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

Отчет о прохождении производственной (технологическая (проектно-технологическая)) практики

Фурман Владислав Константинович

(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 22.03.2022 по 17.04.2022

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:

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

Консультант практической подготовки
от ФГАОУ ВО «СПбПУ»: Пак В.Г.

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 какого-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 производственной практики является улучшение разработанной модели упрощения текстов на японском языке, а также оценка полученных решений.

Для достижения данной цели необходимо выполнить следующие задачи:

- планирование и реализация модификации разработанной системы;
- сбор метрик BLUE и SARI для оценки качества модели;
- сравнение упрощения различных предложений разработанной моделью, а также её модификацией.

ГЛАВА 1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ РАБОТЫ

1.1. Этапы обработки естественного языка

Как правило, в обработке текстов обычно выделяют следующие этапы [17, 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.): предназначены для обработки последовательностей (например, текста). На каждый слой передаётся текущий элемент (слово), а также результат предыдущего слоя. Причём есть обратные связи (поэтому рекуррентные). Они очень медленные — из-за последовательной природы их нельзя распараллелить.
- Долгая краткосрочная память (LTSM) (1997 г.). Это разновидность RNN с элементом «забывания». Они 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. Для этой цели используется так называемый Positional Encoding (см. (1.3) и (1.4)), который для каждого эмбеддинга добавляет информацию о расположении слов в предложении.

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

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

где

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

1.4.6. Обзор архитектуры Transformer'a

В оригинальной статье [14] представлена следующая архитектура Transformer'a (см. рис.1.1).

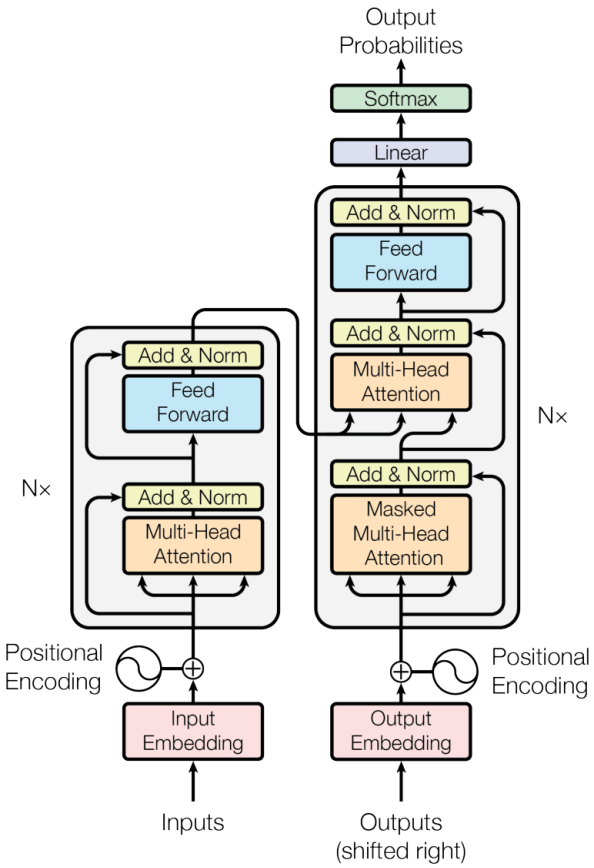

Рис.1.1. Архитектура Transformer'a

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 осмысленный 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 им пользоваться. 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 двух 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
```

```
13    if (mask is not None):
14      temp += mask
15
16    # Применяем softmax к измерению embedding'ов
17    softmax = f.softmax(temp, dim=-1)
18    # Перемножаем softmax с матрицей value
19    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, на главной диагонали которой и ниже «0», а выше — «$-\infty$». 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. (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 имеет вид, как в (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:

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

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

2.2.3. Positional Encoding

Реализовать positional encoding тоже не составляет большого труда:

```
def positionalEncoding(
  sequenceLength: int,
  dModel: int,
  device: torch.device
) -> Tensor:
  # Тензор [[[0.], [1.], [2.], ...]]
  pos = torch \
    .arange(sequenceLength, dtype=torch.float, device=device) \
    .reshape(1, -1, 1)
  # Тензор [[[0., 1., 2., ...]]]
  dim = torch.arange(dModel, dtype=torch.float, device=device).reshape(1, 1, -1)
  # Фаза (аргумент для cos/sin) =
  # [
  #   [
  #     [0., 0., 0., ...],
  #     [1., 1., 1., ...],
  #     [2., 2., 2., ...],
  #     ...
  #   ]
  # ]
  phase = pos / 10000 ** (dim // dModel)

  # [[[sin(...),  cos(...), sin(...),  cos(...), ...], ...]]
  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 [13] (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 [15]. 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.

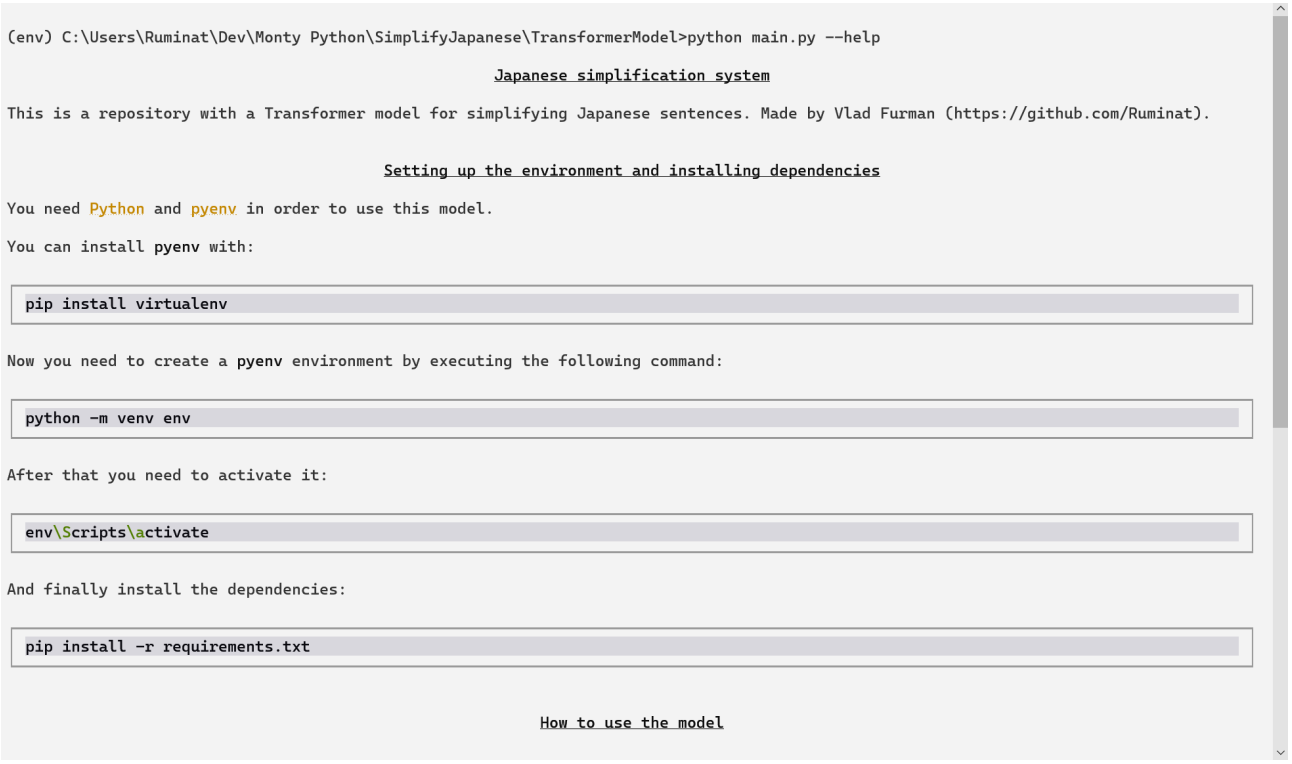

Рис.2.1. Частичный вывод команды `python main.py --help`

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 которому 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).

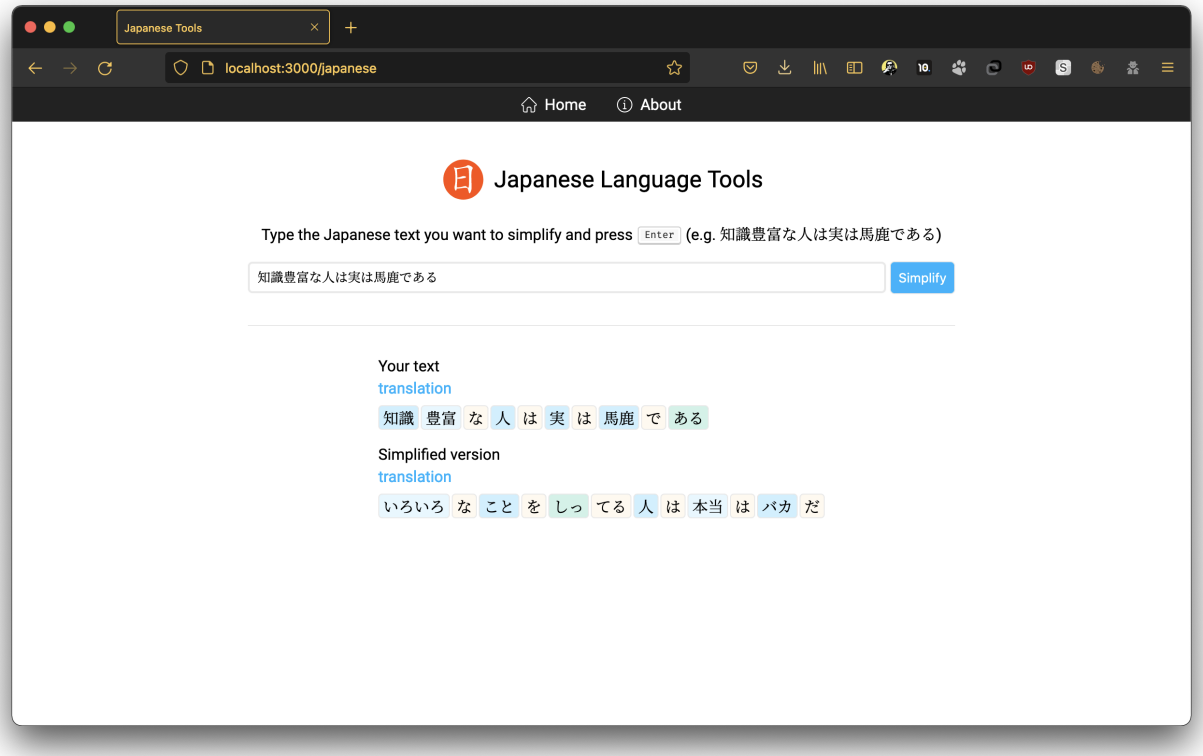

Рис.2.3. Пользовательское приложение

ГЛАВА 3. РЕЗУЛЬТАТЫ И УЛУЧШЕНИЕ МОДЕЛИ

3.1. Недостатки разработанной модели

Разработанная модель обладает следующими недостатками:

– плохо справляется с большими предложениями[3],
– имеет относительно небольшой «словарный запас».

Обе проблемы вызваны довольно маленьким корпусом (50 000 предложений для такой задачи — крайне малое количество). Самым простым и очевидным решением в такой ситуации было бы взять больший корпус, однако в открытом доступе он попросту отсутствует. Поэтому необходимо искать решение проблем в условиях очень небольшого корпуса.

3.2. Варианты модификации модели

Как мы уже говорили ранее, encoder в Transformer'е можно предобучить, чтобы он лучше кодировал входные последовательности слов предложений. Сделать это можно, например, следующим образом:

– взять корпус из предложений на японском языке (к примеру, предложения с Википедии[4]), сопоставить каждое предложение самому себе (то есть упрощение будет вестись в исходные предложения);
– обучить таким образом модель;
– получить encoder, который имеет какое-то представление о японском языке.

Может возникнуть вопрос: как же мы улучшим модель, если будем обучаться на корпусе, в котором никакого упрощения совсем нет? Секрет кроется в том, encoder не отвечает за само упрощение — он лишь кодирует предложения в матрицы чисел. Поэтому мы можем взять этот encoder и дальше уже обучать изначальную модель с его внедрением. В данной работе мы попробуем использовать 2 стратегии внедрения encoder'а в изначальную модель:

[3]На самом деле, эта проблема частично может быть решена разбиением предложения по запятым и отдельному упрощению каждой части, однако лучше, конечно, было бы иметь решение, способное справляться с большими предложениями.

[4]Корпус с предложениями для предобучения модели может быть найден в репозитории модели [9] — файл `modules/Dataset/wikipediaJp/data/wikipediaJp.csv`

1. взять предобученную модель «как есть» и обучить её на корпусе с упрощёнными предложениями;
2. взять изначальную модель и положить в неё предобученный encoder, сгенерировав остальные коэффициенты.

Стоит также отметить, что просто взяв предобученный encoder и положив его в Transformer, мы многого не добьёмся — модель попросту обучится на корпусе с упрощёнными предложениями и пользы от предобученнго encoder'а мы не получим. Чтобы этого не произошло мы уменьшим learning rate для слоёв encoder'а (в данной работе — в 5 раз[5]). В PyTorch это можно сделать следующим образом:

```
encoder = []
rest = []
for name, param in transformer.named_parameters():
  if "encoder" in name:
    encoder.append(param)
  else:
    rest.append(param)

optimizer = torch.optim.Adam(
  [{'params': encoder}, {'params': rest}],
  # ... параметры обучения
)
# здесь мы уменьшаем learning rate у encoder'а в 5 раз
optimizer.param_groups[0]['lr'] = LEARNING_RATE / 5
```

Таким образом мы значительно ограничиваем возможность изменения параметров encoder'а при обучении. Что позволяет нам воспользоваться преимуществом его предобучения.

3.3. Метрики для оценки модели упрощения

Для перевода текстов (в том числе и упрощения) довольно часто используют метрку BLEU [12]. В оригинальной статье Папинени и др. показали наличие корреляции данной метрики с сохранением грамматики и смысла переведённых предложений.

Однако есть и более специфичная метрика, разработанная специально для автоматического упрощения текстов — SARI [16]. Она, по сравнению BLEU, лучше коррелирует с упрощением предложений, а также с сохранением лексической и структурной частей предложений.

[5] Число 5 было найдено подбором: сначала была попытка с learning rate в 10, потом в 2, потом в 4 и, наконец, в 5 — что дало лучший результат для метрик.

3.4. Результаты

Результаты метрик BLUE и SARI для изначальной модели и вариантов её улучшения представлены в табл.3.1.

Таблица 3.1

Метрики полученных моделей

Модель	**BLEU**	**SARI**
Transformer	45,63	63,17
Pretrained Transformer	50,78	67,33
Pretrained Encoder	46,87	64,27

В табл.3.1 введены следующие обозначения:

- Transformer — изначальная модель;
- Pretrained Transformer — предобученная модель, которую дообучаем, замедляя обучение encoder'а;
- Pretrained Encoder — изначальная модель, в которой заменяем encoder на предобученный, замедляя его обучение.

По табл.3.1 видно, что наиболее успешной оказалась модель Pretrained Transformer — улучшение по сравнению с изначальной моделью на 5,15% и 4,16% для BLEU и SARI соответственно. Это может говорить о том, что предобучение положительно сказывается и на decoder'е, так как упрощение в основном оставляет предложение в исходном виде, за исключением упрощённых его частей.

Рассмотрим пример предложения из корпуса для тестирования, упрощение которого улучшилось благодаря модификции модели — см. рис.3.1[6].

[6]Здесь также происходит упрощение слова ますます (на さらに — более распространённое слово), однако обе модели упростили его одинакого, поэтому не будем заострять на этом внимание.

(1) исходное предложение

kanojo wa uchiki na node masumasu kanojo ga suki da
彼女 は 内気 な ので 、 ますます 彼女 が 好き だ

пер. — она робкая, из-за чего я люблю её ещё больше

(2) изначальная модель

kanojo wa **ki ga waruku omou node** sara ni kanojo ga suki da
彼女 は 気 が 悪く 思う ので 、 さら に 彼女 が 好き だ

пер. — она **плохо себя чувствует**, из-за чего я люблю её ещё больше

(3) модифицированная модель (Pretrained Transformer)

kanojo wa **ki ga yowai** node sara ni kanojo ga suki da
彼女 は 気 が 弱い ので 、 さら に 彼女 が 好き だ

пер. — она **скромная**, из-за чего я люблю её ещё больше

Рис.3.1. Пример улучшения упрощения предложения

ЗАКЛЮЧЕНИЕ

В данной производственной практике была достигнута поставленная цель — улучшение разработанной модели, а также оценка полученных решений:

- планирование и реализация модификации разработанной системы;
- сбор метрик BLUE и SARI для оценки качества модели;
- сравнение упрощения различных предложений разработанной моделью, а также её модификацией.

Задачей для следующего этапа является окончательная доработка модели, а также полноценное написание текста ВКР.

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. Bleu: a Method for Automatic Evaluation of Machine Translation / K. Papineni [и др.] // Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics. — Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, 2002. — С. 311—318. — DOI 10.3115/1073083.1073135. — URL: https://aclanthology.org/P02-1040.

13. *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.

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

15. *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.

16. Optimizing Statistical Machine Translation for Text Simplification //. T. 4. — 2016. — C. 401—415. — URL: https://www.aclweb.org/anthology/Q16-1029.

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