NATIONAL UNIVERSITY OF HO CHI MINH CITY
UNIVERSITY OF INFORMATION TECHNOLOGY
FACULTY OF INFORMATION SCIENCE AND ENGINEERING

COURSE PROJECT REPORT

IE229 - ARTIFICIAL INTELLIGENCE

TOPIC:

Evaluation of popular Deep Learning architectures for Natural Machine Translation with IWSLT15 (en – vi) dataset

Lecturer: Takashi Ninomiya
Huỳnh Văn Tín
Lưu Thanh Sơn
Nguyễn Thành Luân

Student group:

1. Nguyễn Văn Khoa ID: 18520929
2. Lê Thị Thu Hằng ID: 18520274
3. Nguyễn Thị Hồng Nhung ID: 18521218

ꕥ Ho Chi Minh City, 7/2021 ꕥ

LECTURE'S COMMENTS

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Preface

Natural Language Processing or NLP is a field of Artificial Intelligence that gives machines the ability to read, understand and derive meaning from human languages.

It is a discipline that focuses on the interaction between data science and human language and is scaling to countless industries. Today, NLP is booming thanks to considerable improvements in the access to data and increases in computational power, allowing practitioners to achieve meaningful results in areas like healthcare, media, finance, and human resources, among others.

In simple terms, NLP represents the automatic handling of natural human languages like speech or text, and although the concept itself is fascinating, the real value behind this technology comes from the use cases. NLP can help humans with lots of tasks, and the fields of application seem to increase daily. Some of them are described below:

- NLP enables the recognition and prediction of diseases based on electronic health records and patient's speech. This capability is being explored in health conditions that go from cardiovascular diseases to depression and even schizophrenia. For example, Amazon Comprehend Medical is a service that uses NLP to extract disease conditions, medications, and treatment outcomes from patient notes, clinical trial reports, and other electronic health records.
- Organizations can determine what customers are saying about a service or product by identifying and extracting information from sources like social media. This sentiment analysis can provide much information about customers' choices and their decision drivers.
- An inventor at IBM developed a cognitive assistant that works like a personalized search engine by learning all about users and then remind

them of a name, a song, or anything they cannot remember the moment they need.

- Companies like Yahoo and Google filter and classify user's emails with NLP by analyzing text in emails that flow through their servers and stopping spam before they even enter the inbox.
- To help to identify fake news, the NLP Group at MIT developed a new system to determine if a source is accurate or politically biased, detecting if a news source can be trusted or not.

And there are many other applications of NLP in everyday life, suggesting that this will be a core area in the future.

In our study, we focus on machine translation task of NLP. We organize the structure of the report as follows:

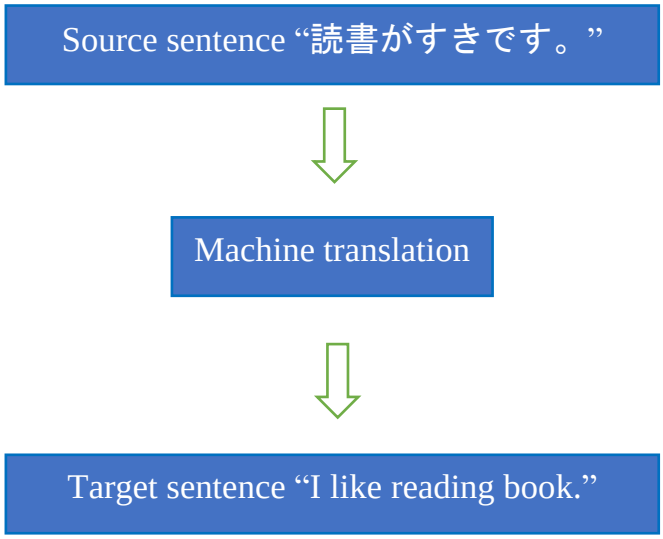
- **Section 1:** Overview.
- **Section 2**: Dataset.
- **Section 3**: Experiments.
- **Section 4**: Results and discussion.
- **Section 5:** Conclusion.

Section 1. OVERVIEW

1.1 Problem Description

Machine Translation is the automatic translation from source language to target language without human involvement.

For example:



There are many methods for machine translation tasks. However, we only approach neural network translation (NMT). NMT consists of two components: an encoder that computes a representation for each source sentence and a decoder that generates one target word at a time.

1.2 Challenges and targets

Challenges

- Neural machine translation is a difficult task, so to find a good model for the task is difficult.
- Training the model for this task requires tons of computation costs, so we have to have suitable hardware (GPU or TPU).

Our targets: Our study plans to build the Attention model and the Transformer model, figure out and fine-tune hyperparameters to achieve better performance with upper 10% of bleu score than baseline models.

Section 2. DATASET

2.1 Dataset description

IWSLT15 data is the dataset for the machine translation task used in the IWSLT2015's Shared Task.

IWSLT15 dataset consists of training data 133,317 sentences, validation data 1,553 sentences and test data 1,268 sentences. In our study, we only use training data and test data.

Table 1: Some examples in IWSLT15 (en – vi) dataset.

Source sentence (English)	Target sentence (Vietnamese)
This is the EUPHORE Smog Chamber in Spain.	Đây là Phòng nghiên cứu khói bụi EUPHORE ở Tây Ban Nha.
And then the hurricane comes through, and the house is in much better condition than it would normally have been in.	Và khi cơn bão tới, và căn nhà ở trong trạng thái vững vàng hơn rất nhiều so với bình thường.

To have a deeper understanding of the in IWSLT15 (en – vi) dataset, we analyze the distribution of the sentence length, and the results are shown in Figure 1 and 2. In both training dataset and test dataset, most English and Vietnamese sentences are less than 100 tokens in length. During the training period in the Attention model, to reduce training time and avoid GPU out of memory in Google Colab, we did not train a few sentences over 50 lengths.

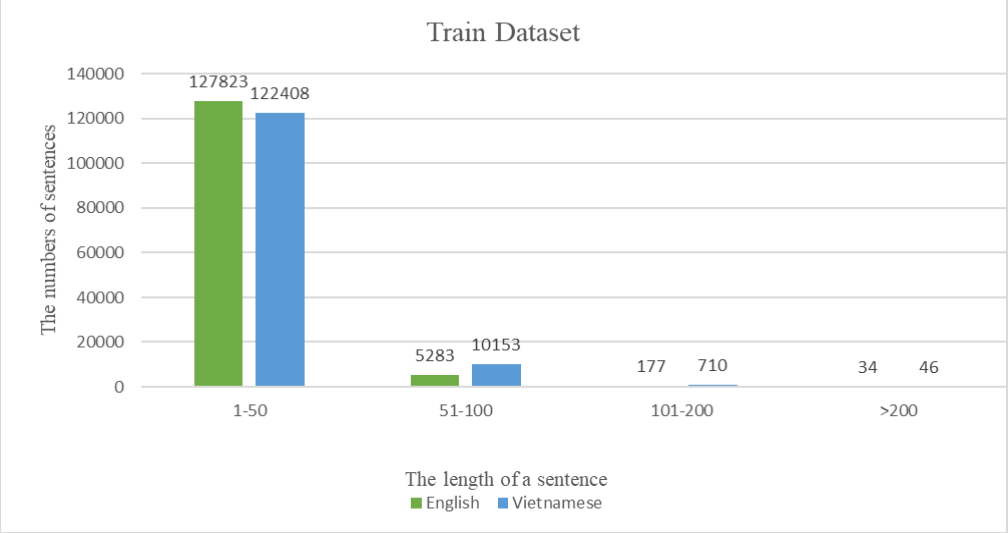


Figure 1: Sentence length distribution in train dataset.

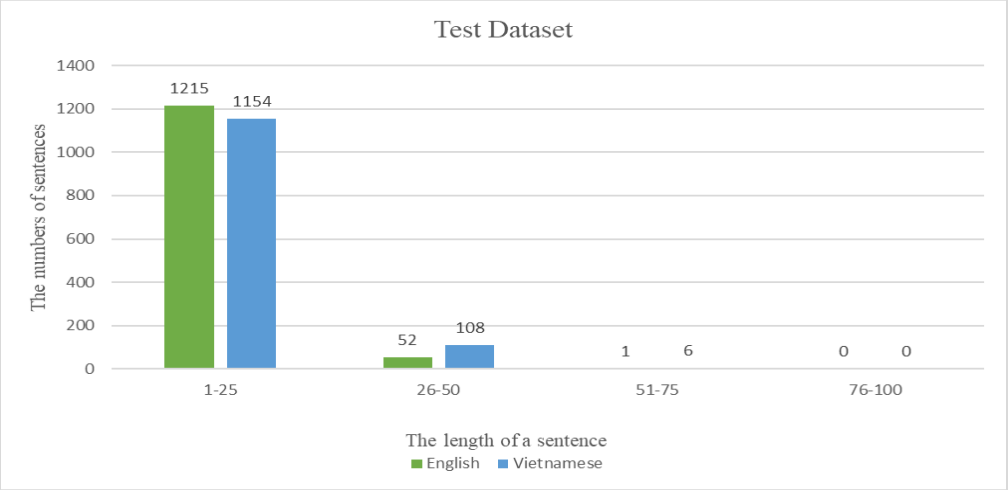


Figure 2: Sentence length distribution in test dataset.

2.2 Data preparation

Text preprocessing aims to convert each sample into a numeric matrix, with each row of the matrix representing a word vector. The text preprocessing steps of this problem are as follows:

- Build a vocabulary set: First, we make a frequency list for each type of token. Then, rearrange the tokens in descending order of appearance frequency, and delete words with a frequency below the threshold. Finally, create the index by assigning an ID for each token, and we have created two variables, a vocabulary list (index to token) and a vocabulary index (token to index).

```
def make_vocab(train_data, min_freq):
  vocab = {}
  for tokenlist in train_data:
    for token in tokenlist:
      if token not in vocab:
        vocab[token] = 0
      vocab[token] += 1
  vocablist = [('<unk>', 0), ('<pad>', 0), ('<cls>', 0), ('<eos>', 0)]
  vocabidx = {}
  for token, freq in vocab.items():
    if freq >= min_freq:
      idx = len(vocablist)
      vocablist.append((token, freq))
      vocabidx[token] = idx
  vocabidx['<unk>'] = 0
  vocabidx['<pad>'] = 1
  vocabidx['<cls>'] = 2
  vocabidx['<eos>'] = 3
  return vocablist, vocabidx

vocablist_en, vocabidx_en = make_vocab(train_en, 3)
vocablist_vi, vocabidx_vi = make_vocab(train_vi, 3)

print('vocab size en:', len(vocablist_en))
print('vocab size vi:', len(vocablist_vi))
```

```
vocab size en: 24420
vocab size vi: 17861
```

Figure 3: Code for creating vocabulary dictionary and Result.

- Add some special tokens to the text: Replace all the tokens not listed in the vocabulary list with the <unk> token. Add the <cls> token to the beginning of the sentences and add the <eos> token to the end.

```
def preprocess(data, vocabidx):
  rr = []
  for tokenlist in data:
    tkl = ['<cls>']
    for token in tokenlist:
      tkl.append(token if token in vocabidx else '<unk>')
    tkl.append('<eos>')
    rr.append((tkl))
  return rr

train_en_prep = preprocess(train_en, vocabidx_en)
train_vi_prep = preprocess(train_vi, vocabidx_vi)
test_en_prep = preprocess(test_en, vocabidx_en)
```

Figure 4: Code for adding special tokens to the sentence.

```
['<cls>', 'Rachel', 'Pike', ':', 'The', 'science', 'behind', 'a', 'climate', 'headline', '<eos>']
['<cls>', 'Khoa_học', 'đằng', 'sau', 'một', 'tiêu_đề', 'về', 'khí_hậu', '<eos>']
```

Figure 5: Two example results of adding special tokens to the sentence.

- Zip English sets and Vietnamese sets.

```
train_data = list(zip(train_en_prep, train_vi_prep))
train_data.sort(key = lambda x: (len(x[0]), len(x[1])))
test_data = list(zip(test_en_prep, test_en, test_vi))

for i in range(5):
  print(train_data[i])
for i in range(5):
  print(test_data[i])
```

Figure 6: Code for zipping English sets and Vietnamese sets.

```
(['<cls>', '<eos>'], ['<cls>', '', '<eos>'])
(['<cls>', '<eos>'], ['<cls>', '.', '<eos>'])
(['<cls>', '<eos>'], ['<cls>', '.', '<eos>'])
(['<cls>', '<eos>'], ['<cls>', 'vỗ', 'tay', '<eos>'])
(['<cls>', '<eos>'], ['<cls>', 'Vỗ', 'tay', '<eos>'])
```

Figure 7: Some example results of English sets and Vietnamese sets.

- Make mini-batch: Convert to mini-batch format – from list of label and token list pairs to list of N pairs of token list and labels list (N is batch size).
- Padding: Since many texts in the mini-batch are analyzed together, the length of each text in the mini-batch must be equal. We align the length to the longest text in the mini batch using inserting the <pad> token.

```
def make_batch(data, batchsize):
  bb = []
  ben = []
  bvi = []
  for en, vi in data:
    ben.append(en)
    bvi.append(vi)
    if len(ben) >= batchsize:
      bb.append((ben, bvi))
      ben = []
      bvi = []
  if len(ben) > 0:
    bb.append((ben, bvi))
  return bb

train_data = make_batch(train_data, BATCHSIZE)
def padding_batch(b):
  maxlen = max([len(x) for x in b])
  for tkl in b:
    for i in range(maxlen - len(tkl)):
      tkl.append('<pad>')

def padding(bb):
  for ben, bvi in bb:
    padding_batch(ben)
    padding_batch(bvi)

padding(train_data)
```

Figure 8: Code for making mini batch and padding.

Here is an example result of making mini-batch and padding.

```
([['<cls>', '<eos>', '<pad>', '<pad>'], ['<cls>', '<eos>', '<pad>', '<pad>'], ['<cls>', '<eos>', '<pad>', '<pad>'], ['<cls>', '<eos>', '<pad>', '<pad>'], ['<cls>', '.', '<eos>', '<pad>'], ['<cls>', 'Yeah', '.', '<eos>'], ['<cls>', 'OK', '.', '<eos>'], ['<cls>', 'Okay', '.', '<eos>']], [['<cls>', 'Vỗ', 'tay', '.', '<eos>', '<pad>'], ['<cls>', 'Vỗ', 'tay', '.', '<eos>', '<pad>'], ['<cls>', 'Khán', 'giả', 'vỗ', 'tay', '<eos>'], ['<cls>', '&quot;', 'tán', 'thưởng', '&quot;', '<eos>'], ['<cls>', '.', '<eos>', '<pad>', '<pad>', '<pad>'], ['<cls>', '<unk>', '<eos>', '<pad>', '<pad>', '<pad>'], ['<cls>', 'OK', '<eos>', '<pad>', '<pad>', '<pad>'], ['<cls>', 'Ok', '<eos>', '<pad>', '<pad>', '<pad>']])
```

- Finally, convert all the words to index using vocabulary index.

```
train_data = [([[vocabidx_en[token] for token in tokenlist]
                for tokenlist in ben],
               [[vocabidx_vi[token] for token in tokenlist]
                for tokenlist in bvi]) for ben, bvi in train_data]

test_data = [([vocabidx_en[token]for token in enprep], en, vi)
for enprep, en, vi in test_data]

for i in range(3):
  print(train_data[i])
for i in range(3):
  print(test_data[i])
```

Figure 9: Code for converting all the words to index.

These are example result in train data, and example result in test data of converting all the words to index.

```
([[2, 3, 1, 1], [2, 3, 1, 1], [2, 3, 1, 1], [2, 3, 1, 1], [2, 48, 3, 1], [2, 2481, 48, 3], [2, 1785, 48, 3], [2, 1527, 48, 3]], [[2, 5245, 461, 72, 3, 1], [2, 5245, 461, 72, 3, 1], [2, 1818, 434, 3424, 461, 3], [2, 595, 1833, 2331, 595, 3], [2, 72, 3, 1, 1, 1], [2, 0, 3, 1, 1, 1], [2, 3213, 3, 1, 1, 1], [2, 1711, 3, 1, 1, 1]])
([2, 109, 49, 96, 198, 6154, 48, 3], ['And', 'I', 'was', 'very', 'proud', '.'], ['Tôi', 'đã', 'rất', 'tự', 'hào', 'về', 'đất', 'nước', 'tôi', '.'])
```

Section 3. EXPERIMENTS

3.1 Baseline models

- Encoder and decoder use recurrent neural network (RNN) with ReLU activation function according to formular (1). Where, x_t is token input and h_t is hidden state of timestep t.

$$RNN(x_t, h_t) = ReLU(x_t + Wh_t + b) \quad (1)$$

```
class RNNEncDec(torch.nn.Module):
  def __init__(self, vocablist_x, vocabidx_x, vocablist_y, vocabidx_y):
    super(RNNEncDec, self).__init__()
    self.encemb = torch.nn.Embedding(len(vocablist_x), 300,
                                     padding_idx=vocabidx_x['<pad>'])
    self.encrnn = torch.nn.Linear(300, 300)
    self.decemb = torch.nn.Embedding(len(vocablist_y), 300,
                                     padding_idx=vocabidx_y['<pad>'])
    self.decrnn = torch.nn.Linear(300, 300)
    self.decout = torch.nn.Linear(300, len(vocablist_y))

  def forward(self, x):
    x, y = x[0], x[1]
    #enc
    e_x = self.encemb(x)
    n_x = e_x.size()[0]
    h = torch.zeros(300, dtype=torch.float32).to(DEVICE)
    for i in range(n_x):
      h = F.relu(e_x[i] + self.encrnn(h))
    #dec
    e_y = self.decemb(y)
    n_y = e_y.size()[0]
    loss = torch.tensor(0., dtype=torch.float32).to(DEVICE)
    for i in range(n_y-1):
      h = F.relu(e_y[i] + self.decrnn(h))
      loss += F.cross_entropy(self.decout(h), y[i+1])
    return loss
```

Figure 10: Code for RNN – based model.

- Encoder and decoder use LSTM with 2 layers and drop out layer.

```
self.encemb = torch.nn.Embedding(len(vocablist_x), 300,
                                 padding_idx = vocabidx_x['<pad>'])
self.dropout = torch.nn.Dropout(0.5)
self.enclstm = torch.nn.LSTM(300,516,2,dropout=0.5)
self.decemb = torch.nn.Embedding(len(vocablist_y), 300,
                                 padding_idx = vocabidx_y['<pad>'])
self.declstm = torch.nn.LSTM(300,516,2,dropout=0.5)
self.decout = torch.nn.Linear(516, len(vocabidx_y))
```

Figure 11: Code for LSTM + drop out model.

3.2 Encoder – decoder model with attention

Bahdanau et al. [1] suggested in the paper 'Neural Machine Translation by Jointly Learning to Align and Translate' that use for the decoder not only context vector learned from the encoder but also an alignment vector to represent the relevant information between input words of the encoder and input words of the decoder. Attention mimics how human translation works when the human translation simultaneously looks at one or several words and then starts the translation. Attention helps to address the information bottleneck problem of RNN-based models.

This model is still the seq2seq architecture, and only the decoder has input that adds outputs of the attention layer.

Attention

Attention has various variants, but in our experiment we use additive attention by Luong et al. [2]. Additive scoring function is calculated according to the formula (2).

$$W_s(\tanh(W_c[E_o + D_h^{t-1}])) \qquad (2)$$

Where, D_h^{t-1} is decoder hidden state from previous step with shape [1, batch size, decoder hidden dimension]. E_o is encoder outputs with shape [source token length, batch size, encoder hidden dimension]. W_s and W_c are learnable parameter and we will use a Linear layer for these.

```
class Attention(nn.Module):
  def __init__(self, encoder_hidden_dim, decoder_hidden_dim): # (512, 512)
    super(Attention, self).__init__()
    self.attn_hidden_vector = nn.Linear(encoder_hidden_dim +
                                        decoder_hidden_dim, decoder_hidden_dim)
# attn_hidden_vector has dimension [source len, batch size, decoder hidden dim]
# > set output dim of linear layer = 1 => [sourse len, batch size]
    self.attn_scoring_fn = nn.Linear(decoder_hidden_dim, 1, bias=False)
  def forward(self, hidden, encoder_outputs):
# hidden = [1, batch size, decoder hidden dim]
    enc_len = encoder_outputs.shape[0]
    hidden = hidden.repeat(enc_len, 1, 1)
# Calculate attention hidden values
    attn_hidden = torch.tanh(self.attn_hidden_vector
                             (torch.cat((hidden, encoder_outputs), dim=2)))

# Calculate the Scoring function. Remove 3rd dimension.
    attn_scoring_vector = self.attn_scoring_fn(attn_hidden).squeeze(2)

# The attn_scoring_vector has dimension of [source len, batch size]
# Since we need to calculate the softmax per record in the batch
# we will switch the dimension to [batch size,source len]
    attn_scoring_vector = attn_scoring_vector.permute(1, 0)

# Softmax function for normalizing the weights to probability distribution
    return F.softmax(attn_scoring_vector, dim=1)
```

Figure 12: Code for class Attention.

Instead of using the RNN/LSTM layer, encoder and decoder use the GRU layer. GRU is an improvement of LSTM with fewer parameters, which helps to reduce training time.

Encoder

The encoder structure is the same as the base model.

```
class Encoder(nn.Module):
    def __init__(self, vocablist_x, vocabidx_x, embedding_dim,
                 encoder_hidden_dim, dropout_prob=0.5):
        super().__init__()

        self.embedding = nn.Embedding(len(vocablist_x), embedding_dim,
                                      padding_idx = vocabidx_x['<pad>'])
        self.rnn = nn.GRU(embedding_dim, encoder_hidden_dim,
                          dropout=dropout_prob)

        self.dropout = nn.Dropout(dropout_prob)

    def forward(self, input_batch): # input_batch is ben
        embedded = self.dropout(self.embedding(input_batch))
        outputs, hidden = self.rnn(embedded)

        return outputs, hidden
```

Figure 13: Code for class Encoder.

Decoder

The decoder contains the attention layer (Figure 14). The input of the GRU layer is concatenated between embedding and the dot product of attention output and encoder outputs (Figure 15). We use the teacher forcing method in model training (Figure 16).

```
class OneStepDecoder(nn.Module):
    def __init__(self, vocablist_y, vocabidx_y, embedding_dim,
                 encoder_hidden_dim, decoder_hidden_dim, attention,
                 dropout_prob=0.5):
        super(OneStepDecoder, self).__init__()

        self.output_dim = len(vocablist_y)
        self.attention = attention

        self.embedding = nn.Embedding(len(vocablist_y), embedding_dim,
                                      padding_idx = vocabidx_y['<pad>'])
# Add the encoder hidden dim and embedding dim
        self.rnn = nn.GRU(encoder_hidden_dim + embedding_dim,
                          decoder_hidden_dim)
# Combine all the features for better prediction
        self.fc = nn.Linear(encoder_hidden_dim + decoder_hidden_dim +
                            embedding_dim, len(vocablist_y))
        self.dropout = nn.Dropout(dropout_prob)
```

Figure 14: Code for function initial of class OneStepDecoder.

```
    def forward(self, input, hidden, encoder_outputs):
# Add the source len dimension
        input = input.unsqueeze(0)
        embedded = self.dropout(self.embedding(input))
# Calculate the attention weights
        a = self.attention(hidden, encoder_outputs).unsqueeze(1)
# We need to perform the batch wise dot product.
# Hence need to shift the batch dimension to the front.
        encoder_outputs = encoder_outputs.permute(1, 0, 2)
# Use PyTorch's bmm function to calculate the weight W.
        W = torch.bmm(a, encoder_outputs)
# Revert the batch dimension.
        W = W.permute(1, 0, 2)
# Concatenate the previous output with W
        rnn_input = torch.cat((embedded, W), dim=2)
        output, hidden = self.rnn(rnn_input, hidden)
# Remove the sentence length dimension and pass them to the Linear layer
        predicted_token = self.fc(torch.cat((output.squeeze(0), W.squeeze(0),
                                             embedded.squeeze(0)), dim=1))

        return predicted_token, hidden, a.squeeze(1)
```

Figure 15: Code for forward of class OneStepDecoder

```
class Decoder(nn.Module):
    def __init__(self, one_step_decoder):
        super().__init__()
        self.one_step_decoder = one_step_decoder

    def forward(self, target, encoder_outputs, hidden,
                teacher_forcing_ratio=0.5):
        batch_size = BATCHSIZE
        input = target[0, :]
        n_y = target.shape[0]
# Tensor to store decoder outputs
        outputs = torch.zeros(n_y,BATCHSIZE,len(vocablist_vi)).to(DEVICE)

        for t in range(1, n_y-1):
# Pass the encoder_outputs. For the first time step the
# hidden state comes from the encoder model.
            output, hidden, a = self.one_step_decoder(input, hidden,
                                                      encoder_outputs)
# Place predictions in a tensor holding predictions for each token
            outputs[t] = output
# Decide if we are going to use teacher forcing or not
            teacher_force = random.random() < teacher_forcing_ratio
# Get the highest predicted token from our predictions
            top1 = output.argmax(1)
# If teacher forcing, use actual next token as next input
# If not, use predicted token
            input = target[t] if teacher_force else top1

        return outputs
```

Figure 16: Code for class Decoder using teacher forcing technique.

During training model, we ignore loss calculation for the <pad> token.

```
# Define the optimizer
optimizer = torch.optim.Adam(model.parameters(), lr = LR)

criterion = nn.CrossEntropyLoss(ignore_index=1)
```

Figure 17: Loss calculation ignore <pad> token.

Seq2seq Model

Class EncoderDecoder has class encoder and decoder class.

```
class EncodeDecoder(nn.Module):
    def __init__(self, encoder, decoder):
        super().__init__()

        self.encoder = encoder
        self.decoder = decoder

    def forward(self, x, teacher_forcing_ratio=0.5):
        encoder_outputs, hidden = self.encoder(x[0])
        return self.decoder(x[1], encoder_outputs, hidden,
                            teacher_forcing_ratio)
```

Figure 18: Code for class seq2seq (encoder and decoder).

Instantiate the model

```
# Instantiate the models
attention_model = Attention(hidden_dim, hidden_dim)
encoder = Encoder(vocablist_x, vocabidx_x, embedding_dim, hidden_dim)
one_step_decoder = OneStepDecoder(vocablist_y, vocabidx_y, embedding_dim,
                                  hidden_dim, hidden_dim, attention_model)
decoder = Decoder(one_step_decoder)

model = EncodeDecoder(encoder, decoder)

model = model.to(DEVICE)
```

Figure 19: Code for instantiate the model.

3.3 TRANSFORMER

Vaswani et al. [3] proposed a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. For translation tasks, the Transformer can be trained significantly faster than architectures based on recurrent or convolutional layers. Transformer removes seq2seq architecture and still uses cross-entropy for loss function. In Transformer, encoder and decoder use three key technologies: Self-attention, positional encoding, and multi-head attention. The encoder's task is to learn the relationship between each token in the source sentence with the rest of the tokens (self attention) and add this relationship to the embedding of each token. The decoder's task is to pass each token of the target sentence to the decoder to search for the next word soon after. This process only stops when it

meets <eos> token. Masked multi-head attention learns the relationship between a token and the tokens in front of this token. The relationship is then added to each token's embedding. Then, implement multi-head attention with the output of the encoder and add information to the embedding of each token. The output is used to predict the next token.

Instead of implementing the Transformer model from scratch, We use the Transformer[1] model of PyTorch. the initial creation of a transformer needs embedding size, number of heads, number of encoder layers, number of decoder layers, feed forward dimension and dropout.

```
self.transformer = Transformer(d_model=emb_size,
                               nhead=nhead,
                               num_encoder_layers=num_encoder_layers,
                               num_decoder_layers=num_decoder_layers,
                               dim_feedforward=dim_feedforward,
                               dropout=dropout)
```

Figure 20: Code for instantiate the model.

We do not need to define encoder component and decoder component.

```
def encode(self, src: Tensor, src_mask: Tensor):
    return self.transformer.encoder(self.positional_encoding(
                        self.src_tok_emb(src)), src_mask)

def decode(self, tgt: Tensor, memory: Tensor, tgt_mask: Tensor):
    return self.transformer.decoder(self.positional_encoding(
                      self.tgt_tok_emb(tgt)), memory,
                      tgt_mask)
```

Figure 21: Code for encoder function and decoder fuction.

[1] https://pytorch.org/docs/stable/generated/torch.nn.Transformer.html

In positional encodings for encoder and decoder, we use pre – defined function (sinusoidal function), according to the formula (3) and formula (4).

$$PE_{(pos,2i)} = sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \quad (3)$$

$$PE_{(pos,2i+1)} = cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \quad (4)$$

Where, pos is the position and i is the dimension.

```
class PositionalEncoding(nn.Module):
    def __init__(self,
                 emb_size: int,
                 dropout: float,
                 maxlen: int = 900):
        super(PositionalEncoding, self).__init__()
        den = torch.exp(- torch.arange(0, emb_size, 2)* math.log(10000) / emb_size)
        pos = torch.arange(0, maxlen).reshape(maxlen, 1)
        pos_embedding = torch.zeros((maxlen, emb_size))
        pos_embedding[:, 0::2] = torch.sin(pos * den)
        pos_embedding[:, 1::2] = torch.cos(pos * den)
        pos_embedding = pos_embedding.unsqueeze(-2)

        self.dropout = nn.Dropout(dropout)
        self.register_buffer('pos_embedding', pos_embedding)

    def forward(self, token_embedding: Tensor):
        return self.dropout(token_embedding + self.pos_embedding[:token_embedding.size(0), :])
```

Figure 22: Code for class Position Encoding.

We need a subsequent word mask during training that will prevent the model from looking into the future words when making predictions. We will also need masks to hide the source and target padding tokens. Using function ‘create_mask’ to create the additive mask for the source sequence (src_mask), the additive mask for the target sequence (tgt_mask), the additive mask for the encoder output (memory_mask), the ByteTensor mask for source keys per batch (src_key_padding_mask), the ByteTensor mask for target keys per batch (tgt_key_padding_mask) and the ByteTensor mask for memory

keys per batch (memory_key_padding_mask). These parameters are the input of the forward propagation.

```
def generate_square_subsequent_mask(sz):
    mask = (torch.triu(torch.ones((sz, sz), device=DEVICE)) == 1).transpose(0, 1)
    mask = mask.float().masked_fill(mask == 0,
                                    float('-inf')).masked_fill(mask == 1,
                                                               float(0.0))
    return mask


def create_mask(src, tgt):
    src_seq_len = src.shape[0]
    tgt_seq_len = tgt.shape[0]

    tgt_mask = generate_square_subsequent_mask(tgt_seq_len)
    src_mask = torch.zeros((src_seq_len, src_seq_len),
                           device=DEVICE).type(torch.bool)

    src_padding_mask = (src == vocabidx_en['<pad>']).transpose(0, 1)
    tgt_padding_mask = (tgt == vocabidx_vi['<pad>']).transpose(0, 1)
    return src_mask, tgt_mask, src_padding_mask, tgt_padding_mask
```

Figure 23: Code for functions creating mask words.

```
def forward(self,
            src: Tensor,
            trg: Tensor,
            src_mask: Tensor,
            tgt_mask: Tensor,
            src_padding_mask: Tensor,
            tgt_padding_mask: Tensor,
            memory_key_padding_mask: Tensor):
    src_emb = self.positional_encoding(self.src_tok_emb(src))
    tgt_emb = self.positional_encoding(self.tgt_tok_emb(trg))
    outs = self.transformer(src_emb, tgt_emb, src_mask, tgt_mask, None,
                            src_padding_mask, tgt_padding_mask,
                            memory_key_padding_mask)
    return self.generator(outs)
```

Figure 24: Code for forward function of model.

The function to generate output sequence using the greedy algorithm is quite similar to the baseline models of function.

```
def evaluate(model, src, max_len, start_symbol):
    src_mask = torch.zeros((src.shape[0], src.shape[0]),
                           device=DEVICE).type(torch.bool)

    memory = model.encode(src, src_mask)
    ys = torch.ones(1, 1).fill_(start_symbol).type(torch.long).to(DEVICE)
    pred = []
    for i in range(max_len-1):
        memory = memory.to(DEVICE)
        tgt_mask = (generate_square_subsequent_mask(ys.size(0)).type(torch.bool)).to(DEVICE)
        out = model.decode(ys, memory, tgt_mask)

        out = out.transpose(0, 1)
        prob = model.generator(out[:, -1])

        _, next_word = torch.max(prob, dim=1)
        next_word = next_word.item()

        ys = torch.cat([ys,
                        torch.ones(1, 1).type_as(src.data).fill_(next_word)],
                        dim=0)
        if next_word == vocabidx_vi['<eos>']:
            break
        pred_y = vocablist_vi[next_word][0]
        pred.append(pred_y)
    return pred
```

Figure 25: Code for function generating output sequence.

3.4 BLEU evaluation metric

To evaluate the performance of models, we use the BLEU evaluation metric [4]. BLEU is a method for automatic evaluation of machine translation. BLEU is a calculation method based on the n-gram match rate. BLEU compares the n-gram of the candidate translation with the n-gram of the reference translation to count the number of matches. These matches are independent of the positions where they occur. BLEU is generally calculated using a match rate of unigram to 4-gram. At this time, as $N = 4$, the BLEU is calculated according to the formula (5).

$$BLEU = BP\ x\ exp\left(\sum_{n=1}^{N}\frac{1}{N}logP_n\right) \qquad (5)$$

Where, P_n is defined as the count of the number of candidate translation n-gram words that occur in any reference translation divided by the total number of n-gram words in the candidate translation.

The brevity penalty BP is a penalty for a translation that is shorter than the reference translation.

We use bleu_score[2] of Torchtext library.

```
bleu = torchtext.data.metrics.bleu_score(pred, ref)
```

Figure 26: Code for calculating bleu score.

3.5 Experimental Settings

We use TPUs of Google Colab pro.

To install the program for the task, we use the PyTorch framework.

We train the networks using the Adam optimizer by minimizing the cross-entropy. The final parameters we implemented are presented in Table 2.

[2] https://pytorch.org/text/stable/data_metrics.html

Table 2: Suitable parameters of each model.

Parameters		
Baseline models	RNN	Epoch = 10 Batch size = 128 Learning rate = 0.0001 Max length of target sentence = 30
	LSTM + Drop out	Epoch = 10 Batch size = 128 Learning rate = 0.0001 Drop out = 0.5 Hidden size = 516 Num_layers = 2 Embedding size = 300 Max length of target sentence = 50
Our models	Attention	Epoch = 20 Batch size = 500 Learning rate = 0.001 Drop out = 0.5 Hidden size = 512 Embedding size = 200 Teacher forcing = 0.5 Max length of target sentence = 50
	Transformer	Epoch = 20 Batch size = 8 Learning rate = 0.0001 Drop out = 0.1 Embedding size = 512

		Nhead = 8 Encoder layers = 2 Decoder layers = 2 Feed forward dimension = 512 Max length of target sentence = 50

Section 4. RESULTS AND DISCUSSION

Through Figure 27 and Figure 28, We notice that the loss value decreases through each epoch. In particular, there is a considerable decline in the first five epochs. From the 15th epoch on, the loss value slightly decreases, demonstrating that to improve model performance, instead of increasing the number of epochs, we should fine-tune the hyperparameters.

Figure 27: Epoch and training loss graph in Attention.

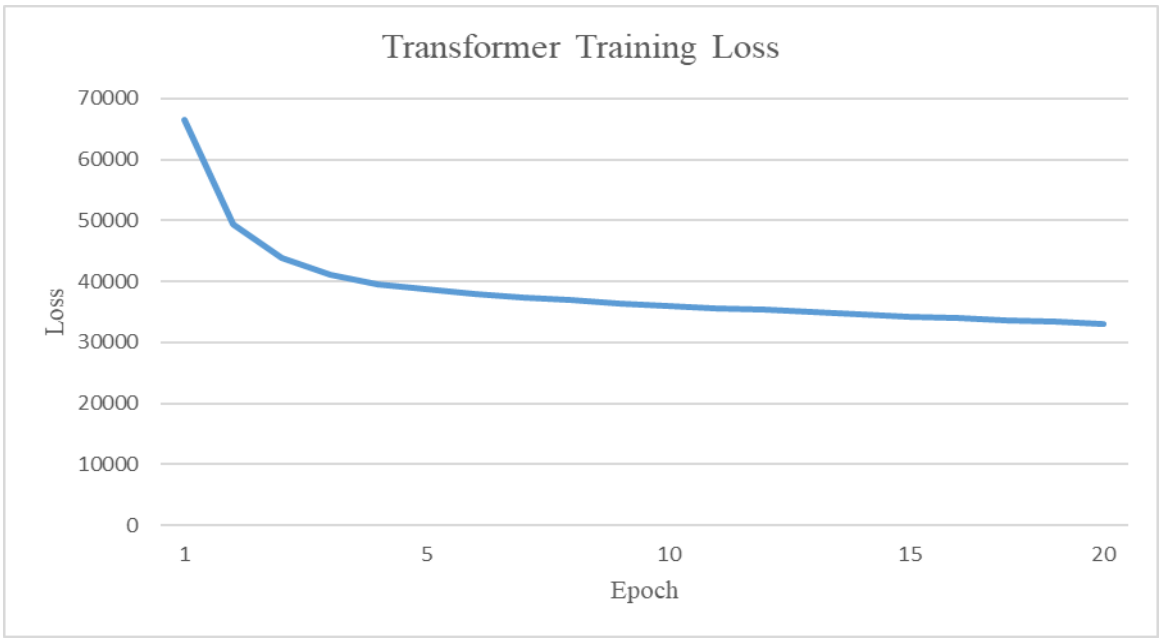


Figure 28: Epoch and training loss graph in Transformer.

With the results shown in table 3, the Attention model and the Transformer model achieve better performance than baseline models, with 13.02% for the Attention model and 21.05% for the Transformer model. The best performing model is Transformer, having a relatively short training time (2.2h).

Table 3: The experimental results in test dataset.

Model	Bleu score (%)	Training time (h)
RNN	0.46	0.0638
LSTM + Drop out	2.63	3.93
Attention	13.02	10
Transformer	**21.05**	2.2

Section 5. CONCLUSION

We achieved our target of building models for the this task of reaching over 10% on bleu evaluation metric. Attention model achieve 13.02%. Trasformer model achieve 21.05% and is the best model.

In future, we aim to use subword technique for solving this task.

References

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

[2] M.-T. Luong, H. Pham, and C. D. Manning, ‘Effective Approaches to Attention-based Neural Machine Translation’, *arXiv:1508.04025 [cs]*, Sep. 2015, Accessed: Jul. 09, 2021. [Online]. Available: http://arxiv.org/abs/1508.04025

[3] A. Vaswani *et al.*, ‘Attention is all you need’, in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[4] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, ‘Bleu: a method for automatic evaluation of machine translation’, in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.