ИСКУССТВЕННЫЕ НЕЙРОННЫЕ СЕТИ

Лекция 12

Механизм внимания и архитектура Transformer

Scaled Dot-Product Attention, Multi-Head Attention, позиционное кодирование и модели Encoder–Decoder

Преподаватель: Аршын Алтыбай

Кафедра компьютерной инженерии,
Алматинский технологический университет, Алматы, Казахстан

Учебный год: 2026

> **Связь с курсом.** Лекции 9–10 были посвящены RNN, LSTM и GRU. В этой лекции изучаем механизм, позволяющий моделировать зависимости между позициями последовательности без обязательного рекуррентного вычисления. Центральный первоисточник — работа Vaswani и соавторов [9]; математическую и инженерную основу дают [6, 3, 4, 5].

1. Цели лекции и предварительные знания

Цель: вывести математическую конструкцию механизма внимания; понять, как она преобразуется в многоголовое самовнимание и блок Transformer; научиться правильно обращаться с размерностями тензоров, масками и позиционными признаками; реализовать компактные численные примеры на NumPy и PyTorch.

Предварительные знания: матричное умножение, скалярное произведение, softmax, цепное правило, градиентный спуск, мини-пакеты, embeddings, архитектуры RNN/LSTM/GRU. В лекции необходимые определения повторяются.

По завершении студент должен уметь:

1. объяснять ограничения фиксированного вектора контекста и последовательной рекурсии;
2. выписывать Q, K, V и проверять их формы;
3. рассчитывать матрицу $QK^\top$ и softmax вручную;
4. обосновывать масштабирование $1/\sqrt{d_k}$;

5. строить padding-mask и causal-mask, избегая утечки будущего;
6. реализовывать Self-Attention и Cross-Attention;
7. выводить формы тензоров для Multi-Head Attention;
8. пояснять синусоидальные и обучаемые позиционные представления;
9. собирать Encoder/Decoder Transformer и обсуждать его сложность;
10. проектировать воспроизводимый эксперимент на задаче последовательностей.

2. Зачем нужен Attention: постановка задачи

2.1 Последовательности и дальние зависимости

Пусть входная последовательность $x_1, \dots, x_n$, где $x_i \in \mathbb{R}^{d_{\text{model}}}$. Задачи включают машинный перевод, классификацию текста, извлечение информации, прогнозирование временных рядов, работу с аудио и изображениями. В RNN состояние определяется рекурсией $h_t = f(x_t, h_{t-1})$. Чтобы сигнал от x_1 влиял на h_n, необходимо пройти много последовательных переходов. Механизмы LSTM и GRU смягчают проблему затухания градиента, но последовательная зависимость вычислений остаётся [6, 4].

В ранних Encoder–Decoder моделях последовательность сворачивалась в единый вектор. Для длинных предложений он мог становиться информационным узким местом. Внимание позволяет каждому выходному шагу выбирать релевантные позиции входной последовательности, формируя *контекстный вектор* с адаптивными весами [8].

2.2 Пример из перевода

Переведём предложение «Студент читает интересную книгу» на другой язык. Когда декодер генерирует слово, соответствующее «книгу», ему может потребоваться сильнее учитывать исходную позицию слова «книгу», чем позиции остальных слов. Однако коэффициенты внимания — это обучаемые статистические веса, а не гарантированная лингвистическая разметка. Аналогично в прогнозировании временного ряда текущий прогноз может использовать разные исторические моменты с различным весом.

> **Основное различие** Attention — операция вычисления взвешенной суммы значений, коэффициенты которой зависят от совместимости запросов и ключей. Transformer — целая архитектура из блоков внимания, нелинейных преобразований, нормализации и остаточных связей. Эти понятия не являются синонимами.

3. Математические основы: softmax и взвешенная сумма

3.1 Взвешенное среднее

Пусть имеются векторы $v_1, \ldots, v_n \in \mathbb{R}^{d_v}$ и веса $\alpha_1, \ldots, \alpha_n$. Контекстный вектор задаётся формулой

$$c = \sum_{j=1}^{n} \alpha_j v_j, \qquad \alpha_j \geq 0, \quad \sum_{j=1}^{n} \alpha_j = 1. \tag{1}$$

Таким образом, c лежит в выпуклой оболочке значений v_j. Это утверждение относится к *одной* операции взвешенной суммы до последующих линейных преобразований и residual connections.

3.2 Softmax: определение и свойства

Из действительных оценок $s_1, \ldots, s_n$ получаем нормированные веса:

$$\alpha_j = \frac{\exp(s_j)}{\sum_{\ell=1}^{n} \exp(s_\ell)}. \tag{2}$$

Так как экспонента положительна, $\alpha_j > 0$ и сумма весов равна единице. Softmax инвариантен относительно добавления общего числа c ко всем оценкам: $\mathrm{softmax}(s + c\mathbf{1}) = \mathrm{softmax}(s)$. В численных программах поэтому используют вычислительно устойчивую формулу $\exp(s_j - \max_\ell s_\ell)$.

> **Производная softmax.** Пусть $\alpha_i = \exp(s_i)/Z$. Тогда
>
> $$\frac{\partial \alpha_i}{\partial s_j} = \alpha_i(\delta_{ij} - \alpha_j), \qquad J_{\mathrm{softmax}} = \mathrm{diag}(\alpha) - \alpha\alpha^\top.$$
>
> Здесь δ_{ij} — символ Кронекера. Эта матрица используется при обратном распространении через механизм внимания.

3.3 Ручной пример

Для оценок $s = (1, 2, 0)$ имеем $\exp(s) \approx (2.7183, 7.3891, 1)$; знаменатель 11.1074, а

$$\alpha \approx (0.2447, 0.6652, 0.0900).$$

Если значения одномерны, $v = (2, 4, -1)$, то контекст

$$c \approx 0.2447 \cdot 2 + 0.6652 \cdot 4 + 0.0900 \cdot (-1) \approx 3.0602.$$

Проверка: контекст находится в интервале $[-1, 4]$. Вес 0.6652 не означает вероятность истинности второго слова; это коэффициент нормированной смеси признаков.

4. От общего внимания к Query, Key и Value

4.1 Формализация поиска релевантной информации

Используется аналогия с поиском: *запрос* (Query, q) описывает то, что ищем; *ключ* (Key, k) характеризует, к чему можно обращаться; *значение* (Value, v) содержит информацию, которую агрегируем. Для набора позиций ключи $k_j \in \mathbb{R}^{d_k}$ и значения $v_j \in \mathbb{R}^{d_v}$. При заданном $q \in \mathbb{R}^{d_k}$ оценки совместимости можно вычислять так:

$$s_j = q^\top k_j, \qquad \alpha_j = \mathrm{softmax}(s)_j, \qquad c(q, K, V) = \sum_j \alpha_j v_j. \tag{3}$$

Важно: q и k_j должны иметь одинаковую размерность d_k для скалярного произведения; размерность v_j может отличаться.

4.2 Аддитивное и скалярное внимание

В аддитивном внимании совместимость вычисляется обучаемой сетью

$$s(q, k) = w_a^\top \tanh(W_q q + W_k k + b), \tag{4}$$

а в dot-product attention — скалярным произведением $q^\top k$. Первый вариант использовался в ранних нейронных системах перевода [8], а масштабированное скалярное внимание стало центральным компонентом Transformer [9].

4.3 Матрицы запросов, ключей и значений

Пусть матрица входных представлений $X \in \mathbb{R}^{n \times d_{\text{model}}}$. Обучаемые проекции имеют формы

$$W_Q \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad W_K \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad W_V \in \mathbb{R}^{d_{\text{model}} \times d_v}. \tag{5}$$

Тогда

$$Q = XW_Q \in \mathbb{R}^{n \times d_k}, \quad K = XW_K \in \mathbb{R}^{n \times d_k}, \quad V = XW_V \in \mathbb{R}^{n \times d_v}. \tag{6}$$

В практических слоях к линейным проекциям могут добавляться смещения. Их наличие влияет на подсчёт параметров, но не на приведённые формы.

5. Scaled Dot-Product Attention: вывод основной формулы

5.1 Матричная форма

Взаимные скалярные произведения запросов и ключей вычисляются как

$$QK^\top \in \mathbb{R}^{n_q \times n_k},$$

где n_q — число запросов и n_k — число ключей. К каждой строке применяем softmax, получаем веса $A \in \mathbb{R}^{n_q \times n_k}$. Результат агрегации значений:

$$\boxed{\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V.} \tag{7}$$

Это фундаментальная формула архитектуры Transformer [9]. Для совместимости размерностей нужны $Q \in \mathbb{R}^{n_q \times d_k}$, $K \in \mathbb{R}^{n_k \times d_k}$, $V \in \mathbb{R}^{n_k \times d_v}$. Итог имеет форму $n_q \times d_v$.

5.2 Почему делим на $\sqrt{d_k}$?

Предположим для иллюстрации, что компоненты q_ℓ и k_ℓ независимы, имеют нулевое среднее и единичную дисперсию. Тогда $q^\top k = \sum_{\ell=1}^{d_k} q_\ell k_\ell$ имеет дисперсию d_k. Для больших d_k значения скалярных произведений могут становиться крупными и softmax — чрезмерно резким. Масштабирование на $1/\sqrt{d_k}$ приводит дисперсию к порядку единицы при сделанных допущениях. Это не универсальная гарантия несатурирующего softmax при любых данных и весах.

5.3 Температура softmax

Более общий вариант $\mathrm{softmax}(s/\tau)$ задаёт температуру $\tau > 0$. При $\tau \to \infty$ распределение приближается к равномерному, при $\tau \to 0^+$ концентрируется на максимальных оценках, если максимум единственный. Стандартный масштаб $\sqrt{d_k}$ выполняет сходную функцию нормировки величины скалярных произведений.

6. Полный численный пример Self-Attention

6.1 Два токена, два признака

Возьмём $n = 2$, $d_k = d_v = 2$ и упрощённые проекции $W_Q = W_K = W_V = I_2$. Пусть

$$X = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad Q = K = V = X.$$

Тогда

$$QK^\top = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \qquad S = \frac{QK^\top}{\sqrt{2}} = \begin{pmatrix} 0.7071 & 0 \\ 0 & 0.7071 \end{pmatrix}.$$

Для каждой строки $\mathrm{softmax}(0.7071, 0)$ получаем

$$a = \frac{e^{0.7071}}{e^{0.7071} + 1} \approx 0.6698, \qquad b \approx 0.3302.$$

Следовательно,

$$A = \begin{pmatrix} a & b \\ b & a \end{pmatrix}, \qquad AV \approx \begin{pmatrix} 0.6698 & 0.3302 \\ 0.3302 & 0.6698 \end{pmatrix}. \tag{8}$$

Оба токена после внимания содержат информацию о двух позициях; они различаются относительным вкладом. Этот пример не использует обучение, но показывает вычислительный смысл операции.

6.2 Тест чувствительности к перестановкам

При перестановке строк X одной и той же матрицей перестановки P и без позиционных признаков

$$Q' = PQ,\ K' = PK,\ V' = PV, \qquad \text{Attention}(PQ, PK, PV) = P\,\text{Attention}(Q, K, V).$$

Это *эквивариантность* относительно перестановки, а не инвариантность отдельных выходных векторов. Отсюда видно, почему порядок нужно кодировать отдельно.

Пример 1. Вычисление внимания вручную на NumPy

```
import numpy as np
X = np.eye(2)
Q, K, V = X, X, X
scores = Q @ K.T / np.sqrt(Q.shape[-1])
z = scores - scores.max(axis=-1, keepdims=True)
A = np.exp(z) / np.exp(z).sum(axis=-1, keepdims=True)
O = A @ V
print(np.round(scores, 4))
print(np.round(A, 4))
print(np.round(O, 4))
```

Ожидаемый вывод

```
[[0.7071 0.    ]
 [0.     0.7071]]
[[0.6698 0.3302]
 [0.3302 0.6698]]
[[0.6698 0.3302]
 [0.3302 0.6698]]
```

7. Маски внимания: padding и причинность

7.1 Проблема различных длин

В одном мини-пакете предложения могут содержать разное число токенов. Чтобы выровнять длины, добавляются специальные токены PAD. Они не должны получать вес как содержательные позиции. Для этого к матрице логитов $S = QK^\top/\sqrt{d_k}$ прибавляют аддитивную маску M:

$$\text{Attention}(Q, K, V; M) = \text{softmax}(S + M)V, \tag{9}$$

где разрешённым позициям соответствует $M_{ij} = 0$, запрещённым — $M_{ij} = -\infty$ (теоретически) или достаточно большое отрицательное число (в определённых численных реализациях). Строки, полностью состоящие из $-\infty$, требуют специальной обработки: обычный softmax не определён в этом случае.

7.2 Causal Mask в авторегрессионной генерации

Для предсказания следующего токена позиция i не должна видеть будущие позиции $j > i$. Поэтому

$$M_{ij} = \begin{cases} 0, & j \le i, \\ -\infty, & j > i. \end{cases} \tag{10}$$

При длине 4 маска имеет вид

$$M = \begin{pmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

С такой маской первые токены не видят будущие. *Важно*: именно правильное расположение маски и сдвиг целевой последовательности препятствуют информационной утечке.

7.3 Численный пример с маской

Пусть $S = \left(\begin{smallmatrix} 2 & 1 \\ 3 & 4 \end{smallmatrix}\right)$. Причинная маска даёт первую строку $(2, -\infty)$, откуда softmax равен $(1, 0)$. Для второй строки softmax от $(3, 4)$ равен приблизительно $(0.2689, 0.7311)$. Следовательно, первый выход зависит только от первого значения, а второй — от обоих.

Практическая ошибка У разных библиотечных функций маска может означать «True = запретить» либо «True = разрешить», а также использовать разный порядок измерений. Никогда не переносите маску из одного API в другое без проверки документации и теста на игрушечном примере.

8. Self-Attention и Cross-Attention

8.1 Самовнимание

При *Self-Attention* запросы, ключи и значения образуются из одного и того же набора X с помощью разных обучаемых проекций. В каждой позиции модель может агрегировать сведения о других разрешённых позициях. В энкодере часто разрешены все позиции, кроме padding; в авторегрессионном декодере запрещены будущие позиции.

8.2 Перекрёстное внимание

При *Cross-Attention* запросы формируются из одного набора Y, а ключи и значения — из другого X:

$$Q = YW_Q, \qquad K = XW_K, \qquad V = XW_V. \tag{11}$$

Если $Y \in \mathbb{R}^{m \times d_{\text{model}}}$ и $X \in \mathbb{R}^{n \times d_{\text{model}}}$, то матрица оценок имеет форму $m \times n$. В классическом Encoder–Decoder декодер таким способом обращается к выходам энкодера [9].

8.3 Сопоставление

Характеристика	Self-Attention	Cross-Attention
Источник Q	тот же набор, что K, V	другой набор
Размер матрицы весов	$n \times n$	$m \times n$
Типичное применение	внутренние связи токенов	связь текста с выходом энкодера
Маскирование	padding, иногда causal	обычно маска ключей/памяти

9. Multi-Head Attention: несколько пространств отношений

9.1 Математическое определение

Одна голова внимания агрегирует информацию на основании одного набора обучаемых проекций. Для H голов:

$$\text{head}_h = \text{Attention}(QW_Q^{(h)}, KW_K^{(h)}, VW_V^{(h)}), \tag{12}$$

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_H)W_O. \tag{13}$$

Обычно для модели ширины d_{model} выбирают $d_k = d_v = d_{\text{model}}/H$ при делимости на H [9]. Но это лишь распространённая конфигурация, а не математическая необходимость.

9.2 Пример форм тензоров

Пусть $B = 16$, $n = 32$, $d_{\text{model}} = 128$, $H = 8$. Каждая голова работает с $d_k = 16$:

$$X : (16, 32, 128), \quad Q : (16, 8, 32, 16), \quad K : (16, 8, 32, 16).$$

После умножения $QK^\top$ по последним двум измерениям получаем $(16, 8, 32, 32)$; результат каждой головы имеет форму $(16, 8, 32, 16)$; после объединения голов — $(16, 32, 128)$.

9.3 Число параметров

При стандартных линейных преобразованиях $W_Q, W_K, W_V, W_O \in \mathbb{R}^{d\times d}$ без смещений

$$P_{\text{MHA}} = 4d^2. \tag{14}$$

При $d = 128$ это $65\,536$ параметров. Если каждое преобразование имеет bias размерности d, то $P = 4d^2 + 4d = 66\,048$. Число голов при фиксированной общей ширине проекций само по себе не умножает число параметров на H.

9.4 Интерпретация голов

Разные головы могут обучиться использовать разные статистические отношения между токенами. Однако нельзя гарантировать, что каждая голова соответствует отдельному лингвистическому понятию. Важность голов определяется экспериментально, и распределения внимания не следует автоматически трактовать как полноценное объяснение предсказания модели.

10. Позиционная информация

10.1 Почему порядок теряется

Без позиционных признаков Self-Attention эквивариантен перестановке входных токенов. Например, последовательности «собака укусила человека» и «человек укусил собаку» содержат те же слова, но различаются порядком и смыслом. Поэтому представление токена должно содержать информацию о позиции либо механизм внимания должен явно учитывать относительный порядок.

10.2 Синусоидальное позиционное кодирование

В оригинальной архитектуре Transformer предложено [9]

$$\mathrm{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right), \tag{15}$$

$$\mathrm{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right). \tag{16}$$

Разные координаты имеют различные масштабы изменения по позиции. Позиционный вектор складывается с embedding токена, поэтому обе матрицы имеют размер (n, d_{model}).

10.3 Ручной пример

Для $d_{\text{model}} = 4$ на позиции $pos = 0$ получаем

$$\mathrm{PE}(0) = (0, 1, 0, 1).$$

На позиции $pos = 1$ пары координат вычисляются как $(\sin 1, \cos 1)$ и $(\sin 0.01, \cos 0.01)$, то есть приблизительно

$$\mathrm{PE}(1) = (0.84147, 0.54030, 0.01000, 0.99995).$$

Пример 2. Синусоидальное кодирование позиций

```python
import torch
import math

def positional_encoding(length, d_model):
    assert d_model % 2 == 0
    pos = torch.arange(length).unsqueeze(1).float()
    i = torch.arange(0, d_model, 2).float()
    div = torch.exp(-math.log(10000) * i / d_model)
    pe = torch.zeros(length, d_model)
    pe[:, 0::2] = torch.sin(pos * div)
    pe[:, 1::2] = torch.cos(pos * div)
    return pe

print(positional_encoding(2, 4).round(decimals=5))
```

10.4 Обучаемые и относительные позиции

Альтернатива — обучаемая таблица позиционных векторов. Она может быть эффективна для фиксированного диапазона позиций, но перенос на существенно более длинные последовательности требует дополнительной осторожности. Относительные позиционные смещения изменяют оценки $QK^\top$ в зависимости от расстояния между позициями. В современных архитектурах применяются и вращательные позиционные представления (RoPE); конкретные способы отличаются и требуют отдельной проверки свойств экстраполяции [12].

11. Полный блок Transformer Encoder

11.1 Остаточные соединения и нормализация

Оригинальный encoder-layer включает Multi-Head Self-Attention и двухслойную позиционную FFN. После каждого подблока используются residual connection и Layer Normalization:

$$U = \mathrm{LayerNorm}\big(X + \mathrm{Dropout}(\mathrm{MHA}(X, X, X))\big), \tag{17}$$

$$Y = \mathrm{LayerNorm}\big(U + \mathrm{Dropout}(\mathrm{FFN}(U))\big). \tag{18}$$

Это Post-LN вариант, близкий к оригинальной статье. В других архитектурах LayerNorm ставится *до* подблока (Pre-LN), что изменяет путь градиента и свойства оптимизации. Нельзя утверждать, что существует единственное расположение нормализации для всех Transformers.

11.2 Позиционно-независимая FFN

Для каждой позиции независимо применяется

$$\mathrm{FFN}(x) = W_2\,\sigma(W_1 x + b_1) + b_2, \tag{19}$$

где $W_1 \in \mathbb{R}^{d_{\mathrm{ff}} \times d_{\mathrm{model}}}$, $W_2 \in \mathbb{R}^{d_{\mathrm{model}} \times d_{\mathrm{ff}}}$. В оригинале используется ReLU, современные варианты нередко применяют GELU или gated-активации. Параметры двух линейных отображений совместно с bias:

$$P_{\mathrm{FFN}} = 2 d_{\mathrm{model}} d_{\mathrm{ff}} + d_{\mathrm{ff}} + d_{\mathrm{model}}.$$

11.3 Layer Normalization

Для вектора $x \in \mathbb{R}^d$ LayerNorm вычисляет среднее и дисперсию по его компонентам:

$$\mu = d^{-1} \sum_{j=1}^{d} x_j, \qquad \sigma^2 = d^{-1} \sum_{j=1}^{d} (x_j - \mu)^2, \tag{20}$$

$$\mathrm{LN}(x) = \gamma \odot \frac{x - \mu}{\sqrt{\sigma^2 + \varepsilon}} + \beta. \tag{21}$$

Параметры $\gamma, \beta \in \mathbb{R}^d$ обучаются. В отличие от Batch Normalization, статистики не вычисляются по всему mini-batch в стандартной форме LayerNorm.

11.4 Подсчёт параметров полного encoder-layer

Для $d = 128$, $d_{\mathrm{ff}} = 512$, $H = 8$, с bias во всех четырёх проекциях внимания и двух FFN:

$$\begin{aligned} P_{\mathrm{MHA}} &= 4d^2 + 4d = 66\,048, \\ P_{\mathrm{FFN}} &= 2 d d_{\mathrm{ff}} + d_{\mathrm{ff}} + d = 131\,712, \\ P_{\mathrm{LN}} &= 2 \times (2d) = 512. \end{aligned}$$

Следовательно, $P_{\mathrm{layer}} = 198\,272$. Здесь принято, что у двух нормализаций есть обучаемые γ, β и остальные специальные параметры отсутствуют. Число параметров embedding-слоя и классификационной головы учитывается отдельно.

12. Transformer Decoder и архитектура Encoder–Decoder

12.1 Три подблока декодера

Классический decoder-layer содержит: (1) masked self-attention над уже известными целевыми токенами; (2) cross-attention с выходом энкодера; (3) FFN. Residual connections и LayerNorm сопровождают каждый подблок [9].

1. **Masked self-attention.** На этапе обучения позиция t обращается только к позициям $\leq t$.
2. **Cross-attention.** Запросы принадлежат состояниям декодера, ключи и значения — выходам энкодера.
3. **FFN.** Выполняется независимое нелинейное преобразование каждого позиционного вектора.

12.2 Teacher Forcing и смещение целей

Пусть целевой текст содержит токены $(\mathrm{BOS}, y_1, \ldots, y_T, \mathrm{EOS})$. Вход декодера смещён относительно целевых меток: по первым токенам нужно предсказать следующие. Например, вход (BOS, y_1, y_2), целевые выходы (y_1, y_2, EOS). Неправильный сдвиг создаёт тривиальную утечку ответа. Обучение обычно использует cross-entropy по непустым целевым позициям с исключением padding-меток.

12.3 Авторегрессионная генерация

При генерации на каждом шаге оценивается $p_\theta(y_t \mid y_{<t}, x)$, затем выбирается следующий токен (greedy search, sampling или beam search). Таким образом,

$$p_\theta(y_{1:T} \mid x) = \prod_{t=1}^{T} p_\theta(y_t \mid y_{<t}, x).$$

Сильно отличать *обучение с teacher forcing* от *инференса с собственными прогнозами*: при ошибке на раннем шаге последующие контексты могут измениться.

13. Encoder-only, Decoder-only и Encoder–Decoder

Свойство	Encoder-only	Decoder-only	Encoder–Decoder
Направление внимания	обычно двустороннее	причинное	двустороннее + причинное + cross
Типичные цели	классификация извлечение	генерация продолжения	условная генерация, перевод
Пример семейства	BERT	GPT	T5
Целевая величина	зависит от задачи	следующий токен	целевая последова-тельность

Эти названия обозначают архитектурные семейства и типичные способы обучения, а не строгие ограничения применимости. Например, decoder-only модели можно адаптировать для классификации, а encoder-only — для извлечения признаков [10, 11].

14. Вычислительная сложность и ограничения

14.1 Стоимость Self-Attention

При длине последовательности n, ширине d и фиксированной общей ширине всех голов произведение $QK^\top$ требует порядка $O(n^2 d)$ операций, а внимание хранит матрицу размера $O(n^2)$ на голову (в наивной реализации). Проекции Q, K, V и выходная проекция требуют $O(nd^2)$. Итого по порядку:

$$O(nd^2 + n^2 d), \qquad \text{наивная память для весов внимания } O(Hn^2).$$

Эти выражения не означают, что Transformer всегда медленнее RNN: параллелизация и аппаратные оптимизации играют существенную роль.

14.2 Пример роста памяти

Для $n = 1024$, $H = 8$, $B = 4$ матрицы внимания содержат

$$B\,H\,n^2 = 4 \cdot 8 \cdot 1024^2 = 33\,554\,432$$

элемента. При float32 одно такое хранилище занимает примерно 134.2 млн байт (около 128 MiB), не считая других тензоров, промежуточных градиентов и дополнительных копий. Для $n = 2048$ объём матрицы увеличится в четыре раза.

14.3 Эффективные реализации

Оптимизированные kernel-реализации, включая FlashAttention, уменьшают обращения к памяти и могут обходиться без явного хранения полной матрицы вероятностей, сохраняя точный результат внимания в пределах численной арифметики выбранного алгоритма [13]. Это не превращает все варианты точного полного внимания в линейные по n по числу скалярных произведений. Также исследуются локальное, разреженное и приближённое внимание, но они имеют свои ограничения и компромиссы.

15. Программная реализация Scaled Dot-Product Attention

15.1 NumPy без обучаемых параметров

Пример 3. Универсальная функция attention на NumPy

```python
import numpy as np

def softmax(x, axis=-1):
    x = x - np.max(x, axis=axis, keepdims=True)
    ex = np.exp(x)
    return ex / ex.sum(axis=axis, keepdims=True)

def attention(q, k, v, allow=None):
    # q: (nq, dk), k: (nk, dk), v: (nk, dv)
    logits = q @ k.T / np.sqrt(q.shape[-1])
    if allow is not None:
        # allow[i,j] == True means query i can see key j.
        if not np.all(allow.any(axis=-1)):
            raise ValueError("At least one key per query is needed")
        logits = np.where(allow, logits, -np.inf)
    probs = softmax(logits)
    return probs @ v, probs
```

Пример 3 (продолжение)

```
X = np.eye(2)
causal = np.tril(np.ones((2, 2), dtype=bool))
output, weights = attention(X, X, X, causal)
print(np.round(weights, 4))
```

Ожидаемый вывод

```
[[1.     0.    ]
 [0.3302 0.6698]]
```

Численный код предполагает, что разрешён хотя бы один ключ для каждого запроса. Векторизация для batch/head выполняется с помощью `@` по последним двум осям.

15.2 Полезные проверки

После реализации студент должен проверить:

1. форма Attention(Q, K, V) равна (n_q, d_v);
2. сумма весов каждой строки равна единице с учётом округления;
3. веса запрещённых позиций равны нулю;
4. при $n_k = 1$ каждый запрос получает единственный v_1;
5. градиент обучаемой реализации имеет конечные значения.

16. Multi-Head Self-Attention на PyTorch

Пример 4. Минимальная реализация Multi-Head Self-Attention

```
import math
import torch
from torch import nn

class MultiHeadSelfAttention(nn.Module):
    def __init__(self, d_model=128, n_heads=8):
        super().__init__()
        assert d_model % n_heads == 0
        self.n_heads = n_heads
        self.d_head = d_model // n_heads
        self.qkv = nn.Linear(d_model, 3 * d_model)
        self.proj = nn.Linear(d_model, d_model)

    def forward(self, x, causal=False):
```

Пример 4 (продолжение)

```
        # x: (batch, seq, d_model)
        B, T, D = x.shape
        qkv = self.qkv(x).view(B, T, 3,
                               self.n_heads, self.d_head)
        q, k, v = qkv.unbind(dim=2)
        q = q.transpose(1, 2)  # (B, heads, T, d_head)
        k = k.transpose(1, 2)
        v = v.transpose(1, 2)
        scores = q @ k.transpose(-2, -1)
        scores = scores / math.sqrt(self.d_head)
        if causal:
            forbid = torch.triu(
                torch.ones(T, T, device=x.device,
                           dtype=torch.bool), diagonal=1)
            scores = scores.masked_fill(forbid, float('-inf'))
        weights = torch.softmax(scores, dim=-1)
        y = weights @ v
        y = y.transpose(1, 2).contiguous().view(B, T, D)
        return self.proj(y)

torch.manual_seed(7)
m = MultiHeadSelfAttention(128, 8)
x = torch.randn(4, 16, 128)
print(tuple(m(x, causal=True).shape))
```

Ожидаемый вывод

```
(4, 16, 128)
```

В этом учебном коде опущены padding mask, dropout и специализированные kernel-оптимизации. Код подходит для понимания форм и причинного ограничения, но сам по себе не является полной реализацией Transformer.

17. Создание Encoder-layer в PyTorch

Пример 5. Один Encoder-layer: Attention + FFN + Residual

```
import torch
from torch import nn

class EncoderBlock(nn.Module):
    def __init__(self, d_model=64, n_heads=4,
                 d_ff=256, dropout=0.1):
        super().__init__()
        self.attn = nn.MultiheadAttention(
```

Пример 5 (продолжение)

```
            d_model, n_heads, dropout=dropout,
            batch_first=True)
        self.ffn = nn.Sequential(
            nn.Linear(d_model, d_ff), nn.GELU(),
            nn.Dropout(dropout), nn.Linear(d_ff, d_model))
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.drop = nn.Dropout(dropout)

    def forward(self, x, key_padding_mask=None):
        # Pre-LN block; key_padding_mask True means "ignore".
        h = self.norm1(x)
        a, _ = self.attn(h, h, h,
                         key_padding_mask=key_padding_mask,
                         need_weights=False)
        x = x + self.drop(a)
        x = x + self.drop(self.ffn(self.norm2(x)))
        return x

block = EncoderBlock()
example = torch.randn(3, 20, 64)
mask = torch.zeros(3, 20, dtype=torch.bool)
mask[:, -2:] = True
print(tuple(block(example, mask).shape))
```

Ожидаемый вывод

```
(3, 20, 64)
```

В отличие от формул Post-LN, приведённых ранее для оригинального Transformer, этот блок использует Pre-LN. Это осознанный архитектурный вариант. Заметим, что маска ключей подавляет обращение к PAD, но для агрегирования потерь по позициям PAD их также нужно исключить из loss.

18. Практический эксперимент: классификация последовательностей

18.1 Постановка

Рассмотрим учебный набор коротких последовательностей целых токенов $x_1, \ldots, x_T$. Задача бинарной классификации: определить, встречается ли определённый маркерный токен. Это не заменяет реальных задач NLP, однако позволяет независимо проверить работу embeddings, positional encoding, self-attention и pooling без большого датасета.

18.2 Честное разбиение

Сначала генерируются непересекающиеся обучающая и тестовая выборки с заранее фиксированным seed. Подбор архитектуры и гиперпараметров должен выполняться на validation, а тест использоваться только для итоговой оценки. В учебной искусственной задаче нельзя делать выводы о превосходстве Transformers над другими архитектурами на реальном естественном языке.

Пример 6. Подготовка синтетических данных для классификации

```
import torch

def make_data(n, length=12, vocab=30, seed=0):
    g = torch.Generator().manual_seed(seed)
    # Tokens 2..29; 1 is a special marker, 0 is padding.
    x = torch.randint(2, vocab, (n, length), generator=g)
    y = torch.randint(0, 2, (n,), generator=g)
    locations = torch.randint(0, length, (n,), generator=g)
    positive = (y == 1).nonzero(as_tuple=True)[0]
    x[positive, locations[positive]] = 1
    return x, y.float()

x_train, y_train = make_data(1200, seed=1)
x_val, y_val = make_data(300, seed=2)
x_test, y_test = make_data(300, seed=3)
print(x_train.shape, y_train.shape)
```

Ожидаемый вывод

```
torch.Size([1200, 12]) torch.Size([1200])
```

18.3 Мини-модель на готовых модулях

Пример 7. Классификатор с TransformerEncoder

```
import torch
from torch import nn

class TinyTransformerClassifier(nn.Module):
    def __init__(self, vocab=30, d_model=32,
                 n_heads=4, max_len=64):
        super().__init__()
        self.token_emb = nn.Embedding(vocab, d_model)
        self.pos_emb = nn.Embedding(max_len, d_model)
        layer = nn.TransformerEncoderLayer(
            d_model=d_model, nhead=n_heads,
            dim_feedforward=64, dropout=0.1,
            batch_first=True, norm_first=True)
```

Пример 7 (продолжение)

```
        self.encoder = nn.TransformerEncoder(layer, num_layers=2)
        self.head = nn.Linear(d_model, 1)

    def forward(self, tokens):
        B, T = tokens.shape
        positions = torch.arange(T, device=tokens.device)
        x = self.token_emb(tokens) + self.pos_emb(positions)[None]
        x = self.encoder(x)
        pooled = x.mean(dim=1)  # no PAD in this dataset
        return self.head(pooled).squeeze(-1)

model = TinyTransformerClassifier()
print(model(torch.ones(5, 12, dtype=torch.long)).shape)
```

Ожидаемый вывод

```
torch.Size([5])
```

18.4 Обучение и отчёт об эксперименте

Пример 8. Обучение и оценка без использования теста в настройке

```
import torch
from torch import nn
from torch.utils.data import TensorDataset, DataLoader

torch.manual_seed(42)
train_loader = DataLoader(
    TensorDataset(x_train, y_train), batch_size=64, shuffle=True)
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3)

for epoch in range(8):
    model.train()
    total = 0.0
    for xb, yb in train_loader:
        optimizer.zero_grad(set_to_none=True)
        logits = model(xb)
        loss = criterion(logits, yb)
        loss.backward()
        optimizer.step()
        total += loss.item() * len(xb)
    model.eval()
    with torch.no_grad():
        val_prob = model(x_val).sigmoid()
        val_acc = ((val_prob >= 0.5) == y_val.bool()).float().mean()
    print(epoch + 1, round(total / len(x_train), 4),
          round(val_acc.item(), 4))
```

Пример 8 (продолжение)

```
27
28 # Run once after choosing hyperparameters on validation.
29 model.eval()
30 with torch.no_grad():
31     prob = model(x_test).sigmoid()
32     test_acc = ((prob >= 0.5) == y_test.bool()).float().mean()
33 print("Final test accuracy:", round(test_acc.item(), 4))
```

Воспроизводимость. Конкретные значения accuracy будут зависеть от версии PyTorch, устройств, случайных начальных весов и способа выполнения кода. Поэтому здесь не приводятся вымышленные итоговые проценты. Запустив код, студент обязан включить в отчёт *свои* реальные значения и сведения о среде выполнения.

19. Обратное распространение в механизме внимания

19.1 Пути градиентов

Внимание задаётся композициями матричных произведений, масштаба, softmax и ещё одного произведения. Пусть

$$S = QK^\top/\sqrt{d_k}, \quad A = \mathrm{softmax}(S), \quad O = AV.$$

Если известен $G_O = \partial L/\partial O$, то

$$\frac{\partial L}{\partial V} = A^\top G_O, \tag{22}$$

$$\frac{\partial L}{\partial A} = G_O V^\top. \tag{23}$$

Далее для каждой строки i softmax имеем

$$G_{S,i} = A_i \odot \big(G_{A,i} - \langle G_{A,i}, A_i\rangle \mathbf{1}\big). \tag{24}$$

Следовательно,

$$\frac{\partial L}{\partial Q} = G_S K/\sqrt{d_k}, \tag{25}$$

$$\frac{\partial L}{\partial K} = G_S^\top Q/\sqrt{d_k}. \tag{26}$$

Это вывод для *одной* операции внимания без маски и dropout; в запрещённых маской элементах логитов градиенты должны быть нулевыми. Для обучаемых проекций $Q = XW_Q$ дополнительно $\partial L/\partial W_Q = X^\top(\partial L/\partial Q)$ и аналогично для остальных матриц.

19.2 Почему полезна проверка градиентов

В собственной реализации легко ошибиться с $\top$, масштабом $\sqrt{d_k}$ и осью softmax. Для малых матриц можно сравнить аналитический градиент с конечной разностью

$$\frac{L(\theta+\varepsilon) - L(\theta-\varepsilon)}{2\varepsilon}.$$

Ошибка должна уменьшаться для разумно выбранного ε, пока не начинают преобладать ошибки округления. Автоматическое дифференцирование PyTorch снимает необходимость писать формулы вручную в промышленном коде, но не отменяет необходимости понимать их.

20. Типичные ошибки при построении Transformer

1. **Неверная ось softmax.** Нормировать нужно по ключам для каждого запроса.
2. **Несовместимые формы тензоров.** После объединения голов нужен правильный порядок осей и contiguous-view или reshape.
3. **Отсутствие масштаба.** Без $1/\sqrt{d_k}$ поведение softmax при больших размерностях может ухудшиться.
4. **Неверная causal mask.** Утечка будущих токенов даёт искусственно высокое качество.
5. **Неправильное значение boolean-mask.** Семантика True различается между API.
6. **Игнорирование padding.** Модель агрегирует бессодержательные токены, а loss может учитывать пустые позиции.
7. **Перепутанные вход и target.** В декодере нужен корректный сдвиг.
8. **Нет positional information.** Без неё Self-Attention не различает порядок на уровне исходной операции.
9. **Сравнение моделей на разных датасетах.** Тест не должен служить инструментом подбора гиперпараметров.
10. **Неправильная интерпретация attention weights.** Большой вес сам по себе не доказывает причинного влияния признака.
11. **Неучтённый memory footprint.** Квадратичные по длине промежуточные матрицы способны исчерпать память ускорителя.
12. **Отсутствие экспериментов с baseline.** Более сложная модель не обязана превосходить LSTM или простой классификатор.

21. Проектирование корректных экспериментов

21.1 Какие модели сравнивать

В задаче классификации последовательностей разумно сравнить BoW/логистическую регрессию, LSTM/GRU и Transformer при одинаковых разделениях train/validation/test. Требуются одинаковые принципы метрик и настройки. При сильном дисбалансе Accuracy недостаточно: полезны Precision, Recall, F1 и PR-AUC; для генерации применяются иные метрики.

21.2 Ablation Study

Последовательно удаляют или изменяют компоненты: позиционное кодирование; количество голов; число слоёв; размер FFN; dropout; длину контекста. Чтобы сделать вывод причинно обоснованным, нужно сохранять прочие условия как можно ближе одинаковыми и учитывать разброс между случайными инициализациями.

21.3 Документация результатов

Отчёт должен содержать версии библиотек, размер данных, способ разделения, seed, параметры модели, число обучаемых параметров, устройство обучения, ограничения вычислительных ресурсов, значения метрик и таблицу экспериментов. Любая таблица с цифрами должна основываться на реально выполненном запуске, а не на ожидаемом результате.

22. Практические расчётные задачи с решениями

22.1 Задача 1: формы матриц

Условие. $n_q = 5$, $n_k = 7$, $d_k = 16$, $d_v = 24$. Найти размерности Q, K, V, S, A, O.

Решение. $Q : (5, 16)$, $K : (7, 16)$, $V : (7, 24)$, $S = QK^\top/4 : (5, 7)$, $A : (5, 7)$, $O = AV : (5, 24)$. Деление на $\sqrt{16} = 4$ не изменяет форму.

22.2 Задача 2: размерность головы

Условие. $d_{\text{model}} = 384$, $H = 6$, $n = 20$, $B = 8$. Найти формы Q и A после разделения на головы.

Решение. $d_k = 384/6 = 64$, $Q : (8, 6, 20, 64)$, а $A : (8, 6, 20, 20)$. После объединения выход имеет форму $(8, 20, 384)$.

22.3 Задача 3: softmax и контекст

Условие. Оценки $s = (0, \ln 2)$, значения $v_1 = (1, 0)$ и $v_2 = (0, 3)$. Найти выход.

Решение. $e^0 = 1$, $e^{\ln 2} = 2$, поэтому $\alpha = (1/3, 2/3)$, $c = (1/3)(1, 0) + (2/3)(0, 3) = (1/3, 2)$.

22.4 Задача 4: причинная маска

Условие. Запрос на позиции $i = 3$ в последовательности длины 5. Какие позиции доступны в causal self-attention при индексации с единицы?

Решение. Доступны позиции $1, 2, 3$. Позиции $4, 5$ запрещены маской. Если дополнительно токен 2 является PAD, его нужно исключить padding mask.

22.5 Задача 5: число параметров

Условие. Найти число параметров стандартного MHA при $d = 256$, включая bias в четырёх линейных проекциях.

Решение. $P = 4d^2 + 4d = 4 \cdot 65536 + 4 \cdot 256 = 263\,168$. При фиксированной общей ширине проекций это число не меняется только из-за выбора числа голов.

22.6 Задача 6: стоимость памяти

Условие. $B = 2$, $H = 4$, $n = 512$. Сколько чисел в полном тензоре внимания? Сколько MiB занимает один такой тензор float32?

Решение. $2 \cdot 4 \cdot 512^2 = 2\,097\,152$ чисел. Четыре байта на число: $8\,388\,608$ байт, то есть ровно 8 MiB. Накладные расходы, градиенты и копии здесь не учтены.

23. Вопросы для контроля

1. Что называют последовательными данными? Приведите три предметных примера.
2. Почему рекуррентное вычисление затрудняет параллелизацию по времени?
3. Какой информационный недостаток раннего Encoder–Decoder помогает смягчить attention?
4. Определите запрос, ключ и значение математически.
5. Что означает термин score function?
6. Как вычисляется контекстный вектор?
7. Почему веса softmax неотрицательны и суммируются в единицу?
8. Докажите инвариантность softmax к сдвигу всех логитов.
9. Выведите якобиан softmax.
10. Запишите формулу Scaled Dot-Product Attention.
11. Проверьте размерности $QK^\top$ и AV.
12. Каким образом масштаб $\sqrt{d_k}$ связан с дисперсией скалярного произведения?
13. Чем аддитивное внимание отличается от dot-product?
14. Почему суммарная размерность голов часто равна d_{model}?
15. Как объединяются результаты нескольких голов?
16. Чем Self-Attention отличается от Cross-Attention?
17. Для чего нужна padding mask?
18. Для чего нужна causal mask?
19. Что произойдёт при полностью замаскированной строке logits?
20. Почему проверка семантики маски особенно важна при смене API?
21. Докажите эквивариантность attention к перестановке без позиционных признаков.
22. Почему синусоидальное позиционное кодирование использует разные частоты?
23. Сравните обучаемые и синусоидальные позиции.
24. Какие подблоки входят в Transformer Encoder?
25. Что представляет собой позиционно-независимая FFN?
26. Выведите формулу числа параметров FFN.
27. Для чего применяются residual connections?
28. Чем LayerNorm отличается от BatchNorm?
29. Чем Pre-LN отличается от Post-LN?
30. Какие три подблока входят в классический decoder-layer?
31. Что такое teacher forcing и сдвиг целевой последовательности?

32. Как факторизуется авторегрессионная вероятность последовательности?
33. Назовите типичные задачи для encoder-only и decoder-only архитектур.
34. Как растёт размер матрицы внимания с увеличением длины последовательности?
35. Почему FlashAttention не устраняет все ограничения длинного контекста?
36. Как вычисляется градиент по V для операции $O = AV$?
37. Почему attention weights не эквивалентны причинному объяснению?
38. Что такое информационная утечка при подготовке последовательных данных?
39. Какие baseline-модели следует рассмотреть перед Transformer?
40. Как построить корректное ablation study?

24. Задания для самостоятельной работы

1. **Ручной расчёт.** Для трёх векторов размерности 2 вычислите полную матрицу Scaled Dot-Product Attention и проверьте сумму строк.
2. **Проверка устойчивости.** Реализуйте softmax без вычитания максимума и с ним; сравните результаты для логитов $(900, 901, 899)$.
3. **Маскирование.** Напишите тест, который подтверждает нулевые веса будущих токенов при causal mask.
4. **Размерности.** Разработайте таблицу форм тензоров при $B = 8$, $n = 50$, $d = 256$, $H = 8$.
5. **Параметры.** Подсчитайте число параметров одного encoder-layer для $d = 256$, $d_{\text{ff}} = 1024$ с bias и двумя LayerNorm.
6. **Self-Attention.** Создайте NumPy-реализацию для Q, K, V разной длины.
7. **Cross-Attention.** Используйте два независимых массива длинами 5 и 8; объясните форму выхода.
8. **Позиции.** Сравните модель без positional encoding и с ним на задаче, где метка зависит от порядка токенов.
9. **PyTorch.** Реализуйте один Pre-LN Encoder-layer и проверьте формы входа и выхода.
10. **Маски padding.** Создайте пакет последовательностей разной длины, верно исключите PAD из агрегирования и из функции потерь.
11. **Градиенты.** Проверьте по конечным разностям один параметр собственной реализации внимания.
12. **Сравнение.** Сравните LSTM и Transformer на одинаковом искусствен-

ном датасете, приведите хотя бы две случайные инициализации.

13. **Ablation.** Исследуйте влияние числа голов $H = 1, 2, 4, 8$ при фиксированной ширине d.
14. **Память.** Оцените размер полного attention tensor при длинах $128, 256, 512, 1024$ и постройте график.
15. **Мини-проект.** Создайте классификатор последовательностей с пользовательской загрузкой данных и подготовьте отчёт о трёх экспериментах.

25. Итоги и связь со следующей лекцией

Механизм внимания вычисляет адаптивную взвешенную агрегацию значений и может связывать произвольные разрешённые позиции в пределах контекста. Self-Attention формирует контекстные признаки из одной последовательности; Cross-Attention переносит информацию между двумя наборами представлений. Multi-Head Attention, позиционные признаки, FFN, residual connections и нормализация образуют базовые блоки Transformer. При работе с этой архитектурой критичны правильные размерности, маскирование, предотвращение утечки данных и оценка вычислительных затрат.

Следующая тема курса — *генеративные состязательные сети (GAN)* либо, в зависимости от календарно-тематического плана, дальнейшее изучение генеративных архитектур и их прикладного применения.

26. Литература

Список литературы

[1] Траск Э. *Грокаем глубокое обучение* (Grokking Deep Learning). Manning, 2019.

[2] Рашид Т. *Создаём нейронную сеть* (Make Your Own Neural Network). 2016.

[3] Géron A. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. 3rd ed. O'Reilly, 2022.

[4] Chollet F. *Deep Learning with Python*. 2nd ed. Manning, 2021.

[5] Stevens E., Antiga L., Viehmann T. *Deep Learning with PyTorch*. Manning, 2020.

[6] Goodfellow I., Bengio Y., Courville A. *Deep Learning*. MIT Press, 2016. https://www.deeplearningbook.org/

[7] Nielsen M. A. *Neural Networks and Deep Learning*. Determination Press, 2015. http://neuralnetworksanddeeplearning.com/

[8] Bahdanau D., Cho K., Bengio Y. Neural Machine Translation by Jointly Learning to Align and Translate. ICLR, 2015. https://arxiv.org/abs/1409.0473

[9] Vaswani A. et al. Attention Is All You Need. NeurIPS, 2017. https://arxiv.org/abs/1706.03762

[10] Devlin J. et al. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. NAACL, 2019. https://arxiv.org/abs/1810.04805

[11] Raffel C. et al. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. JMLR 21(140), 2020. https://jmlr.org/papers/v21/20-074.html

[12] Su J. et al. RoFormer: Enhanced Transformer with Rotary Position Embedding. 2021. https://arxiv.org/abs/2104.09864

[13] Dao T. et al. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. NeurIPS, 2022. https://arxiv.org/abs/2205.14135

[14] PyTorch Documentation. MultiheadAttention, TransformerEncoder and TransformerEncoderLayer. https://docs.pytorch.org/docs/stable/nn.html