EEE-586 Term Project Report: Headline Generation from Distinctive Summaries

Burak Künkçü
21501843

Furkan Şahinuç
21400503

Selim Furkan Tekin
21501391

***Abstract*—In this paper, we demonstrate a LSTM based encoder-decoder network for the headline generation task. State-of-the-art BERT language model is used extract context vectors. Attention mechanism is used together with encoder-decoder network to capture relevancy information in input sequence and improve overall system performance. Several summaries are generated and according to their TF-IDF score, the best summary is used in headline generation instead of evaluating network on the whole content. Generated headlines are compared according to the BLEU metric.**

I. Introduction

Seq2Seq encoder-decoder networks achieved state of the art performances for tasks involving transformation of one sequence into another such as translation tasks from a language to another etc. Such models often use recurrent network cells such as RNNs and LSTMs to construct encoder-decoder architectures which generates hidden vector representations and decomposes output sequences word-by-word in sequential time steps. In addition, it is possible to optimize such networks by adding attention mechanism which became highly popular in neural network training recently. Attention mechanism allows the network to look the input sequence selectively and learn alignments between input and output sequences.

In this term project, we have studied headline generation from distinctive summaries of articles. We have used LSTM recurrent cells to implement our encoder-decoder network that is used for sequence-to-sequence transformation. As a very special type of text summarization, headline generation is still a challenging task. It can be observed that, even though the nature of this task involves plenty of training data (lots of articles being published every day), only a very specific part of the published articles is essential for composing the article headline. Therefore, in order to generate a consistent and explanatory headline for an article, it is required to perform a successful selection of highlights in the context. In our work, we employ attention mechanism in order to selectively search the text for such highlights. We have used BERT, a state of the art language model, in order to generate contextualized word embeddings. Thus we can effectively exploit the context concept which is essential for the headline generation task. Also, instead of directly generating headlines from a whole article, first we generate several summaries of the article using pre-trained summarization methods. Then, we adaptively select the best summary for the article and use it as the input sequence for our headline generation model. In order to choose best summary, we use a TF-IDF based method. We use BLEU, a frequently used evaluation metric used in machine translation, in order to compare our generated headlines with original labeled headlines.

We implement our model using *pytorch* framework. We train and evaluate our model on the all-the-news dataset (available at: `https://www.kaggle.com/snapcrack/all-the-news`)

II. Related Work

There exist numerous studies on headline generation. These can be classified under three main approaches rule-based, statistics-based, and summarization-based. In this work we focus on summarization approach and think headlines as short summaries of texts. There are two main approaches to generating summary of a text, extraction-based and abstraction-based. Extraction based summaries are refined sentences formed by the words in text while abstraction based methods aims to generate summaries using paraphrasing and shortening.

One of the key example study of abstraction-based approach is [1] where they used neural encoder-decoder architecture. They used attention mechanism between first sentence of the article and given headlines. After the power of attention mechanism is shown, [2] implemented some side properties of language, such as abstract meaning, as an extension to attention based model. They used abstract meaning representation (AMR) parser to modify attention based model and improved the headline generation benchmarks compared with the baseline neural attention based model. Another extension to [1] made by [3]. They proposed conditional recurrent neural network (RNN) and designed a convolutional attention based encoder. Encoder computes scores over the words of input sentence considering position in the sentence and then these scores informs the decoder which part of the input sentence it should focus on and generate the next word. However, in [4] showed that attention based encoder-decoder approaches can be problematic due to unrelated filling of details when detail is missing in news. Another assumption that these [1], [2], [3], [4] works are making is, first sentences are the most informative. [5] showed that important information is distributed across several sentences and proposed multi sentence comparison model. A recent approach [6] proposed a model architecture where it takes multiple summaries as input to encoder and pass output states through control layer which modifies attention equation and feed resulting context

vector to decoder. These summaries can be extraction or abstraction based. In the control model they used Long-Short Term Memory (LSTM) units and suggest that this provides an hierarchy between summaries. However they did not mention LSTM functionality in detail.

III. Problem Definition

We can represent our dataset as $\mathcal{D} = \{\mathbf{X}_i, \mathbf{Y}_i\}_{i=1}^{N}$ where $|N|$ is the total number of news in our dataset. Let $\mathbf{X}_i = \{\mathbf{x}_j\}_{j=1}^{T}$ denote $|T|$ length of input text. Let $\mathbf{Y}_i = \{\mathbf{y}_k\}_{k=1}^{T'}$ represent the title of the news and $T < T'$. Where each $\mathbf{x}_j$ and $\mathbf{y}_k$ are one hot vectors of length $|V|$ where V is the vocabulary and $0 \leq j \leq T$, $0 \leq k \leq T'$. First we create candidate summaries $\{\mathbf{S}_1, \mathbf{S}_2, ...\mathbf{S}_K\}$ where $|K|$ is the number of candidate summaries. For each summary $\mathbf{S} = \{\mathbf{s}_l\}_{l=1}^{L}$ where $|L|$ is the length of the summary text there are synthesized words $\mathbf{s}_l \in V$. Our goal is to learn,

$$P(\mathbf{Y}|\mathbf{X}) = P(\mathbf{Y}|\{\mathbf{S}_1, \mathbf{S}_2, ...\mathbf{S}_K\}) \tag{1}$$

by maximising,

$$\hat{\mathbf{Y}} = \text{argmax}_{\mathbf{Y}}\, P(\mathbf{Y}|\{\mathbf{S}_1, \mathbf{S}_2, ...\mathbf{S}_K\}) \tag{2}$$

where $\hat{\mathbf{Y}}$ represents predicted headline.

IV. Proposed Methodology

The methodology we followed in the main framework consists of encoder - decoder architecture with attention mechanism. As it is well known, such kind of architectures are special form of recurrent neural networks. Before examining how we implement this model, it would be better to clarify how the data is utilized and prepared for training.

A. Data & Pre-processing

In our project, the name of the used dataset is "All The News". It can be obtained from https://www.kaggle.com/snapcrack/all-the-news. This dataset contains news from several subjects such as politics, sport, music etc. There are approximately 140 thousands news and their respective titles. 10 % of the data was allocated as validation set, 10 % of the data was allocated as validation set and the remaining 80 % was used for training purposes.

First of all the dataset was subjected to a pre-processing. In pre-processing, all noisy characters and punctuation marks were removed. All letters are made lowercase and stop words are expelled from the dataset. Additionally, maximum word length is limited to 20 characters and <start> and <end> tokens were added in order to where the news start or finish.

Before summarization process it would be beneficial to mention some features of the news articles and titles. There are 142471 articles in our dataset. Every article has its unique title. However, 15 of the has no titles. This situation may stem from some errors while colleting the data. Since the number of articles without titles is very low compared to total size of dataset. The effect of removal of erroneous data is negligible. Besides these general statistics, we also calculated average title and article length. Furthermore, it is important to know how many titles are extracted from the article sentences as a whole or paraphrased. How many words are common among titles and articles. These statistics can be observed in Table I.

TABLE I: Dataset Statistics

Average title length (in words)	10.2
Average article length (in words)	732.4
Average number of words in titles contained by article (per-cent)	73.2
Number of articles contains whole title	633

Here, we see that the number of titles which exactly come from articles are relatively low compared to total size of dataset. This situation shows that article titles use paraphrasing for important sentences. That's why learning the contexts of words while generating titles have significance.

After pre-processing, next operation is that summarizing the news. As it can be understood from the title of the project, our main objective is to generate and predict headlines from the summaries of the news. The reason why we used summaries instead of raw news is that feeding the network only by raw text would be too costly in terms of computation. Furthermore, whole news articles contain not only important parts of the news but also relatively redundant words too. Therefore, generating articles from the significant parts of the news would lead to obtain more informative titles. Nevertheless, "All The News" dataset does not contain summaries of news. For this reason, we create summaries for each news. While summarizing the news we utilized "LexRankSummarizer", "LsaSummarizer", "SumBasicSummarizer" libraries. Nonetheless, we need to choose the best reflecting summary among these three candidate. In order to choose best summary, we used a TF-IDF based method. As it can be remembered from the lectures. TF-IDF is a method that indicates a word's importance and specificity for document where it belongs to. Term frequency of a word respective to a document is calculated as number of the word in that document divided by total word count in all documents.

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_k} \tag{3}$$

In equation 3, $n_{i,j}$ denotes the number of occurences word i in document j. Number in the denominator denotes the total number of words in all documents. Inverse document frequency of a word is formulated as equation 4.

$$idf_i = \log \frac{|D|}{|d : t_i \in d|} \tag{4}$$

Here D represents the set of all documents, cardinality of D is divided by the number of documents in which the term t_i takes place at least one time. Then, logarithm of the resulting value is taken. TF-IDF score of word for a document is calculated by multiplication of $tf_{i,j}$ and idf_i. It is reasonable to infer that if a word occurs many times in a document, that word can be an important word for that document. If

this stated word does not take place in other documents so many times, one can understand that stated word is a specific word for the first documents. Such kind of words carries a lot of information compared to other words in the documents. Naturally, high occurrence number does not enlighten whole problem. For example, stop words such as "the, and, of" take place so many times in every document, so these words does not give any information about the documents. Therefore, not only term frequency but also inverse document frequency should be taken into consideration.

In our case, we have chosen best summary among three candidates with the help TF-IDF like method. We treated union of three candidates as a document because all three candidates are related to one news article. Then, every TF-IDF scores of words were calculated. For every summary candidate, TF-IDF scores of words are summed and divided by total number of words in summary candidate. Here, normalization process is critical. Since the number of words in summaries are not the same, summaries consisting of more words would result in having high total TF-IDF score. That's why total TF-IDF scores should be normalized by total number words in summaries. In these calculations, <start> and <end> tokens were not included.

B. Proposed Model

Until this point, followed methodology is related to preparation of dataset to initialize training. After making the dataset ready to train. next step is to map words into embeddings. The most common and traditional pre-trained word embeddings are GloVe [7] and word2vec [8]. Altough these tools are very powerful, they are not state of the art embeddings anymore. Language models such as ELMO [9], BERT [10] or ELECTRA [11] implement transformer models and generate contextualized word embeddings. In title generation task, since context concept is essential property, we decided to used one of these embedding models. However, those architectures have some effects in terms of model size and computation power. That's why we used "easy-bert" model for mapping words into embeddings [12]. This model directly gives the regular embeddings calculated with BERT parameters for each token or sequence. In fact, original BERT model does not give individual embeddings for words. It provides a pre-trained language model for fine tuning of the various language task. On the other hand traditional word embeddings do not take context into consideration in sufficient level. easy-bert incorporates the useful sides of traditional and contextualized models [12]. Therefore, we could obtain contextualized models for far cheaper way.

We have implemented a recurrent encoder-decoder neural network architecture with attention mechanism which is inspired by the Neural Machine Translation system idea that models the probability $p(y|x)$ of translating a source context $(x_1, ...x_k)$ to a target context $(y_1, ..., y_m)$. Our encoder-decoder neural network architecture consists of two parts: encoder estimates a representation z for each source context and decoder generates the headline word-by-word with log-probabilities:

$$logp(y_i|y_{0,...,i-1}, z) \tag{5}$$

Therefore, the modelled translation probability can be decomposed as:

$$logp(y|x) = \sum_{i}^{m} logp(y_i|y_{0,...,i-1}, z) \tag{6}$$

It is possible to use different recurrent architectures such as RNN, GRU or LSTM for generating the hidden representation z. Then, it is possible to decode each output word with probability:

$$p(y_i|y_{0,...,i-1}, z) = softmax(f(h_i)) \tag{7}$$

where h_i is the current hidden state representation of the used recurrent architecture and $f(\cdot)$ is a function that maps the hidden state representation to a vocabulary-sized vector. In our work, we have used LSTM as our recurrent architecture and all hidden states are computed throughout the LSTM unit. So far, the model only employs the encoder-decoder architecture. In order to employ the attention mechanism, we add an additional hidden state representation as follows

$$\tilde{h_i} = tanh(W_c h_i^{concat}) \tag{8}$$

where $h_i^{concat} = [c_i; h_i]$ and c_i is the context vector that captures the relevant information in source context for the attention mechanism to help generating the target headline. W_c is a linear operation that maps the concatenated representation back to the original hidden state dimensions. Then, it is possible to write this additional representation into equation (7):

$$p(y_i|y_{0,...,i-1}, z) = softmax(W_f \tilde{h_i}) \tag{9}$$

where W_f is a matrix operator representation for the function $f(\cdot)$. Then in order to find context vectors c_i we use the global attention approach where all encoder hidden states are used while calculating c_i. Current hidden states h_i are then compared with source hidden states h_j^{source} in order to derive an alignment vector a_i with size equals to the source context length:

$$(a_i)_j = \frac{\exp(score(h_i, h_j^{source}))}{\exp(\sum_{j'} score(h_i, h_{j'}^{source}))} \tag{10}$$

where $score(\cdot, \cdot)$ is a context based function defined as:

$$score(h_i, h_j^{source}) = h_i^T W_a h_j^{source} \tag{11}$$

Then, the context-vector c_i is computed as a weighted sum of all encoder hidden states where the weights are the related entries of the alignment vector a_i. Figure 1 illustrates the attention mechanism that is being explained. After combining the encoder-decoder architecture with the attention mechanism, we have used an input feeding approach where the output vectors that composes the generated headline are concatenated with the source context sequence and feed back to the encoder-decoder architecture in the following time steps. Figure 2

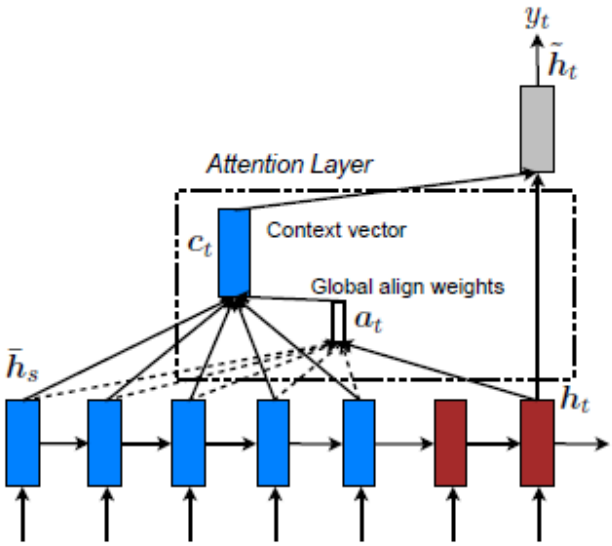

Fig. 1: Attention Mechanism

illustrates the entire model architecture together with input feeding architecture.

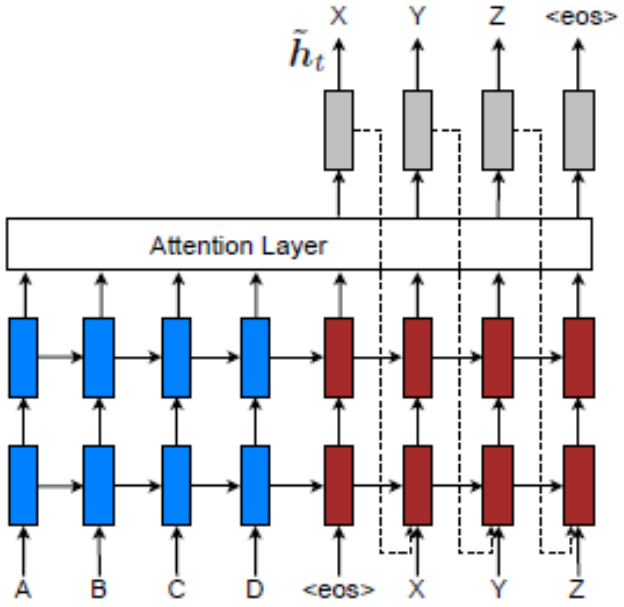

Fig. 2: Proposed Model

C. Evaluation

In order to compare our generated titles with original labeled ones we need a similarity metric. In such kind of applications BLEU metric is used frequently [13]. This metric is developed for machine translation tasks first but used in many different NLP tasks. BLEU is language independent metric and co-incides highly with human evaluation. BLEU mainly, looks for matched n-grams between reference and candidate text.

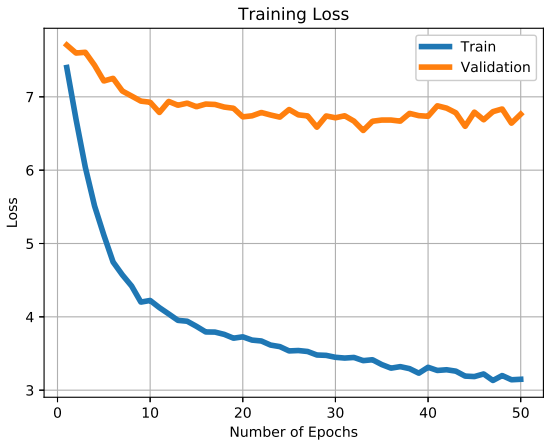

Fig. 3: Training Loss

TABLE II: Sample Generated Headlines

Original Title	georgia man shoots another times hot pocket snapped
Predicted Title	georgia man pleads
Original Title	london mayor boris johnson backs brexit boosting campaign
Predicted Title	brexit britain theresa thatcher
Original Title	conservatives embrace principled populism
Predicted Title	trump populism trumpisms
Original Title	know every sex city outfit
Predicted Title	city outfit
Original Title	solid jobs created april unemployment falls
Predicted Title	unemployment falls

Here, n-gram concept is the same as the one we learned in lectures. However, comparison is positional independent. In addition reference text may be comprise of multiple sentences or sentence blocks. BLEU score of a candidate with respect to the reference lies between 0.0 and 1.0. As it can be expected, value 1.0 represents complete match (it means that candidate is the same as reference) while value 0.0 indicates no correlation between candidate and reference [13]. In our project, original labels were used as references. Scores of candidates (our generated titles) is used for evaluation of success of our model. Obtained results will be demonstrated in next section.

V. Results

Our loss curve is shown in Figure 3. We have observed that as the training process increases our model tends to over-fit. However we prevent it by using early stop and implementing regularisation techniques such as drop-out layer. In our training we followed teacher forcing technique. For evaluation our predictions are recurrently produced as shown in Figure 2. From the experiments made on the test set we have achieved **0.35** BLEU-1 score. Table II shows the original title and our predictions.

VI. Discussion & Conclusion

In conclusion, we have implemented headline (or title) generator for news article. Since news articles are relatively long sequences. We used neural models with keeping memory.

For this purpose, we utilized RNNs with attention mechanism. Actually, RNNs are not successful in storing long term memory. In summarization tasks, since the input coming through model are long paragraphs, long term memory is a required feature for such kind of tasks. In this project, our main objective is to choose best informative text from summarizers and feed to our model which can keep long term memory. After this process, we would obtain best title.

Naturally, there are some drawbacks while generating headlines from scratch. Some possible, errors stem from similar news articles. Some keywords such as "Trump", "politics", "action" etc. take place a lot of times. Although news articles are gathered from different sources, actors of the news are same. This may make some words to have high number of occurrences. Therefore such words may be repeated at generated headlines. On the other hand, in such kind of tasks, implementing long sequences requires more complex models. We tried to overcome this issue by using summaries of the articles. Based on the summarization techniques performance of the generated headlines may change. As a future work, this problem may be examined. Overall, we have met our requirements and obtained reasonable results for this task as shown in "Results" section.

REFERENCES

[1] A. M. Rush, S. Chopra, and J. Weston, "A neural attention model for abstractive sentence summarization," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, (Lisbon, Portugal), pp. 379–389, Association for Computational Linguistics, Sept. 2015.

[2] S. Takase, J. Suzuki, N. Okazaki, T. Hirao, and M. Nagata, "Neural headline generation on abstract meaning representation," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, (Austin, Texas), pp. 1054–1059, Association for Computational Linguistics, Nov. 2016.

[3] S. Chopra, M. Auli, and A. M. Rush, "Abstractive sentence summarization with attentive recurrent neural networks," in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, (San Diego, California), pp. 93–98, Association for Computational Linguistics, June 2016.

[4] K. Lopyrev, "Generating news headlines with recurrent neural networks," *arXiv preprint arXiv:1512.01712*, 2015.

[5] E. Alfonseca, D. Pighin, and G. Garrido, "Heady: News headline abstraction through event pattern clustering," in *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1243–1253, 2013.

[6] J. Tan, X. Wan, and J. Xiao, "From neural sentence summarization to headline generation: A coarse-to-fine approach.," in *IJCAI*, pp. 4109–4115, 2017.

[7] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, (Doha, Qatar), pp. 1532–1543, Association for Computational Linguistics, Oct. 2014.

[8] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems 26* (C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, eds.), pp. 3111–3119, Curran Associates, Inc., 2013.

[9] M. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, (New Orleans, Louisiana), pp. 2227–2237, Association for Computational Linguistics, June 2018.

[10] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, (Minneapolis, Minnesota), Association for Computational Linguistics, June 2019.

[11] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "ELECTRA: Pre-training text encoders as discriminators rather than generators," in *ICLR*, 2020.

[12] R. Rua, "easy-bert," Apr. 2019.

[13] 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*, (Philadelphia, Pennsylvania, USA), pp. 311–318, Association for Computational Linguistics, July 2002.

VII. APPENDIX

A. Contributions

We all equally contributed to the project, here is the individual contribution of each partner:

- : Burak Künkçü: Dataset preparation, helping model implementation, model explanation in report
- Selim Furkan Tekin: Helping the implementation of the model. Training, testing and obtaining the results. Problem definition and related work.
- Furkan Şahinuç: Implementing TF-IDF algorithm on candidate summaries. Initialization of pre-trained vectors which are input to the model. First part of the methodology and discussion parts of the report.

```python
import os
import random
import pandas as pd
import pickle as pkl

from collections import Counter
from transformers.preprocessing import Preprocess
from transformers.summarizer import Summarizer
from transformers.summary_selector import SummarySelector


class LoadData:
    def __init__(self, dataset_path, **data_params):
        self.summary_path = os.path.join(dataset_path, 'summary_set.pkl')
        self.vocab_path = os.path.join(dataset_path, 'vocab.pkl')
        self.all_sum_save_path = os.path.join(dataset_path, 'all_sum_set.pkl')
        self.raw_data_path = os.path.join(dataset_path, 'all-the-news')

        self.content_len = data_params.get('content_len', 50)
        self.title_len = data_params.get('title_len', 15)
        self.num_samples = data_params.get('num_samples', -1)
        self.sentence_num = data_params.get('num_sentence', 3)
        self.unk_threshold = data_params.get('unk_threshold', 10)

        self.test_ratio = data_params.get('test_ratio', 0.1)
        self.val_ratio = data_params.get('val_ratio', 0.1)
        self.shuffle = data_params.get('shuffle', True)

        if not os.path.isfile(self.vocab_path) or not os.path.isfile(self.summary_path):
            contents, titles = self.__load_from_csv()
            print('\nCreating data ...')
            self.summaries, self.titles, self.word2int, self.int2word =
            ↪  self.__create_summary_set(contents, titles)

            with open(self.vocab_path, 'wb') as f:
                pkl.dump([self.word2int, self.int2word], f)

            with open(self.summary_path, 'wb') as f:
                pkl.dump([self.summaries, self.titles], f)

        else:
            print('\nLoading data from pickle ...')
            with open(self.vocab_path, 'rb') as f:
                self.word2int, self.int2word = pkl.load(f)
            with open(self.summary_path, 'rb') as f:
                self.summaries, self.titles = pkl.load(f)

        # split test train validation
        self.data_dict, self.label_dict = self.__split_data()
```

```
    def __load_from_csv(self):
        # read all articles 1, 2 and 3
        file_paths = os.listdir(self.raw_data_path)
        file_paths = [os.path.join(self.raw_data_path, file_name) for file_name in file_paths]

        df_list = []
        for file in file_paths:
            df = pd.read_csv(file, index_col=0)
            df_list.append(df)
        articles = pd.concat(df_list, ignore_index=True)

        contents = articles['content'].values[:self.num_samples]
        titles = articles['title'].values[:self.num_samples]

        return contents, titles

    def __create_summary_set(self, in_contents, in_titles):
        process_obj = Preprocess(content_len=self.content_len, title_len=self.title_len)

        print("Creating summaries ...")
        summarizer = Summarizer(num_sentence=self.sentence_num)
        sum_collection, sum_titles = summarizer.transform(X=in_contents, y=in_titles)

        summary_label = []
        summary_content = []
        for i in range(summarizer.num_summarizers):
            titles = []
            contents = []
            for sum_list, title in zip(sum_collection, sum_titles):
                contents.append(sum_list[i])
                titles.append(title)

            print('\nPreprocessing summary: {}'.format(i))
            contents, titles = process_obj.transform(X=contents, y=titles)

            summary_content.append(contents)
            summary_label.append(titles)

        with open(self.all_sum_save_path, 'wb') as f:
            pkl.dump([summary_content, summary_label], f)

        print('\nSelecting summaries ...')
        selector = SummarySelector()
        summary_content = selector.transform(summary_content)
        summary_label = summary_label[0]

        # create vocab
        word2int, int2word = self.__create_vocab(summary_content, summary_label)

        content_int = [[word2int[word] for word in content] for content in summary_content]
        title_int = [[word2int[word] for word in title] for title in summary_label]

        return content_int, title_int, word2int, int2word
```

```
    def __create_vocab(self, contents, titles):
        contents, titles = self.__rmv_less_frequent(contents, titles)
        vocab = self.__get_counts(contents, titles)

        word2int = {k: v for v, k in enumerate(vocab.keys())}
        int2word = {v: k for k, v in word2int.items()}

        return word2int, int2word

    def __rmv_less_frequent(self, contents, titles):
        vocab = self.__get_counts(contents, titles)

        # replace the less frequent words with '<unk>'
        for i in range(len(contents)):
            for j in range(len(contents[i])):
                word = contents[i][j]
                if vocab[word] <= self.unk_threshold:
                    contents[i][j] = '<unk>'
            for k in range(len(titles[i])):
                word = titles[i][k]
                if vocab[word] <= self.unk_threshold:
                    titles[i][k] = '<unk>'

        return contents, titles

    def __split_data(self):
        dataset_length = len(self.summaries)

        if self.shuffle:
            zipped = list(zip(self.summaries, self.titles))
            random.shuffle(zipped)
            self.summaries, self.titles = zip(*zipped)
            self.summaries = list(self.summaries)
            self.titles = list(self.titles)

        test_count = int(dataset_length * self.test_ratio)
        val_count = int(dataset_length * self.val_ratio)

        data_dict = dict()
        data_dict['test'] = self.summaries[:test_count]
        data_dict['validation'] = self.summaries[test_count:test_count + val_count]
        data_dict['train'] = self.summaries[test_count + val_count:]

        label_dict = dict()
        label_dict['test'] = self.titles[:test_count]
        label_dict['validation'] = self.titles[test_count:test_count + val_count]
        label_dict['train'] = self.titles[test_count + val_count:]

        return data_dict, label_dict

    @staticmethod
    def __get_counts(contents, titles):
        # create_vocab
        all_words = []
        for c, t in zip(contents, titles):
            all_words += c + t

        vocab = Counter(all_words)
        vocab = {k: v for k, v in sorted(vocab.items(), key=lambda x: x[1], reverse=True)}

        return vocab
```

```python
import os
import torch
import numpy as np
import torch.nn as nn

from transformers.word2vec import Word2VecTransformer


class Embedding(nn.Module):
    def __init__(self, int2word, device, embed_name='glove'):
        nn.Module.__init__(self)
        self.vector_path = 'embedding/embed_{}.npy'.format(embed_name)
        self.embed_name = embed_name

        self.int2word = int2word
        self.vocab_size = len(int2word)

        if os.path.isfile(self.vector_path):
            print('Loading saved embedding vectors')
            self.weights = np.load(self.vector_path, allow_pickle=True)
            self.weights = torch.from_numpy(self.weights)
        else:
            print('Creating embed tensor')
            self.weights = self.create_embed_tensor()
            np.save(self.vector_path, self.weights)

        self.embed_dim = self.weights.shape[1]
        self.embedding = nn.Embedding.from_pretrained(self.weights).to(device)

    def forward(self, captions):
        """
        :param captions: (B, S)
        :return: vectors: (B, S, Vector_dim)
        """
        vectors = self.embedding(captions)
        return vectors

    def create_embed_tensor(self):
        transformer = Word2VecTransformer(self.embed_name)
        vectors = []
        for i in range(self.vocab_size):
            print("\r{:.2f}%".format(i * 100 / self.vocab_size), flush=True, end='')
            vec = transformer.transform(self.int2word[i])
            vectors.append(vec)
        embed = np.stack(vectors, axis=0)
        return torch.from_numpy(embed)


if __name__ == '__main__':
    import pickle

    vocab = pickle.load(open('data/vocab.pkl', 'rb'))
    word2int, int2word = vocab
    Embedding(int2word)
```

```python
import random
import torch
import torch.nn as nn

from models.embed import Embedding


class Encoder(nn.Module):
    def __init__(self, vocab, hidden_dim, num_layers, device, embed_name):
        super(Encoder, self).__init__()
        self.num_layers = num_layers
        self.hidden_dim = hidden_dim

        self.embedding_layer = Embedding(vocab, device, embed_name=embed_name)
        self.embed_dim = self.embedding_layer.embed_dim

        self.rnn = nn.LSTM(input_size=self.embed_dim,
                           hidden_size=self.hidden_dim,
                           num_layers=num_layers,
                           batch_first=True)

    def forward(self, input_tensor):
        """
        :param input_tensor: (B,)
        :param hidden: ((l, b, d), (l, b, d))
        :return:
        """
        embed = self.embedding_layer.embedding(input_tensor).float()
        output, hidden = self.rnn(embed)

        return output, hidden


class Attention(nn.Module):
    def __init__(self, enc_hidden, dec_hidden):
        super(Attention, self).__init__()
        self.enc_hidden = enc_hidden
        self.dec_hidden = dec_hidden

        self.w_in = nn.Linear(self.dec_hidden, self.enc_hidden, bias=False)
        self.w_out = nn.Linear(self.dec_hidden+self.enc_hidden, self.dec_hidden)

        self.softmax = nn.Softmax()
        self.tanh = nn.Tanh()

    def forward(self, dec_out, enc_out):
        """
        :param torch.Tensor dec_out: (batch, t_out, d_dec)
        :param torch.Tensor enc_out: (batch, t_in, d_enc)
        :return:
        """
        batch, t_out, d_dec = dec_out.shape
        batch, t_in, d_enc = enc_out.shape

        dec_out_ = dec_out.contiguous().view(batch*t_out, d_dec)
        energy = self.w_in(dec_out_)
        energy = energy.view(batch, t_out, d_enc)

        # swap dimensions: (batch, d_enc, t_in)
        enc_out_ = enc_out.transpose(1, 2)

        # (batch, t_out, d_enc) x (batch, d_enc, t_in) -> (batch, t_out, t_in)
        attn_energies = torch.bmm(energy, enc_out_)

        alpha = self.softmax(attn_energies.view(batch*t_out, t_in))
        alpha = alpha.view(batch, t_out, t_in)

        # (batch, t_out, t_in) * (batch, t_in, d_enc) -> (batch, t_out, d_enc)
        context = torch.bmm(alpha, enc_out)

        # \hat{h_t} = tanh(W [context, dec_out])
        concat_c = torch.cat([context, dec_out], dim=2)
        concat_c = concat_c.view(batch*t_out, d_enc + d_dec)
        concat_c = self.w_out(concat_c)
        attn_h = self.tanh(concat_c.view(batch, t_out, d_dec))
```

```python
class Decoder(nn.Module):
    def __init__(self, vocab, hidden_dim, enc_hidden, num_layers, dropout_prob, device,
    ↪  embed_name):
        super(Decoder, self).__init__()
        self.hidden_dim = hidden_dim
        self.enc_hidden = enc_hidden
        self.num_layers = num_layers
        self.drop_prob = dropout_prob

        self.embedding_layer = Embedding(vocab, device, embed_name=embed_name)
        self.embed_dim = self.embedding_layer.embed_dim
        self.rnn = nn.LSTM(input_size=self.embed_dim,
                           hidden_size=self.hidden_dim,
                           num_layers=self.num_layers,
                           batch_first=True)

        self.attention = Attention(dec_hidden=self.hidden_dim, enc_hidden=self.enc_hidden)
        self.out_lin = nn.Linear(self.hidden_dim, self.embedding_layer.vocab_size)
        self.dropout = nn.Dropout(self.drop_prob)

    def forward(self, input_tensor, hidden, enc_out):
        # Teacher-forcing,
        input_tensor = input_tensor.unsqueeze(1)
        embed = self.embedding_layer.embedding(input_tensor).float()
        embed = self.dropout(embed)

        dec_out, dec_hid = self.rnn(embed, hidden)
        attn_out, alpha = self.attention(dec_out, enc_out)
        output = self.out_lin(attn_out)

        return output, hidden


class Seq2Seq(nn.Module):
    def __init__(self, vocabs, device, **model_params):
        super(Seq2Seq, self).__init__()
        self.word2int, self.intword = vocabs
        self.enc_hidden = model_params['encoder_hidden_dim']
        self.dec_hidden = model_params['decoder_hidden_dim']
        self.enc_num_layer = model_params['encoder_num_layer']
        self.dec_num_layer = model_params['decoder_num_layer']
        self.drop_prob = model_params['dropout_prob']
        self.embed_name = model_params['embed_name']
        self.device = device

        self.encoder = Encoder(vocab=self.intword,
                               hidden_dim=self.enc_hidden,
                               num_layers=self.enc_num_layer,
                               device=self.device,
                               embed_name=self.embed_name)
        self.decoder = Decoder(vocab=self.intword,
                               hidden_dim=self.dec_hidden,
                               enc_hidden=self.enc_hidden,
                               num_layers=self.dec_num_layer,
                               dropout_prob=self.drop_prob,
                               device=self.device,
                               embed_name=self.embed_name)

    def forward(self, contents, titles, tf_ratio=0.0):
        """

        :param contents: (b, t')
        :param titles: (b, t)
        :param tf_ratio: teacher forcing ratio
        :return:
        """
        batch, title_len = titles.shape

        enc_out, hidden = self.encoder(contents)

        # <start> is mapped to id=2
        dec_inputs = torch.ones(batch, dtype=torch.long) * self.word2int['<start>']
        dec_inputs = dec_inputs.to(self.device)
        pred, hidden = self.decoder(dec_inputs, hidden, enc_out)

```

```python
import re
import emoji
import string
import nltk

from nltk.tokenize import TweetTokenizer
from nltk.corpus import stopwords

eng_stopwords = set(stopwords.words('english'))


class Preprocess:
    def __init__(self, content_len, title_len):
        self.content_len = content_len
        self.title_len = title_len
        self.tokenizer = TweetTokenizer()

        # remove url, long-mention, mention, hash-tag, numbers
        self.re_list = [r'http\S+', r'\(@ ?[^\s].+\)', r'@ ?[^\s]+',
                        r'# ?[^\s]+', '[{}]'.format(re.escape(string.punctuation))]

        # replace '- The New York Times'
        self.re_title = r'\-[a-z A-Z]+'

        # before tokenizing, remove non textual emoji
        self.rmv_emoji = lambda x: emoji.get_emoji_regexp().sub(r'', x)

        # after tokenizing
        self.rmv_stop = lambda x: [w for w in x if w not in eng_stopwords]
        self.rmv_pun = lambda x: [w for w in x if w not in string.punctuation]
        self.rmv_short_long = lambda x: [w for w in x if 20 >= len(w) >= 3]

        self.rmv_list = [self.rmv_stop, self.rmv_pun, self.rmv_short_long]

    def transform(self, X, y):
        """
        X is the content
        y is the title

        :param X: [str, str, ..., str]
        :param y: [str, str, ..., str]
        :return:[[token , ..., token], ...], [[token , ..., token], ...]
        """
        labels = []
        clean_data = []
        for count, (content, title) in enumerate(zip(X, y)):
            print('\r{:.2f}%'.format(count * 100 / len(X)), flush=True, end='')
            token_content = self._preprocess(str(content).lower(), mode='content')
            title_content = self._preprocess(str(title).lower(), mode='title')
            clean_data.append(token_content)
            labels.append(title_content)

        return clean_data, labels

    def _preprocess(self, sentence, mode='content'):
        """
        :param sentence: string
        :return: [token, ..., token]
        """
        if mode == 'title':
            self.re_list.insert(0, self.re_title)

        # remove url, long-mention, mention, hash-tag, non-textual emoji, punctuation
        for re_op in self.re_list:
            sentence = re.sub(re_op, '', sentence)
        sentence = self.rmv_emoji(sentence)

        # tokenize the sentence
        tokens = self.tokenizer.tokenize(sentence)
        for rmv_op in self.rmv_list:
            tokens = rmv_op(tokens)

        # lower every word
        tokens = [token.lower() for token in tokens]

```

```python
from sumy.parsers.plaintext import PlaintextParser
from sumy.nlp.tokenizers import Tokenizer
from sumy.summarizers.lex_rank import LexRankSummarizer
from sumy.summarizers.lsa import LsaSummarizer
from sumy.summarizers.sum_basic import SumBasicSummarizer


class Summarizer:
    def __init__(self, num_sentence, trim_len=5000):
        self.num_sentence = num_sentence
        self.trim_len = trim_len
        self.tokenizer = Tokenizer('english')

        self.summarizers = [LexRankSummarizer(), LsaSummarizer(), SumBasicSummarizer()]
        self.num_summarizers = len(self.summarizers)

    def transform(self, X, y):
        """
        Create 5 summaries for each content. If content has less than
        self.num_sentence it is not added to collection

        :param X: [str, str, ...]
        :param y: [str, str, ...]
        :return: X=[[sum1, sum2, ..., sum5], ...], y=[str, str, ...]
        """
        labels = []
        sum_collection = []
        for count, (content, title) in enumerate(zip(X, y)):
            print('\r{:.2f}%'.format(count * 100 / len(X)), flush=True, end='')

            trimed_content = content[:self.trim_len]

            parser = PlaintextParser.from_string(trimed_content, self.tokenizer)

            sum_list = []
            long_content = True
            for summarizer in self.summarizers:
                summary = summarizer(parser.document, self.num_sentence)
                if len(summary) >= self.num_sentence:
                    sum_txt = ' '.join(sentence._text for sentence in summary)
                    sum_list.append(sum_txt)
                else:
                    long_content = False

            if long_content:
                labels.append(y[count])
                sum_collection.append(sum_list)

        return sum_collection, labels
```

```python
import math


class SummarySelector:
    def __init__(self):
        self.start = 0
        self.end = 1

    def transform(self, summaries):

        vocab = dict()
        vocab['<start>'] = self.start
        vocab['<end>'] = self.end
        count = 2
        for ind in range(len(summaries)):
            for row in summaries[ind]:
                for word in row:
                    if word not in vocab:
                        vocab[word] = count
                        count += 1
        idf = dict()

        total_word_count = 0
        total_document_count = len(summaries[0])

        candidate_corpus = []

        for num in range(len(summaries[0])):
            temp = summaries[0][num] + summaries[1][num] + summaries[2][num]
            candidate_corpus.append(temp)

        for word in list(vocab.keys()):
            belonging_doc = 0

            for doc in candidate_corpus:
                if word in doc:
                    belonging_doc += 1

            if belonging_doc == 0:
                continue

            idf[word] = math.log(total_document_count / belonging_doc)

        for candidate in candidate_corpus:
            total_word_count += len(candidate) - 6  # start end x3

        print(total_word_count)

        chosen_summaries = []
        for num in range(len(summaries[0])):
            scores = []
            candidates = []
            candidates.append(summaries[0][num][:])
            candidates.append(summaries[1][num][:])
            candidates.append(summaries[2][num][:])

            for candidate in candidates:
                score = 0
                for word in candidate:
                    if word == '<start>' or word == '<end>':
                        continue
                    score += (candidate_corpus[num].count(word) / total_word_count) * idf[word]

                scores.append(score / (len(candidate) - 2))

            best_index = scores.index(max(scores))

            chosen_summaries.append(candidates[best_index])

        return chosen_summaries
```

```python
import os
import pickle as pkl
import numpy as np
from gensim.models import KeyedVectors

np.random.seed(42)


class Word2VecTransformer:
    def __init__(self, embed_name='glove'):
        project_dir = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
        self.embed_name = embed_name
        if embed_name == 'glove':
            self.vector_path = os.path.join(project_dir, 'embedding/glove_dict.pkl')
            self.model = pkl.load(open(self.vector_path, 'rb'))
            self.vector_size = len(next(iter(self.model.values())))

        elif embed_name == 'bert':
            from easybert import Bert
            self.vector_path = 'https://tfhub.dev/google/bert_multi_cased_L-12_H-768_A-12/1'
            self.vector_size = 768

            print('Downloading BERT Model')
            self.model = Bert(self.vector_path)

        elif embed_name == 'fasttext':
            self.model_path = os.path.join(project_dir, 'embedding/fasttext.pkl')
            self.vector_path = os.path.join(project_dir, 'embedding/fasttext.vec')

            if os.path.isfile(self.model_path):
                model_file = open(self.model_path, 'rb')
                self.model = pkl.load(model_file)
            else:
                print('Loading Fasttext model')
                self.model = KeyedVectors.load_word2vec_format(self.vector_path, binary=False,
                ↪  unicode_errors='replace')
                model_file = open(self.model_path, 'wb')
                pkl.dump(self.model, model_file)

            self.vector_size = self.model.vector_size

        self.special_tokens = ['<start>', '<end>', '<pad>', '<unk>']
        self.special_tok_dict = {}
        for token in self.special_tokens:
            self.special_tok_dict[token] = np.random.rand(self.vector_size)

    def transform(self, x):
        """
        returns bert vector of input word

        :param str x: input word
        :return:
        """
        if x in self.special_tokens:
            r = self.special_tok_dict[x]
        else:
            if self.embed_name == 'bert':
                with self.model:
                    r = self.model.embed(x)
            elif self.embed_name == 'glove':
                if x in self.model:
                    r = self.model[x]
                else:
                    r = np.random.rand(self.vector_size)
            elif self.embed_name == 'fasttext':
                if x in self.model.wv:
                    r = self.model.wv[x]
                else:
                    r = np.random.rand(self.vector_size)
        return r
```

```python
from dataset import HeadlineDataset
from torch.utils.data import DataLoader


class BatchGenerator:
    def __init__(self, data_dict, label_dict, **params):
        self.data_dict = data_dict
        self.label_dict = label_dict

        self.batch_size = params.get('batch_size', 16)
        self.num_works = params.get('num_works', 4)
        self.shuffle = params.get('shuffle', True)

        self.dataset_dict, self.dataloader_dict = self.__create_data()

    def generate(self, data_type):
        """
        :param data_type: can be 'test', 'train' and 'validation'
        :return: img tensor, label numpy_array
        """
        selected_loader = self.dataloader_dict[data_type]
        yield from selected_loader

    def num_iter(self, data_type):
        dataset = self.dataset_dict[data_type]
        return len(dataset) // self.batch_size

    def __create_data(self):

        im_dataset = {}
        for i in ['test', 'train', 'validation']:
            im_dataset[i] = HeadlineDataset(articles=self.data_dict[i],
                                            titles=self.label_dict[i])

        im_loader = {}
        for i in ['test', 'train', 'validation']:
            im_loader[i] = DataLoader(im_dataset[i],
                                      batch_size=self.batch_size,
                                      shuffle=self.shuffle,
                                      num_workers=self.num_works,
                                      drop_last=True)
        return im_dataset, im_loader
```

```python
model_params = {
    'encoder_hidden_dim': 512,
    'decoder_hidden_dim': 512,
    'encoder_num_layer': 1,
    'decoder_num_layer': 1,
    'dropout_prob': 0.3,
    'embed_name': 'fasttext'
    }

train_params = {
    'num_epoch': 50,
    'learn_rate': 0.0003,
    'train_tf_ratio': 0.5,
    'val_tf_ratio': 0.1,
    'clip': 5,
    'eval_every': 400
}

data_params = {
    "content_len": 50,
    "title_len": 15,
    "num_samples": 100,
    "num_sentence": 3,
    "test_ratio": 0.1,
    "val_ratio": 0.1,
    "shuffle": True,
    "unk_threshold": 15
}

batch_params = {
    'batch_size': 128,
    'num_works': 0,
    'shuffle': True,
}
```

```python
import os
import random
import pandas as pd
import pickle as pkl

from collections import Counter
from transformers.preprocessing import Preprocess
from transformers.summarizer import Summarizer
from transformers.summary_selector import SummarySelector


class LoadData:
    def __init__(self, dataset_path, **data_params):
        self.summary_path = os.path.join(dataset_path, 'summary_set.pkl')
        self.vocab_path = os.path.join(dataset_path, 'vocab.pkl')
        self.all_sum_save_path = os.path.join(dataset_path, 'all_sum_set.pkl')
        self.raw_data_path = os.path.join(dataset_path, 'all-the-news')

        self.content_len = data_params.get('content_len', 50)
        self.title_len = data_params.get('title_len', 15)
        self.num_samples = data_params.get('num_samples', -1)
        self.sentence_num = data_params.get('num_sentence', 3)
        self.unk_threshold = data_params.get('unk_threshold', 10)

        self.test_ratio = data_params.get('test_ratio', 0.1)
        self.val_ratio = data_params.get('val_ratio', 0.1)
        self.shuffle = data_params.get('shuffle', True)

        if not os.path.isfile(self.vocab_path) or not os.path.isfile(self.summary_path):
            contents, titles = self.__load_from_csv()
            print('\nCreating data ...')
            self.summaries, self.titles, self.word2int, self.int2word =
            ↪  self.__create_summary_set(contents, titles)

            with open(self.vocab_path, 'wb') as f:
                pkl.dump([self.word2int, self.int2word], f)

            with open(self.summary_path, 'wb') as f:
                pkl.dump([self.summaries, self.titles], f)

        else:
            print('\nLoading data from pickle ...')
            with open(self.vocab_path, 'rb') as f:
                self.word2int, self.int2word = pkl.load(f)
            with open(self.summary_path, 'rb') as f:
                self.summaries, self.titles = pkl.load(f)

        # split test train validation
        self.data_dict, self.label_dict = self.__split_data()

    def __load_from_csv(self):
        # read all articles 1, 2 and 3
        file_paths = os.listdir(self.raw_data_path)
        file_paths = [os.path.join(self.raw_data_path, file_name) for file_name in file_paths]

        df_list = []
        for file in file_paths:
            df = pd.read_csv(file, index_col=0)
            df_list.append(df)
        articles = pd.concat(df_list, ignore_index=True)

        contents = articles['content'].values[:self.num_samples]
        titles = articles['title'].values[:self.num_samples]

        return contents, titles

    def __create_summary_set(self, in_contents, in_titles):
        process_obj = Preprocess(content_len=self.content_len, title_len=self.title_len)

        print("Creating summaries ...")
        summarizer = Summarizer(num_sentence=self.sentence_num)
        sum_collection, sum_titles = summarizer.transform(X=in_contents, y=in_titles)

        summary_label = []
        summary_content = []
```

```python
import sys
import pickle as pkl

from config import data_params, batch_params, model_params, train_params
from load_data import LoadData
from batch_generator import BatchGenerator
from trainer import train
from tester import test


def main(mode):
    dataset_path = 'data'
    data = LoadData(dataset_path=dataset_path, **data_params)

    print('Creating Batch Generator...')
    batch_gen = BatchGenerator(data_dict=data.data_dict,
                               label_dict=data.label_dict,
                               **batch_params)

    if mode == 'train':
        train(vocabs=[data.word2int, data.int2word],
              batch_gen=batch_gen,
              train_params=train_params,
              model_params=model_params)

    elif mode == 'test':
        print('Loading model')
        model_file = open('results/seq2seq.pkl', 'rb')
        model = pkl.load(model_file)
        print('Testing model...')
        test(model, data.int2word, batch_gen)


if __name__ == '__main__':
    if len(sys.argv) == 1:
        print('no arguments given, default process of training has started')
        run_mode = 'train'
    else:
        run_mode = sys.argv[1]
    main(run_mode)
```

```python
import torch
import numpy as np

from transformers.bleu import compute_bleu
from trainer import translate


device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


def test(net, vocab, batch_gen):
    net.eval()
    net.to(device)

    bleu_all_score = []
    for idx, (content, title) in enumerate(batch_gen.generate('test')):

        print('\rtest:{}'.format(idx), flush=True, end='')

        content, title = content.to(device), title.to(device)
        title_pre = net(content, title, tf_ratio=0.1)

        referance_caps, translate_caps = translate(outputs=[title, title_pre],
                                                   int2word=vocab,
                                                   top_k=10,
                                                   print_count=10)
        print('\n')
        for i in range(len(referance_caps)):
            bleu_score_list = []
            for j in range(1, 5):
                bleu, _, _ = compute_bleu(referance_caps[i], translate_caps[i], max_order=j)
                bleu_score_list.append(bleu)
            bleu_all_score.append(bleu_score_list)

    bleu_all_score = np.array(bleu_all_score)
    bleu_score_arr = np.mean(bleu_all_score, axis=0)

    for i in range(4):
        print("BLUE {}: {:.2f}".format(i+1, bleu_score_arr[i]))
```

```python
import torch
import pickle
import numpy as np
import collections
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

from models.seq2seq import Seq2Seq

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


def train(vocabs, batch_gen, train_params, model_params):
    word2int, int2word = vocabs
    num_epoch = train_params['num_epoch']
    learn_rate = train_params['learn_rate']
    clip = train_params['clip']
    eval_every = train_params['eval_every']
    train_tf_ratio = train_params['train_tf_ratio']
    val_tf_ratio = train_params['val_tf_ratio']

    net = Seq2Seq(vocabs=vocabs, device=device, **model_params).to(device)
    net.train()

    opt = optim.Adam(net.parameters(), lr=learn_rate)
    weights = calc_class_weights(batch_gen.data_dict, batch_gen.label_dict)
    criterion = nn.CrossEntropyLoss(weight=weights, ignore_index=word2int['<pad>'])

    print('Training is starting ...')
    train_loss_list = []
    val_loss_list = []
    for epoch in range(num_epoch):
        running_loss = 0

        for idx, (x_cap, y_cap) in enumerate(batch_gen.generate('train')):
            print('\rtrain:{}/{}'.format(idx, batch_gen.num_iter('train')), flush=True, end='')
            x_cap, y_cap = x_cap.to(device), y_cap.to(device)

            opt.zero_grad()
            output = net(x_cap, y_cap, train_tf_ratio)

            loss = criterion(output.view(-1, output.size(2)), y_cap.view(-1).long())
            loss.backward()

            nn.utils.clip_grad_norm_(net.parameters(), clip)
            opt.step()

            running_loss += loss.item()

            if (idx+1) % eval_every == 0:
                print('\n')
                val_loss = evaluate(net, word2int, batch_gen, weights, val_tf_ratio)
                print("\nEpoch: {}/{}...".format(epoch + 1, num_epoch),
                      "Step: {}...".format(idx),
                      "Loss: {:.4f}...".format(running_loss / idx),
                      "Val Loss: {:.4f}\n".format(val_loss))

        print('\nCreating sample captions')
        sample(net, vocabs, generator=batch_gen.generate('validation'))
        print('\n')

        train_loss_list.append(running_loss / idx)
        val_loss_list.append(val_loss)

        loss_file = open('results/losses.pkl', 'wb')
        model_file = open('results/seq2seq.pkl', 'wb')
        pickle.dump([train_loss_list, val_loss_list], loss_file)
        pickle.dump(net, model_file)

    print('Training finished, saving the model')
    model_file = open('seq2seq.pkl', 'wb')
    pickle.dump(net, model_file)


```