Natural Language Processing CA#5

Student Name:
Pouya Haji Mohammadi Gohari

SID:810102113

Date of deadline
Monday 10th June, 2024

Dept. of Computer Engineering

University of Tehran

Contents

1 Dataset

In this section, we are going to investigate through our translation dataset which can achieve from this link.To investigate data, we should first load it with utilizing following code:

```
drive.mount('/content/drive')
!wget -P /content/drive/My\ Drive/ https://object.pouta.csc.fi/OPUS-MIZAN/v1/moses
   ↪ /en-fa.txt.zip
!unzip "/content/drive/My Drive/en-fa.txt.zip"
```

First of all, we will allow 'colab' to access our google drive.Afterward, the dataset will be downloaded by wget command.With unzipping it, the data will be prepared to use.
Based on the two file created after unzipping(Persian and English text file) we will answer some questions as follow:
Q1:The number of lines for each file can be obtained with following command:

```
!wc -l MIZAN.en-fa.en
!wc -l MIZAN.en-fa.fa
```

These lines are shown in table 1:

File Name	Number of Lines
'MIZAN.en-fa.fa' file	1,021,597
'MIZNA.en-fa.en' file	1,021,597

Table 1: Number of Lines for Each File

In order to see what is three first lines of each, we utilize following command:

```
!head MIZAN.en-fa.en -n 3
!head MIZAN.en-fa.fa -n 3
```

These three lines have been illustrated in figure 1:

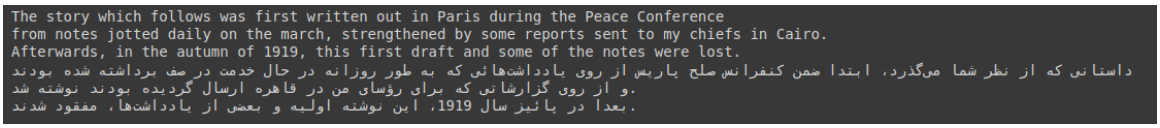

Figure 1: First Three Lines from Each File

Q2:In order to tokenized with white spaces, we have implemented code below:

```
def number_of_tokens_whitespaces(example:list[str]) -> int:
    """
    This function will simply tokenize each sentence with respect to white spaces
    Args:
        example: It sould be list of string as representation of a sentence
    Output:
        This function will return a inteeger since we wan to
    """
```

```
    example = example.replace('\u200c', '')
    example = example.replace('\u202C', '')
    token_counts = len(example.split())
    return token_counts
```

Since we are dealing with Persian texts in our dataset, we consider to delete some characters like ’202c’ and ’200c’ which stands for half spaces and right to left respectively. We will count the tokens in each line using length of the line where splitted function.
Finally with utilizing ’matplotlib’ we can plot histogram for number of tokens for each line.The code below will help us to do so:

```
def plot_histogram(tokens_list:list[int], language:str) -> None:
    """
    Creating a plot for each language.
    Args:
        tokens_list: list of token's number for each line
        language: Specify the language string
    Output:
        Creating a plot
    """
    plt.figure(figsize=(10,8))
    plt.hist(tokens_list, bins=30, alpha=0.75)
    plt.xlabel('Each line')
    plt.ylabel('Frequency of tokens')
    plt.title(f'Token Count Histogram for {language} Lines')
    plt.grid(True)
    plt.show()
```

From opening a file with specify path, we can calculate number of tokens of each line and finally plot histogram for each.The figure 2 and figure 3 are histogram for each file.

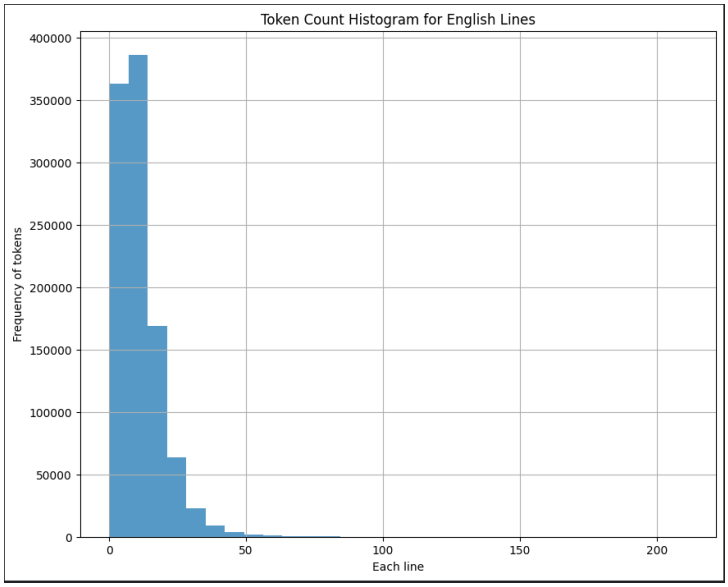

Figure 2: Token Count for English Lines

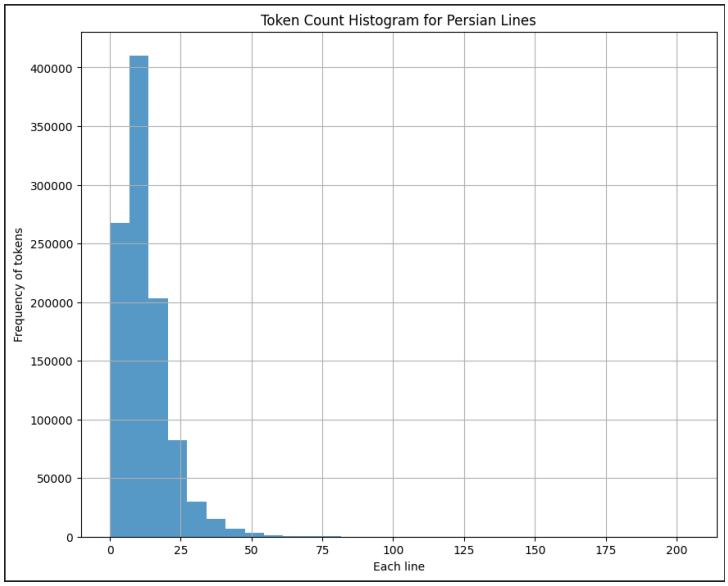

Figure 3: Token Count for Persian Lines

Q3:It would be very good idea to prune the outliers based on the target language and have less noisy number of tokens per line.So based on target language we can prune the lines having more than 50 tokens or less than 10. Based on the 'ignoring_outlier' function, we can create a dictionary for both target and source languages to save index alongside their number of tokens for each line.

```
def ignoring_outliers(src_dict:dict, trg_dict:dict, low_threshold:int,
    ↪ high_threshold:int) -> tuple:
    """
```

```
    This function will remove outliers with respect to the low and high threshold
       ↪ based on the trg_dict
      Args:
          src_dict: a dictionary of source contains {index:number_of_tokens}
          trg_dict: a dictionary of target just like src_dict
          low_threshold: if number of tokens are less than this threshold it would
             ↪  be pruned.
          high_threshold: if number of token are higher than this threshold it
             ↪ would be also condiser to be pruned.
      Output:
          This fucntion will return two dictionary in tuple data type.
    """
    revised_src_dict, revised_trg_dict = {}, {}
    indices = []
    for index, n_tokens in trg_dict.items():
        if n_tokens > high_threshold or n_tokens < low_threshold:
          continue
        else:
          revised_trg_dict[index] = n_tokens
          indices.append(index)
    for index in indices:
        revised_src_dict[index] = src_dict[index]

    return (revised_src_dict, revised_trg_dict)
```

Now from length of each dictionary, we can obtain filtered lines. Filtered lines can be chosen with respect to the corresponding index as follow:

```
with open('/content/MIZAN.en-fa.en', 'r', encoding='utf-8') as eng_file:
  eng_lines = [line for line in eng_file.readlines()]
with open('/content/MIZAN.en-fa.fa', 'r', encoding='utf-8') as per_file:
  per_lines = [line for line in per_file.readlines()]
filtered_eng_lines = [eng_lines[index] for index in revised_src_dict.keys()]
filtered_per_lines = [per_lines[index] for index in revised_trg_dict.keys()]
```

The filtered lines has been reported in table 2:

Language	Filtered Lines
Persian file	1,021,597
English file	1,021,597

Table 2: Number of Filtered Lines for Each File

Q4:In order to shuffle filtered dataset, we will zip filtered lines in order to pair them(since each English sentence translated into one Persian sentence where they are indeed parallel).Moreover, we will set random seed to 42 since for each run we obtain same shuffled data.The code below will clarify what we did:

```
import random
random.seed(42)
```

```
random.shuffle(paired_lines)
n_train_samples = 500000
n_validation_samples= 5000
n_test_samples = 10000
train_set = paired_lines[:n_train_samples]
val_set = paired_lines[n_train_samples :n_validation_samples+n_train_samples]
test_set = paired_lines[n_train_samples + n_validation_samples : n_train_samples +
    ↪  n_validation_samples + n_test_samples]
```

From implemented code above, we will assign 500,000 lines of first paired lines to the train set and next 5,000 sentences would be consider as validation set and finally the next 10,000 would be test set.
Q5:We will save these into 6 different file as follow:

- train.fa: Including Persian lines of train set.
- train.en: Including English lines of train set.
- valid.fa: Including Persian lines of validation set.
- valid.en: Including English lines of validation set.
- test.fa: Including Persian lines of test set.
- test.en: Including English lines of test set.

Following code will satisfy our purpose(We will save them in raw_data folder):

```
train_path_en = '/content/raw_data/train.en'
train_path_fa = '/content/raw_data/train.fa'
val_path_en = '/content/raw_data/valid.en'
val_path_fa = '/content/raw_data/valid.fa'
test_path_en ='/content/raw_data/test.en'
test_path_fa ='/content/raw_data/test.fa'
paths = [train_path_en, train_path_fa, val_path_en, val_path_fa, test_path_en,
    ↪ test_path_fa]
train_en, train_fa = zip(*train_set)
val_en, val_pa = zip(*val_set)
test_en, test_pa = zip(*test_set)
data = [train_en, train_fa, val_en, val_pa, test_en, test_pa]
for index, path in enumerate(paths):
  with open(path, 'w', encoding='utf-8') as myfile:
    for line in data[index]:
      myfile.write(line)
```

2 Part 1: Training BPE tokenizer and Pre-processing dataset

In this section we will train a BPE[1] tokenizer by utilizing the sentence piece library. One will be trained on Persian train set while another will be trained on English train set. We will set the vocab size to 10K as description said.The following code will show you the way:

```
train_en_file = '/content/raw_data/train.en'
train_fa_file = '/content/raw_data/train.fa'
model_prefix_en = '/content/tokenizer/english_tokenizer'
model_prefix_fa = '/content/tokenizer/persian_tokenizer'
vocab_size = 10000
model_type = 'bpe'
character_coverage = 1.0
spm.SentencePieceTrainer.train(
    input=train_en_file,
    model_prefix=model_prefix_en,
    vocab_size=vocab_size,
    model_type=model_type,
    character_coverage=character_coverage
)
spm.SentencePieceTrainer.train(
    input=train_fa_file,
    model_prefix=model_prefix_fa,
    vocab_size=vocab_size,
    model_type=model_type,
    character_coverage=character_coverage
)
```

First we will define the input file path for trainer of sentence piece and output file path as input and model prefix arguments.Then vocab size will be set where in our case is 10k. We want to train BPE tokenizer therefore we pass 'bpe' as model type. Character coverage is 1, passing higher value for this parameter(ranging 0 to 1) means that more unique characters from the training data will be included in the vocabulary which can be important for languages with many unique characters which we consider to set it to 1.
After training our BPE tokenizer for each train set(English and Persian), we consider to make a directory called 'tokenized' and encode each file for raw_data directory. Firstly, we will load two trained model of BPE tokenizer with following code:

```
eng_tokenizer = spm.SentencePieceProcessor(model_file='/content/tokenizer/
    ↪ english_tokenizer.model')
per_tokenizer = spm.SentencePieceProcessor(model_file='/content/tokenizer/
    ↪ persian_tokenizer.model')
```

Then utilizing the following code we can tokenize each file store it in specified path.

```
def tokenize_file(input_file:str, output_file:str, tokenizer:spm.
    ↪ SentencePieceProcessor):
  """
```

[1] Byte-Pair Encoding

```
    This function will tokenize each file based on your tokenizer.
        Args:
             input_file: Sepcify the path of your raw data
             output_file: Specify where you want to write the tokenized data
             tokenizer: pass your trained tokenizer
        Output:
             A file will be created in specify location where each line is
                ↪ tokenized format of raw data
  """
  with open(input_file, 'r', encoding='utf-8') as raw_file:
    lines = raw_file.readlines()
  cleaned_lines = [line.replace('\u200c', '') for line in lines]
  super_cleaned_lines = [line.replace('\u202C', '') for line in cleaned_lines]
  with open(output_file, 'w', encoding='utf-8') as tokenized_file:
    for line in tqdm(super_cleaned_lines):
      tokenized_line = tokenizer.encode(line.strip(), out_type=str)
      tokenized_file.write(' '.join(tokenized_line) + '\n')
```

As we said in previous section we will replace two characters representing half spaces and rtl with space.Finally utilizing code above, we can tokenize each raw file into tokenized directory.

```
tokenize_file(train_path_en, train_path_en_tokenized, eng_tokenizer)
tokenize_file(val_path_en, val_path_en_tokenized, eng_tokenizer)
tokenize_file(test_path_en, test_path_en_tokenized, eng_tokenizer)
tokenize_file(train_path_fa, train_path_fa_tokenized, per_tokenizer)
tokenize_file(val_path_fa, val_path_fa_tokenized, per_tokenizer)
tokenize_file(test_path_fa, test_path_fa_tokenized, per_tokenizer)
```

Now we can proceed to use fair seq library in order to pre-process the tokenized data.Let's first use the following command and explain each arguments later.

```
!fairseq-preprocess --source-lang en --target-lang fa \
                    --trainpref ./tokenized/train \
                    --validpref ./tokenized/valid \
                    --testpref  ./tokenized/test \
                    --destdir   ./data_bin/ \
                    --nwordssrc 10000\
                    --nwordstgt 10000
```

First line will specify the source language that we are translating from with **–source-lang** where it is en(English) and the target language that we are tanslating to with **–target-lang** where it is fa(Persian).After that we will specify the train, validation and test path for **–trainpref, –validpref, –testpref** respectively.**–destdir** will get a path where this command will create some files for us which are useful for training phase.At the end **–nwordssrc, –nwordstgt** will be set to 10k as description said so.The usage of these two commands are as follow:

- nwordsrc: Setting this parameter to 10k, means the source language vocabulary will be limited to 10,000 mos t frequent tokens or words.Moreover, this parameter would mainly used to specify how many words will be retained for source language.
- nwordtrg: Like nwordsrc, this will be used to specfiy how many words will be retained for target language.

The usage and definition of these two parameters are according to documentation of fairseq. Now lets answer what files these preprocess command line will create for us.This function will create three main files as follow:

1. Dictionary Files: These files contain the vocabulary for source and target languages where each line of these files represents as token and their corresponding frequency.

2. Binary Files: For either source or target with respect to train, validation and test set will create multiple binary files.These files consist of pre-processed data in binary format.The binary format is efficient for input and output operation during training.

3. Index Files: These index files provide a mapping that correspond to their binary files where helps to access the data in efficient way.

Moreover, binary files include tokenized sentences from their corresponding files(for example train.fa and train.en). These tokens will be represented as their tokens ids based on the vocabulary file.(dict.en.text and dict.fa.text) Index files includes offsets that points to start of the sentences corresponding to their binary files.
The overall files created by this function has been illustrated in figure 4:

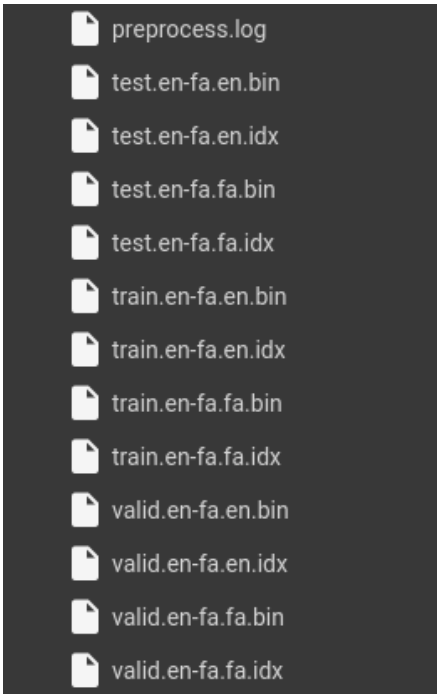

Figure 4: Total Files Created by fairseq-preprocess

3 Part 2: Training LSTM ENCODER-DECODER

After pre-process our dataset in previous section, it is time to proceed into training a encoder-decoder model based on LSTM.Let us introduce you to our implemented code then explain why we choose some parameters:

```
!fairseq-train \
    "./data_bin/" \
    --arch lstm --share-decoder-input-output-embed \
    --encoder-layers 6 --decoder-layers 6 \
    --optimizer adam --adam-betas '(0.9, 0.98)' --clip-norm 0.0 \
    --lr 2e-3 --lr-scheduler inverse_sqrt --warmup-updates 4000 --warmup-init-lr 5
       ↪ e-08 \
    --dropout 0.3 --weight-decay 0.0001 \
    --criterion label_smoothed_cross_entropy --label-smoothing 0.2 \
    --max-tokens 4096 \
    --max-sentences 128 \
    --update-freq 4\
    --num-workers 2\
    --eval-bleu \
    --eval-bleu-args '{"beam": 5, "max_len_a": 1.2, "max_len_b": 10}' \
    --eval-bleu-detok moses \
    --eval-bleu-print-samples \
    --best-checkpoint-metric bleu --maximize-best-checkpoint-metric \
    --fp16 --memory-efficient-fp16 \
    --max-epoch 5 \
    --save-dir ./data_bin/checkpoints/ \
    --tensorboard-logdir ./data_bin/logs
```

As description said we have chose some parameters as follow:

1. Number of Epochs: The total epoch we consider model to be trained is 5 as description said.
2. Criterion: label_smoothed_cross_entropy with label_smoothing of 0.2.
3. Optimizer: The adam was set for optimizer with parameter beta set to 0.9 and 0.98.
4. Layers: Decoder and Encoder layers must be set to 6 as said.

As said we have chosen same hyper parameters for each model we are willing to train.Thus after training each model with same parameters, we decided to choose learning rate of $2e-3$ with initial learning rate of $5e-8$ and 4000 for warm up steps.(This parameters was set in way that both model will be effective in their own way.Moreover, we have tried so many parameters for each model and obtain the best parameters not just for model based on LSTM but also best for model that is based on Transformers)
We will use blue score for validation set to see the performance with beam width of 5.Each epoch will saved under the 'data_bin/checkpoints/' and logs would be saved under the 'data_bin/checkpoints/' in order to use to future purposes with tensor board.The best checkpoint will be saved under the same directory of checkpoints.
Some other hyper paramterers and their explaination:

1. max tokens: This parameter sets a limit for maximum number of tokens in each batch.This parameter will help the GPU memory usage.

2. max sentences: According to the fairseq documentation the batch size and max sentences are equal parameters since they represent how much example(sentences) must be in a batch.Furthermore, this parameter will set a maximum limit for how many sentences should be in a batch.This will help us when we dealing a dataset where the variation of sequence length is high.

3. update freq: This parameter is as same as gradient accumulation.For example in our scenario it has been set to 4 meaning that loss of 4 batches will be accumulated then weights will be updated.

4. number of workers: This parameter sets number of worker threads for data loading.More workers will speed up data loading by parallelizing the process.When dealing with large dataset this can help us to load data more efficient.

Moreover, max tokens and max sentences are two maximum limit of number of tokens and number of sentences can be seen with in a batch.If a batch reach max sentences first the batch will be processed even with lower max tokens and vice versa.
In order to answer the provided question, we have tried so many values for these two parameters and even with higher values for them, the loss' scale would be decreased so much leading to change the order for loss of training.Thus we should have a trade off between the speed of training and order of how loss decreasing.Even though the higher values for these parameters are possible and could be benefit us to dealing with large dataset but they must choose wisely where they to do not effect the training phase.In summary, in order to control the dataset's volume we can indeed use these two parameters where can help the GPU memory usage with respect to limiting the number of tokens and number of sentences within a batch.
Since we could not dowmnload the loss from log for training and validation set, we will use following function to do so:(These csv files will be uploaded alongside with code and report)

```
import os
import pandas as pd
from tensorboard.backend.event_processing import event_accumulator

log_dir = './data_bin/logs/train'

ea = event_accumulator.EventAccumulator(log_dir)
ea.Reload()
print("Available scalar keys:")
print(ea.Tags()['scalars'])


scalars = ea.Scalars('loss')
df = pd.DataFrame([(s.step, s.value) for s in scalars], columns=['step', 'loss'])
df.to_csv('lstm_training_loss.csv', index=False)

log_dir = './data_bin/logs/valid'

ea = event_accumulator.EventAccumulator(log_dir)
ea.Reload()
print("Available scalar keys:")
print(ea.Tags()['scalars'])

```

```
scalars = ea.Scalars('loss')
df = pd.DataFrame([(s.step, s.value) for s in scalars], columns=['step', 'loss'])
df.to_csv('lstm_validation_loss.csv', index=False)

%load_ext tensorboard
%tensorboard --logdir ./data_bin/logs --port 12345
```

Note that we have downloaded these files from 'colab'.The loss for training alongside with validation loss has been illustrated in figure 5:

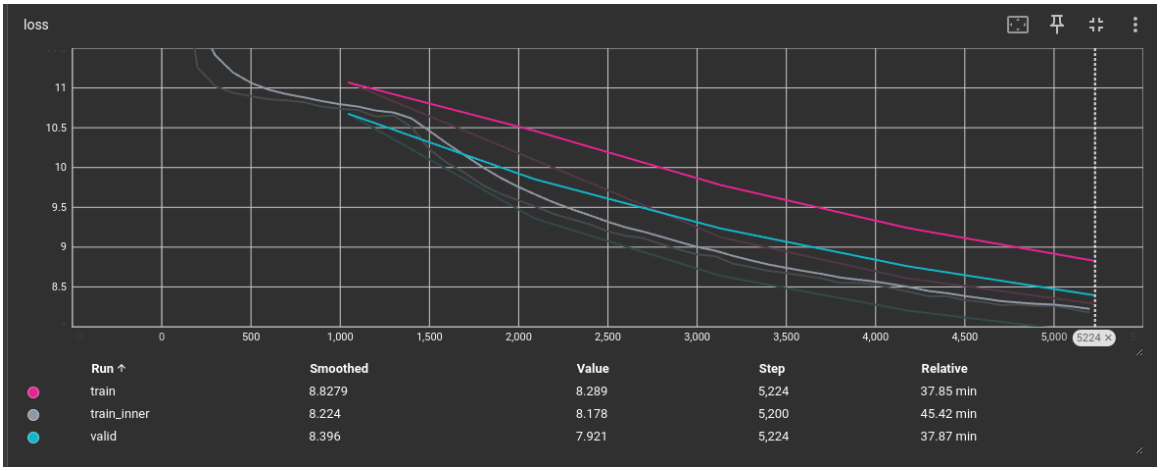

Figure 5: Training and Validation Loss

According to the loss for train set and validation set, we could have more epochs to train it more but as description 5 epochs would be enough.

4 Part 3: Training Transformer ENCODER-DECODER

We will avoid the explanation of following code for training encoder-decoder model based on the transfomer since whole parameters are same as previous section.The implemented code is as follow:

```
!fairseq-train \
    "./data_bin/" \
    --arch transformer --share-decoder-input-output-embed \
    --encoder-layers 6 --decoder-layers 6 \
    --optimizer adam --adam-betas '(0.9, 0.98)' --clip-norm 0.0 \
    --lr 2e-3 --lr-scheduler inverse_sqrt --warmup-updates 4000 --warmup-init-lr 5
       ↪ e-08 \
    --dropout 0.3 --weight-decay 0.0001 \
    --criterion label_smoothed_cross_entropy --label-smoothing 0.2 \
    --max-tokens 4096 \
    --max-sentences 128 \
    --update-freq 4\
    --num-workers 2\
    --eval-bleu \
    --eval-bleu-args '{"beam": 5, "max_len_a": 1.2, "max_len_b": 10}' \
    --eval-bleu-detok moses \
    --eval-bleu-print-samples \
    --best-checkpoint-metric bleu --maximize-best-checkpoint-metric \
    --fp16 --memory-efficient-fp16 \
    --max-epoch 5 \
    --save-dir ./data_bin/checkpoints2/ \
    --tensorboard-logdir ./data_bin/logs2
```

We will consider checkpoint2 and logs2 for this model.The loss for training and validation are depicted in figure 6:

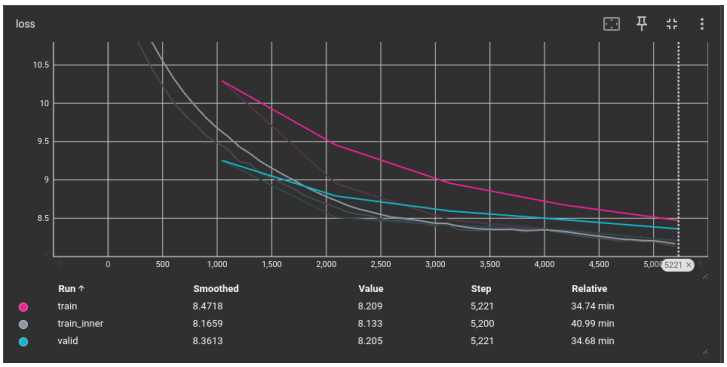

Figure 6: Transformer loss for validation and training set

The loss for validation and training has been uploaded alongside with code and report.As depicted in figure 6, but we did not yield good performance with BLEU score where will be discussed in the later sections.(Evaluation section)

5 Evaluation Metric and Examine Test dataset

In final section, we will dive into examine the performance on two score called BLEU and COMBAT scores.Where these metric's definition are as follow:

1. BLEU score: This is a metric for evaluating quality of a machine translating a language to another.More precisely, BLEU calculates the n-gram precision, between the candidate and the reference translation.These n-grams are commonly unigrams, bigrams, trigrams, 4-grams.This score equation is as follow:

$$BLEU = BP * \exp \sum_{i=1}^{n} w_n \log p_n$$

   where p_n is the modified precision for n-grams of length n, w_n is the weight for n-grams of length n and N is the maximum length of n-grams which considered.The BP is brevity prenalty(These are according to the slide of NLP course)

2. COMBAT: This metric is based on neural network model to evaluate the quality of machine translation.Moreover, this metric will use a pre-trained multilingual encoder to predict the quality of a translation by comparing it with reference translation and the source.COMET takes triple input consist of the source sentence, the reference translation and the candidate translation.These models has been trained on the human judgement quality estimation datasets.In the end for each triplet input these scores can be averaged to produce an overall evaluation score for current model we have been training.

There is some strength in COMBAT metric since it can evaluate the model based on semantic understanding where the BLEU will not capture them.BLUE score can be indeed insensitive to word order where the COMBAT not also capture semantic similarities but also aligns more closely to human judgement(since they are trained based on human judgements!)
The best BLEU score has been reported in table 3:

Model	Best BLEU
LSTM encoder-decoder	4.76
Transformer encoder-decoder	1.34

Table 3: Best BLEU score for two trained models

These BLEU score for validation per epoch for these two model has been illustrated in figure 7 and figure 8 respectively for LSTM and Transformer:

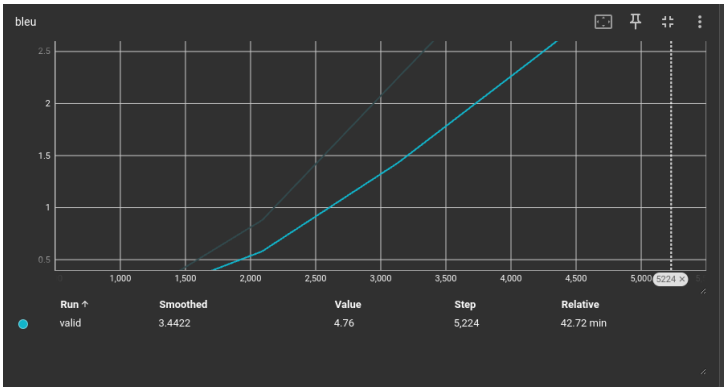

Figure 7: LSTM BLEU score per epoch of validation

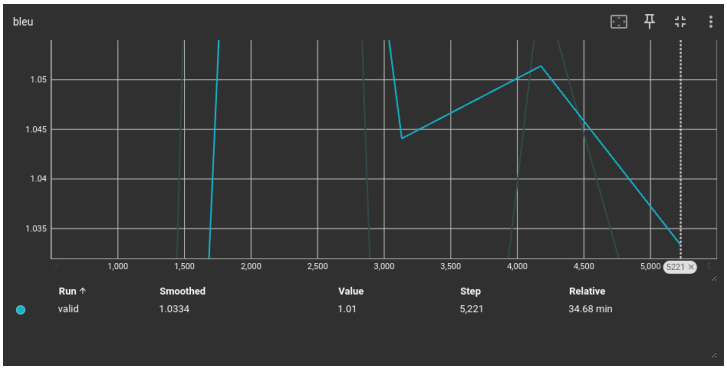

Figure 8: Transformer BLEU score per epoch of validation

As you can see for lstm BLEU score is increasing where the BLEU score for transformer is not.(Maybe it is due that the same BLEU score function for these model is not good metric while we might get better score with some other metric such as COMBAT).

Attention: We have downloaded the loss for train and validation for both model with following code:

```
log_dir = './data_bin/logs2/train'

ea = event_accumulator.EventAccumulator(log_dir)
ea.Reload()
print("Available scalar keys:")
print(ea.Tags()['scalars'])

scalars = ea.Scalars('loss')
df = pd.DataFrame([(s.step, s.value) for s in scalars], columns=['step', 'loss'])
df.to_csv('transformer_training_loss.csv', index=False)

log_dir = './data_bin/logs2/valid'

ea = event_accumulator.EventAccumulator(log_dir)
ea.Reload()
```

```
print("Available scalar keys:")
print(ea.Tags()['scalars'])

scalars = ea.Scalars('loss')
df = pd.DataFrame([(s.step, s.value) for s in scalars], columns=['step', 'loss'])
df.to_csv('transformer_validation_loss.csv', index=False)
```

After the CSV files has been created, we will download it via 'colab' env. We will generate the the hypothesis for test set via following command:

```
!fairseq-generate data_bin \
    --path data_bin/checkpoints/checkpoint_best.pt \
    --batch-size 64 --beam 5 \
    --results-path ./LSTM_result
```

This command simply use a batch size of 64 and beam width of 5 based on the best checkpoints the model has. The last line of generated text consist BLEU score where this score for each model is reported in table 4:

Model	BLEU score for test set
LSTM encoder-decoder	4.15
Transformer encoder-decoder	1.28

Table 4: Best BLEU score for two trained models

Again the idea of consider which model is better can not be obtained from this score since it is based on the n-grams precision.A better score where defined earlier in this section must be considered.This COMBAT score is indeed very powerful metric since, it aligns more closely with human judgements.Before jumping to the COMBAT score, the following implemented code will help us to get the BLUE score from the generated text by fairseq generate command.

```
def get_blue_score(file_path:str) -> float:
  """
    This method will get the the file path for the output of fairseq-generate
       ↪ function and return the blue screen
    output is blue score and its data type is float.
    input is string contains folder path.
  """
  with open(file_path, 'r', encoding='utf-8') as read_file:
    lines = read_file.readlines()
  tokens = lines[-1].strip().split()
  index = tokens.index('BLEU4')
  return float(tokens[index+2].replace(',',''))
```

After this we must prepare these generated text via previous command for the combat models.These triplet inputs must be specified:

1. Source: In generated text, source will be marked as 'S-' for each line.
2. Target: The target sentence which is a reference sentence of translated source is marked as 'T-' at the first of each line.

3. Hypothesis or machine-translated target: The translated sentence by the machine translated are marked 'H-' at the beginning of each line.

Now with the following code we can have dictionary that store each of them:

```
def create_dictionary(file_path:str) -> dict:
  """
    Simply get a path file and create a dictionary based on sources and targets
       ↪ and hypothesis exist in that file.
    Output: dictionary contain three key: hypothesis, sources and targets.
    Input: path for a file.
  """
  with open(file_path, 'r', encoding='utf-8') as read_file:
    lines = read_file.readlines()
  output_dict = {
      'hypothesis': [],
      'sources': [],
      'targets': []
  }
  for line in lines:
    tokens = line.strip().split()
    if tokens[0][0:2] == 'S-':
      output_dict['sources'].append(' '.join(tokens[1:]))
    elif tokens[0][0:2] == 'T-':
      output_dict['targets'].append(' '.join(tokens[1:]))
    elif tokens[0][0:2] == 'H-':
      output_dict['hypothesis'].append(' '.join(tokens[2:]))
    else:
      continue
  return output_dict
```

After saving each line of sources, targets and hypothesis sentences in a dictionary data type, we can decode them with pre-trained bpe model by sentence piece library where was implemented in section Part 1. As we translating English to Persian our hypothesis and target must be decoded by the Persian pre-trained model while for the source we must decode them with English pre-trained model.The following code will illustrate the way:

```
def decode_each(hypo_trg_src:dict) -> dict:
  """
    This function will decode each with our bpe.
  """
  decode_dict = {key:[] for key in hypo_trg_src.keys()}
  for hypo in hypo_trg_src['hypothesis']:
    tokenized_hypo = hypo.strip().split()
    decode_dict['hypothesis'].append(per_tokenizer.decode(tokenized_hypo, out_type
       ↪ =str))
  for src in hypo_trg_src['sources']:
    tokenized_src = src.strip().split()
    decode_dict['sources'].append(eng_tokenizer.decode(tokenized_src, out_type=str
       ↪ ))
```

```
  for trg in hypo_trg_src['targets']:
    tokenized_trg = trg.strip().split()
    decode_dict['targets'].append(per_tokenizer.decode(tokenized_trg, out_type=str
       ↪ ))
  return decode_dict
```

Now we have a dictionary where each key in it has decoded line!Now based on implemented code for the Unbabel-Comet library we find out we must give the input in the format of a list contains dictionaries.Thus we consider to do this:

```
data = []
for src, ref, mt in zip(decoded_lstm_output['sources'], decoded_lstm_output['
   ↪ targets'], decoded_lstm_output['hypothesis']):
    data.append({
        "src": src,
        "ref": ref,
        "mt": mt
    })
```

Now our data has been prepared for combat models to predict the scores for us.Based on the frequently asked questions part in the COMET documentation link we find out for general purpose MT evaluation they recommend us to use 'Unbabel/wmt22-comet-da'. Now utilizing the following code we will load this model in order to predict the scores for us:

```
from comet import download_model, load_from_checkpoint
model_path = download_model("Unbabel/wmt22-comet-da")
model = load_from_checkpoint(model_path)
output = model.predict(data, batch_size=8, gpus=1)
```

Now lets proceed into obtain the combat score for each model where has been reported in table 5:

Model	combat score
LSTM encoder-decoder	0.52
Transformer encoder-decoder	0.43

Table 5: COMBAT score for two trained models

As you can see in the table5, the combat score is very good for these two models but the combat score for LSTM is higher than transformer which can indicate that the encoder-decoder based on RNN(LSTM, GRU) can some times outperform the Transformer models.In the class, when Dr.Faili was teaching us machine translation, I remembered that he said, when the parallel data volume is low, encoder-decoder model based on RNN can outperform the transformers.Note that we are telling this based on just 5 epochs and for more investigating which model has better performance we must try both model trained same amount of epoch.Then evaluate them again.
Based on the two last table 5 and table 4 both metrics ranging 0 to 1.(Note that the BLEU score is the percentage). Both of these metric agree that the LSTM encoder-decoder performs better than the transformer encoder-decoder on the test set.
The BLUE score as said before focused on ngram overlap and might not fully capture semantic and contextual accuracy.

The COMET score provides a better alignment with human judgement and confirms that the LSTM model will generate higher quality translation.