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 ***Робототехника и комплексная автоматизация***

КАФЕДРА ***Системы автоматизированного проектирования (РК-6)***

ОТЧЕТ ПО ПРОИЗВОДСТВЕННОЙ ПРАКТИКЕ

Студент ***Онюшев Артем Андреевич***
фамилия, имя, отчество

Группа ***РК6-86Б***

Тип практики ***Преддипломная***

Название предприятия ***НИИ АПП МГТУ им. Н.Э. Баумана***

Студент _____ ***Онюшев А.А.***
подпись, дата *фамилия, и.о.*

Руководитель практики
от кафедры _____ ***Витюков Ф.А.***
подпись, дата *фамилия, и.о.*

Оценка _____

2024 г.

«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

УТВЕРЖДАЮ
Заведующий кафедрой *РК6*
_____ __ *А.П. Карпенко* _
« ___ » _____ 2024 г.

З А Д А Н И Е
на прохождение производственной практики
Преддипломная
Тип практики

Студент

Онюшев Артем Андреевич ***4*** курса группы ***РК6-86Б***
Фамилия Имя Отчество № курса индекс группы

в период с ***13 мая 2024*** г. по ***26 мая 2024*** г.

Предприятие: ***НИИ АПП МГТУ им. Н.Э. Баумана***
Подразделение:
_____
(отдел/сектор/цех)
Руководитель практики от предприятия (наставник):
Киселев Игорь Алексеевич, директор НИИ АПП МГТУ им. Н.Э.Баумана
(Фамилия Имя Отчество полностью, должность)
Руководитель практики от кафедры:
Витюков Фёдор Андреевич, старший преподаватель
(Фамилия Имя Отчество полностью, должность)
Задание:
1. *Изучить теорию об архитектуре нейронной сети трансформер;*
2. *Разработать свою собственную нейронную сеть на основе архитектуры трансформер;*
3. *Разобрать ряд задач, которые решает трансформер.*

Дата выдачи задания ***14 мая 2024*** г.

Руководитель практики от предприятия _____/ ***И.А. Киселев*** /

Руководитель практики от кафедры _____/ ***Ф.А. Витюков*** /

Студент _____/ ***А.А. Онюшев*** /

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Нейронные сети являются мощным инструментом в области искусственного интеллекта и машинного обучения. Они моделируют работу человеческого мозга, позволяя компьютерам обрабатывать и анализировать сложные данные. Исследования по нейронным сетям имеют огромный потенциал для решения различного рода задач.

Один из важнейших инструментов машинного обучения — трансформеры. Популярность трансформеров взлетела до небес в связи с появлением больших языковых моделей вроде ChatGPT, GPT-4 и LLama. Эти модели созданы на основе трансформерной архитектуры и демонстрируют отличную производительность в понимании и синтезе естественных языков.

Помимо понимания и синтеза естественных языков, НС с трансформерной архитектурой имеют большой успех и спрос во многих других доменах машинного обучения, таких как распознавание человеческой речи, анализ изображений и видео.

В данной работе мы поймем, что такое архитектура трансформера и как она устроена. Научимся правильно использовать трансформер и разберем ряд задач, которые решает трансформер.

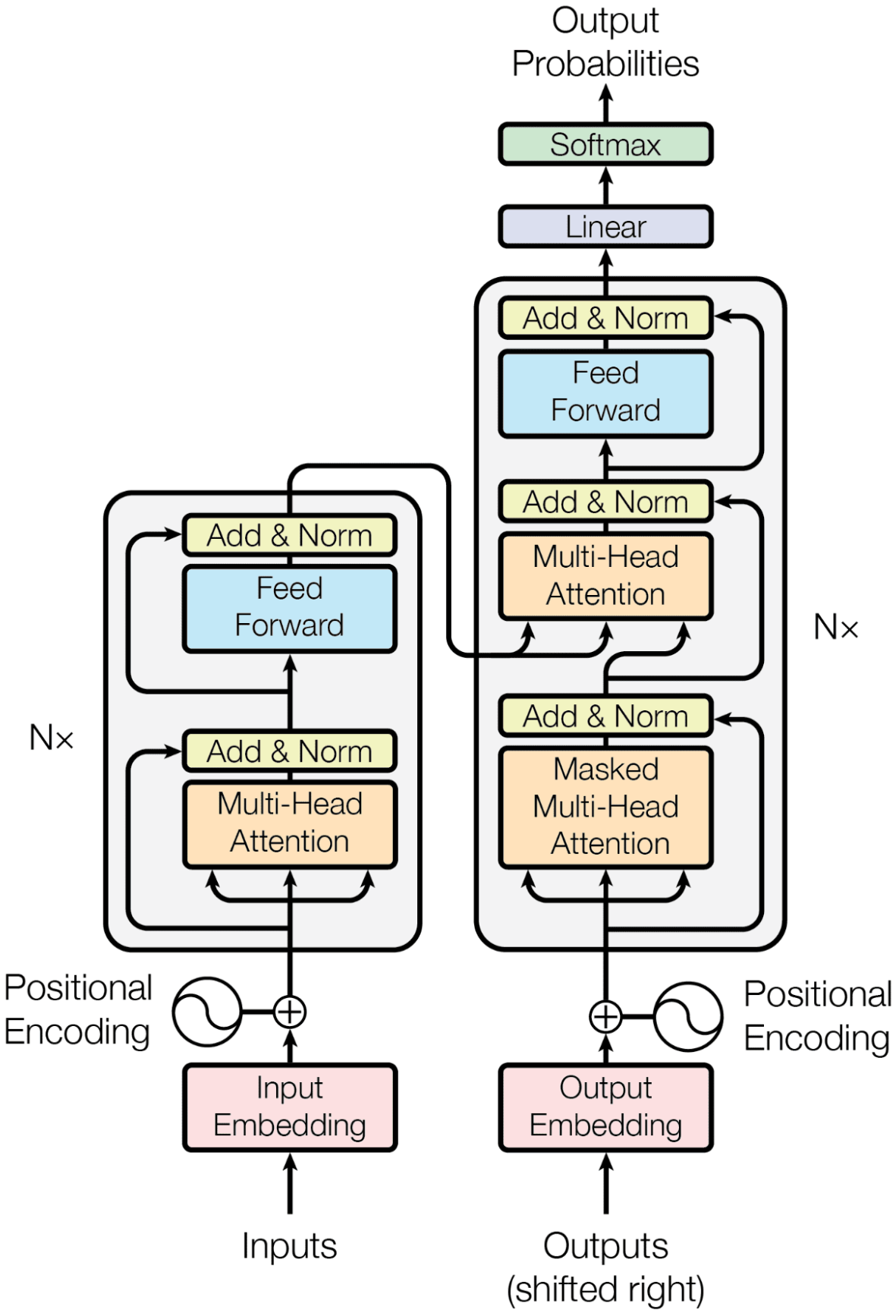

Рисунок 1. Архитектура трансформерной НС.

1. ПОСТАНОВКА ЗАДАЧИ

Требуется изучить и разработать свою нейронную сеть, используя архитектуру трансформера. Разобрать каждый блок такой архитектуры и представить его реализацию в коде.

2. КРАТКИЙ ОТЧЕТ О ВЫПОЛНЕННЫХ РАБОТАХ

Полную версию кода можно найти по ссылке: https://github.com/Relax-FM/Transformer_tutor.

2.1. Что такое трансформер?

Перед реализацией архитектуры трансформера на практике, стоит узнать, что такое трансформер. Трансформер — это такой вид нейросетевой архитектуры, который хорошо подходит для обработки последовательностей данных. Пожалуй, самый популярный пример таких данных это предложение, которое можно считать упорядоченным набором слов.

Трансформеры создают цифровое представление каждого элемента последовательности, инкапсулируют важную информацию о нём и окружающем его контексте. Получившиеся представления затем можно передать в другие нейросети, которые воспользуются этой информацией для решения разных задач, в том числе для синтеза и классификации. Создавая такие информативные представления, трансформеры помогают последующим нейросетям лучше понять скрытые паттерны и взаимосвязи во входных данных. И поэтому они лучше синтезируют последовательные и взаимосвязанные результаты.

Главное преимущество трансформеров заключается в их способности обрабатывать длительные зависимости в последовательностях. Кроме того, они очень производительны, могут обрабатывать последовательности параллельно. Это особенно полезно в задачах вроде машинного перевода, анализа настроений и синтеза текста.

2.2. Что поступает в трансформер?

Прежде чем подать данные в трансформер, нужно сначала преобразовать их в последовательность токенов — набор целых чисел, представляющих входные данные. Будем рассматривать сценарий использования трансформера для обработки естественного языка.

Получив последовательность целых чисел, представляющих входные данные, мы можем превратить их в эмбеддинги — это способ представления информации, облегчающий её обработку алгоритмами машинного обучения. Эмбеддинги передают смысл токенов в сжатом формате, представляя информацию в виде последовательности чисел. Сначала они синтезируются как случайная последовательность, а значимое представление формируется во время обучения. Однако у эмбеддингов есть наследственное ограничение: они не учитывают контекст, в котором синтезировались токены.

В зависимости от задачи, при превращении токенов в эмбеддинги нам может потребоваться сохранить порядок токенов. Это особенно важно в обработке естественных языков, иначе мы придём к методу «мешка слов». Чтобы этого не допустить, мы применяем к эмбеддингам позиционное кодирование (на рисунке 1 Positional Encoding). Есть разные способы это сделать, но основная идея в том, что у нас есть ещё один набор эмбеддингов, представляющих положение каждого токена во входной последовательности. Этот второй набор комбинируется с эмбеддингами токенов.

Рисунок 2. Получаем итоговый вектор эмбеддинга.

Другая сложность в том, что у токенов могут быть разные значения в зависимости от соседних токенов. Например:

- Андрей не любит арбуз, **он** слишком сладкий.
- Андрей не любит арбуз, **он** любит дыню.

Здесь слово «он» используется в двух абсолютно разных контекстах, поэтому имеют разные значения. В первом предложении слово «он» подразумевает арбуз. Во втором же – Андрей. Трансформер решает эту проблему с помощью механизма «Attention» или же «Внимание».

2.3. «Внимание»

Пожалуй, самый важный механизм в трансформенной архитектуре — это внимание (на рисунке 1 – светло-оранжевый блок). Он позволяет нейросети понять, какая часть входной последовательности наиболее релевантна задаче. Механизм внимания определяет для каждого токена последовательности, какие другие токены необходимы для его понимания в данном контексте. Прежде чем мы перейдем тому, как это реализовано в трансформере, давайте сначала разберемся, чего пытается добиться механизм внимания.

Этот механизм можно представить как метод, который заменяет каждый эмбеддинг токена на эмбеддинг, содержащий информацию о соседних токенах, вместо использования одинакового эмбеддинга для каждого токена вне зависимости от контекста. Если бы мы знали, какие токены релевантны текущему, то узнать его контекст можно с помощью средневзвешенного — или, в общем случае, линейной комбинации — этих эмбеддингов.

На практике мы часто параллельно запускаем несколько таких блоков «внимания» (self-attention), чтобы трансформер одновременно обрабатывал разные части входной последовательности — это называют multi-head attention. Идея проста: выходы нескольких независимых блоков self-attention конкатенируются и передаются через линейный слой. Он позволяет модели комбинировать контекстуальную информацию из каждого блока внимания.

Мы поверхностно рассмотрели, что пытается добиться механизм внимания. Давайте теперь разберемся, как именно это реализовано.

Листинг 1 – Класс слоя «Внимания»

```
1  class MultiHeadAttentionLayer(nn.Module):
2      def __init__(self, hid_dim, n_heads, dropout, device):
3          super().__init__()
4          assert hid_dim % n_heads == 0
5          self.hid_dim = hid_dim
6          self.n_heads = n_heads
7          self.head_dim = hid_dim // n_heads
8          self.fc_q = nn.Linear(hid_dim, hid_dim)
9          self.fc_k = nn.Linear(hid_dim, hid_dim)
10         self.fc_v = nn.Linear(hid_dim, hid_dim)
11         self.fc_o = nn.Linear(hid_dim, hid_dim)
12         self.dropout = nn.Dropout(dropout)
13         self.scale =
```

```
    torch.sqrt(torch.FloatTensor([self.head_dim])).to(device)
14
15      def forward(self, query, key, value, mask = None):
16
17          batch_size = query.shape[0]
18
19          #query = [batch size, query len, hid dim]
20          #key = [batch size, key len, hid dim]
21          #value = [batch size, value len, hid dim]
22          Q = self.fc_q(query)
23          K = self.fc_k(key)
24          V = self.fc_v(value)
25          #Q = [batch size, query len, hid dim]
26          #K = [batch size, key len, hid dim]
27          #V = [batch size, value len, hid dim]
28          Q = Q.view(batch_size, -1, self.n_heads, self.head_dim)
    .permute(0, 2, 1, 3)
29          K = K.view(batch_size, -1, self.n_heads, self.head_dim)
    .permute(0, 2, 1, 3)
30          V = V.view(batch_size, -1, self.n_heads, self.head_dim)
    .permute(0, 2, 1, 3)
31          #Q = [batch size, n heads, query len, head dim]
32          #K = [batch size, n heads, key len, head dim]
33          #V = [batch size, n heads, value len, head dim]
34          energy = torch.matmul(Q, K.permute(0, 1, 3, 2))
    / self.scale
35          #energy = [batch size, n heads, query len, key len]
36          if mask is not None:
37              energy = energy.masked_fill(mask == 0, -1e10)
38          # [batch size, 1, trg len, trg len]
39          attention = torch.softmax(energy, dim = -1)
40          #attention = [batch size, n heads, query len, key len]
41          x = torch.matmul(self.dropout(attention), V)
42          #x = [batch size, n heads, query len, head dim]
43          x = x.permute(0, 2, 1, 3).contiguous()
44          #x = [batch size, query len, n heads, head dim]
45          x = x.view(batch_size, -1, self.hid_dim)
46          #x = [batch size, query len, hid dim]
47          x = self.fc_o(x)
48          #x = [batch size, query len, hid dim]
49          return x, attention
```

2.4.Полносвязная НС

После блока «внимания», если посмотреть на рисунок 1, идет блок с полносвязной нейронной сетью. Этот блок включает в себя самую простую нейронную архитектуру MLP (Multilayer perception), обрабатывающую представления, полученные на выходе из слоя «внимания».

Этот слой нужен, чтобы среди связей слов в предложении найти самые нужные, найти какие-то закономерности и просто добавляет «свободные» нейроны для «мыслительных» процессов.

Реализация этого блока в коде выглядит так:

Листинг 2 – Класс полносвязной НС

```
1       class PositionwiseFeedforwardLayer(nn.Module):
2           def __init__(self, hid_dim, pf_dim, dropout):
3               super().__init__()
4
5               self.fc_1 = nn.Linear(hid_dim, pf_dim)
6               self.fc_2 = nn.Linear(pf_dim, hid_dim)
7
8               self.dropout = nn.Dropout(dropout)
9
10          def forward(self, x):
11
12              #x = [batch size, seq len, hid dim]
13
14              x = self.dropout(torch.relu(self.fc_1(x)))
15
16              #x = [batch size, seq len, pf dim]
17
18              x = self.fc_2(x)
19
20              #x = [batch size, seq len, hid dim]
21
22          return x
```

2.5.Какие бывают трансформеры?

Мы рассмотрели основные блоки из которых создают трансформерные архитектуры. Существует много разных трансформерных архитектур, и большинство можно разделить на три типа.

1) Энкодеры

Модели-энкодеры синтезируют контекстуальные эмбеддинги, которые можно использовать в последующих задачах вроде классификации или распознавания именованных сущностей, поскольку механизм внимания может обрабатывать всю входящую последовательность. Именно этот тип архитектуры мы рассмотрели в этой статье. Самое популярное семейство чистых трансформеров-энкодеров — это BERT и его разновидности.

Передав данные через один или несколько блоков-трансформеров, мы получаем сложную матрицу контекстуализированных эмбеддингов, которая содержит по эмбеддингу на каждый токен последовательности. Но чтобы использовать эти данные для последующих задач вроде классификации нужно сделать одно предсказание. Обычно берут первый токен и передают через классификатор, в котором есть слои Dropout и Linear. Результат работы этих слоев можно пропустить через МЛФ для превращения в вероятности классов.

2) Декодеры

Этот тип архитектур почти идентичен предыдущему, главное отличие в том, что декодеры используют маскированный (или причинный) слой self-attention, поэтому механизм внимания может принимать только текущий и предыдущие элементы входной последовательности. То есть контекстуальные эмбеддинги учитывают только предыдущий контекст. К популярным моделям-декодерам относится семейство GPT.

Обычно этого добиваются с помощью маскирования оценок внимания с помощью двоичной нижнетреугольной матрицы и замены немаскированных элементов отрицательной бесконечностью (потом при прогоне через МЛФ получают для этих позиций оценки внимания равные нулю).

3) Энкодеры-декодеры

Изначально трансформеры были представлены как архитектура для машинного перевода и использовали и энкодеры, и декодеры. С помощью энкодеров создается промежуточное представление, прежде чем с помощью декодера переводить в желаемый формат. Хотя энкодеры-декодеры сегодня менее распространены, архитектуры вроде T5 показывают, что задачи вроде ответов на вопросы, подведения итогов и классификации можно представить в виде преобразование последовательности в последовательность и решить с помощью описанного подхода.

Главное отличие архитектур типа энкодер-декодер заключается в том, что декодер использует энкодер-декодерное внимание: при вычислении внимания используется результат энкодера (K и V) и входные данные декодера (Q). Сравните с self-attention, когда для всех входных данных используется одна и та же входная матрица эмбеддингов. При этом общий процесс синтеза очень похож на процесс в архитектурах декодеров.

На рисунке 1 представлена, как раз, архитектура такого трансформера.

3. АНАЛИЗ РЕЗУЛЬТАТОВ

Результат практической работы показал отличную эффективность в решении множества задач связанных с NLP (Natural Language Processing). Подробный разбор всех блоков архитектуры позволяет более легко понять работу и устройство сети. Рассмотрены задачи, которые могут быть решены такой архитектурой.

ЗАКЛЮЧЕНИЕ

В ходе выполнения эксплуатационной практики прошло ознакомление с написанием искусственного интеллекта на основе архитектуры трансформера. Изучены возможности модуля PyTorch языка программирования Python. Разобраны главные математические и биологические основы такой НС.

Создана НС с архитектурой трансформера с возможностью обучения на различных датасетах. Изучены различные блоки такой сети. Изучены необходимые для этого функции. Разобрана теоретическая составляющая трансформерных нейронных сетей.

Разобраны различные области применения НС. Изучены тонкости настройки трансформера для разных типов задач.

СПИСОК ЛИТЕРАТУРЫ

1. PyTorch Documentation [электронный ресурс] // URL: https://pytorch.org/docs/stable/index.html. Дата обращения: 16.05.2024 - 26.05.2024.
2. Deep Learning School Tutors [электронный ресурс] // URL: https://stepik.org/course/196142/syllabus. Дата обращения: 16.05.2024 - 26.05.2024.
3. Хабр статья про трансформер [электронный ресурс] // URL: https://habr.com/ru/companies/mws/articles/770202/. Дата обращения: 24.05.2024 – 26.05.2024.
4. PyTorch Tutorials [Электронный ресурс] – URL: https://pytorch.org/tutorials/ (дата обращения: 14.05.2024 - 20.05.2024)
5. Pandas Documentation [Электронный ресурс] – URL: https://pandas.pydata.org/docs/ (дата обращения: 15.05.2024 - 22.05.2024)