transformation and softmax function to convert the decoder output to predicted next-token probabilities.

In order for the model to make use of the order of the sequence, they add "positional encodings"(Fig .20) to the input embeddings at the bottoms of the encoder and decoder stacks. The positional encodings have the same dimension d_{model} as the embeddings, so that the two can be summed.

4.8. Training

In codeblock (Fig. 20)a Object is created for holding a batch of data with mask during training. Next in training loop (Fig. 21), loss function is created to keep an eye on loss continuously for every epoch. A generic loss compute function is created which takes care of parameter updates.

4.9. Optimizer

In Normopt() (Fig. 23) They use an Adam optimizer described in the paper, with $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10e - 9$. And then they create a scheduler to vary

```python
class PositionalEncoding(nn.Module):
    "Implement the PE function."
    def __init__(self, d_model, dropout, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        # Compute the positional encodings once in log space.
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) *
                             -(math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + Variable(self.pe[:, :x.size(1)],
                         requires_grad=False)
        return self.dropout(x)
```

Figure 21. Positional Encoding

```python
class Batch:
    "Object for holding a batch of data with mask during training."
    def __init__(self, src, trg=None, pad=0):
        self.src = src
        self.src_mask = (src != pad).unsqueeze(-2)
        if trg is not None:
            self.trg = trg[:, :-1]
            self.trg_y = trg[:, 1:]
            self.trg_mask = \
                self.make_std_mask(self.trg, pad)
            self.ntokens = (self.trg_y != pad).data.sum()

    @staticmethod
    def make_std_mask(tgt, pad):
        "Create a mask to hide padding and future words."
        tgt_mask = (tgt != pad).unsqueeze(-2)
        tgt_mask = tgt_mask & Variable(
            subsequent_mask(tgt.size(-1)).type_as(tgt_mask.data))
        return tgt_mask
```

Figure 22. batch and masking

the learning rate over the training process according to:

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$$

This corresponds to increasing the learning rate linearly for the first $warmup\_steps$ training steps, and decreasing it thereafter proportionally to the inverse square root of the step number. They used $warmup\_steps = 4000$.

4.10. Label Smoothing

Label Smoothing() (Fig. 24)is a regularization technique where they tackle the problem of over fitting and overconfidence. They implement label smoothing using the KL div loss. they create a distribution that has confidence of the correct word and the rest of the smoothing mass distributed throughout the vocabulary. The block first decide

```python
def run_epoch(data_iter, model, loss_compute):
    "Standard Training and Logging Function"
    start = time.time()
    total_tokens = 0
    total_loss = 0
    tokens = 0
    for i, batch in enumerate(data_iter):
        out = model.forward(batch.src, batch.trg,
                            batch.src_mask, batch.trg_mask)
        loss = loss_compute(out, batch.trg_y, batch.ntokens)
        total_loss += loss
        total_tokens += batch.ntokens
        tokens += batch.ntokens
        if i % 50 == 1:
            elapsed = time.time() - start
            print("Epoch Step: %d Loss: %f Tokens per Sec: %f" %
                    (i, loss / batch.ntokens, tokens / elapsed))
            start = time.time()
            tokens = 0
    return total_loss / total_tokens
```

Figure 23. training loop

```python
class NoamOpt:
    "Optim wrapper that implements rate."
    def __init__(self, model_size, factor, warmup, optimizer):
        self.optimizer = optimizer
        self._step = 0
        self.warmup = warmup
        self.factor = factor
        self.model_size = model_size
        self._rate = 0

    def step(self):
        "Update parameters and rate"
        self._step += 1
        rate = self.rate()
        for p in self.optimizer.param_groups:
            p['lr'] = rate
        self._rate = rate
        self.optimizer.step()

    def rate(self, step = None):
        "Implement `lrate` above"
        if step is None:
            step = self._step
        return self.factor * \
            (self.model_size ** (-0.5) *
            min(step ** (-0.5), step * self.warmup ** (-1.5)))

def get_std_opt(model):
    return NoamOpt(model.src_embed[0].d_model, 2, 4000,
            torch.optim.Adam(model.parameters(), lr=0, betas=(0.9, 0.98), eps=1e-9))
```

Figure 24. Optimizer

the smoothing and confidence level. Here, instead of making the probabilities for expected 1 and 0 for others, they set the log-probabilities of expected to be between 1 - smoothing and others to smoothing / (size - 2). Label smoothing starts to penalizing the model when it gets very high confidence about a model. The function returns the KL Div Loss of the prediction array.

4.11. Loss Computation

Here the Loss Computation Block (Fig. 25) will compute the KL Div loss and pass this loss for backward propagation. For optimizer they have used the Adam optimizer. This function will return the loss for each training.

```python
class LabelSmoothing(nn.Module):
    "Implement label smoothing."
    def __init__(self, size, padding_idx, smoothing=0.0):
        super(LabelSmoothing, self).__init__()
        self.criterion = nn.KLDivLoss(size_average=False)
        self.padding_idx = padding_idx
        self.confidence = 1.0 - smoothing
        self.smoothing = smoothing
        self.size = size
        self.true_dist = None

    def forward(self, x, target):
        assert x.size(1) == self.size
        true_dist = x.data.clone()
        true_dist.fill_(self.smoothing / (self.size - 2))
        true_dist.scatter_(1, target.data.unsqueeze(1), self.confidence)
        true_dist[:, self.padding_idx] = 0
        mask = torch.nonzero(target.data == self.padding_idx)
        if mask.dim() > 0:
            true_dist.index_fill_(0, mask.squeeze(), 0.0)
        self.true_dist = true_dist
        return self.criterion(x, Variable(true_dist, requires_grad=False))
```

Figure 25. Label Smoothing

```python
class SimpleLossCompute:
    "A simple loss compute and train function."
    def __init__(self, generator, criterion, opt=None):
        self.generator = generator
        self.criterion = criterion
        self.opt = opt

    def __call__(self, x, y, norm):
        x = self.generator(x)
        loss = self.criterion(x.contiguous().view(-1, x.size(-1)),
                              y.contiguous().view(-1)) / norm
        loss.backward()
        if self.opt is not None:
            self.opt.step()
            self.opt.optimizer.zero_grad()
        return loss.data[0] * norm
```

Figure 26. Loss Computation

5. Results Replicated from Existing Transformer code of FairSeq

We experiment with an existing implementation of Transformer by FairSeq [2]. This transformer architecture (*transformer_iwslt_de_en*) has 6 encoders and decoders and was trained originally on *IWSLT14 German to English dataset* by the authors. We have trained the model from scratch on a subset of IIT Bombay English-Hindi Corpus [8] called the *Book Translations (Gyaan-Nidhi Corpus)*. We chose Hindi to be the source language and English to be the target language

The IIT Bombay English-Hindi corpus contains parallel corpus for English-Hindi collected from a variety of existing sources and corpora developed at the Center for Indian Language Technology, IIT Bombay over the years. Here we specifically pick one such corpus *Book Translations (Gyaan-Nidhi Corpus)*, as we observed it to contain good readable sentences

Gyaan-Nidhi Corpus	Number of sentences
Training data	207,804
Validation data	9,874
Testing data	9,445

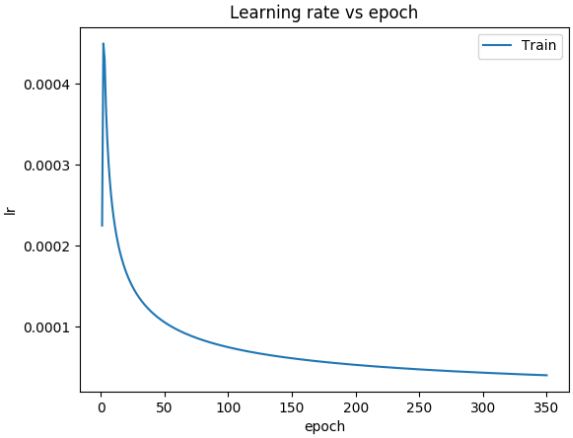

Figure 27. The learning of Hi-En model over 350 epochs

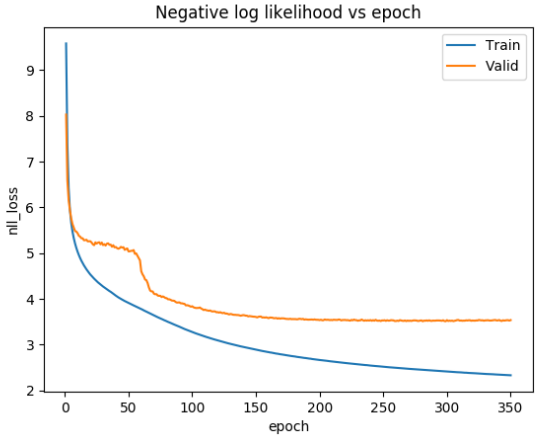

Figure 28. The negative log likelihood loss of Hi-En model over 350 epochs

Avg. words per sentence	English	Hindi
Training data	19.841	20.245
Validation data	19.744	20.074
Testing data	20.030	20.459

NLL Loss	Training	Validation
6E-6D with L2_alpha 1e-4	2.901	5.241
7E-7D with L2_alpha 1e-4	2.883	5.334
8E-8D with L2_alpha 1e-4	3.237	5.227
6E-6D with L2_alpha 1e-3	**2.332**	**3.538**

Hi-En model	BLEU-1	BLEU-4
Training data	0.4693	0.2429
Validation data	0.4236	0.1419
Testing data	0.4251	0.1437

Book Translations (Gyaan-Nidhi Corpus) has 227,123 sentences in total. We split this data into training, validation and test as shown above. We also observe that the average number of words per sentence is about 20 in both languages.

We apply Byte Pair Encoding (BPE) jointly on both the languages training data and learn a total of 10 BPE tokens. We used ADAM optimizer with betas as 0.9 and 0.98. We used a learning rate scheduler which increases linearly till 5e-4 in 4000 updates (warmup updates) and then decreases proportional to inverse square root (see figure 6). We use a droupt of 0.3. We use label smoothed cross entropy function with label smoothing as 0.1. We limit the maximum number of tokens to 4096.

We find that with the L2 regularization parameter alpha as 1e-3, trained for 350 epochs provide the best results. The plot of negative log likelihood (NLL) loss vs epochs

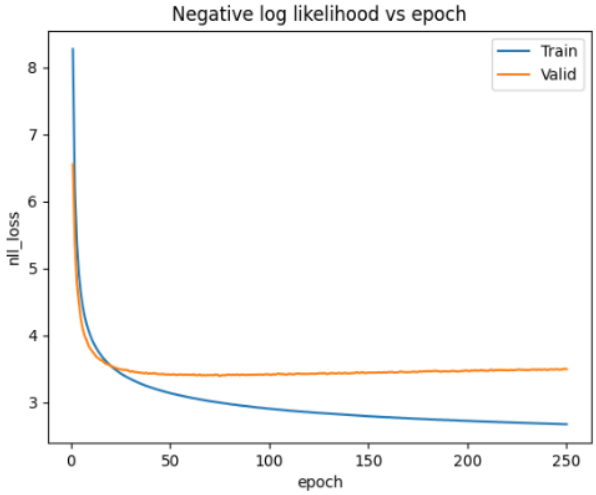

Figure 29. The negative log likelihood loss of Hi-En model over 250 epochs using BBPE

is shown in figure 7. Training took 7 hours on RTX3090 GPU for the best model.

We also experimented with Transliteration of Hindi to of Hindi to English using Aksharmukha [1] python package. Transliteration is a type of conversion of a text from one script to another that involves swapping letters in predictable ways. This was mainly used for transliterating Hindi to English characters so that words like names, places which repeat during the time of vocab building for both languages will have a single entry only. Please find below example for transliteration:

S : उत्तर भारत कम अशांत नहीं था।

T : uttara bhArata kama azAMta nahIM thA.

Hi-En transliterated model	BLEU-4
Training data	0.3075
Validation data	0.1214
Testing data	0.1211

From the results we can see that transliteration didnot result in an improvement in scores for both validation and test dataset, So we thought of ignoring transliteration and improve the quality of data by preprocessing.

The quality of the data on which the model get trained, determines the performance of the model. Therefore text processing is an important step in our methodology, as it affects the quality of the data going into the model. We experiment with and without text processing. In text processing we handle the below cases,

1) Converting all characters to lowercase (in case of English)

2) Converting special characters like $, &, £ to their corresponding word (dollar, and, euro) (in case of English)

3) Replacing '-' between two numbers by 'to' (in case of English) and by '' (in case of Hindi) and removing '-' between words (in both languages)

Hi-En Preprocessed dataset model	BLEU-4
Training data	0.3133
Validation data	0.1982
Testing data	0.1982

From the results of preprocessing, we can see that the model scores are improved and this can be further used in future model improvements.

There are various types of encodings available to convert a sentence to a sequence of tokens. Few of the tokenization methods are Character-level tokenization, Byte-level Byte Pair Encoding (BBPE), Byte Pair Encoding (BPE) and Word-level tokenization. Among these, the most frequently used tokenization is the BPE in the NLP community.

We tried replacing Byte Pair Encoding (BPE) with Byte Level Byte Pair Encoding (BBPE) with 4096 as vocabulary size on the preprocessed dataset. All the other parameters are same as what we used in Byte Pair Encoding (BPE) but we have used only 2 encoders and 2 decoders. We observed that some new relatable words were introduced replacing target ground truth sentence words. Below are some statistics with Byte Level Byte Pair Encoding (BBPE):

source sentence	इस तथ्य ने सिपाहियों की धार्मिक भावनाओं को उभारा और उनमें क्रोध पैदा हुआ।	उन्होंने बंबई विश्वविद्यालय से स्नातक की परीक्षा पास की और अपना सारा जीवन राष्ट्र की सेवा में लगा दिया।
ground truth	The sepoys were angered at this affront to their religious feelings.	He graduated from the Bombay University and devoted his entire life to the service of the country.
BPE translated sentence	The fact created a stir among the sepoys and angered them.	He passed the graduate examination from the Bombay University and devoted his whole life to the service of the nation.
BBPE translated sentence	this enlightened the religious sentiments of the soldiers and anger was created in them.	he passed the graduate from the university of bombay and spent his life in the service of the nation.
source sentence	इतने व्यापक और विविधतापूर्ण श्रोतावर्ग से वे घनिष्ठ और निरंतर संपर्क कैसे बनाकर रखे?	पेंशन के लिए कम-से-कम 10 वर्ष की सेवा जरुरी है।
ground truth	How was he to maintain a close and continuing contact with so vast and varied an audience?	A minimum 10 years' service is required for entitlement to pension.
BPE translated sentence	How did he keep in close and continuous touch with the vast and varied audience?	For pension at least 10 years the service is required.
BBPE translated sentence	how did he keep close and constant contact with such a wide and varied audience?	service of at least 10 years is required for pension.

Figure 30. Sample Translation from Hi to En

Avg. words per sentence	English	Hindi
Training data	29.20	178.84
Validation data	29.17	177.74
Testing data	29.54	180.90

NLL Loss	Training	Validation
2E-2D with L2_alpha 1e-4	2.67	**3.492**

Hi-En BBPE model	BLEU-4
Training data	0.2726
Validation data	0.2142
Testing data	0.2121

6. Implementation of Transformer architecture using PyTorch

We have implementing the transformer architecture using PyTorch building blocks. Link to our github repo provided here. Data used to train our implementation of Transformer is a sequence of numbers in [1, 10] and we try to learn identity mapping between source and target sequence

7. Conclusion

In this work we explained transformer's overall architecture and it's implementation. Transformer is the first model came up based on attention completely. Transformers re-

placed RNNs which are doing well in language translation tasks till that day. Because of it's markable performance in translation they are tried in vision tasks too. Over the time multiple transformer variants like ViT, DeTR, Swin Transformer are developed in Vision community which gave hopes for transformer to standout as an single overall model for both translation and vision tasks. But with latest paper like A ConvNet for 20s convolutions came back with a whooping performance over transformers particularly in vision community.

References

[1] Aksharmukha transliteration python package https://aksharamukha.appspot.com/python. 10

[2] Fairseq implementation of transformer https://github.com/pytorch/fairseq/blob/main. 9

[3] Jay Alammar. The illustrated transformer https://jalammar.github.io/illustrated-transformer/, 2018. 2

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014. 2

[5] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc Le. Massive exploration of neural machine translation architectures. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1442–1451, Copenhagen, Denmark, Sept. 2017. Association for Computational Linguistics. 2

[6] Kyunghyun Cho Changhan Wang and Jiatao Gu. Neural machine translation with byte-level subwords. *CoRR*, abs/1909.03341, 2019. 5

[7] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014. cite arxiv:1406.1078Comment: EMNLP 2014. 2, 3

[8] Anoop Kunchukuttan, Pratik Mehta, and Pushpak Bhattacharyya. The IIT Bombay English-Hindi parallel corpus. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan, May 2018. European Language Resources Association (ELRA). 9

[9] Zhouhan Lin, Minwei Feng, Cícero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. A structured self-attentive sentence embedding. In *ICLR (Poster)*. OpenReview.net, 2017. 2

[10] Bernadine Racoma. What is human translation and why is it important? https://www.daytranslations.com/blog/human-translation-important/, 2015. 1

[11] Alexander M. Rush. Annotated transformer. *CoRR*, abs/1909.03341, 2019. 5

[12] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, page 3104–3112, Cambridge, MA, USA, 2014. MIT Press. 2, 5

[13] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. 1, 2

[14] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Googleś neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016. 2

8. Contributions

Exhaustive literature survey done by B.Sasideep and Deevanshu M. Gupta. Implementation of Transformer using PyTorch building block provided here is done by Arvind Roshaan Results Replicated from Existing Transformer code of FairSeq is done by Sanyog and Srikanth