Artificial Intelligence Final Report Assignment 問題 3 (Problem 3)
レポート解答用紙 (Report Answer Sheet)

Group Leader
学生証番号 (Student ID): 18521172
名前(Name): Đặng Văn Nhân

Group Members
学生証番号 (Student ID): 18521106
名前(Name): Nguyễn Lê Minh

学生証番号 (Student ID): 18521432
名前(Name): Võ Hồng Thiên

問題 3 (Problem 3)のレポート

Link our program and model: https://drive.google.com/drive/folders/1l0ez2xsxR6DkOV0-kF6H07VZQvVqdQwX?usp=sharing

Abstract: Our task is to write a program (PyTorch) that achieves higher accuracy (BLEU) on the IWSLT15(en-vi) data set. The program is an improved version of the program in the 13th lecture. We use transformer for this problem, as it is a well-known method for machine translation and many other tasks. Recent methods such as BERT, GPT-3, T5,... are all transformer-based. Through the implementation process, we achieved the highest result for the problem of 24.14%(BLEU). However, this outcome can still be improved with further development.

Dataset:

The IWSLT15(en-vi) dataset includes pairs of English and Vietnamese sentences. The dataset can be used for the machine translation task. Following are the fields included in the dataset:

Dataset	Sentences(en-vi)
Train	133317
Test	1268

Sample:

Here are some examples of datasets:

English Sentence	Vietnamese sentence
'Rachel Pike: The science behind a climate headline'	'Khoa học đằng sau một tiêu đề về khí hậu'
'Headlines that look like this when they have to do with climate change, and headlines that look like this when they have to do with air quality or smog.'	'Có những dòng trông như thế này khi bàn về biến đổi khí hậu, và như thế này khi nói về chất lượng không khí hay khói bụi.'
'They are both two branches of the same field of atmospheric science.'	'Cả hai đều là một nhánh của cùng một lĩnh vực trong ngành khoa học khí quyển.'

Disadvantages of Recurrent models:

Recurrent models typically take in a sequence in the order it is written and use that to output a sequence. Each element in the sequence is associated with its step in computation time t. These models 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.

The sequential nature of models does not allow for parallelization within training examples, which becomes critical at longer sequence lengths, as memory constraints limit batching

across examples. In other words, if we rely on sequences and we need to know the beginning of a text before being able to compute something about the ending of it, then we can not use parallel computing. The program would have to wait until the initial computations are complete. This is not good, because if the text is too long, then:

- It will take a long time to process it.
- We will lose a good amount of information mentioned earlier in the text approach the end.

Therefore, attention mechanisms have become critical for sequence modeling in various tasks, allowing modeling of dependencies without caring too much about their distance in the input or output sequences.

What is attention model?

According to lecture 14 by teacher Takashi Ninomiya: Attention solves long-distance problems by directly referencing the encoder's internal state at the output of the decoder.

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

Attention is an added layer that lets a model focus on what is important.

Queries, Values, and Keys are used for information retrieval inside the Attention layer.

The Attention finds matches even between languages with very different grammatical structures.

Transformer – Our model for translation task

With large sequences, the information tends to get lost within the network and vanishing gradients problems arise related to the length of input sequences. LSTMs and GRUs help a little with these problems. But even those architectures stop working well when they try to process very long sequences.

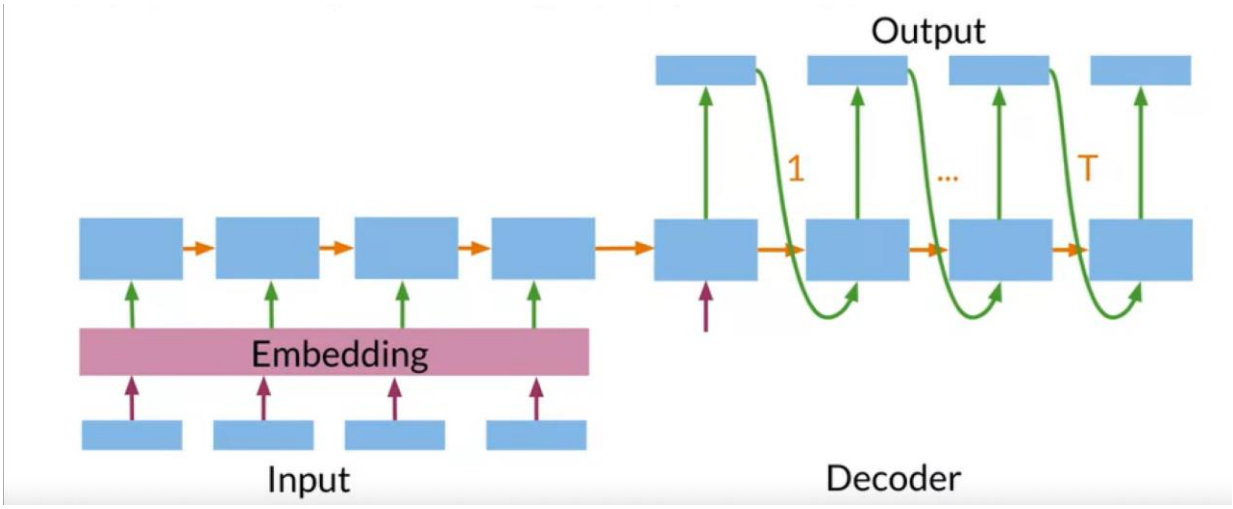

Figure 1. Machine translation model using RNN.

Transformer helps us to solve the problems that occur with RNN (Figure 1):

- Parallel computing is difficult to implement.
- For long sequences in RNNs there is loss of information.
- In RNN there is the problem of vanishing gradient.

Details of the RNN model were presented in lecture 12.

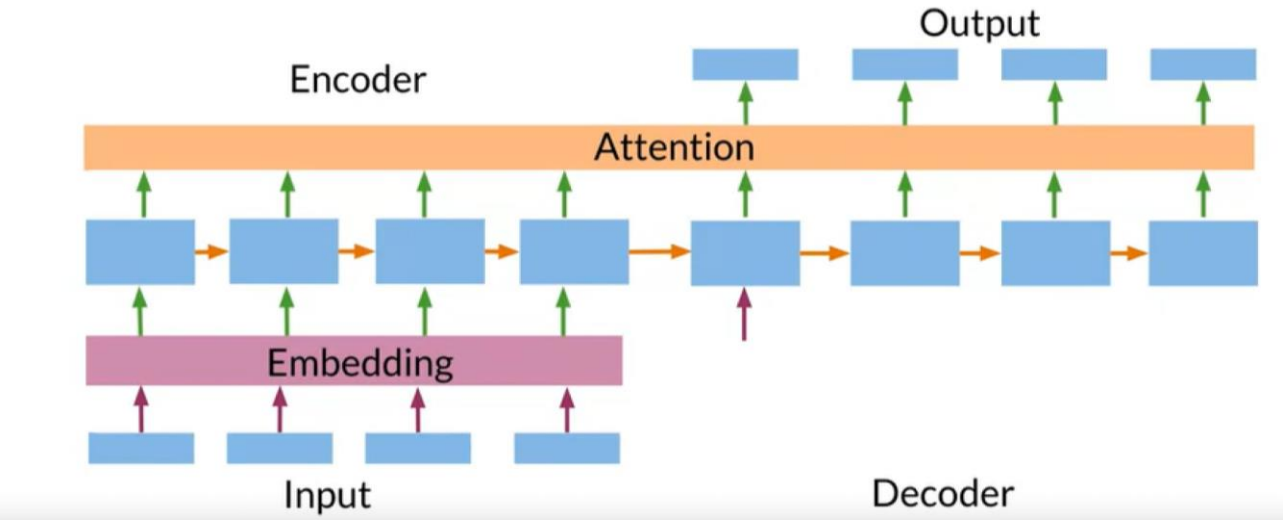

Figure 2. Machine translation using transformer.

According to lecture 14 by teacher:

Transformer is a new type of neural machine translation model that is neither RNN nor CNN and is the baseline model for current neural machine translation. A major feature is that translation is performed only by calculating attention. Transformer uses the following key technologies:

- Self Attention
- Positional Encoding
- Multi-head Attention

Details on how transformers work were taught by teacher Takashi Ninomiya in lesson 14.

Evaluate

We use the BLEU measure to evaluate the model's translation results at the request of the teacher. The closer the BLEU score is to one, the better model is. The closer to zero, the worse it is.

To get the BLEU score, the candidates and the references are usually based on an average of uni, bi, tri or even four-gram precision. We would sum over the unique n-gram counts in the candidate and divide by the total number of words in the candidate. The same concept could apply to unigrams, bigrams, etc. One issue with the BLEU score is that it does not take into account semantics, so it does not take into account the order of the n-grams in the sentence.

Program

```
import requests
import torch
import torch.nn.functional as F
import torchtext
import torch.nn as nn
from torch.nn import Transformer
import math
from torch import Tensor

url = "https://nlp.stanford.edu/projects/nmt/data/iwslt15.en-vi/"
train_en = [line.split() for line in requests.get(url + "train.en").text.spli
tlines()]
train_vi = [line.split() for line in requests.get(url + "train.vi").text.spli
tlines()]
test_en = [line.split() for line in requests.get(url + "tst2013.en").text.spl
itlines()]
test_vi = [line.split() for line in requests.get(url + "tst2013.vi").text.spl
itlines()]

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)

MODELNAME = 'iwslt15-en-vi-transformer.model'
EPOCH= 10
BATCHSIZE = 16
LR = 0.0001
SRC_VOCAB_SIZE = len(vocablist_en)
TGT_VOCAB_SIZE = len(vocablist_vi)
EMB_SIZE = 512
NHEAD = 8
FFN_HID_DIM = 512
NUM_ENCODER_LAYERS = 3
NUM_DECODER_LAYERS = 3
BOS_IDX = 2
EOS_IDX = 3
PAD_IDX = 1
DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'

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)

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))

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)

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, e
n, vi in test_data]

class PositionalEncoding(nn.Module):
    def __init__(self,
                 emb_size: int,
                 dropout: float,
                 maxlen: int = 5000):
        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_embed
ding.size(0), :])

# helper Module to convert tensor of input indices into corresponding tensor
of token embeddings
class TokenEmbedding(nn.Module):
    def __init__(self, vocab_size: int, emb_size):
        super(TokenEmbedding, self).__init__()
        self.embedding = nn.Embedding(vocab_size, emb_size)
        self.emb_size = emb_size

    def forward(self, tokens: Tensor):
        return self.embedding(tokens.long()) * math.sqrt(self.emb_size)

# Seq2Seq Network
class Seq2SeqTransformer(nn.Module):
    def __init__(self,
                 num_encoder_layers: int,
                 num_decoder_layers: int,
                 emb_size: int,
                 nhead: int,
                 src_vocab_size: int,
                 tgt_vocab_size: int,
                 dim_feedforward: int = 512,
```

```
                 dropout: float = 0.1):
        super(Seq2SeqTransformer, self).__init__()
        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)
        self.generator = nn.Linear(emb_size, tgt_vocab_size)
        self.src_tok_emb = TokenEmbedding(src_vocab_size, emb_size)
        self.tgt_tok_emb = TokenEmbedding(tgt_vocab_size, emb_size)
        self.positional_encoding = PositionalEncoding(
            emb_size, dropout=dropout)

    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_ke
y_padding_mask)
        return self.generator(outs)

    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)

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(tor
ch.bool)

    src_padding_mask = (src == PAD_IDX).transpose(0, 1)
    tgt_padding_mask = (tgt == PAD_IDX).transpose(0, 1)
    return src_mask, tgt_mask, src_padding_mask, tgt_padding_mask

loss_fn = torch.nn.CrossEntropyLoss(ignore_index=PAD_IDX)
```

```
def train():
    transformer = Seq2SeqTransformer(NUM_ENCODER_LAYERS, NUM_DECODER_LAYERS,
EMB_SIZE,
                                     NHEAD, SRC_VOCAB_SIZE, TGT_VOCAB_SIZE, FFN_H
ID_DIM).to(DEVICE)
    for p in transformer.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)
    optimizer = torch.optim.Adam(transformer.parameters(), lr=LR, betas=(0.9,
 0.98), eps=1e-9)
    for epoch in range(EPOCH):
        losses = 0
        step = 0
        for ben, bvi in train_data:
            ben = torch.tensor(ben, dtype=torch.int64).transpose(0,1).to(DEVI
CE)
            bvi = torch.tensor(bvi, dtype=torch.int64).transpose(0,1).to(DEVI
CE)
            bvi_input = bvi[:-1, :]
            src_mask, tgt_mask, src_padding_mask, tgt_padding_mask = create_m
ask(ben, bvi_input)
            logits = transformer(ben, bvi_input, src_mask, tgt_mask, src_padd
ing_mask, tgt_padding_mask, src_padding_mask)
            optimizer.zero_grad()
            bvi_out = bvi[1:, :]
            loss = loss_fn(logits.reshape(-1, logits.shape[-
1]), bvi_out.reshape(-1))
            loss.backward()
            optimizer.step()
            losses += loss.item()
            if step%100 == 0:
                print('step:', step, ': batchloss', loss.item())
            step += 1
        print('epoch', epoch + 1, ': loss', losses)
    torch.save(transformer.state_dict(), MODELNAME)
train()

def greedy_decode(model, src, src_mask, max_len, start_symbol):
    src = src.to(DEVICE)
    src_mask = src_mask.to(DEVICE)

    memory = model.encode(src, src_mask)
    ys = torch.ones(1, 1).fill_(start_symbol).type(torch.long).to(DEVICE)
    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 == EOS_IDX:
```

```
            break
    return ys

def test():
    total = 0
    correct = 0
    model = Seq2SeqTransformer(NUM_ENCODER_LAYERS, NUM_DECODER_LAYERS, EMB_SI
ZE,
                                    NHEAD, SRC_VOCAB_SIZE, TGT_VOCAB_SIZE, FFN_H
ID_DIM).to(DEVICE)
    model.load_state_dict(torch.load(MODELNAME))
    model.eval()
    ref = []
    pred = []
    for enprep, en, vi in test_data:
        lis = []
        input = torch.tensor([enprep], dtype=torch.int64).transpose(0, 1).to(
DEVICE)
        num_tokens = len(enprep)
        src_mask = (torch.zeros(num_tokens, num_tokens)).type(torch.bool)
        tgt_tokens = greedy_decode(
        model,  input, src_mask, max_len=num_tokens + 5, start_symbol=BOS_IDX
).flatten()
        p = list(tgt_tokens.cpu().numpy())
        for i in p:
            lis.append(list(vocabidx_vi.keys())[list(vocabidx_vi.values()).in
dex(i)])
        print('INPUT', en)
        print('REF', vi)
        print('MT', lis[1:-1])
        ref.append([vi])
        pred.append(lis)
    bleu = torchtext.data.metrics.bleu_score(pred, ref)
    print('total:', len(test_data))
    print('bleu:', bleu)

test()
```

Program explanation

For data loading and data preprocessing, we do the same as instructed in lecture 13 by Mr. Takashi Ninomiya.

In this program we build transformer model for machine translation task. Therefore, the RNN class in lecture 13 will be rewritten.

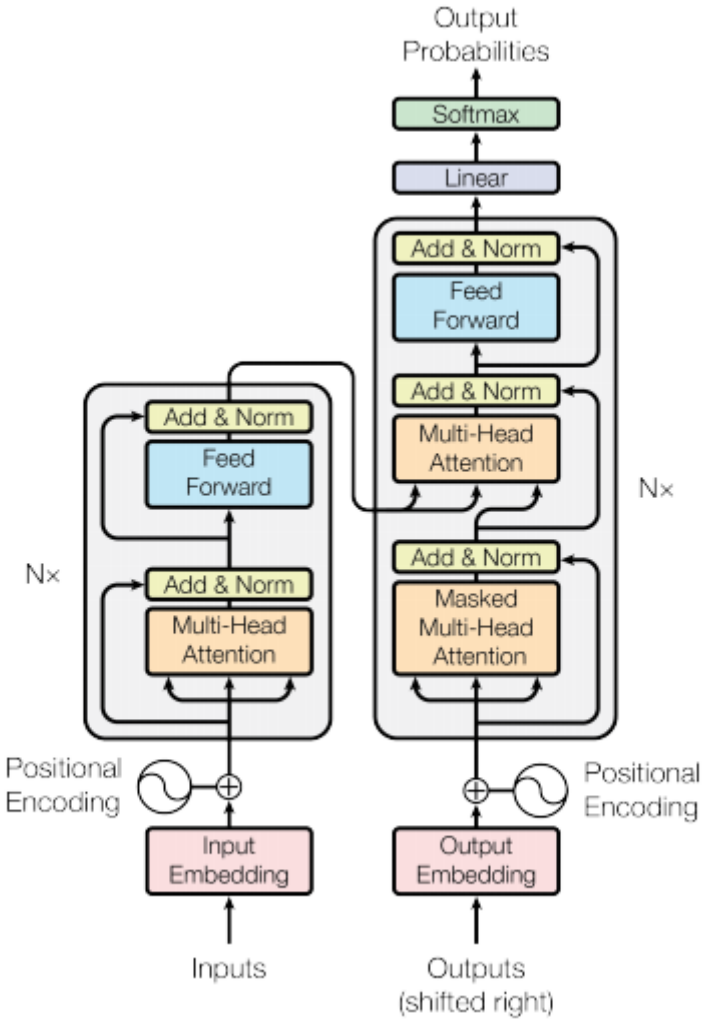

Figure 3 . The Transformer – model architecture

Looking at the Transformer model first introduced by the Google team in the article "Attention is all you need", we need to build a PositionalEncoding class.

Class PositionalEncoding adds positional encoding to the token embedding to introduce a notion of word order.

Next, we build **Class TokenEmbedding** to convert tensor of input indices into the corresponding tensor of token embeddings.

In **Class Seq2SeqTransformer**, we pass in the necessary arguments for training such as

num_encoder_layers, num_decoder_layers represent the number of encoder and decoder layers, nhead represents the number of heads Attention the model has, ... Transformer model is prewritten by Pytorch. We just need to import the model from the torch.nn module and use it.

During training, we need a subsequent word mask that will prevent model to look into the future words when making predictions. We will also need masks to hide source and target padding tokens. So, function **generate_square_subsequent_mask** and function **create_mask** will help us do that.

In the **train function**, similar lesson 13 we learned. First, call the previously built Seq2SeqTransformer class, and passing in the predefined arguments.

Then if the parameters in the transformer currently have dimensions greater than 1, we initialize them again: nn.init.xavier_uniform_(p).

Then, we train the model and use the **cross-entropy loss** function for the backpropagation task.

We use decoding after all the calculations have been performed on the encoder hidden states and when we are ready to predict the next token. Greedy decoding is the simplest way to decode the model's predictions as it selects the most probable word at every step.

Finally, we build the **test function** to translate the input sentence from the test_data. We convert the id of the English input sentence into tensor form, then through the

greedy_decode function, we get the corresponding tensor form of the Vietnamese sentence.

We use the command:

```
p = list(tgt_tokens.cpu().numpy())
    for i in p:
        lis.append(list(vocabidx_vi.keys())[list(vocabidx_vi.values()).index(i)])
```

to convert the tensor of the prediction sentence to Vietnamese to numpy, then get each word corresponding to the numbers from the previously processed vocabidx_vi dataset.

All predicted results, we add to the pred array: pred.append(lis).

The real results are added to the ref array: ref.append([vi]).

And we use the pre-written **BLEU measure** function to evaluate after the program translates the entire sentence from English to Vietnamese.

Result

Nhead	Emb_size	FFN_Hid_Dim	Epoch	Batchsize	BLEU
8	512	512	10	16	0.2317
8	512	512	15	16	0.2414
8	512	512	18	16	0.2387
8	512	512	10	20	0.2286
8	512	512	20	16	0.2378
8	256	512	15	16	0.2254

The highest result we achieved was 24.14%. This is the run that achieved the highest results, we also saved the models in the results table in the google drive link at the top of the report file. Teacher can link the model directly to the "load model command" in the test function to save running time. Details of each run, we save in the file answer sheet

```
epoch 1 : loss 36005.88170170784
epoch 2 : loss 25579.82274461165
epoch 3 : loss 22232.30920217
epoch 4 : loss 20342.22800121084
epoch 5 : loss 19119.14760239795
epoch 6 : loss 18242.384003633168
epoch 7 : loss 17582.066704203608
epoch 8 : loss 17089.501768143382
epoch 9 : loss 16684.8883768674
epoch 10 : loss 16341.545362715144
epoch 11 : loss 16054.266527586617
epoch 12 : loss 15775.618200219702
epoch 13 : loss 15528.635360068642
epoch 14 : loss 15301.558325021062
epoch 15 : loss 15074.107944233343

total: 1268
bleu: 0.24139326810836792
```

Conclude:

We have implemented and modified various parameters to improve the program. Regarding the batchsize, we found that most programs set the batchsize to 128. However, we cannot make the batch size larger than 24, as that would cause an error: Cuda is out of memory. Therefore, we use batchsize of 16 and 20. Experiments show that batchsize 16 gives higher results. Similarly, if you increase Nhead from 8 to 16, we will also get a Cuda memory error. Furthermore, we also consider Nhead=8, Emb_size=512, FFN_Hid_Dim=512 to be ideal sizes because when we reduce Emb_size from 512 to 256, the result is markedly reduced from 24.14% to 22, 54%. Another parameter that heavily influences the results is epoch. We vary the number of epochs to range from 10-20, and the results show that 15 is the most ideal number of epochs to achieve high BLEU accuracy.

Development:

Because of cuda memory limit, we can't try on common batchsize sizes like 32, 64, 128,... Similarly, for nhead count, emb_size, when increased, overflows memory. If the memory issue is fixed, we believe it can be tried on more parameters and the results can be higher.

To select words to translate we use Greedy decoding method. This method is the simplest because it selects the most probable word at each step. But the best word at each step may not be the best for longer sequences.

In addition to the method we use, there are some other methods that are said to be better such as random sampling, Beam search decoding, Grid Beam search decoding ,... If these methods are applied, we believe the accuracy body will increase. In fact, the authors of machine translation articles have applied the above methods to increase the results.

Another option is to use models developed from Transformer and modern models such as BERT, GPT-3, T5, BPE,... Some of these models were taught in lesson 15 by the teacher. However, we have not been able to apply it yet. If these new models are applied, we believe the results will be greatly improved.

Reference:

Google Team. (2017). *Attention is all you need.*