ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»
Институт компьютерных наук и технологий

Отчет о прохождении производственной (научно-исследовательская работа магистра) практики

Фурмана Владислава Константиновича

(Ф.И.О. обучающегося)

2 курс, 3540203/00101

(номер курса обучения и учебной группы)

02.04.03 Математическое обеспечение и администрирование информационных систем

(направление подготовки (код и наименование))

Место прохождения практики: ФГАОУ ВО «СПбПУ», ИКНиТ, ВШИИ,

(указывается наименование профильной организации или наименование структурного подразделения

г. Санкт-Петербург, ул. Обручевых, д. 1, лит. В

ФГАОУ ВО «СПбПУ», фактический адрес)

Сроки практики: с 24.01.2022 по 21.03.2022

Руководитель практической подготовки от ФГАОУ ВО «СПбПУ»: Белых Игорь Николаевич, к.ф.-м.н., доцент ВШИИ

(*Ф.И.О., уч. степень, должность)*

Консультант практической подготовки от ФГАОУ ВО «СПбПУ»: Пак Вадим Геннадьевич, к.ф.-м.н., доцент ВШИИ

(*Ф.И.О., уч. степень, должность)*

Руководитель практической подготовки от профильной организации: нет

Оценка: хорошо

Руководитель практической подготовки
от ФГАОУ ВО «СПбПУ»: Белых И.Н.

Консультант практической подготовки
от ФГАОУ ВО «СПбПУ»:

Руководитель практической подготовки
от профильной организации:

Обучающийся: Фурман В.К.

Дата:

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

С помощью естественного языка можно выразить любую мысль, любую идею. Любое изображение или звук можно описать словами. Текст является всеобъемлющим средством передачи информации. Что означает, что обработка текстов на естественных языках (Natural Language Processing, NLP) является крайне важной и актуальной проблемой.

Существует большое множество задач в области обработки текстов, например:

- перевод с одного языка на другой (например, перевод с русского на японский или обратно);
- или же монолингвистический перевод (перевод с языка в него же), как, например, упрощение текстов (понижение сложности слов, выражений, грамматики, сохраняя при этом исходный смысл текста);
- классификация текстов (положительный или отрицательный отзыв, фильтрация спама и т. д.);
- генерация текстов (например, из заданного заголовка сгенерировать статью);
- реферирование текстов (из большого по объёму документа или набора документов выделить ёмкую основную мысль);

Для японского и китайского языков особый интерес представляет задача упрощения текстов, так как эти языки используют иероглифическую письменность, где для чтения текстов нужно знать чтение и значение отдельных иероглифов (в японском языке большинство иероглифов имеют несколько чтений, порой даже больше 10). Это может значительно сузить круг возможных читателей какой-либо текста — дети изучают иероглифы, начиная с первого класса школы и до самого выпуска. То же касается и иностранцев, имеющих довольно ограниченное знание иероглифов. Причём даже взрослые японцы и китайцы могут испытывать трудности с иероглифами, особенно связанные с юридическими документами. Количество иероглифов довольно высоко, в среднем, взрослый японец знает порядка 2 000 иероглифов, взрослый китаец — 8 000 (хотя самих иероглифов значительно больше — не менее 80 000, — но большинство из них используются крайне редко). Поэтому есть высокая потребность в упрощении текстов для увеличения количества их потенциальных читателей.

Более того, упрощение текстов может повысить эффективность других задач NLP, как, например, реферирование, извлечение информации, машинный перевод и т. д.

Целью данной научно-исследовательской работы является окончательная программная реализация системы для упрощения предложений на японском языке.

Для достижения данной цели необходимо выполнить следующие задачи:

- описать архитектуру реализуемой системы, обзор инструментов для разработки;
- сделать обзор на архитектуру модели Transformer, как с теоритической, так и с практической точек зрения;
- описать реализацию системы упрощения (модель, сервер и приложение);
- провести демонстрацию примеров работы разработанной системы на предложениях как из корпуса, так и вне корпуса.

ГЛАВА 1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ РАБОТЫ

1.1. Этапы обработки естественного языка

Как правило, в обработке текстов обычно выделяют следующие этапы [15, с. 9]:

1. Графематический анализ. Здесь осуществляется анализ на уровне символов, в том числе и токенизация, то есть разбиение набора символов (текста) на последовательность отдельных структурированных частей (слово, знак препинания, число, гиперссылка, адрес электронной почты и т. д.).
2. Морфологический анализ. Здесь происходит анализ на уровне слов (не токенов). Стоит выделить следующие процессы, проходящие на этом этапе:
   - Лемматизация — это нахождение нормальной (начальной) формы слова (леммы), к примеру, лемма у слова «сбегать» — «бегать». В японском это относится в основном к глаголам, так как у существительных, как правило, всего одна словоформа.
   - Приписывание граммем. Граммемы — грамматическая характеристика, например, род, падеж, число. Граммемы могут помочь в разрешении неоднозначностей, которые возникают в морфологии.
3. Фрагментационный анализ. Осуществляется на уровне фраз, частей предложений. Этот этап неразрывно связан с синтаксическим анализом, а иногда и вовсе говорят о них, как об одном целом. Сюда, например, может входить обработка причастных или деепричастных оборотов.
4. Синтаксический анализ. Осуществляется на уровне предложений. Здесь, как правило, строится дерево зависимостей одних слов от других, а также исключается морфологическая неоднозначность.
5. Семантический анализ. Осуществляется на уровне всего текста. Самый сложный и неоднозначный из этапов, здесь появляется формальное представление смысла текста, как правило, в виде сементического графа. На сегодняшний день задачи семантического анализа чаще всего решаются искусственными нейронными сетями, о чём мы и поговорим далее.

1.2. Искусственные нейронные сети

Вкратце, искусственная нейронная сеть (ИНС) представляет собой систему соединённых и взаимодействующих между собой простых нейронов. Каждый нейрон получает на вход несколько чисел (входные данные или же выходы других нейронов), суммирует эти числа с определёнными коэффициентами (нахождение оптимальных коэффициентов — обучение ИНС), после чего применяет к сумме функцию активации (любую нелинейную функцию) и передаёт результат на вход другому нейрону (или же на выход ИНС).

Существует большое множество различных архитектур ИНС (свёрточные, рекуррентные, рекурсивные, графовые и т. д.), однако в данной работе будет сконцентрировано внимание на так называемых Transformer'ах, используемых, как правило, для языковой обработки, в частности для задач перевода и упрощения текстов.

1.3. Какие модели используют для обработки естественных языков

Существует большое количество моделей, использующихся для обработки естественных языков. Одними из наиболее популярных (до появления Transformer'ов) были следующие:

- Рекуррентные нейронные сети (RNN) (1982 г.):
  - Обработка последовательностей (например, текста).
  - На каждый слой передаётся текущий элемент (слово) + результат предыдущего слоя.
  - Причём есть обратные связи — поэтому рекуррентные.
  - Очень медленные — из-за последовательной природы нельзя распараллелить.
- Долгая краткосрочная память (LTSM) (1997 г.):
  - Разновиднсть RNN с элементом «забывания».
  - Ещё медленнее (из-за более сложного устройства каждого слоя модели).

Для больших объёмов данных эти модели подходят не очень хорошо из-за своей последовательной природы обработки (требует огромных вычислительных ресурсов). На замену им пришли так называемые Transformer'ы (которые мы обсудим в следующем разделе).

1.4. О модели Transformer

Transformer — относительно новая (2017 г.) архитектура глубоких ИНС, разработанная в Google Brain. Так же, как и рекуррентные ИНС (РНС), Transformer'ы предназначены для обработки последовательностей (к примеру, текста), то есть Transformer'ы относятся к sequence-to-sequence (seq2seq) моделям. В отличие от РНС, Transformer'ы не требуют обработки последовательностей по порядку, благодаря чему они распараллеливаются легче, чем РНС, и могут быстрее значительно обучаться.

1.4.1. Где применяются Transformer'ы

Используются Transformer'ы, например, в Яндексе (там его используют для лучшего ранжирования запросов, то есть поиск идёт не только по тексту, как обычной строке, но и по смыслу этого текста), во многих переводчиках (Яндекс, Google, DeepL и т. д.), а также в GPT-3 — самой большой на сегодняшний день модели генерации текстов на английском языке.

1.4.2. Устройство Transformer'а

Transformer состоит из encoder'а и decoder'а. Encoder получает на вход последовательность слов в виде векторов (word2vec). Decoder получает на вход часть этой последовательности и выход encoder'а. Encoder и decoder состоят из слоев. Слои encoder'а последовательно передают результат следующему слою в качестве его входа. Слои decoder'а последовательно передают результат следующему слою вместе с результатом encoder'а в качестве его входа.

Каждый encoder состоит из механизма внимания (attention) (вход из предыдущего слоя) и ИНС с прямой связью (вход из механизма внимания). Каждый decoder состоит из механизма внимания (вход из предыдущего слоя), механизма внимания к результатам encoder'а и ИНС с прямой связью (вход из механизма внимания).

1.4.3. Механизм внимания

Каждый механизм внимания параметризован матрицами весов запросов W_Q, весов ключей W_K, весов значений W_V. Для вычисления внимания входного вектора

X к вектору Y, вычисляются вектора $Q = W_Q X$, $K = W_K X$, $V = W_V Y$. Эти вектора используются для вычисления результата внимания по формуле (1.1).

Если коротко, то внимание устанавливает взаимоотношения слов в предложении. Оно показывает нам насколько важен каждый элемент (по отдельности).

$$\text{Attention}(Q, K, V) = \underbrace{\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)}_{\text{scores}} V, \tag{1.1}$$

где

- scores «оценивают» важность элементов (там лежат значения от 0 до 1).
- Q (Query), K (Key), V (Value) — матрицы входных элементов.
- d_k — нижняя размерность одной из этих матриц (длина части embedding'a).

1.4.4. MultiHead Attention

«Сердце» Transformer'ов — MultiHead Attention. Несколько слоёв внимания позволяют «следить» за разными частями входной последовательности (независимо от других). Добавляя больше этих слоёв, мы «следим» за бóльшим количеством частей последовательности.

$$\text{MultiHead}(Q, K, V) = \text{Concat}\left(\text{head}_1, \ldots, \text{head}_h\right) W_0, \tag{1.2}$$

где

- $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$.
- W_i — матрицы коэффициентов для обучения.

1.4.5. Positional Encoding

Так как в Transformer'е нет ни рекурренции (recurrence), ни свёртки, нам нужно что-то, что будет использовать порядок элементов в последовательности.

$$PE(p, 2i) = \sin\left(\frac{p}{10\,000^{2i/d_{\text{model}}}}\right), \tag{1.3}$$

$$PE(p, 2i+1) = \cos\left(\frac{p}{10\,000^{(2i+1)/d_{\text{model}}}}\right), \tag{1.4}$$

где

- p (position) — позиция,
- i (dimension) — размер предложения.

1.4.6. Обзор архитектуры Transformer'a

В оригинальной статье [13] представлена следующая архитектура Transformer'a (см. рис.1.1).

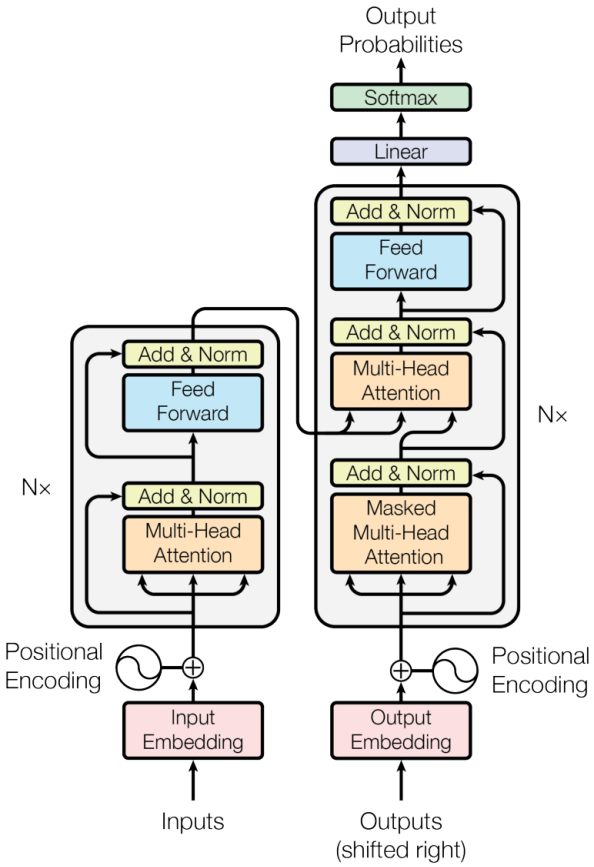

Рис.1.1. Архитектура Transformer'a

На рис.1.1 мы видим, что входные данные превращаются в эмбеддинги, после чего к ним суммируются positional encoding, после чего результат этой операции прогоняется через механизм MultiHead Attention, и, в итоге, через линейный слой со слоем Softmax мы получаем выходные вероятности элементов (слов в предложении).

1.4.7. Верхнеуровневый взгляд на Transformer

Какой высокоуровневый смысл несут encoder и decoder? Encoder представляет собой модель, «изучающую» исходный язык, его контекст (какие слова встечаются рядом в предложении, как следуют предложения друг за другом). То есть обучая encoder, мы обучаем модель «понимать» язык. Decoder же, в свою очередь, учится превращать контекст, полученный encoder'ом в какой-то осмыс-

ленный набор слов (в том числе и на другом языке). Иными словами, encoder получает информацию о предложениях, а decoder превращает эту информацию во что-то полезное (переведённое предложение, упрощённый текст, ответ на вопрос пользователя и т.д.)

1.4.8. Как encoder учится понимать контекст

Чтобы научить encoder понимать смысл текста на каком-либо языке, нужно его каким-то образом обучить (предоставить данные для обучения). Отличная новость заключается в том, что данные для обучения можно получить относительно несложно, причём в очень больших объёмах. В модели BERT [11], к примеру, сделали следующее: для большого набора языков выгрузили все статьи википедии для каждого из них. После этого процесс обучения заключается в том, чтобы передавать encoder'у предложения, но не в исходном виде, а с замаскированными словами (вспомним маску, о которой мы говорили ранее), чтобы encoder пытался угадать, какое слово должно стоять в предложении. Тоже самое делается и с предложениями — модели подаются 2 предложения, и она должна определить, следует ли оно предложение за другим. Вся прелесть в том, что весь этот процесс обучения не нуждается в ручной обработке — маскировка случайных слов и подставление двух случайных предложений реализуется программно, причём довольно просто.

После чего эти большие объёмы данных прогоняются через encoder, обучая его, а на выходе мы получаем модель, имеющую довольно обширное знание о законах языка, о том, как строятся слова в предложениях, а также как строятся сами предложения. Далее нужно лишь дообучить decoder для решения нужной нам задачи (fine-tuning), что можно сделать даже на относительно небольшом массиве данных. В итоге мы получаем модель, превосходящую современные решения в мире NLP.

ГЛАВА 2. ДЕТАЛИ ПРАКТИЧЕСКОЙ РЕАЛИЗАЦИИ

2.1. Инструменты и исходный код

2.1.1. Выбор инструментов для системы упрощения

В данной работе используются следующие инструменты:

1. Машинное обучение. Для обучения модели будет использоваться Python в связке с фреймворком для машинного обучения PyTorch [5], предоставляющий широкие возможности для реализации нейронных сетей, в том числе, там присутствует поддержка ранее упомянутых Transformer'ов.
2. Токенизация. Для токенизации японского текста будет использоваться библиотека MeCab [4] (он также вычисляет части речи токенов).
3. Эмбеддинги. Готовые модели с эмбеддингами могут быть взяты из Python–библиотеки Spacy [6] (в том числе там есть эмбеддинги для японского языка).
4. Сервер. Тут, опять же, будет использован Python с фреймворком Falcon [1], позволяющим создавать легковесный back-end. То есть будет реализован REST API сервер.
5. Веб-приложение. Будет использоваться TypeScript [7] с библиотекой Lit [3] (библиотека для веб-компонентов).

Система будет доступна в браузере в виде простого веб-приложения, то есть модель будет обучена на Python, а пользоваться обученной моделью можно будет в любом современном браузере (пользователю ничего не нужно будет устанавливать).

2.1.2. Объяснение выбранной архитектуры

Описанная архитектура «приложение-сервер» выбрана неслучайно. Дело в том, что в отличие, к примеру, от настольного приложения, пользователю ничего не нужно скачивать — он просто открывает веб-приложение и может пользоваться приложением. Обученная модель используется на сервере, что позволяет её чаще обновлять, а возможно даже и заменить на более удачную.

Это также может позволить другим разработчикам взять готовую часть решения — например, взять уже существующее приложение и использовать там свою модель для упрощения. Или же, наоборот, использовать в своём существующем приложении сервер, представленный в данной работе, просто делая к нему запрос

по ранее упомянутому пути (так как сервер не имеет привязки к приложению, сделать это довольно просто, нужно лишь настроить домены, которым будет дан доступ к серверу).

2.1.3. Исходный код системы

Исходный код разработанной системы был размещён в 2-х репозиториях на GitHub:

- приложение можно найти в репозитории [8],
- модель с сервером — в репозитории [9].

Разделение на 2 репозитория было сделано затем, чтобы обновление одного из компонентов (сервер/приложение) не затрагивало другой компонет системы. К примеру, если кто-то будет использовать сервер с упрощением, то ему вовсе не обязательно знать о том, что обновилось пользовательское приложение (которым он, возможно, и вовсе не пользуется).

2.2. Как устроен Transformer изнутри

Для ускорения вычислений в PyTorch используются не матрицы размерности $N \times M$, а тензоры размерности $B \times M \times N$ (где B — размер батча), то есть матрицы обрабатываются батчами размера B. Ускорение происходит за счёт оптимизированного вычисления батчей на видеокартах в PyTorch. Однако для простоты изложения будем считать, что работаем мы с матрицами.

2.2.1. Механизм внимания

Вернёмся к формуле (1.1). Программно её можно реализовать следующим образом:

```
1  def scaledDotProductAttention(
2    query: Tensor,
3    key: Tensor,
4    value: Tensor,
5    mask: Optional[Tensor] = None
6  ) -> Tensor:
7    # Считаем scale, на который будем делить
8    scale = query.size(-1) ** 0.5
9    # Перемножаем матрицы query и key, делим их на scale
10   temp = query.bmm(key.transpose(1, 2)) / scale
11
12   # Применяем маску, если она есть
```

```
    if (mask is not None):
      temp += mask

    # Применяем softmax к измерению embedding'ов
    softmax = f.softmax(temp, dim=-1)
    # Перемножаем softmax с матрицей value
    return softmax.bmm(value)
```

Обратим внимание на то, что в коде используется некая маска. О том, что это и зачем она нужна, поговорим в следующем разделе.

2.2.2. Маска в механизме внимания

Как же выглядит маска? Просто создаётся треугольная матрица формы size×size, в левой части которой «0», а в правой — «$-\infty$». Нужно это для того, чтобы при обучении не показывать полностью переведённые (упрощённые) предложения модели (защита от переобучения). То есть «$-\infty$» при суммировании маски со scores заставляет модель «принебречь» частями предложения. Матрица имеет следующий вид (см. рис.2.1).

$$\begin{bmatrix} 0 & -\infty & -\infty & \dots & -\infty & -\infty \\ 0 & 0 & -\infty & \dots & -\infty & -\infty \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & -\infty & -\infty \\ 0 & 0 & 0 & \dots & 0 & -\infty \\ 0 & 0 & 0 & \dots & 0 & 0 \end{bmatrix}_{\texttt{size}\times\texttt{size}} \tag{2.1}$$

Например, матрица маски размера 4×4 представлена на рис.2.2.

$$\begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.2}$$

Реализовать создание такой матрицы довольно несложно, в PyTorch это можно сделать следующим образом:

```
def generateSquareSubsequentMask(size: int, device: torch.device) -> Tensor:
  # Создаём треугольную матрицу
  mask = (torch.triu(torch.ones((size, size), device=device)) == 1).transpose(0, 1)
  # Переводим её в формат float с 0-ми и -inf
  mask = mask.float() \
```

```
6       .masked_fill(mask == 0, float("-inf")) \
7       .masked_fill(mask == 1, float(0.))
8     return mask
```

2.2.3. Positional Encoding

Реализовать positional encoding тоже не составляет большого труда:

```
1  def positionalEncoding(
2    sequenceLength: int,
3    dModel: int,
4    device: torch.device
5  ) -> Tensor:
6    # Тензор [[[0.], [1.], [2.], ...]]
7    pos = torch \
8      .arange(sequenceLength, dtype=torch.float, device=device) \
9      .reshape(1, -1, 1)
10   # Тензор [[[0., 1., 2., ...]]]
11   dim = torch.arange(dModel, dtype=torch.float, device=device).reshape(1, 1, -1)
12   # Фаза (аргумент для cos/sin) =
13   # [
14   #   [
15   #     [0., 0., 0., ...],
16   #     [1., 1., 1., ...],
17   #     [2., 2., 2., ...],
18   #     ...
19   #   ]
20   # ]
21   phase = pos / 10000 ** (dim // dModel)
22
23   # [[[sin(...),  cos(...), sin(...),  cos(...), ...], ...]]
24   return torch.where(dim.long() % 2 == 0, torch.sin(phase), torch.cos(phase))
```

2.3. Устройство разработанной модели

2.3.1. Обучение

Изначальные коэффициенты для модели генерируются с помощью Glorot initialization [12] (функция `nn.init.xavier_uniform_` в PyTorch).

Для обучения используется оптимизатор Adam (класс `torch.optim.Adam` в PyTorch) со следующими параметрами:

– Batch Size = 64,

– Learning Rate = 10^{-4},

– Weight Decay[1] = 0,
– Betas = (0,9; 0,98) (как в оригинальной статье [10]),
– Epsilon = 10^{-9} (как в оригинальной статье [10]).

В качестве функции потерь была выбрана функция перекрёстной-энтропии (класс `torch.nn.CrossEntropyLoss` в PyTorch).

В коде обучение выполняется с помощью функций `evaluate`, `train` и `trainEpoch` (см. Приложение 1, файл `modules/Seq2SeqTransformer/utils.py`).

2.3.2. Топология ИНС

В модели используется топология из оригинальной статьи [10] со следующими параметрами:

– Epochs[2] (количество эпох) = 30,
– Embeddings Size (размер эмбеддингов) = 512,
– Attention Heads (количество механизмов внимания) = 8,
– Dim Forward (размер слоя feed forward) = 512,
– Encoder Layers (количество слоёв encoder'a) = 6,
– Decoder Layers (количество слоёв decoder'a) = 6.

2.3.3. Работа с корпусом

Как уже было сказано ранее, используется корпус SNOW [14]. Авторы этого корпуса выложили его на HuggingFace [2], поэтому использовать его не составляет большого труда (см. Приложение 1, файл `modules/Dataset/snowSimplifiedJapanese/main.py`). Этот корпус состоит из 2-х частей:

1. T15 (50 000 предложений) — будем использовать для обучения и валидации (train / validation — 95% / 5%);
2. T23 (35 000 предложений) — будем использовать для тестирования итоговой модели.

[1] Weight Decay (регуляризация) была отключена, так как в нашем распоряжении имеется довольно небольшой корпус. Попытки установить хотя бы какое-то небольшое значение для этого параметра значительно ухудшали качество итоговой обученной модели.

[2] Это максимальное количество эпох. Если валидация модели не улучшается 3 последних эпохи, то обучение модели останавлиается во избежание переобучения.

2.3.4. Работа с японским языком

В первой главе мы обсуждали сложности работы с японским языком, тем не менее, существуют готовые решения, способные значительно упростить нам жизнь.

Токенизация с помощью MeCab выполняется на сервере, чтобы передать пользователю информацию о частях речи. Токенизация и преобразование токенов в эмбеддинги выполняется при обучении и упрощении (здесь MeCab нам не подходит, так как он поддерживает лишь токенизацию). Эмбеддинги поддерживаются в Spacy (как и токенизация, но без частей речи, поэтому и используются 2 инструмента). Код работы с японским языком может быть найден Приложении 1, файлы `modules/Language/utils.py` и `modules/Language/definitions.py`.

Стоит также отметить, что на вход модели подаются не только токены со словами, запятыми, точками и т. д. Есть также так называемые специальные символы:

1. `<unk>` (unkown) — неизвестный токен (например, слово, которого нет в словаре);
2. `<pad>` (padding) — этим символом выравнивают предложения до одной длины (просто вставляют их в конец всех предложений), это нужно по той причине, что модель работает с предложениями одной длины;
3. `<bos>` (Beginning Of Sentence) — символ начала предложения;
4. `<eos>` (End Of Sentence) — символ конца предложения.

2.4. Консольный интерфейс

Моделью можно управлять через консоль. Например, с помощью команды `python main.py --train` можно запустить обучение модели. А через команду `python main.py --load` можно загрузить уже обученную модель (скачать уже обученную модель можно из репозитория на GitHub [9] в разделе «Releases»). Также можно посмотреть инструкцию к использованию модели через `python main.py --help`, на рис.2.1 показан частичный вывод этой команды.

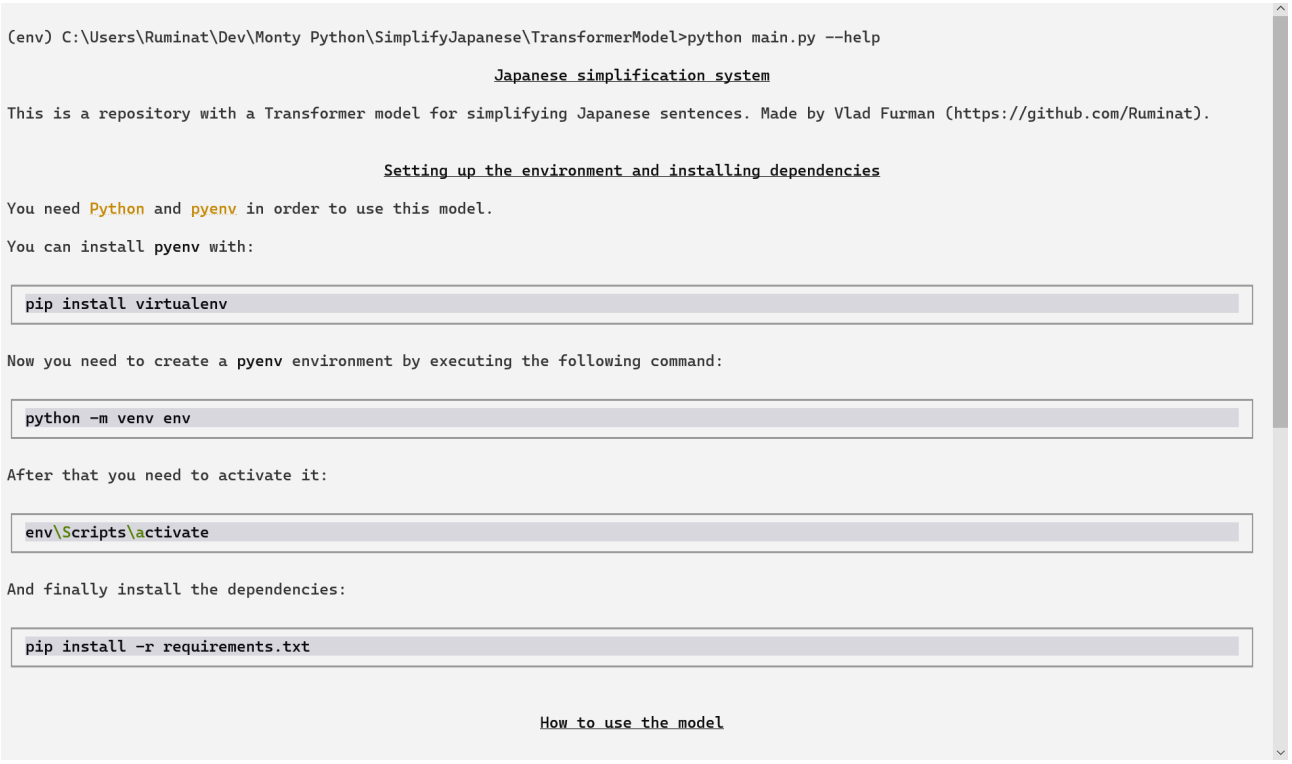

Рис.2.1. Частичный вывод команды `python main.py --help`

Поддерживаются также следующие флаги для команды `python main.py`:

1. `--version` или просто `--v` — выводит текущую версию системы,
2. `--no-print` — отключает вывод упрощения тестовых предложений.

2.5. Сервер

Как уже было сказано ранее, сервер использует фреймворк Falcon. Имеется лишь один путь — `/processJapaneseText`, по которму пользовательское приложение передаёт запрос с японским текстом, после чего сервер, используя функции `getMeCabTokens` и `transformer.translate`, упрощает это предложение и возвращает результат приложению.

2.5.1. Настройка сервера

Чтобы использовать сервер, его необходимо сначала настроить. Весь процесс подготовки окружения для сервера описан в `README.md` репозитория [9]. Нужно сделать следующее:

1. `pip install virtualenv` — установить virtual env,
2. `python -m venv env` — создать virtual env,
3. `env\Scripts\activate` — активировать virtual env,

4. `pip install -r requirements.txt` — установить необходимые зависимости.

Теперь сервер может быть запущен командами:

– `python main.py --server` — в режиме разработки,
– `waitress-serve --port=8000 server:app` — в production-окружении.

2.5.2. Как «общаться» с сервером

Как уже было сказано ранее, сервер имеет путь `/processJapaneseText`. Вот пример запроса по этому пути через консольную утилиту `curl`:

`curl http://localhost:5000/processJapaneseText?text=お前はもう死んでいる`

Пример ответа показан на рис.2.2, где можно увидеть, что возвращается JSON со следующими полями:

– `originalText` — исходное предложение,
– `simplifiedText` — упрощённое предложение,
– `originalTextTokens` — исходные токены с частями речи,
– `simplifiedTextTokens` — упрощённые токены с частями речи.


```
{
  "originalText": "お前はもう死んでいる",
  "simplifiedText": "あなたはもう死んでいる",
  "originalTextTokens": [
    { "partOfSpeech": "代名詞", "token": "お前" },
    { "partOfSpeech": "助詞-係助詞", "token": "は" },
    { "partOfSpeech": "副詞", "token": "もう" },
    { "partOfSpeech": "動詞-一般", "token": "死ん" },
    { "partOfSpeech": "助詞-接続助詞", "token": "で" },
    { "partOfSpeech": "動詞-非自立可能", "token": "いる" }
  ],
  "simplifiedTextTokens": [
    { "partOfSpeech": "代名詞", "token": "あなた" },
    { "partOfSpeech": "助詞-係助詞", "token": "は" },
    { "partOfSpeech": "副詞", "token": "もう" },
    { "partOfSpeech": "動詞-一般", "token": "死ん" },
    { "partOfSpeech": "助詞-接続助詞", "token": "で" },
    { "partOfSpeech": "動詞-非自立可能", "token": "いる" }
  ]
}
```


Рис.2.2. Пример ответа на запрос `/processJapaneseText`

2.6. Пользовательское приложение

Пользовательское приложение выглядит довольно минималистично — есть форма ввода предложения, после нажатия на кнопку «Simplify» или нажатия

«Ctrl» + «Enter» отправляется запрос на сервер (по пути `/processJapaneseText`), после чего ответ отображается в виде, как на рис.2.3.

Есть также возможность посмотреть перевод предложений (исходного и упрощённого) — при нажатии на ссылку «translation» открывается страница Google Translate с выбранным предложением — это может использоваться как некая опорная линия упрощения (что смысл не потерялся).

Цвета в предложениях (исходном и упрощённом) на рис.2.3 указывают на часть речи какого-либо токена (слова).

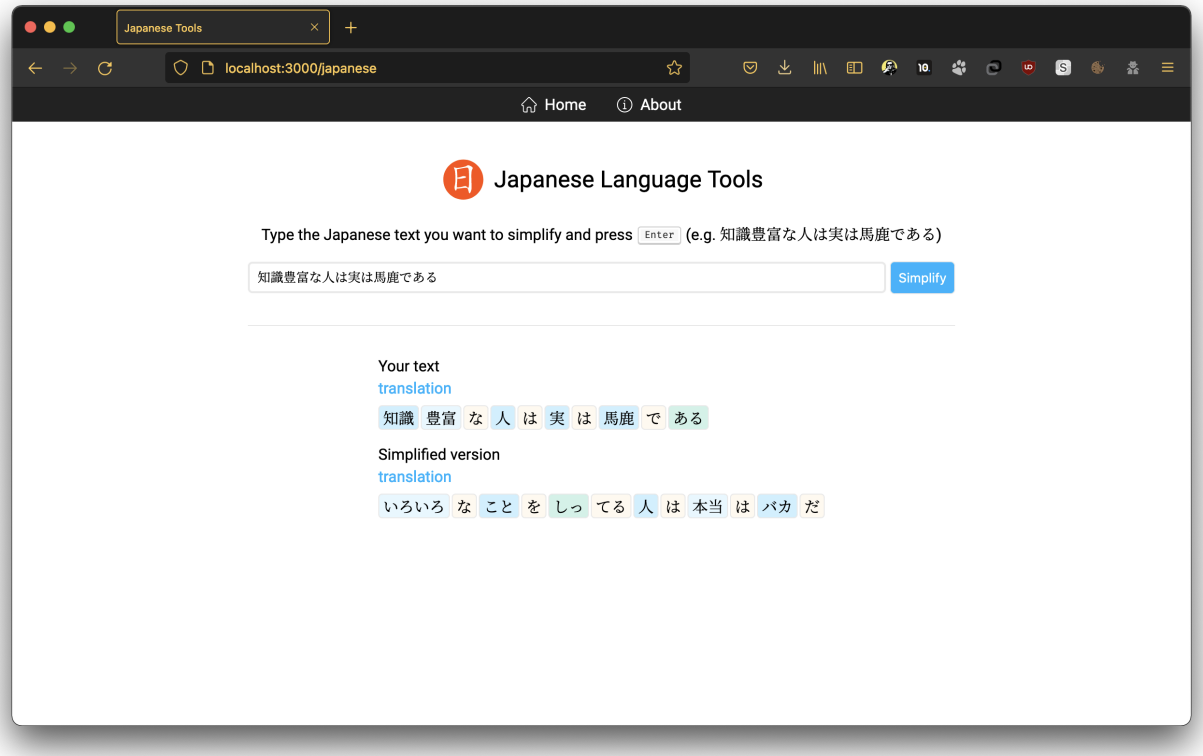

Рис.2.3. Пользовательское приложение

ЗАКЛЮЧЕНИЕ

В данной научно-исследовательской работе была выполнена поставленная цель — окончательная программная реализация системы для упрощения предложений на японском языке. Для достижения этой цели были выполнены следующие задачи:

- описание архитектуры реализуемой системы, обзор инструментов для разработки;
- обзор архитектуры модели Transformer, как с теоритической, так и с практической точек зрения;
- описание реализации системы упрощения (модель, сервер и приложение);
- демонстрация примеров работы разработанной системы на предложениях как из корпуса, так и вне корпуса.

Задачей для следующего этапа является исследование разработанной системы и, возможно, её оптимизация.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Falcon. Python web framework. — URL: https://falcon.readthedocs.io/en/stable/ (дата обращения: 06.02.2022).

2. HuggingFace. The AI community building the future. — URL: https://huggingface.co (дата обращения: 20.03.2022).

3. Lit. JS web-components library. — URL: https://lit.dev (дата обращения: 06.02.2022).

4. MeCab. Japanese language tokenization. — URL: https://github.com/taku910/mecab (дата обращения: 21.05.2021).

5. PyTorch. Python machine learning framework. — URL: https://pytorch.org (дата обращения: 06.02.2022).

6. Spacy. Industrial-Strength Natural Language Processing. — URL: https://spacy.io/ (дата обращения: 06.02.2022).

7. TypeScript. Superset of JavaScript with types. — URL: https://www.typescriptlang.org (дата обращения: 06.02.2022).

8. Репозиторий разработанного пользовательского приложения. — URL: https://github.com/Ruminat/experiments (дата обращения: 20.03.2022).

9. Репозиторий разработанных модели с сервером. — URL: https://github.com/Ruminat/japanese-simplification (дата обращения: 20.03.2022).

10. Attention Is All You Need / A. Vaswani [и др.]. — 2017. — arXiv: 1706.03762 [cs.CL].

11. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding / J. Devlin [и др.] // CoRR. — 2018. — Т. abs/1810.04805. — arXiv: 1810.04805. — URL: http://arxiv.org/abs/1810.04805.

12. *Glorot X.*, *and Y. B.* Understanding the difficulty of training deep feedforward neural networks // Journal of Machine Learning Research. — 2010. — eprint: 9:249-256. — URL: https://www.researchgate.net/publication/215616968_Understanding_the_difficulty_of_training_deep_feedforward_neural_networks.

13. *Maruyama T.*, *Yamamoto K.* Extremely Low Resource Text simplification with Pre-trained Transformer Language Model. — 2019. — DOI 10.1109/IALP48816.2019.9037650.

14. *Maruyama T.*, *Yamamoto K.* Simplified Corpus with Core Vocabulary // Proceedings of the Eleventh International Conference on Language Resources and

Evaluation (LREC 2018). — Miyazaki, Japan: European Language Resources Association (ELRA), 2018. — URL: https://www.aclweb.org/anthology/L18-1185.

15. *Батура Т. В.* Математическая лингвистика и автоматическая обработка текстов на естественном языке. — Новосибирск: Новосиб. гос. ун-т., 2016. — 166 с.

Приложение 1

Исходный код

Файл `main.py` (главный файл для запуска обучения модели или старта сервера):

```
import sys
from importlib import import_module

__version__ = "0.0.1"
hadErrors = False

try:
  from rich.console import Console
  console = Console()
except ImportError as e:
  print("Module `Console` is not installed!")
  class MyConsole:
    def print(content):
      print(content)
  console = MyConsole
  hadErrors = True

try:
  from rich.markdown import Markdown
except ImportError as e:
  print("Module `Markdown` is not installed!")
  Markdown = str
  hadErrors = True

def printHelp():
  with open("./README.md", encoding = 'utf-8') as readme:
    console.print(Markdown(readme.read()))
  if (hadErrors):
    print("""
      --> You probably didn't follow the instructions above, <--
      --> so be sure to check them out                        <--
    """)

if (hadErrors or "--help" in sys.argv or "--h" in sys.argv):
  printHelp()
elif ("--version" in sys.argv or "--v" in sys.argv):
  print(f"Current version is {__version__}")
elif ("--server" in sys.argv or "--serve" in sys.argv):
  print("Starting the server...\n")
  serverModule = import_module("apps.SimplificationServer.main")
  app = getattr(serverModule, "startSimplificationServerApp")
  app()
else:
  print("Loading the transformer model...\n")
  transformerModule = import_module("apps.Transformer.main")
  app = getattr(transformerModule, "startTransformerApp")
  app()
```

Файл `definitions.py` (основные определения и константы):

```
import torch
```

```
from modules.Dataset.snowSimplifiedJapanese.main import \
    snowSimplifiedJapaneseDataset
from modules.Language.definitions import JAPANESE_SIMPLIFIED, JAPANESE_SOURCE
from modules.Language.utils import (getTextTransform, getTokenTransform,
                                    getVocabTransform)


# You can put a trained model into MODELS_DIR with file name DEFAULT_MODEL_FILENAME
# so you won't have to train it each time
MODELS_DIR = "./build"
DEFAULT_MODEL_FILENAME = "transformer.pt"

# Source and target languages for translation (Japanese and simplified Japanese)
SRC_LANGUAGE = JAPANESE_SOURCE
TGT_LANGUAGE = JAPANESE_SIMPLIFIED

# The dataset we're gonna train the model on
DATASET = snowSimplifiedJapaneseDataset

# Transforms for the source sentence (text -> embeddings)
tokenTransform = getTokenTransform(SRC_LANGUAGE, TGT_LANGUAGE)
vocabTransform = getVocabTransform(SRC_LANGUAGE, TGT_LANGUAGE, tokenTransform, DATASET)
textTransform = getTextTransform(SRC_LANGUAGE, TGT_LANGUAGE, tokenTransform, vocabTransform)

# The seed for PyTorch
SEED = 0
# Batch size for learning
BATCH_SIZE = 64

# Optimizer parameters
WEIGHT_DECAY = 0
LEARNING_RATE = 0.0001
BETAS = (0.9, 0.98)
EPSILON = 1e-9

# The number of training epochs
NUM_EPOCHS = 30
# The size of the embedding vectors
EMB_SIZE = 512
# The number of attention heads
NHEAD = 8
# Size of the feed forward layer
DIM_FEEDFORWARD = 512
# The number of encoder/decoder layers
NUM_ENCODER_LAYERS = 6
NUM_DECODER_LAYERS = NUM_ENCODER_LAYERS

# Supplementary constants
SRC_VOCAB_SIZE = len(vocabTransform[SRC_LANGUAGE])
TGT_VOCAB_SIZE = len(vocabTransform[TGT_LANGUAGE])
DEVICE_CPU = "cpu"
DEVICE_GPU = "cuda"
# Which device to use for training/evaluation (uses CUDA when available, otherwise CPU)
DEVICE = torch.device(DEVICE_GPU if torch.cuda.is_available() else DEVICE_CPU)
```

Файл `utils.py` (утилиты для запуска модели):

```
import torch

from definitions import (BATCH_SIZE, BETAS, DATASET, DEFAULT_MODEL_FILENAME,
                         DEVICE, DIM_FEEDFORWARD, EMB_SIZE, EPSILON,
                         LEARNING_RATE, MODELS_DIR, NHEAD, NUM_DECODER_LAYERS,
```

```
                        NUM_EPOCHS, SEED, SRC_LANGUAGE, SRC_VOCAB_SIZE,
                        TGT_LANGUAGE, TGT_VOCAB_SIZE, WEIGHT_DECAY, textTransform)
from modules.Language.definitions import PAD_IDX
from modules.Seq2SeqTransformer.main import Seq2SeqTransformer
from modules.Seq2SeqTransformer.utils import (initializeTransformerParameters,
                                              train)


def initiatePyTorch() -> None:
  torch.manual_seed(SEED)
  torch.cuda.empty_cache()
  print(f"Running PyTorch on {DEVICE} with seed={SEED}")

def prettyPrintTranslation(transformer: Seq2SeqTransformer, sourceSentence: str) -> None:
  src = f"«{sourceSentence.strip()}»"
  result = f"«{transformer.translate(sourceSentence).strip()}»"
  print("Translating:", src, "->", result)

def loadTransformer(fileName: str = DEFAULT_MODEL_FILENAME) -> Seq2SeqTransformer:
  print("Loading Transformer...")
  transformer = torch.load(f"{MODELS_DIR}/{fileName}")
  transformer.eval()
  print("Transformer is ready to use")
  return transformer

def getTrainedTransformer(fileName: str = DEFAULT_MODEL_FILENAME) -> Seq2SeqTransformer:
  transformer = Seq2SeqTransformer(
    batchSize=BATCH_SIZE,
    srcLanguage=SRC_LANGUAGE,
    tgtLanguage=TGT_LANGUAGE,
    num_encoder_layers = NUM_DECODER_LAYERS,
    num_decoder_layers = NUM_DECODER_LAYERS,
    embeddingSize = EMB_SIZE,
    nhead = NHEAD,
    srcVocabSize = SRC_VOCAB_SIZE,
    tgtVocabSize = TGT_VOCAB_SIZE,
    dim_feedforward = DIM_FEEDFORWARD,
    device=DEVICE
  )
  print("Created the Transformer model")
  initializeTransformerParameters(transformer)
  print("Initialized parameters")

  transformer = transformer.to(DEVICE)
  lossFn = torch.nn.CrossEntropyLoss(ignore_index=PAD_IDX)
  optimizer = torch.optim.Adam(
    transformer.parameters(),
    lr=LEARNING_RATE,
    weight_decay=WEIGHT_DECAY,
    betas=BETAS,
    eps=EPSILON
  )
  print("Created lossFn and optimizer")

  print("Training the model...")
  train(transformer, optimizer, lossFn, textTransform, NUM_EPOCHS, DATASET)
  print("The model has trained well")

  torch.save(transformer, f"{MODELS_DIR}/{fileName}")

  return transformer
```

Файл apps/SimplificationServer/main.py (сервер на Falcon):

```
from flask import Flask, request, jsonify
from flask_cors import cross_origin

from modules.Parser.utils import getMeCabTokens
from utils import initiatePyTorch, loadTransformer

def startSimplificationServerApp():
  app = Flask(__name__)

  initiatePyTorch()
  transformer = loadTransformer()

  @app.route("/processJapaneseText", methods=["GET"])
  @cross_origin()
  def getProcessJapaneseText():
    text = request.args.get("text").strip()

    if (text == ""):
      return jsonify({
        "originalText": "",
        "simplifiedText": "",
        "originalTextTokens": [],
        "simplifiedTextTokens": []
      })

    simplifiedText = transformer.translate(text)
    textTokens = getMeCabTokens(text)
    simplifiedTextTokens = getMeCabTokens(simplifiedText)

    return jsonify({
      "originalText": text,
      "simplifiedText": simplifiedText,
      "originalTextTokens": textTokens,
      "simplifiedTextTokens": simplifiedTextTokens
    })

  app.run()
```

Файл modules/Parser/definitions.py (MeCab-токен):

```
from typing import Optional

class MeCabToken:
  def __init__(self, token: str, partOfSpeech: Optional[str] = None):
    self.token = token
    self.partOfSpeech = partOfSpeech
```

Файл modules/Parser/utils.py (токенизация через MeCab):

```
import MeCab

wakati = MeCab.Tagger("-Owakati")
tagger = MeCab.Tagger()

# Text to MeCab tokens:
# "" -> [{ "token": "", "partOfSpeech": "" }]
def getMeCabTokens(text: str) -> list[dict]:
  result = []
  for tokenParts in tagger.parse(text).split("\n"):
```

```
    parts = tokenParts.split("\t")
    token = parts[0]

    if token == "" or token == "EOS":
      continue

    resultToken = {}
    resultToken["token"] = token

    if len(parts) >= 5:
      resultToken["partOfSpeech"] = parts[4]

    result.append(resultToken)
  return result
```

Файл `apps/Transformer/main.py` (запуск обучения/загрузки модели из сохранённого файла из директории `/build`):

```
import sys

from definitions import DATASET
from modules.Language.definitions import JAPANESE_SIMPLIFIED, JAPANESE_SOURCE
from modules.Metrics.bleu import getBleuScore
from utils import (getTrainedTransformer, initiatePyTorch, loadTransformer,
                   prettyPrintTranslation)


def startTransformerApp():
  initiatePyTorch()

  if ("--train" in sys.argv):
    print("\n-- TRAIN MODE --\n")
    transformer = getTrainedTransformer()
  elif ("--load" in sys.argv):
    print("\n-- LOADING THE SAVED MODEL --\n")
    transformer = loadTransformer()
  else:
    print("\n-- DEFAULT (TRAIN) MODE --\n")
    transformer = getTrainedTransformer()

  # -- Testing the model --
  blueScore = getBleuScore(transformer, DATASET, JAPANESE_SOURCE, JAPANESE_SIMPLIFIED)
  print(f"BLEU score: {blueScore}")

  if ("--no-print" not in sys.argv):
    print("\nSentences that are not in the dataset\n")

    prettyPrintTranslation(transformer, "")
    prettyPrintTranslation(transformer, "")
    prettyPrintTranslation(transformer, "")
    prettyPrintTranslation(transformer, "")
    prettyPrintTranslation(transformer, "")

    print("\nSentences from the dataset\n")

    prettyPrintTranslation(transformer, "")
```

Файл `modules/Seq2SeqTransformer/main.py` (модель sequence2sequence Transformer — финальная модель для упрощения):

```
from typing import List
import torch
import torch.nn as nn
from definitions import (SRC_LANGUAGE, TGT_LANGUAGE, textTransform,
                         vocabTransform)
from modules.Embedding.main import TokenEmbedding
from modules.Language.definitions import BOS_IDX, BOS_SYMBOL, EOS_SYMBOL
from modules.PositionalEncoding.main import PositionalEncoding
from modules.Seq2SeqTransformer.utils import greedyDecode
from torch import Tensor

from torch.nn import Transformer


# The final model to be trained
class Seq2SeqTransformer(nn.Module):
  def __init__(
    self,
    batchSize: int,
    srcLanguage: str,
    tgtLanguage: str,
    num_encoder_layers: int,
    num_decoder_layers: int,
    embeddingSize: int,
    nhead: int,
    srcVocabSize: int,
    tgtVocabSize: int,
    dim_feedforward: int = 512,
    dropout: float = 0.1,
    device: torch.device = torch.device("cpu")
  ):
    super(Seq2SeqTransformer, self).__init__()
    self.transformer = Transformer(
      d_model=embeddingSize,
      nhead=nhead,
      num_encoder_layers=num_encoder_layers,
      num_decoder_layers=num_decoder_layers,
      dim_feedforward=dim_feedforward,
      dropout=dropout
    )
    self.generator = nn.Linear(embeddingSize, tgtVocabSize)
    self.src_tok_emb = TokenEmbedding(srcVocabSize, embeddingSize)
    self.tgt_tok_emb = TokenEmbedding(tgtVocabSize, embeddingSize)
    self.positional_encoding = PositionalEncoding(embeddingSize, dropout=dropout)
    self.batchSize = batchSize
    self.srcLanguage = srcLanguage
    self.tgtLanguage = tgtLanguage
    self.device = device

  def forward(
    self,
    src: Tensor,
    trg: Tensor,
    srcMask: Tensor,
    tgtMask: Tensor,
    srcPaddingMask: Tensor,
    tgtPaddingMask: Tensor,
    memory_key_padding_mask: Tensor
  ):
    srcEmbedding = self.positional_encoding(self.src_tok_emb(src))
    tgtEmbedding = self.positional_encoding(self.tgt_tok_emb(trg))
    outs = self.transformer(
```

```
      srcEmbedding,
      tgtEmbedding,
      srcMask,
      tgtMask,
      None,
      srcPaddingMask,
      tgtPaddingMask,
      memory_key_padding_mask
    )
    return self.generator(outs)

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

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

  # Method for translating from srcLanguage to tgtLangauge (Japanese -> simplified Japanese)
  def translate(self, srcSentence: str):
    self.eval()
    src = textTransform[SRC_LANGUAGE](srcSentence).view(-1, 1)
    num_tokens = src.shape[0]
    srcMask = (torch.zeros(num_tokens, num_tokens)).type(torch.bool)
    tgtTokens = greedyDecode(
      self,
      src,
      srcMask,
      maxLen=num_tokens + 5,
      startSymbol=BOS_IDX,
      device=self.device
    ).flatten()
    tokens = vocabTransform[TGT_LANGUAGE].lookup_tokens(list(tgtTokens.cpu().numpy()))

    return self.tokensToText(tokens)

  # Turns a list of tokens into a single string
  # ["what", "is", "love"] -> "what is love"
  def tokensToText(self, tokens: List[str]) -> str:
    result = ""
    for token in tokens:
      if token == BOS_SYMBOL or token == EOS_SYMBOL:
        continue
      if token == "":
        break
      result += token
    return result
```

Файл `modules/Seq2SeqTransformer/utils.py` (утилиты для обучения модели упрощения):

```
from timeit import default_timer as timer

import torch
from modules.Dataset.main import MyDataset
```

```
from modules.Language.definitions import BOS_IDX, PAD_IDX
from modules.Language.utils import getCollateFn
from torch import Tensor, nn
from torch.utils.data import DataLoader


# Generates the following mask:
# tensor([[0., -inf, -inf, -inf, -inf],
#         [0.,   0., -inf, -inf, -inf],
#         [0.,   0.,   0., -inf, -inf],
#         [0.,   0.,   0.,   0., -inf],
#         [0.,   0.,   0.,   0.,   0.]])
def generateSquareSubsequentMask(size: int, device: str) -> Tensor:
  mask = (torch.triu(torch.ones((size, size), device=device)) == 1).transpose(0, 1)
  mask = mask.float().masked_fill(mask == 0, float("-inf"))
  mask = mask.masked_fill(mask == 1, float(0.0))
  return mask

# Generates the seq2seq transformer masks
def createMask(src: Tensor, tgt: Tensor, device: str):
  src_seq_len = src.shape[0]
  tgt_seq_len = tgt.shape[0]

  tgt_mask = generateSquareSubsequentMask(tgt_seq_len, device)
  src_mask = torch.zeros((src_seq_len, src_seq_len), device=device).type(torch.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

# Initializes the Transformer's parameters with the Glorot initialization.
def initializeTransformerParameters(transformer: nn.Module) -> None:
  for parameter in transformer.parameters():
    if parameter.dim() > 1:
      nn.init.xavier_uniform_(parameter)

# Function to generate output sequence (simplified sentence) using the greedy algorithm.
def greedyDecode(model, src, srcMask, maxLen, startSymbol, device: str) -> Tensor:
  src = src.to(device)
  srcMask = srcMask.to(device)

  memory = model.encode(src, srcMask)
  ys = torch.ones(1, 1).fill_(startSymbol).type(torch.long).to(device)
  for i in range(maxLen - 1):
    memory = memory.to(device)
    tgtMaskBase = generateSquareSubsequentMask(ys.size(0), device)
    tgtMask = (tgtMaskBase.type(torch.bool)).to(device)
    out = model.decode(ys, memory, tgtMask)
    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 == BOS_IDX:
      break
  return ys

# Evaluates the model.
def evaluate(
  model: torch.nn,
  lossFn: torch.nn,
```

```
  textTransform: dict,
  dataset: MyDataset
) -> float:
  model.eval()
  losses = 0

  valIter = dataset.getValidationSplit()
  collateFn = getCollateFn(model.srcLanguage, model.tgtLanguage, textTransform)
  valDataloader = DataLoader(valIter, batch_size=model.batchSize, collate_fn=collateFn)

  for src, tgt in valDataloader:
    src = src.to(model.device)
    tgt = tgt.to(model.device)

    tgtInput = tgt[:-1, :]
    srcMask, tgtMask, srcPaddingMask, tgtPaddingMask = createMask(
      src,
      tgtInput,
      device=model.device
    )
    logits = model(
      src,
      tgtInput,
      srcMask,
      tgtMask,
      srcPaddingMask,
      tgtPaddingMask,
      srcPaddingMask
    )

    tgt_out = tgt[1:, :]
    loss = lossFn(logits.reshape(-1, logits.shape[-1]), tgt_out.reshape(-1))
    losses += loss.item()

  return losses / len(valDataloader)

# Trains the model
def train(
  model: torch.nn,
  optimizer: torch.optim,
  lossFn: torch.nn,
  textTransform: dict,
  epochs: int,
  dataset: MyDataset
) -> None:
  bestValue = 1729
  bestValueEpoch = 1

  for epoch in range(1, epochs + 1):
    startTime = timer()
    trainLoss = trainEpoch(model, optimizer, lossFn, textTransform, dataset)
    endTime = timer()
    trainTime = endTime - startTime

    startTime = timer()
    valueLoss = evaluate(model, lossFn, textTransform, dataset)
    endTime = timer()
    evaluationTime = endTime - startTime

    trainLossPrint = f"train loss: {trainLoss:.3f} ({trainTime:.1f}s)"
    valLossPrint = f"val loss: {valueLoss:.3f} ({evaluationTime:.1f})"
    print((f"epoch-{epoch}: ${trainLossPrint}, ${valLossPrint}"))
```

```
    if (valueLoss < bestValue):
      bestValue = valueLoss
      bestValueEpoch = epoch
    if (epoch - bestValueEpoch > 3):
      print("The model stopped improving, so we stop the learning process.")
      break

# One train epoch
def trainEpoch(
  model: torch.nn,
  optimizer: torch.optim,
  lossFn: torch.nn,
  textTransform: dict,
  dataset: MyDataset
) -> float:
  model.train()
  losses = 0
  train_iter = dataset.getTrainSplit()
  collateFn = getCollateFn(model.srcLanguage, model.tgtLanguage, textTransform)
  trainSplit = DataLoader(train_iter, batch_size=model.batchSize, collate_fn=collateFn)

  for src, tgt in trainSplit:
    src = src.to(model.device)
    tgt = tgt.to(model.device)

    tgtInput = tgt[:-1, :]
    srcMask, tgtMask, srcPaddingMask, tgtPaddingMask = createMask(
      src,
      tgtInput,
      device=model.device
    )
    logits = model(
      src,
      tgtInput,
      srcMask,
      tgtMask,
      srcPaddingMask,
      tgtPaddingMask,
      srcPaddingMask
    )

    optimizer.zero_grad()

    tgt_out = tgt[1:, :]
    loss = lossFn(logits.reshape(-1, logits.shape[-1]), tgt_out.reshape(-1))
    loss.backward()

    optimizer.step()
    losses += loss.item()

  return losses / len(trainSplit)
```

Файл `modules/PositionalEncoding/main.py` (модель positional encoding):

```
import math

import torch
import torch.nn as nn
from torch import Tensor
```

```
# helper Module that adds positional encoding to the token embedding
# to introduce a notion of word order
class PositionalEncoding(nn.Module):
  def __init__(
    self,
    embeddingSize: int,
    dropout: float,
    maxlen: int = 5000
  ):
    super(PositionalEncoding, self).__init__()
    den = torch.exp(-torch.arange(0, embeddingSize, 2) * math.log(10000) / embeddingSize)
    pos = torch.arange(0, maxlen).reshape(maxlen, 1)
    posEmbedding = torch.zeros((maxlen, embeddingSize))
    posEmbedding[:, 0::2] = torch.sin(pos * den)
    posEmbedding[:, 1::2] = torch.cos(pos * den)
    posEmbedding = posEmbedding.unsqueeze(-2)

    self.dropout = nn.Dropout(dropout)
    self.register_buffer("posEmbedding", posEmbedding)

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

Файл `modules/Embedding/main.py` (модель эмбеддингов):

```
import math

import torch.nn as nn
from torch import Tensor


# 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)
```

Файл `modules/Language/definitions.py` (определения языков):

```
# Languages for src/tgt
JAPANESE_SOURCE = "original_ja"
JAPANESE_SIMPLIFIED = "simplified_ja"

# Special symbols
UNK_SYMBOL = "<unk>" # unknown symbol
PAD_SYMBOL = "<pad>" # padding symbol
BOS_SYMBOL = "<bos>" # «Begginning Of a Sentence» symbol
EOS_SYMBOL = "<eos>" # «End Of a Sentence» symbol

# Indicies for special symbols
UNK_IDX = 0
PAD_IDX = 1
BOS_IDX = 2
EOS_IDX = 3
# Make sure the tokens are in order of their indices to properly insert them in vocab
SPECIAL_SYMBOLS = [UNK_SYMBOL, PAD_SYMBOL, BOS_SYMBOL, EOS_SYMBOL]
```

```
# Spacy dataset for Japanese
SPACY_JP = "ja_core_news_lg"

# Converts a language to a Spacy dataset
LANGUAGE_TO_SPACY_DATASET = {
  JAPANESE_SOURCE: SPACY_JP,
  JAPANESE_SIMPLIFIED: SPACY_JP
}
```

Файл `modules/Language/utils.py` (утилиты для обработки языков):

```
from typing import Iterable, List

import torch
from modules.Dataset.main import MyDataset
from modules.Language.definitions import (BOS_IDX, EOS_IDX,
                                          LANGUAGE_TO_SPACY_DATASET, PAD_IDX,
                                          SPECIAL_SYMBOLS, UNK_IDX)
from torch.nn.utils.rnn import pad_sequence
from torchtext.data.utils import get_tokenizer
from torchtext.vocab import build_vocab_from_iterator


# Returns a Spacy tokenizer for the given language
def getSpacyTokenizer(language: str):
  return get_tokenizer("spacy", language=LANGUAGE_TO_SPACY_DATASET[language])

# Create source and target language tokenizer. Make sure to install the dependencies.
# pip install -U spacy
# python -m spacy download ja_core_news_lg
def getTokenTransform(srcLanguage: str, tgtLanguage: str):
  return {
    srcLanguage: getSpacyTokenizer(srcLanguage),
    tgtLanguage: getSpacyTokenizer(tgtLanguage),
  }

def getVocabTransform(
  srcLanguage: str,
  tgtLanguage: str,
  tokenTransform: dict,
  dataset: MyDataset
):
  vocabTransform = {}
  for language in [srcLanguage, tgtLanguage]:
    # Training data Iterator
    trainIter = dataset.getTrainSplit()
    # Create torchtext"s Vocab object
    vocabTransform[language] = build_vocab_from_iterator(
      yieldTokens(tokenTransform, trainIter, language),
      min_freq=1,
      specials=SPECIAL_SYMBOLS,
      special_first=True
    )

  # Set UNK_IDX as the default index. This index is returned when the token is not found.
  # If not set, it throws RuntimeError when the queried token is not found in the Vocabulary.
  for language in [srcLanguage, tgtLanguage]:
    vocabTransform[language].set_default_index(UNK_IDX)

  return vocabTransform
```

```
# Function to iterate through tokens
def yieldTokens(tokenTransform: dict, dataIter: Iterable, language: str) -> List[str]:
  tokenizer = tokenTransform[language]
  for dataSample in dataIter:
    sample = dataSample[language]
    yield tokenizer(sample)

# src and tgt language text transforms to convert raw strings into tensors indices
def getTextTransform(
  srcLanguage: str,
  tgtLanguage: str,
  tokenTransform: dict,
  vocabTransform: dict
):
  textTransform = {}
  for language in [srcLanguage, tgtLanguage]:
    textTransform[language] = sequentialTransforms(
      tokenTransform[language], # Tokenization
      vocabTransform[language], # Numericalization
      tensorTransform
    ) # Add BOS/EOS and create tensor
  return textTransform

# helper function to club together sequential operations
def sequentialTransforms(*transforms):
  def func(textInput: str):
    for transform in transforms:
      textInput = transform(textInput)
    return textInput
  return func

# function to add BOS/EOS and create tensor for input sequence indices
def tensorTransform(tokenIds: List[int]):
  return torch.cat(
    (
      torch.tensor([BOS_IDX]),
      torch.tensor(tokenIds),
      torch.tensor([EOS_IDX])
    )
  )

# function to collate data samples into batch tensors
def getCollateFn(srcLanguage: str, tgtLanguage: str, textTransform: dict):
  def collateFn(batch):
    srcBatch, tgtBatch = [], []
    for sus in batch:
      srcSample, tgtSample = sus[srcLanguage], sus[tgtLanguage]
      srcBatch.append(textTransform[srcLanguage](srcSample.rstrip("\n")))
      tgtBatch.append(textTransform[tgtLanguage](tgtSample.rstrip("\n")))

    srcBatch = pad_sequence(srcBatch, padding_value=PAD_IDX)
    tgtBatch = pad_sequence(tgtBatch, padding_value=PAD_IDX)
    return srcBatch, tgtBatch
  return collateFn
```

Файл `modules/Dataset/main.py` (класс корпуса):

```
from typing import Optional
from modules.Dataset.definitions import TDatasetFn


class MyDataset:
```

```
  def __init__(
    self,
    getTrainSplit: TDatasetFn,
    getValidationSplit: Optional[TDatasetFn],
    getTestSplit: Optional[TDatasetFn]
  ):
    self.getTrainSplit = getTrainSplit
    self.getValidationSplit = getValidationSplit
    self.getTestSplit = getTestSplit
```

Файл `modules/Dataset/definitions.py` (типы для корпусов):

```
from typing import Callable, Union

from datasets import Dataset, DatasetDict, IterableDataset, IterableDatasetDict

TDataset = Union[DatasetDict, Dataset, IterableDatasetDict, IterableDataset]

TDatasetFn = Callable[[], TDataset]
```

Файл `modules/Dataset/snowSimplifiedJapanese/main.py` (корпус упрощения предложений на японском языке SNOW (T15 и T23)):

```
import functools
from datasets import concatenate_datasets, load_dataset

from modules.Dataset.main import MyDataset

SNOW_DATASET = "snow_simplified_japanese_corpus"

SNOW_T15 = "snow_t15"
SNOW_T23 = "snow_t23"

VALIDATION_PERCENT = 5
TEST_PERCENT = 5

@functools.cache
def getTrainSplit():
  t15Dataset = load_dataset(SNOW_DATASET, SNOW_T15, split=f"train[{VALIDATION_PERCENT}%:]")
  t23Dataset = load_dataset(SNOW_DATASET, SNOW_T23, split=f"train[{TEST_PERCENT}%:]")
  return concatenate_datasets([t15Dataset, t23Dataset])

@functools.cache
def getValidationSplit():
  return load_dataset(SNOW_DATASET, SNOW_T15, split=f"train[:{VALIDATION_PERCENT}%]")

@functools.cache
def getTestSplit():
  return load_dataset(SNOW_DATASET, SNOW_T23, split=f"train[:{TEST_PERCENT}%]")

snowSimplifiedJapaneseDataset = MyDataset(
  getTrainSplit=getTrainSplit,
  getValidationSplit=getValidationSplit,
  getTestSplit=getTestSplit
)
```