



**Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Московский государственный технический университет  
имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)**

---

ФАКУЛЬТЕТ Информатика, искусственный интеллект и системы управления  
КАФЕДРА Системы обработки информации и управления

**Домашнее задание  
по дисциплине «Методы машинного обучения»**

Выполнил: Гринин О.Е,  
Группа: ИУ5-22М

Москва, 2022 г.

## ЗАДАНИЕ

Домашнее задание по дисциплине направлено на анализ современных методов машинного обучения и их применение для решения практических задач. Домашнее задание включает три основных этапа:

1. выбор задачи;
2. теоретический этап;
3. практический этап.

Этап выбора задачи предполагает анализ ресурса `paperswithcode` [1]. Данный ресурс включает описание нескольких тысяч современных задач в области машинного обучения. Каждое описание задачи содержит ссылки на наиболее современные и актуальные научные статьи, предназначенные для решения задачи (список статей регулярно обновляется авторами ресурса). Каждое описание статьи содержит ссылку на репозиторий с открытым исходным кодом, реализующим представленные в статье эксперименты. На этапе выбора задачи обучающийся выбирает одну из задач машинного обучения, описание которой содержит ссылки на статьи и репозитории с исходным кодом.

Теоретический этап включает проработку как минимум двух статей, относящихся к выбранной задаче. Результаты проработки обучающийся излагает в теоретической части отчета по домашнему заданию, которая может включать:

- описание общих подходов к решению задачи;
- конкретные топологии нейронных сетей, нейросетевых ансамблей или других моделей машинного обучения, предназначенных для решения задачи;
- математическое описание, алгоритмы функционирования, особенности обучения используемых для решения задачи нейронных сетей, нейросетевых ансамблей или других моделей машинного обучения;
- описание наборов данных, используемых для обучения моделей;

- оценка качества решения задачи, описание метрик качества и их значений;
- предложения обучающегося по улучшению качества решения задачи.

Практический этап включает повторение экспериментов авторов статей на основе представленных авторами репозитория с исходным кодом и возможное улучшение обучающимися полученных результатов. Результаты проработки обучающийся излагает в практической части отчета по домашнему заданию, которая может включать:

- исходные коды программ, представленные авторами статей, результаты документирования программ обучающимися с использованием диаграмм UML, путем визуализации топологий нейронных сетей и другими способами;
- результаты выполнения программ, вычисление значений для описанных в статьях метрик качества, выводы обучающегося о воспроизводимости экспериментов авторов статей и соответствии практических экспериментов теоретическим материалам статей;
- предложения обучающегося по возможным улучшениям решения задачи, результаты практических экспериментов (исходные коды, документация) по возможному улучшению решения задачи.

Отчет по домашнему заданию должен содержать:

1. Титульный лист.
2. Постановку выбранной задачи машинного обучения, соответствующую этапу выбора задачи.
3. Теоретическую часть отчета.
4. Практическую часть отчета.
5. Выводы обучающегося по результатам выполнения теоретической и практической частей.
6. Список использованных источников.

# 1. ПОСТАНОВКА ЗАДАЧИ

В результате анализа содержимого ресурса «Papers with code» была выбрана область обработки естественных языков (NLP – Natural Language Processing). В данной области было решено изучить решения задачи распознавания именованных сущностей (NER – Named Entity Recognition)

Natural Language Processing

## Named Entity Recognition

Edit

600 papers with code • 59 benchmarks • 92 datasets

Named entity recognition (NER) is the task of tagging entities in text with their corresponding type. Approaches typically use BIOES-style notation, which differentiates the beginning (B) and the inside (I) of entities. O is used for non-entity tokens.

Example:

**Mark Watney visited Mars**  
B-PER I-PER O B-LOC

( Image credit: Zalando )

### Benchmarks

These leaderboards are used to track progress in Named Entity Recognition

Trend	Dataset	Best Model	Paper	Code	Compare
	CoNLL 2003 (English)	ACE + document-context			<a href="#">See all</a>
	Ontonotes v5 (English)	BERT-MRC+DSC			<a href="#">See all</a>
	NCBI-disease	Spark NLP			<a href="#">See all</a>

### Content

- Introduction
- Benchmarks
- Datasets
- Subtasks
- Libraries
- Papers
  - Most implemented
  - Social
  - Latest
  - No code

Задача NER – выделить спаны сущностей в тексте (спан – непрерывный фрагмент текста). Допустим, есть новостной текст, и мы хотим выделить в нем сущности (некоторый заранее зафиксированный набор — например, персоны, локации, организации, даты и так далее). Задача NER – понять, что участок текста “1 января 1997 года” является датой, “Кофи Аннан” – персоной, а “ООН” – организацией.

Есть несколько причин, почему NER является одной из самых популярных задач NLP.

Во-первых, извлечение именованных сущностей — это шаг в сторону “понимания” текста. Это может как иметь самостоятельную ценность, так и помочь лучше решать другие задачи NLP.

Так, если мы знаем, где в тексте выделены сущности, то мы можем найти важные для какой-то задачи фрагменты текста. Например, можем выделить только те абзацы, где встречаются сущности какого-то определенного типа, а потом работать только с ними.

Кроме того, сущности — это жесткие и надежные коллокации, их выделение может быть важно для многих задач. Допустим, у вас есть название именованной сущности и, какой бы она ни была, скорее всего, она непрерывна, и все действия с ней нужно совершать как с единым блоком. Например, переводить название сущности в название сущности. Вы хотите перевести «Магазин “Пятерочка”» на французский язык единым куском, а не разбить на несколько не связанных друг с другом фрагментов. Умение определять коллокации полезно и для многих других задач — например, для синтаксического парсинга.

Без решения задачи NER тяжело представить себе решение многих задач NLP, допустим, разрешение местоименной анафоры или построение вопросно-ответных систем.

В качестве статей для анализа решений данной задачи возьмем:

- 1) Unified Named Entity Recognition as Word-Word Relation Classification
- 2) BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding
- 3) Nested Named Entity Recognition
- 4) Discontinuous Named Entity Recognition as Maximal Clique Discovery

Основной же статьей для анализа будет статья “Unified Named Entity Recognition as Word-Word Relation Classification”, остальные являются вспомогательными для понимания предметной области и проблематики задачи.

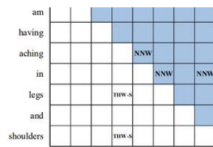


Figure 2: An example to show our relation classification for NER. We leverage a word-pair grid to show the relations between each word pair. NNW: Next-Neighboring-Word, THW-\*: Tail-Head-Word-\*

## Unified Named Entity Recognition as Word-Word Relation Classification

★ 215

19 Dec 2021

ljynlp/w2ner • PyTorch • 19 Dec 2021

So far, named entity recognition (NER) has been involved with three major types, including flat, overlapped (aka. nested), and discontinuous NER, which have mostly been studied individually. Recently, a growing interest has been built for unified NER, tackling the above three jobs concurrently with one single model. Current best-performing methods mainly include span-based and sequence-to-sequence models, where unfortunately the former merely focus on boundary identification and the latter may suffer from exposure bias. In this work, we present a novel alternative by modeling the unified NER as word-word relation classification, namely  $W^2NER$ . The architecture resolves the kernel bottleneck of unified NER by effectively modeling the neighboring relations between entity words with Next-Neighboring-Word (NNW) and Tail-Head-Word-\* (THW-\*) relations. Based on the  $W^2NER$  scheme we develop a neural framework, in which the unified NER is modeled as a 2D grid of word pairs. We then propose multi-granularity 2D convolutions for better refining the grid representations. Finally, a co-predictor is used to sufficiently reason the word-word relations. We perform extensive experiments on 14 widely-used benchmark datasets for flat, overlapped, and discontinuous NER (8 English and 6 Chinese datasets), where our model beats all the current top-performing baselines, pushing the state-of-the-art performances of unified NER.

[Paper](#)

[Code](#)

## Unified Named Entity Recognition as Word-Word Relation Classification

19 Dec 2021 · Jingye Li, Hao Fei, Jiang Liu, Shengqiong Wu, Meishan Zhang, Chong Teng, Donghong Ji, Fei Li · [Edit social preview](#)

So far, named entity recognition (NER) has been involved with three major types, including flat, overlapped (aka. nested), and discontinuous NER, which have mostly been studied individually. Recently, a growing interest has been built for unified NER, tackling the above three jobs concurrently with one single model. Current best-performing methods mainly include span-based and sequence-to-sequence models, where unfortunately the former merely focus on boundary identification and the latter may suffer from exposure bias. In this work, we present a novel alternative by modeling the unified NER as word-word relation classification, namely  $W^2NER$ . The architecture resolves the kernel bottleneck of unified NER by effectively modeling the neighboring relations between entity words with Next-Neighboring-Word (NNW) and Tail-Head-Word-\* (THW-\*) relations. Based on the  $W^2NER$  scheme we develop a neural framework, in which the unified NER is modeled as a 2D grid of word pairs. We then propose multi-granularity 2D convolutions for better refining the grid representations. Finally, a co-predictor is used to sufficiently reason the word-word relations. We perform extensive experiments on 14 widely-used benchmark datasets for flat, overlapped, and discontinuous NER (8 English and 6 Chinese datasets), where our model beats all the current top-performing baselines, pushing the state-of-the-art performances of unified NER.

[PDF](#)

[Abstract](#)

### Code

[Edit](#)

[ljynlp/w2ner](#) [official](#)

★ 215

[PyTorch](#)

### Tasks

[Edit](#)

[Chinese Named Entity Recognition](#)

[Classification](#)

[Named Entity Recognition](#)

[NER](#)

[Nested Named Entity Recognition](#)

[Relation Classification](#)

## 2. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

В рамках домашнего задания рассматриваются материалы статей:

- Unified Named Entity Recognition as Word-Word Relation Classification
- BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding
- Nested Named Entity Recognition
- Discontinuous Named Entity Recognition as Maximal Clique Discovery

До сих пор распознавание именованных объектов (NER) было связано с тремя основными типами, включая flat, nested, и discontinuous NER, которые в основном изучались индивидуально. В последнее время растет интерес к унифицированному NER, который одновременно решает три вышеуказанные задачи с помощью одной модели.

Современные наиболее эффективные методы в основном включают модели, основанные на промежутках и последовательностях, где, к сожалению, первые просто фокусируются на идентификации границ, а вторые могут страдать от предвзятости предположений. В статье была предложена новая альтернатива, моделирующая унифицированную классификацию NER в виде отношения слов к словам, а именно W2NER.

Архитектура повторно решает узкое место ядра unified NER, эффективно моделируя соседние отношения между конечными словами с помощью отношений Next-Neighboring-Word (NNW) и Tail-Head-Word-\* (THW-\*).

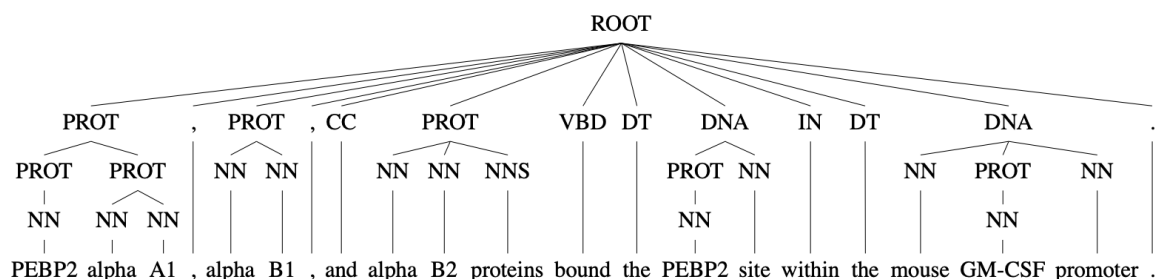
На основе схемы W2NER мы разрабатываем нейронную структуру, в которой унифицированный NER моделируется как 2D-сетка пар слов. Затем мы предлагаем многослойные 2D-свертки для лучшего уточнения представлений сетки. Наконец, ко-предиктор используется для достаточного обоснования отношений между словами.

## 2.1. Nested Named Entity Recognition

Распознавание именованных сущностей - это задача поиска сущностей, таких как люди и организации, в тексте. Часто организации вложены друг в друга, например, Банк Китая и Университет Вашингтона, обе организации с вложенными местоположениями. Вложенные объекты также распространены в биомедицинских данных, где различные биологические объекты, представляющие интерес, часто состоят друг из друга. В GENIA corpus (Ohta et al., 2002), который помечен такими типами сущностей, как белок и ДНК, примерно 17% сущностей встроены в другую сущность. В корпусе испанских и каталонских газетных текстов AnCorra (Март и др., 2007) почти половина объектов встроена. Однако в работе по распознаванию именованных сущностей (NER) почти полностью игнорировались вложенные сущности и вместо этого было решено сосредоточиться на самых внешних сущностях.

В статье “Nested Named Entity Recognition” мы знакомимся с новым решением проблемы распознавания вложенных именованных сущностей. Модель явно представляет вложенную структуру, позволяя объектам влиять не только на метки окружающих их слов, как в CRF, но и на объекты, содержащиеся в них, и в которых они содержатся. Мы представляем каждое предложение в виде дерева синтаксического анализа, со словами в виде листьев и с фразами, соответствующими каждой сущности (и узлом, который соединяет все предложение).

Ниже представлен пример древовидного представления вложенных именованных сущностей из статьи:





## 2.2. Discontinuous Named Entity Recognition

Распознавание именованных сущностей (NER) остается сложной задачей, когда упоминания сущностей могут быть непоследовательными. Существующие методы разбивают процесс распознавания на несколько последовательных этапов. При обучении они прогнозируют, основываясь на промежуточных результатах, в то время как при выводе полагаются на выходные данные модели предыдущих шагов, что приводит к смещению экспозиции. Чтобы решить эту проблему, мы сначала построим сегментный граф для каждого предложения, в котором каждый узел обозначает сегмент (непрерывный объект сам по себе или часть прерывистых объектов), а ребро связывает два узла, принадлежащих одному и тому же объекту.

Узлы и ребра могут быть сгенерированы соответственно за один этап с помощью схемы разметки сетки и изучены совместно с использованием новой архитектуры под названием Мас. Тогда прерывистый NER можно переформулировать как непараметрический процесс обнаружения максимальных клик в графе и объединения промежутков в каждой клике. Эксперименты с тремя контрольными показателями показывают, что наш метод превосходит самые современные результаты (SOTA), улучшая показатели F1 на 3,5 процента и обеспечивая 5-кратное ускорение по сравнению с моделью SOTA.

В теории графов клика - это подмножество вершин неориентированного графа, где каждые две вершины в клике смежны, в то время как максимальная клика - это та, которая не может быть расширена путем включения еще одной смежной вершины. Это означает, что каждая вершина в максимальной клике имеет тесные отношения друг с другом, и никакая другая вершина не может быть добавлена, что аналогично отношениям между сегментами в непрерывном объекте. Основываясь на этом понимании, мы утверждаем, что прерывистый NER может быть эквивалентно интерпретирован как обнаружение максимальных кликов из

графа сегментов, где узлы представляют сегменты, которые либо образуют объекты сами по себе, либо присутствуют как части прерывистой сущности, а ребра соединяют сегменты, принадлежащие одному и тому же объекту.

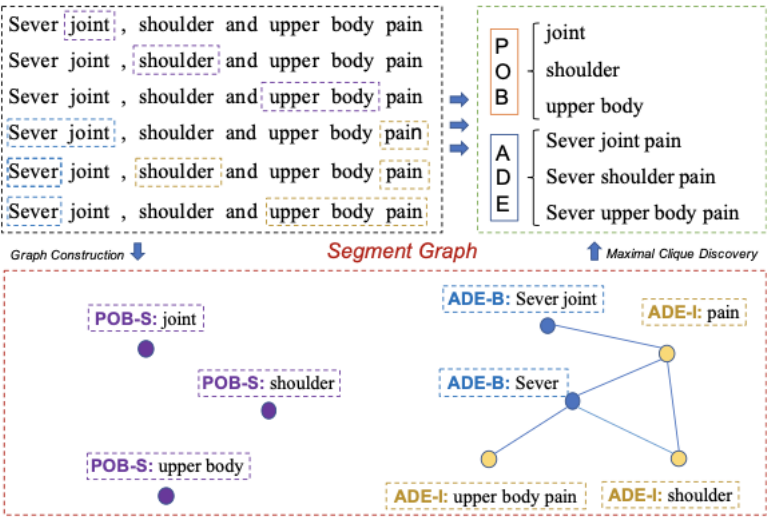
Учитывая, что процесс поиска максимальной клики обычно непараметрический (Брон и Кербощ, 1973), прерывистый NER фактически разбивается на две подзадачи: извлечение сегмента и предсказание ребер, чтобы соответственно создать узлы и ребра графа сегментов. Результаты их прогнозирования могут быть сгенерированы независимо с помощью предложенной нами схемы разметки сетки и будут использоваться вместе для построения сегментного графика, так что алгоритм обнаружения максимальной клики может быть применен для восстановления желаемых объектов. Общий процесс экстракции показан на рисунке 2. Далее мы сначала представим нашу схему разметки сетки и ее рабочий процесс декодирования. Затем мы подробно рассмотрим Мас, разрывную модель NER, основанную на обнаружении максимальной клики, основанную на этой схеме маркировки.

Основной проблемой является то, что упоминания сущностей могут перекрываться друг с другом. Чтобы сделать нашу модель способной извлекать такие перекрывающиеся сегменты, мы строим двумерную таблицу тегов. Паре токенов ( $t_i$ ,  $t_j$ ) будет присвоен набор меток, если сегмент от  $t_i$  до  $t_j$  принадлежит к соответствующим категориям.

На практике схема маркировки BIS используется для представления того, является ли сегмент непрерывным объектом, упомянутым (X-S), или находится в начале (X-B) или внутри (X-I) разрозненного объекта типа X. Например, (верхняя часть тела) присваивается тегу POB-S, поскольку "верхняя часть тела" является непрерывной сущностью типа Part of Body (POB). И тег (Sever, joint) - ADE-B, поскольку "Sever joint" является начальным сегментом прерывистого упоминания "Sever joint pain" типа Неблагоприятного лекарственного события (ADE). Между тем,

“соединение” также распознается как объект, поскольку на месте (соединение, соединение) есть тег POB-S, таким образом, проблема извлечения перекрывающихся сегментов решена.

Пример процесса экстракции представлен на рисунке ниже:



Пример разметки для извлечения сегмента:

	Sever	joint	,	shoulder	and	upper	body	pain
Sever	ADE-B	ADE-B						
joint		POB-S						
,								
shoulder				ADE-I POB-S				
and								
upper							POB-S	ADE-I
body								
pain								ADE-I

### 2.3. Трансформеры

Трансформер (Transformer) – модель, которая использует механизм внимания для повышения скорости обучения. Однако самое большое преимущество Трансформеров заключается в их высокой эффективности в условиях параллелизации.

В модели трансформера энкодер отображает входную последовательность символьных представлений  $(x_1, \dots, x_n)$  в последовательность непрерывных представлений  $z = (z_1, \dots, z_n)$ . Учитывая  $z$ , декодер затем генерирует выходную последовательность символов  $(y_1, \dots, y_m)$  по одному элементу за раз. На каждом этапе модель является авторегрессионной, используя ранее сгенерированные символы в качестве дополнительных входных данных при создании следующих символов.

Глобально архитектура трансформера включает в себя кодирующий и декодирующий компонент и связь между ними.

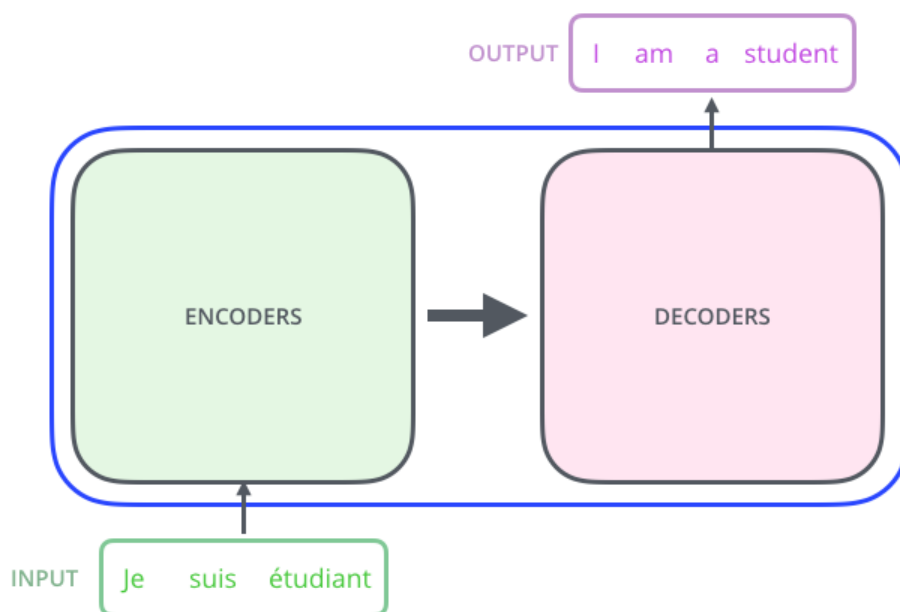


Рисунок 1. Модель трансформера в общем виде

Кодирующий компонент представляет из себя набор из 6 энкодеров. Декодирующий компонент – это набор декодеров, представленных в том же количестве.

При декомпозиции энкодер и декодер в стандартном трансформере содержат по шесть слоев, то есть наборов математических операций.

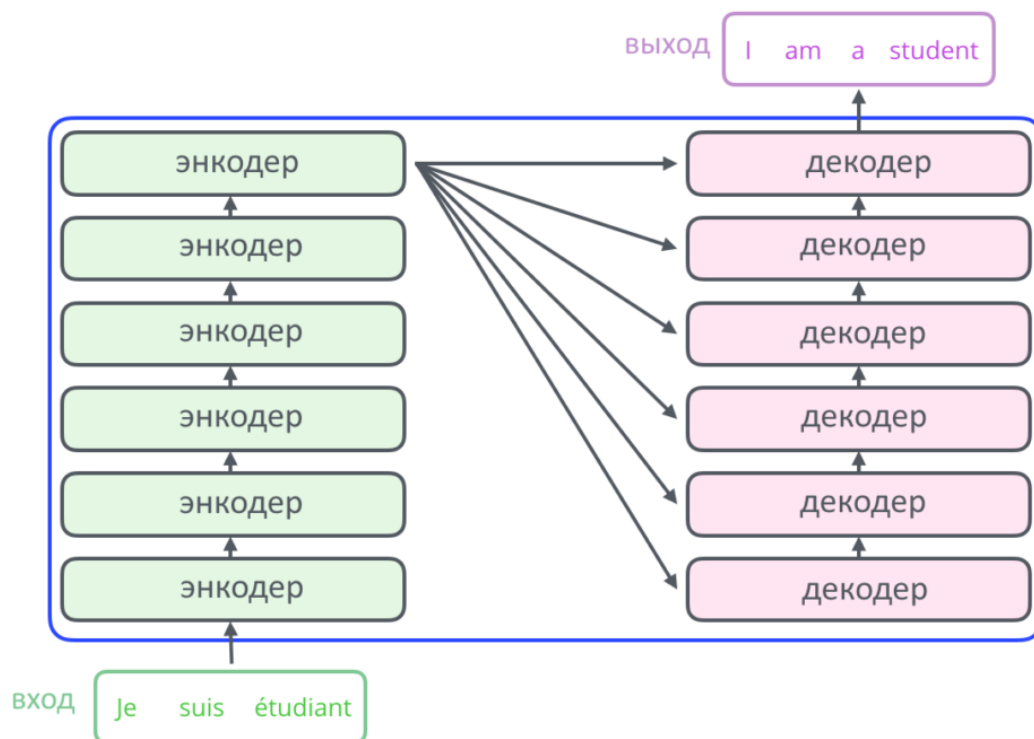


Рисунок 2 - Декомпозиция энкодера и декодера

Внутри каждого слоя энкодера есть две составные части: слой внимания и нейросеть с прямой связью.



Рисунок 3 - Архитектура энкодера

## Self-attention

В механизме Attention, внимание было направлено с одной части нейронной сети на другую ее часть. Декодер переводил фразу и одновременно решал, на какой части оригинала сосредоточиться, чтобы решать задачу точнее.

В трансформере тоже работает подобное, но не в кодирующей части. В трансформерах же применяется self-attention («внимание на себя»). Внимание энкодера обращено на предыдущий слой энкодера, то есть на свое предыдущее состояние.

На самом первом слое предыдущих нет, поэтому трансформер получает «сырые» вектора слов и концентрирует внимание на некоторых из них. Новый вариант входного предложения, где какие-то слова уже помечены как важные, попадает в нейросеть с прямой связью, чтобы моделировать более сложные функции.[6] Функции на слое внимания энкодера — линейные, поэтому бессмысленно передавать результат от одной линейной функции к другой по цепочке, добавление новых слоев не усложняет модель и не приближает ее к комплексной реальности. Эта проблема решается, если между линейными слоями внимания добавить слои с нелинейными функциями активации. Оттуда переделанный вектор попадает на следующий слой энкодера. Таким образом вектор проходит через все 6 энкодеров.

Следующий слой снова добавляет веса каким-то словам, результат идет дальше.

Смысл повторения — в том, чтобы каждый новый слой энкодера обратил внимание на что-нибудь интересное, но по-своему.[13]

## Механизма внимания: query, key, value

В модели трансформера вводится три новых понятия: запрос (query), ключ (key) и значение (value). На запрос Q, происходит поиск самые близкие к нему ключей K и выдаются соответствующие значения V.

Для расчета self-attention используется «dot product attention». На каждом слое внимания есть три разных матрицы весов — матрицы веса «запроса», «ключа» и «значения». Они заполняются случайно при первом запуске, и выучиваются во время обучения. На каждом слое они разные: так слои учатся обращать внимание на разные вещи и дополняют друг друга.

Вектор с прошлого слоя энкодера умножается на три этих матрицы и получается вектор «запроса» (Q), вектор «ключа» (K) и вектор «значения» (V).

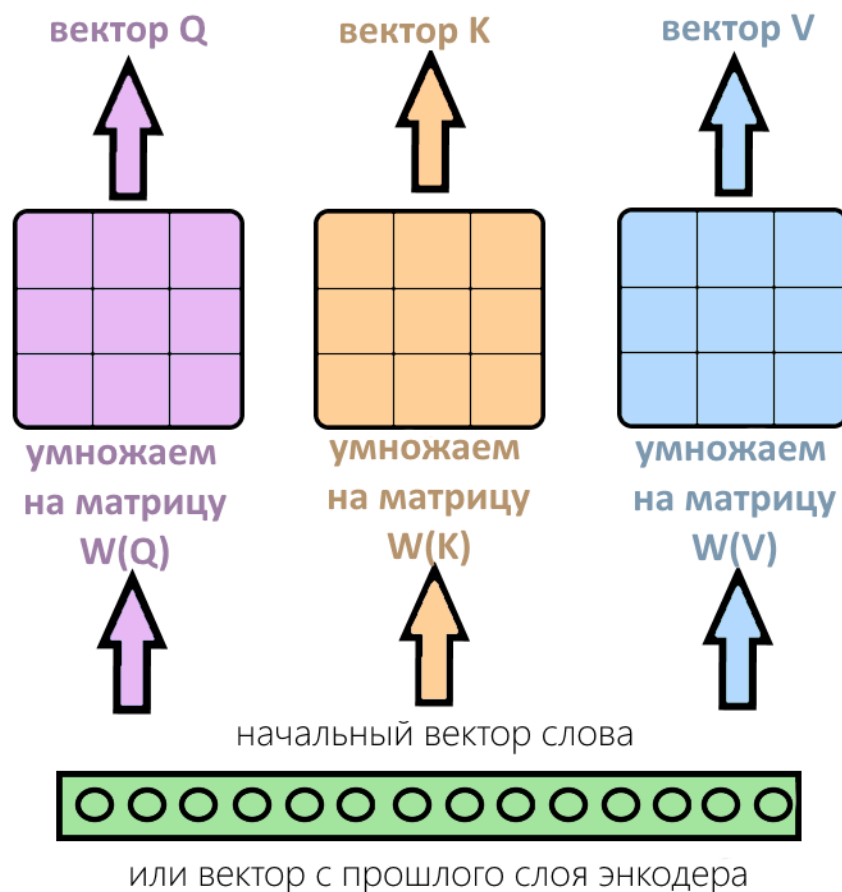


Рисунок 4. Модель механизма внимания в трансформере

В трансформере входные слова обрабатываются параллельно и независимо друг от друга. Данное свойство модели является одним из важнейших преимуществ по сравнению с рекуррентными нейросетями.

Слова из предложения можно обрабатывать параллельно, если объединить их в матрицы. Один вектор слова — одна строчка. Таким образом из векторов слов система получает матрицу предложений.

Если необходимо найти «запрос»  $Q$  под каждое входное слово — происходит умножение матрицы предложения на матрицу весов. Результатом данного произведения будет матрица запросов  $Q$ , в которой одна строчка равняется одному запросу, а матрицы умножаются быстрее, чем отдельные вектора по очереди. То же самое работает с матрицами  $K$  и  $V$ .

### **Multi-head attention («многоголовое» внимание)**

Для лучшей работы в трансформерах используется Multi-head attention, то есть происходит умножение исходных векторов на восемь разных небольших матриц весов  $W(Q)$ ,  $W(K)$ ,  $W(V)$ . Восемь получившихся матриц  $Q$ ,  $K$ ,  $V$  параллельно проходят через механизм скалярного внимания. В конце их результаты конкатенируются и попадают на слой с нелинейной функцией активации.[5]

Выбирая случайные начальные веса каждой «головы внимания», можно заставить систему «обратить внимание» на разные аспекты слова и получить восемь его разных «запросов», «ключей» и «значений»: это полезно и повышает качество анализа.



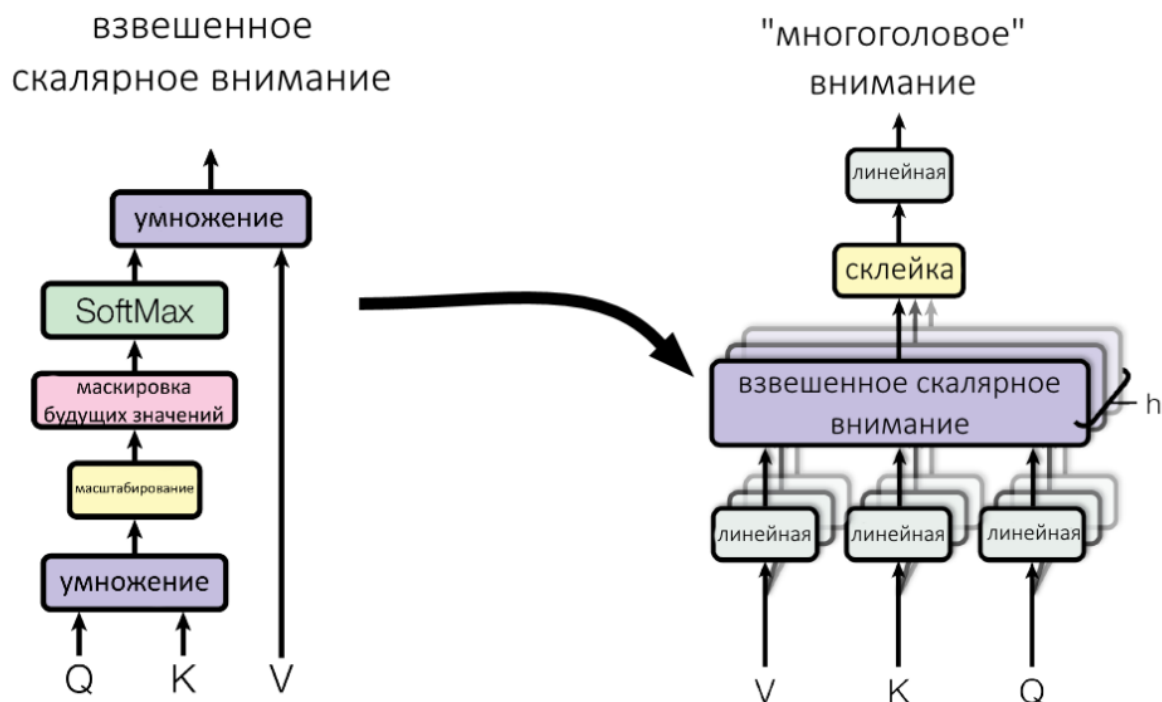


Рисунок 5. Модель механизма “Multi-head attention”

Итак, входные вектора слов, находящиеся в матрице, проходят через линейную проекцию  $h=8$  раз. Это значит, что исходную матрицу 8 раз умножают на разные маленькие матрицы, получая восемь троек Q, K, V.

## BERT

BERT (Bidirectional Encoder Representation Transformers) — модель представления языка, которая предназначена для предварительного обучения глубоких двунаправленных представлений на простых немаркированных текстах путем совмещения левого и правого контекстов во всех слоях. Это позволяет настраивать предварительно обученную модель BERT с помощью лишь одного дополнительного выходного слоя и получать наиболее актуальные результаты для широкого спектра задач.

Стандартные модели представления языка, существовавшие до BERT, например OpenAI GPT, были однонаправленными. Это ограничивало выбор архитектур, которые можно использовать для предварительного обучения. Например, в OpenAI GPT каждый токен мог

обслуживать только предыдущий токен (слева направо) в слое внутреннего внимания модели.[4]

Такой подход создает ряд ограничений, поэтому для предварительного обучения BERT используют маскированную языковую модель (MLM): случайным образом маскируется некоторое количество токенов во входных данных, затем модель должна предсказать исходное значение замаскированных слов, исходя из контекста. Это предоставляет возможность совмещать левый и правый контексты, что в свою очередь позволяет предварительно обучить двунаправленную модель представлений.

В структуре BERT есть два этапа:

1. Предварительное обучение — модель обучается на немаркированных данных, выполняя различные задачи.
2. Точная настройка — модель загружается с предварительно обученными параметрами и обучается на помеченных данных из последующих задач.

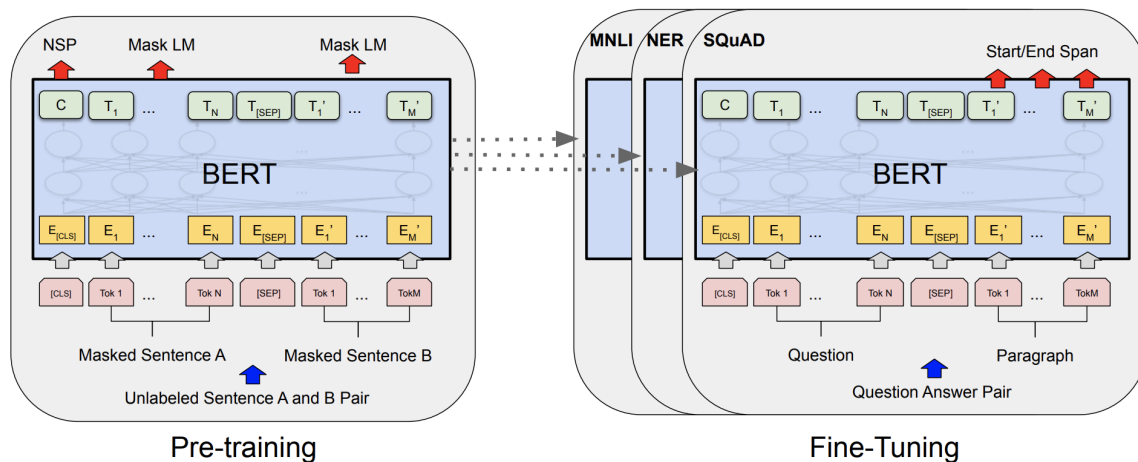


Рисунок 6. Архитектура модели BERT

Чтобы BERT научился обрабатывать различные задачи, на входе можно подавать как одно, так и пару предложений в одной и той же последовательности токенов. Модель использует эмбединг WordPiece со словарным запасом в 30 000 токенов. Первый токен в каждой

последовательности — это специальный токен для классификации предложения [CLS], окончательное скрытое состояние которого используется в качестве представления совокупной последовательности для задач классификации. Предложения в последовательности различают двумя способами. Можно разделять их с помощью специального токена [SEP], а можно добавить заученный эмбединг к каждому токenu, чтобы указать, принадлежит ли он предложению A или предложению B. Последний скрытый вектор специального токена обозначается как C, а последний скрытый вектор некоторого входного токена  $i$  обозначается как  $T_i$ . [12]

Для данного токена его входное представление строится путем суммирования соответствующих эмбедингов токена, сегмента и позиции.

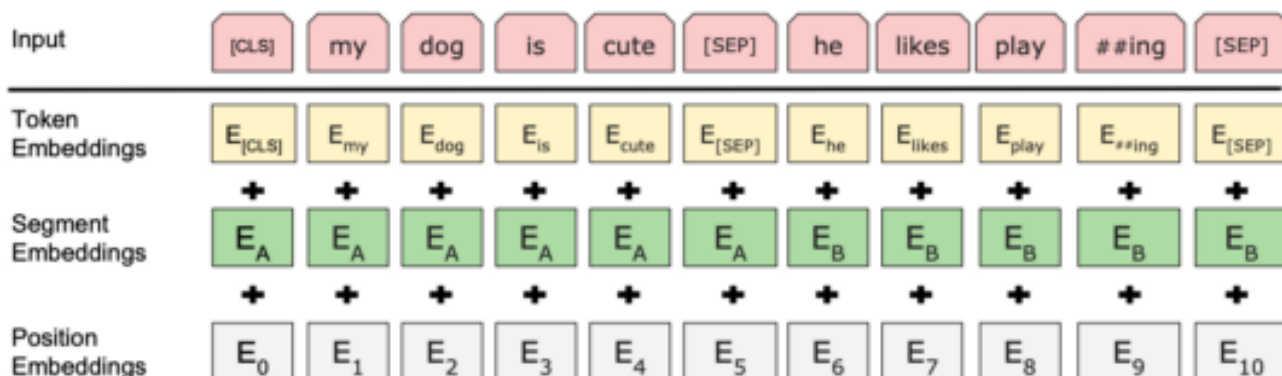


Рисунок 7. Схема представления токенов в модели BERT

Для того чтобы предобучить модель BERT существует две задачи:

1. Маскированная языковая модель (MLM) — Чтобы обучить глубокое двунаправленное представление, случайный процент (15% в рассматриваемом примере) входных токенов маскируют и тренируют нейросеть предсказывать закрытые маской токены. Финальные скрытые векторы, соответствующие маскированным токенам, передаются в выходной softmax слой из словаря.
2. Предсказание следующего предложения (NSP) — Чтобы обучить модель понимать отношения между предложениями,

ее предварительно обучают бинаризованной задаче прогнозирования следующего предложения. При подготовке примеров предложений А и В для предварительного обучения в 50% случаев В — это фактическое следующее предложение, которое следует за А, а в 50% случаев В — случайное предложение из корпуса.

В BERT точная настройка выполняется путем простой замены соответствующих входных и выходных данных в зависимости от того, включают ли последующие задачи один текст или текстовые пары. Для каждой конкретной задачи подключают входные и выходные данные соответственно и полностью настраивают модель.[2]

## 2.4. Unified Named Entity Recognition

До сих пор распознавание именованных объектов (NER) было связано с тремя основными типами, включая flat, nested, и discontinuous NER, которые в основном изучались индивидуально. В последнее время растет интерес к унифицированному NER, который одновременно решает три вышеуказанные задачи с помощью одной модели.

Современные наиболее эффективные методы в основном включают модели, основанные на промежутках и последовательностях, где, к сожалению, первые просто фокусируются на идентификации границ, а вторые могут страдать от предвзятости предположений. В статье была предложена новая альтернатива, моделирующая унифицированную классификацию NER в виде отношения слов к словам, а именно W2NER.

Архитектура повторно решает узкое место ядра unified NER, эффективно моделируя соседние отношения между конечными словами с помощью отношений Next-Neighboring-Word (NNW) и Tail-Head-Word-\* (THW-\*).

На основе схемы W2NER мы разрабатываем нейронную структуру, в которой унифицированный NER моделируется как 2D-сетка пар слов.

Затем мы предлагаем многозернистые 2D-свертки для лучшего уточнения представлений сетки. Наконец, ко-предиктор используется для достаточного обоснования отношений между словами.

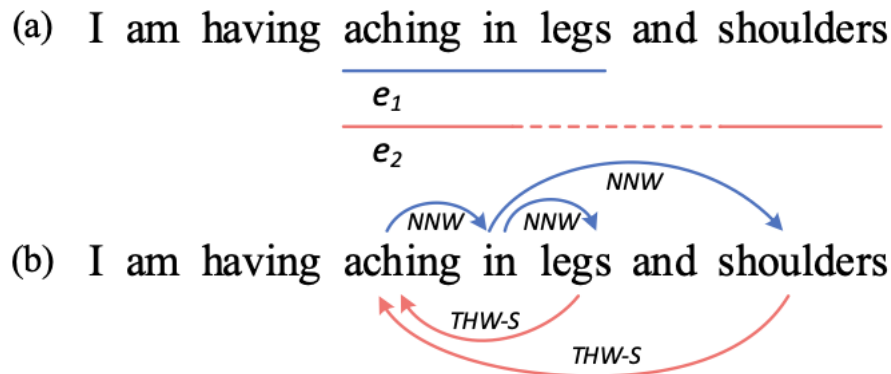


Рисунок 8. Пример использования NNW и THW-S

В основе приведенного выше наблюдения мы исследуем альтернативный унифицированный формализм NER с новой архитектурой классификации отношений слово-слово, а именно W2NER. Наш метод разрешает унифицированный NER, эффективно моделируя как идентификацию границ объекта, так и соседние отношения между словами объекта. В частности, W2NER делает прогнозы для двух типов отношений, включая Следующее-Соседнее-Слово (NNW) и Слово-Хвост-Голова-\* (THW-\*), как показано на рисунке 1(b). Отношение NNW адресует идентификацию слова сущности, указывая, совпадают ли два слова-аргумента в сущности (например, aching→in), в то время как отношение THW-\* учитывает определение границ сущности и типа, показывая, являются ли два слова-аргумента границами хвоста и головы соответственно объекта. “\*” сущность (например, ноги→ боль, Симптом).

Основываясь на схеме W2NER, мы далее представляем нейронную структуру для унифицированного NER. Во-первых, BERT (Devlin et al. 2019) и BiLSTM (Lample et al. 2016) используются для предоставления контекстуализированных представлений слов, на основе которых мы

строим 2-мерную (2D) сетку для пар слов. После этого мы разрабатываем 2D-свертки с несколькими грануляциями для уточнения представлений пар слов, эффективно фиксируя взаимодействия как между близкими, так и между удаленными парами слов. Ко-предиктор, наконец, обосновывает отношения между словами и выдает все возможные упоминания сущностей, в которых для получения дополнительных преимуществ совместно используются классификаторы biaffine и multi-layer perceptron (MLP).

	I	am	having	aching	in	legs	and	shoulders
I								
am								
having								
aching					NNW			
in						NNW		NNW
legs				THW-S				
and								
shoulders				THW-S				

Рисунок 9. Классификация отношений с помощью NNW и THW-S

Пример, показывающий метод классификации отношений для NER. Мы используем сетку пар слов, чтобы визуализировать отношения между каждой парой слов.

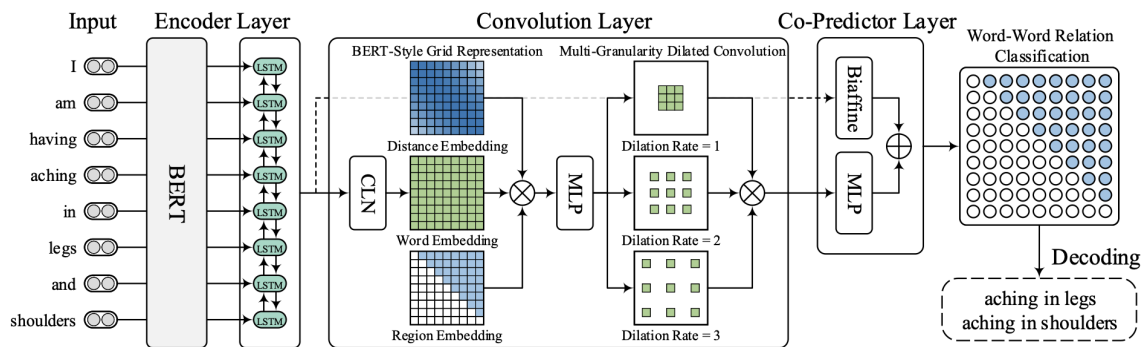


Рисунок 10. Архитектура модели Unified NER

## 2.5. Наборы данных и результаты эксперимента

В работе используется набора данных для всех трех подзадач NER:

- **Наборы данных FlatNER** - используем CoNLL-2003 (Sang and Meulder 2003) и OntoNotes 5.0 (Pradhan et al. 2013b) на английском языке, OntoNotes 4.0 (Weischedel et al. 2011), MSRA (Levow 2006), Weibo (Peng and Dredze 2015; He and Sun 2017) и Resume (Zhang и Ян 2018) на китайском языке. ACL Anthology. Используется Справочный корпус антологии ACL в качестве набора данных.
- **Наборы данных NestedNER** - проводим эксперименты на ACE 2004 (Doddington et al. 2004), ACE 2005 (Walker et al. 2011), GENIA (Kim et al. 2003). Для GENIA мы следуем примеру Yan et al. (2021), чтобы использовать пять типов сущностей и разделить train /dev /test как 8.1: 0.9:1.0. Для ACE 2004 и ACE 2005 на английском языке используем то же деление данных, что и Lu и Roth (2015); Yu et al. (2020 год). Для ACE 2004 и ACE 2005 на китайском языке разделили train/dev/test как 8.0:1.0:1.0.

- **Наборы данных DiscontinuousNER** - экспериментируем с тремя наборами данных для прерывистых NER, а именно CADEC (Karimi et al. 2015), share13 (Pradhan et al. 2013a) и ShARe14 (Mowery et al. 2014), все из которых получены из документов биомедицинской или клинической области. Для разделения данных мы используем сценарии предварительной обработки, предоставленные Dai et al. (2020). Около 10% объектов в этих наборах данных являются прерывистыми.

		CADEC			ShARe13			ShARe14		
		P	R	F1	P	R	F1	P	R	F1
• <b>Sequence Labeling</b>	Tang et al. (2018)	67.80	64.99	66.36	-	-	-	-	-	-
• <b>Span-based</b>	Li et al. (2021a)	-	-	69.90	-	-	82.50	-	-	-
• <b>Hypergraph-based</b>	Wang and Lu (2019)	72.10	48.40	58.00	83.80	60.40	70.30	79.10	70.70	74.70
• <b>Seq2Seq</b>	Yan et al. (2021)	70.08	71.21	70.64	82.09	77.42	79.69	77.20	83.75	80.34
	Fei et al. (2021)	<b>75.50</b>	71.80	72.40	<b>87.90</b>	77.20	80.30	-	-	-
• <b>Others</b>	Dai et al. (2020)	68.90	69.00	69.00	80.50	75.00	77.70	78.10	81.20	79.60
	Wang et al. (2021)	70.50	<b>72.50</b>	71.50	84.30	78.20	81.20	78.20	<b>84.70</b>	81.30
	W <sup>2</sup> NER (ours)	74.09	72.35	<b>73.21</b>	85.57	<b>79.68</b>	<b>82.52</b>	<b>79.88</b>	83.71	<b>81.75</b>

Рисунок 11. Сравнение модели W2NER с другими архитектурами



### 3. ПРАКТИЧЕСКАЯ ЧАСТЬ

#### Обучение BERT для решения задачи NER

Распознавание именованных объектов (NER) использует определенную схему аннотаций, которая определяется (по крайней мере, для европейских языков) на уровне слов. Широко используемая схема аннотаций называется IOB-tagging, что означает "Inside-Outside-Beginning". Каждый тег указывает, находится ли соответствующее слово внутри, снаружи или в начале определенного именованного объекта. Причина, по которой это используется, заключается в том, что именованные объекты обычно содержат более 1 слова.

Давайте посмотрим на пример. Если у вас есть предложение типа "Барак Обама родился на Гавайях", то соответствующие теги будут [B-PERS, I-PERS, O, O, O, B-GEO]. B-PERS означает, что слово "Барак" - это начало личности, I-PERS означает, что слово "Обама" находится внутри человека, "O" означает, что слово "был" находится вне именованной сущности, и так далее. Таким образом, у одного обычно столько тегов, сколько слов в предложении.

Здесь мы будем использовать набор данных NER из Kaggle, который уже находится в формате IOB. аписную книжку. Давайте распечатаем первые несколько строк этого csv-файла:

```
[ ] data = pd.read_csv("ner_datasetreference.csv", encoding='unicode_escape')
data.head()
```

	Sentence #	Word	POS	Tag
0	Sentence: 1	Thousands	NNS	O
1	NaN	of	IN	O
2	NaN	demonstrators	NNS	O
3	NaN	have	VBP	O
4	NaN	marched	VBN	O

Выведем имеющиеся теги в датасете:

```
[ ] print("Number of tags: {}".format(len(data.Tag.unique())))
    frequencies = data.Tag.value_counts()
    frequencies

Number of tags: 17
0      887908
B-geo    37644
B-tim    20333
B-org    20143
I-per    17251
B-per    16990
I-org    16784
B-gpe    15870
I-geo     7414
I-tim     6528
B-art      402
B-eve      308
I-art      297
I-eve      253
B-nat      201
I-gpe      198
I-nat       51
Name: Tag, dtype: int64
```

Теперь, когда наши данные предварительно обработаны, мы можем превратить их в тензоры высоты тона, чтобы предоставить их модели. Давайте начнем с определения некоторых ключевых переменных, которые будут использоваться позже в процессе обучения / оценки:

```
[ ] MAX_LEN = 128
    TRAIN_BATCH_SIZE = 4
    VALID_BATCH_SIZE = 2
    EPOCHS = 1
    LEARNING_RATE = 1e-05
    MAX_GRAD_NORM = 10
    tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')
```

Создаем 2 набора данных, один для обучения и один для тестирования. Давайте использовать соотношение 80/20:

```

▶ train_size = 0.8
train_dataset = data.sample(frac=train_size, random_state=200)
test_dataset = data.drop(train_dataset.index).reset_index(drop=True)
train_dataset = train_dataset.reset_index(drop=True)

print("FULL Dataset: {}".format(data.shape))
print("TRAIN Dataset: {}".format(train_dataset.shape))
print("TEST Dataset: {}".format(test_dataset.shape))

training_set = dataset(train_dataset, tokenizer, MAX_LEN)
testing_set = dataset(test_dataset, tokenizer, MAX_LEN)

FULL Dataset: (47571, 2)
TRAIN Dataset: (38057, 2)
TEST Dataset: (9514, 2)

```

Определим параметры для нашего даталодера:

```

[ ] train_params = {'batch_size': TRAIN_BATCH_SIZE,
                    'shuffle': True,
                    'num_workers': 0
                  }

test_params = {'batch_size': VALID_BATCH_SIZE,
               'shuffle': True,
               'num_workers': 0
              }

training_loader = DataLoader(training_set, **train_params)
testing_loader = DataLoader(testing_set, **test_params)


```


Определяем модель:

```

▶ model = BertForTokenClassification.from_pretrained('bert-base-uncased', num_labels=len(labels_to_ids))
model.to(device)

```

Downloading: 100%  433/433 [00:06<00:00, 71.5B/s]

Downloading: 100%  440M/440M [00:05<00:00, 74.4MB/s]

Some weights of the model checkpoint at bert-base-uncased were not used when initializing BertForTokenClassification: ['c  
- This IS expected if you are initializing BertForTokenClassification from the checkpoint of a model trained on another t  
- This IS NOT expected if you are initializing BertForTokenClassification from the checkpoint of a model that you expect  
Some weights of BertForTokenClassification were not initialized from the model checkpoint at bert-base-uncased and are ne  
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

```

BertForTokenClassification(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(30522, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
        )
      )
    )
  )
)

```

Переходим к дообучению модели:

```
[ ] inputs = training_set[2]
    input_ids = inputs["input_ids"].unsqueeze(0)
    attention_mask = inputs["attention_mask"].unsqueeze(0)
    labels = inputs["labels"].unsqueeze(0)

    input_ids = input_ids.to(device)
    attention_mask = attention_mask.to(device)
    labels = labels.to(device)

    outputs = model(input_ids, attention_mask=attention_mask, labels=labels)
    initial_loss = outputs[0]
    initial_loss

tensor(2.6094, device='cuda:0', grad_fn=<NllLossBackward>)
```

```
def train(epoch):
    tr_loss, tr_accuracy = 0, 0
    nb_tr_examples, nb_tr_steps = 0, 0
    tr_preds, tr_labels = [], []
    model.train()

    for idx, batch in enumerate(training_loader):

        ids = batch['input_ids'].to(device, dtype = torch.long)
        mask = batch['attention_mask'].to(device, dtype = torch.long)
        labels = batch['labels'].to(device, dtype = torch.long)

        loss, tr_logits = model(input_ids=ids, attention_mask=mask,
labels=labels)
        tr_loss += loss.item()

        nb_tr_steps += 1
        nb_tr_examples += labels.size(0)

        if idx % 100==0:
            loss_step = tr_loss/nb_tr_steps
            print(f"Training loss per 100 training steps: {loss_step}")

        flattened_targets = labels.view(-1)
        active_logits = tr_logits.view(-1, model.num_labels)
        flattened_predictions = torch.argmax(active_logits, axis=1)
        active_accuracy = labels.view(-1) != -100

        labels = torch.masked_select(flattened_targets,
active_accuracy)
        predictions = torch.masked_select(flattened_predictions,
active_accuracy)

        tr_labels.extend(labels)
        tr_preds.extend(predictions)

        tmp_tr_accuracy = accuracy_score(labels.cpu().numpy(),
predictions.cpu().numpy())
        tr_accuracy += tmp_tr_accuracy

        torch.nn.utils.clip_grad_norm_(
            parameters=model.parameters(), max_norm=MAX_GRAD_NORM
        )
```

```

optimizer.zero_grad()
loss.backward()
optimizer.step()

epoch_loss = tr_loss / nb_tr_steps
tr_accuracy = tr_accuracy / nb_tr_steps
print(f"Training loss epoch: {epoch_loss}")
print(f"Training accuracy epoch: {tr_accuracy}")


```

### Процесс обучения:

```

▶ for epoch in range(EPOCHS):
    print(f"Training epoch: {epoch + 1}")
    train(epoch)

```

 Training epoch: 1  
 Training loss per 100 training steps: 0.41946473717689514  
 Training loss per 100 training steps: 0.5749164560053608  
 Training loss per 100 training steps: 0.4594639188688786  
 Training loss per 100 training steps: 0.38003166179571835  
 Training loss per 100 training steps: 0.34181345879399866  
 Training loss per 100 training steps: 0.3160412432086801  
 Training loss per 100 training steps: 0.2929958432150413  
 Training loss per 100 training steps: 0.27827913607716476  
 Training loss per 100 training steps: 0.2661318151598864  
 Training loss per 100 training steps: 0.25599023683221966  
 Training loss per 100 training steps: 0.24448720548584774  
 Training loss per 100 training steps: 0.23736871189719488  
 Training loss per 100 training steps: 0.22972667161887086  
 Training loss per 100 training steps: 0.22326894107200979  
 Training loss per 100 training steps: 0.21723312479134738  
 Training loss per 100 training steps: 0.2124901934335136  
 Training loss per 100 training steps: 0.20869656914681065  
 Training loss per 100 training steps: 0.20421362344804125  
 Training loss per 100 training steps: 0.2012133353282757  
 Training loss per 100 training steps: 0.19797089583397712  
 Training loss per 100 training steps: 0.19545926196199992  
 Training loss per 100 training steps: 0.19256210732612863  
 Training loss per 100 training steps: 0.1895416755830511  
 Training loss per 100 training steps: 0.18687579922276068  
 Training loss per 100 training steps: 0.1840702812224223  
 Training loss per 100 training steps: 0.18219109044307139  
 Training loss per 100 training steps: 0.1800966161664508  
 Training loss per 100 training steps: 0.17866380249558003  
 Training loss per 100 training steps: 0.17666383030911145  
 Training loss per 100 training steps: 0.17523060850439526  
 Training loss per 100 training steps: 0.17382653130085562

### Валидация:

```

def valid(model, testing_loader):
    # put model in evaluation mode
    model.eval()

    eval_loss, eval_accuracy = 0, 0
    nb_eval_examples, nb_eval_steps = 0, 0
    eval_preds, eval_labels = [], []

    with torch.no_grad():

```

```

        for idx, batch in enumerate(testing_loader):

            ids = batch['input_ids'].to(device, dtype = torch.long)
            mask = batch['attention_mask'].to(device, dtype =
torch.long)
            labels = batch['labels'].to(device, dtype = torch.long)

            loss, eval_logits = model(input_ids=ids,
attention_mask=mask, labels=labels)

            eval_loss += loss.item()

            nb_eval_steps += 1
            nb_eval_examples += labels.size(0)

            if idx % 100==0:
                loss_step = eval_loss/nb_eval_steps
                print(f"Validation loss per 100 evaluation steps:
{loss_step}")

                # compute evaluation accuracy
                flattened_targets = labels.view(-1) # shape (batch_size *
seq_len,)
                active_logits = eval_logits.view(-1, model.num_labels) #
shape (batch_size * seq_len, num_labels)
                flattened_predictions = torch.argmax(active_logits, axis=1)
# shape (batch_size * seq_len,)

                # only compute accuracy at active labels
                active_accuracy = labels.view(-1) != -100 # shape
(batch_size, seq_len)

                labels = torch.masked_select(flattened_targets,
active_accuracy)
                predictions = torch.masked_select(flattened_predictions,
active_accuracy)

                eval_labels.extend(labels)
                eval_preds.extend(predictions)

                tmp_eval_accuracy = accuracy_score(labels.cpu().numpy(),
predictions.cpu().numpy())
                eval_accuracy += tmp_eval_accuracy

            labels = [ids_to_labels[id.item()] for id in eval_labels]
            predictions = [ids_to_labels[id.item()] for id in eval_preds]

            eval_loss = eval_loss / nb_eval_steps
            eval_accuracy = eval_accuracy / nb_eval_steps
            print(f"Validation Loss: {eval_loss}")
            print(f"Validation Accuracy: {eval_accuracy}")

        return labels, predictions

```

**Результаты обучения:**

```
▶ from segeval.metrics import classification_report  
print(classification_report(labels, predictions))
```

```
👤
```

	precision	recall	f1-score	support
org	0.63	0.59	0.61	3964
gpe	0.94	0.93	0.93	3021
geo	0.81	0.89	0.85	7378
tim	0.86	0.85	0.85	4070
per	0.73	0.78	0.75	3367
micro avg	0.79	0.82	0.80	21800
macro avg	0.79	0.82	0.80	21800

## **4. ВЫВОДЫ**

В рамках домашнего задания был выполнен обзор теоретических и практических материалов, связанных со статьей «Unified Named Entity Recognition as Word-Word Relation Classification».

В статье рассматривалась подход для разработки универсального решения задачи NER. Также в процессе работы со статьей были изучены дополнительные материалы статей, из которых мы выявили какие бывают подзадачи для NER, основную проблематику, с которой необходимо бороться и подходы для решения этих проблем.



## 5. СПИСОК ИСТОЧНИКОВ

1. Browse State-of-the-Art. – Текст. Изображение: электронные // Papers With Code : [сайт]. – URL: <https://paperswithcode.com/sota> (дата обращения: 25.05.2022).
2. Transformer в картинках. – Текст. Изображение: электронные // Хабр : [сайт]. – URL: <https://habr.com/ru/post/486358/> (дата обращения: 25.05.2022).
3. datasets. – Текст. Изображение: электронные // GitHub : [сайт]. – URL: <https://github.com/huggingface/datasets> (дата обращения: 25.05.2022).
4. malteos. – Текст. Изображение: электронные // Hugging Face : [сайт]. – URL: <https://huggingface.co/malteos> (дата обращения: 25.05.2022).