Часть I. Теория, история и внутреннее устройство нейросети трансформер

1. Что такое языковая модель и зачем нам GPT

Языковая модель — это программа, которая умеет вычислять вероятность следующего слова при заданном контексте.
Например:
`"кошка села на ..."` — модель говорит, что `"стул"` более вероятно, чем `"метеорит"`.

GPT — это **авторегрессионная** языковая модель: она порождает текст по одному токену, каждый раз подставляя только что сгенерированное слово обратно в качестве входных данных.

---

2. Краткая история

2.1. Рекуррентные сети (RNN, LSTM, GRU)

Сначала тексты обрабатывались рекуррентными сетями (RNN).
Сеть читала последовательность слово за словом, обновляя «скрытое состояние», которое хранило память о предыдущих словах.

Проблемы:

- Очень медленно (нельзя обрабатывать слова параллельно, потому что каждое состояние зависит от предыдущего).
- Трудно запоминать связи между дальними словами (проблема долгой зависимости).

2.2. Seq2Seq и Attention

Для задач перевода придумали конструкцию **кодер-декодер**:

- Кодер (обычно RNN) превращает всё исходное предложение в один вектор.
- Декодер (тоже RNN) по этому вектору порождает перевод.

Затем к этому добавили **механизм внимания (Bahdanau, 2014)** — декодер при каждом шаге «подсматривал» сразу на все скрытые состояния кодера, а не только на конечный вектор. Это резко улучшило качество перевода.

2.3. Архитектура трансформеа (Vaswani et al., 2017)

Революционная статья “**Attention Is All You Need**” предложила вообще отказаться от рекуррентности и использовать **только attention**.
Такой подход:

- Обрабатывает все токены параллельно.
- Легко захватывает дальние связи.
- Хорошо масштабируется (можно обучать на огромных датасетах).

Исходный Transformer состоял из двух половин: **кодер** (превращает входной текст в представление) и **декодер** (генерирует перевод, пользуясь этим представлением).

2.4. GPT — «generative pre-trained transformer»

Создатели GPT (Radford et al., 2018) взяли **только декодерную часть** Transformer’a и обучили её на предсказание следующего слова в огромном корпусе текстов без разметки.
Почему только декодер?
Потому что для генерации текста нам не нужно внешнее представление от кодера — модель учится предсказывать следующее слово, глядя только на предыдущие. Так появился **GPT-1**, затем **GPT-2**, **GPT-3** и современные версии. Архитектурно они почти не изменились: выросли размеры, улучшились данные, но основа — всё тот же авторегрессионный Transformer-декодер.

---

3. Как текст попадает на вход нейросети

Для машины текст — это просто последовательность чисел. Весь процесс превращения предложения в понятный сети вид состоит из трёх шагов.

3.1. Токенизация

Текст разбивается на **токены** — минимальные смысловые кусочки. Это могут быть:

- Целые слова (например, `"где"`),
- Части слов (например, `"игр" + "ать"`),
- Знаки препинания,
- Специальные символы.

Для каждого токена в словаре модели есть фиксированный **целочисленный индекс**.

Пример:

```
Текст:   "кот сидит"
Токены:  ["кот", "сидит"]
Индексы: [1205, 3807]
```

```
Индексы: [1205, 3807]
```

(Реальные индексы зависят от словаря; обычно нейросеть добавляет токен «конец строки» или специальные маркеры.)

3.2. Эмбеддинги (векторы слов)

Каждый индекс превращается в вектор чисел длиной, например, 512 (параметр `d_model`). Для этого используется обучаемая таблица эмбеддингов — большая матрица `(размер_словаря × d_model)`.
Индекс `1205` даёт одну строчку этой матрицы — вектор из 512 чисел, который и будет подан на вход сети.

3.3. Позиционные эмбеддинги

Сеть не знает, в каком порядке идут слова, потому что все токены обрабатываются параллельно. Поэтому к векторам слов **добавляют** ещё одни векторы, кодирующие позицию (первое, второе, ... слово).
Обычно это также обучаемая таблица размером `(максимальная_длина × d_model)`.
Затем векторы слов и позиций складываются поэлементно.

Итак, после трёх шагов мы имеем тензор формы `(batch, длина_последовательности, d_model)`, и он идёт в стопку декодер-блоков.

---

4. Механизм внимания или ядро GPT

Внимание помогает каждому слову «взглянуть» на все актуальные предыдущие слова и решить, какие из них сейчас важны.

4.1. Идея запрос-ключ-значение

У нас есть последовательность из T токенов, каждый представлен вектором размера `d_model`.
Для **каждой** позиции мы вычисляем три вектора (проекции):

- **Запрос (query, Q)** — «что я ищу?».
- **Ключ (key, K)** — «что я могу предложить?».
- **Значение (value, V)** — «моя содержательная информация».

Все эти векторы получаются умножением исходного вектора на три обучаемые матрицы.

4.2. Применение

Внимание одного токена `i` к токену `j` вычисляется так:

1. Скалярное произведение `Q_i · K_j` = насколько запрос «подходит» к ключу.
2. Деление на `√d_k` (где `d_k` = `d_model / n_heads`), чтобы градиенты не взрывались.
3. Применяется softmax по строке — получаем веса (вероятности), с которыми `i` смотрит на все `j`.
4. Выход для токена `i` = сумма `V_j`, взвешенных полученным вниманием.

Матрично для всех токенов сразу:

$$\text{Attention}(Q, K, V) = \text{softmax}\left((Q \cdot K^T) / \sqrt{d_k} \right) \cdot V$$

4.3. Каузальная (причинная) маска

GPT должен предсказывать следующее слово, глядя только на предыдущие, и не подглядывать в будущее. Поэтому на этапе self-attention мы запрещаем токену `i` видеть токены `j > i`.
Делается это добавлением **минус бесконечности** (`-∞`) в матрицу коэффициентов для запрещённых позиций до softmax. Тогда после softmax эти веса становятся ≈0, и токен «не видит» будущие слова.

Маска в простейшем виде — верхнетреугольная матрица из нулей (запрещено) и единиц (разрешено), которая накладывается на `attn_scores`.

4.4. Многоголовое внимание

Чтобы модель могла улавливать разные виды связей (синтаксические, смысловые, позиционные), делается несколько «голов» внимания параллельно.
Каждая голова — независимые линейные проекции в своё подпространство размером `d_k`. Результаты всех голов склеиваются обратно и проходят через ещё один линейный слой.

Таким образом, вектор каждого токена после блока внимания содержит информацию обо всех разрешённых контекстных связях.

---

5. Устройство одного блока декодера и как слои идут по очереди

GPT целиком состоит из **N одинаковых блоков**, поставленных друг за другом. Каждый блок содержит два подслоя:

1. **Masked Multi-Head Self-Attention** (с остаточной связью и LayerNorm).
2. **Feed Forward Network** (полносвязная сеть с нелинейностью, также с остаточной связью

и LayerNorm).

Остаточная связь (`x = x + sublayer(x)`) помогает градиентам не затухать, а LayerNorm стабилизирует обучение.

Поток данных через один блок:

```
Вход: тензор x формы (B, T, d_model)

┌─────────────────────┐
│ LayerNorm           │
├─────────────────────┤
│ Masked Multi-Head   │
│ Self-Attention      │──── + (x)  ──► x1
└─────────────────────┘

┌─────────────────────┐
│ LayerNorm           │
├─────────────────────┤
│ Feed-Forward        │──── + (x1) ──► Выход
└─────────────────────┘
```

И так **N раз подряд**: выход блока 1 — вход блока 2, и так далее.

После последнего блока снова применяется LayerNorm (финальная нормализация), а затем линейный слой `(d_model → vocab_size)`, который превращает финальные векторы в вероятности слов.

---

6. Иллюстрация прохождения данных

```
Текст: "кот сидит на"
    ↓ Tokenizer
[1205, 3807, 1681]
    ↓ Embedding
   (1, 3, 512)          ← сложили эмбеддинги токенов + позиционные
    ↓
┌──────────────────────┐
│   Decoder Block 1    │
│   (Attention + FFN)  │
└──────────┬───────────┘
           ↓
┌──────────────────────┐
│   Decoder Block 2    │
│   ...                │
└──────────┬───────────┘
           ↓
          ...
           ↓
```

```
┌──────────────────────┐
│   Decoder Block N    │
└──────────┬───────────┘
           ↓
    LayerNorm (финальный)
           ↓
    Linear(512 → 50000)    (vocab_size)
           ↓
    softmax → вероятности
           ↓
   предсказание: "стуле" (индекс 9842)
```

Теперь, вооружённые теорией, перейдём к практике.

Часть II. Практическая реализация на PyTorch

Мы напишем учебную модель MiniGPT — очень похожую на настоящие, но с минимальными зависимостями. Для примеров будем считать:

- `vocab_size = 10000` (словарь)
- `d_model = 256`
- `n_heads = 8`
- `d_ff = 1024` (внутренняя размерность FFN)
- `n_layers = 6`
- `max_len = 512`

```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

1. Подготовка эмбеддингов и маски

```
class Embeddings(nn.Module):
    def __init__(self, vocab_size, d_model, max_len):
        super().__init__()
        self.tok_emb = nn.Embedding(vocab_size, d_model)   # (V, d_model)
        self.pos_emb = nn.Embedding(max_len, d_model)      # (max_len, d_model)

    def forward(self, x):  # x: (batch, seq_len)
        seq_len = x.size(1)
        positions = torch.arange(0, seq_len, device=x.device).unsqueeze(0)  # (1, seq_len)
```

```
        tok = self.tok_emb(x)          # (B, T, d_model)
        pos = self.pos_emb(positions)  # (1, T, d_model)
        return tok + pos

def generate_causal_mask(seq_len):
    """ Возвращает маску (1, 1, seq_len, seq_len), где 1 – разрешено, 0 – запрещено """
    mask = torch.tril(torch.ones(seq_len, seq_len)).view(1, 1, seq_len, seq_len)
    return mask
```

2. Многоголовое причинное внимание

```
class CausalMultiHeadAttention(nn.Module):
    def __init__(self, d_model, n_heads):
        super().__init__()
        assert d_model % n_heads == 0
        self.d_model = d_model
        self.n_heads = n_heads
        self.d_k = d_model // n_heads

        # Линейные слои для Q, K, V и выходной проекции
        self.q_proj = nn.Linear(d_model, d_model)
        self.k_proj = nn.Linear(d_model, d_model)
        self.v_proj = nn.Linear(d_model, d_model)
        self.out_proj = nn.Linear(d_model, d_model)

    def forward(self, x, mask=None):
        B, T, C = x.shape  # C = d_model

        # Проекции и разделение на головы
        Q = self.q_proj(x).view(B, T, self.n_heads, self.d_k).transpose(1, 2)  # (B, n_hea
        K = self.k_proj(x).view(B, T, self.n_heads, self.d_k).transpose(1, 2)
        V = self.v_proj(x).view(B, T, self.n_heads, self.d_k).transpose(1, 2)

        # 1. Scaled dot-product
        attn_scores = (Q @ K.transpose(-2, -1)) * (self.d_k ** -0.5)  # (B, n_heads, T, T)

        # 2. Каузальная маска: запрещаем смотреть в будущее
        if mask is not None:
            attn_scores = attn_scores.masked_fill(mask == 0, float('-inf'))

        # 3. Softmax и взвешенная сумма
        attn_weights = F.softmax(attn_scores, dim=-1)  # (B, n_heads, T, T)
        attn_output = attn_weights @ V                  # (B, n_heads, T, d_k)

        # 4. Объединяем головы и пропускаем через линейный слой
        attn_output = attn_output.transpose(1, 2).contiguous().view(B, T, C)
        return self.out_proj(attn_output)
```

3. Feed-Forward сеть (FFN)

```
class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff):
        super().__init__()
        self.linear1 = nn.Linear(d_model, d_ff)
        self.linear2 = nn.Linear(d_ff, d_model)

    def forward(self, x):
        return self.linear2(F.gelu(self.linear1(x)))
```

4. Один блок декодера

```
class DecoderBlock(nn.Module):
    def __init__(self, d_model, n_heads, d_ff):
        super().__init__()
        self.ln1 = nn.LayerNorm(d_model)
        self.attn = CausalMultiHeadAttention(d_model, n_heads)
        self.ln2 = nn.LayerNorm(d_model)
        self.ff = FeedForward(d_model, d_ff)

    def forward(self, x, mask=None):
        # Attention + Add & Norm
        x = x + self.attn(self.ln1(x), mask)
        # FFN + Add & Norm
        x = x + self.ff(self.ln2(x))
        return x
```

5. Собираем MiniGPT

```
class MiniGPT(nn.Module):
    def __init__(self, vocab_size, d_model, n_heads, d_ff, n_layers, max_len):
        super().__init__()
        self.embed = Embeddings(vocab_size, d_model, max_len)
        self.layers = nn.ModuleList([
            DecoderBlock(d_model, n_heads, d_ff) for _ in range(n_layers)
        ])
        self.ln_final = nn.LayerNorm(d_model)
        self.head = nn.Linear(d_model, vocab_size)

    def forward(self, x):
        # Создаём маску для текущей длины последовательности
        mask = generate_causal_mask(x.size(1)).to(x.device)
        # Эмбеддинги
        x = self.embed(x)                    # (B, T, d_model)
        # Передаём через все N блоков
        for layer in self.layers:
```

```
            x = layer(x, mask)
        # Финальная нормализация и проекция на словарь
        x = self.ln_final(x)
        logits = self.head(x)                # (B, T, vocab_size)
        return logits
```

6. Пример использования (инференс, генерация)

```
def generate_text(model, prompt_tokens, max_new_tokens, max_len, device):
    model.eval()
    tokens = torch.tensor([prompt_tokens], device=device)  # (1, T)

    for _ in range(max_new_tokens):
        # Обрезаем, если больше max_len
        inp = tokens[:, -max_len:]                         # (1, current_len)
        with torch.no_grad():
            logits = model(inp)                            # (1, current_len, vocab_size)
            # Берём предсказание только для последнего токена
            next_logits = logits[:, -1, :]                 # (1, vocab_size)
            probs = F.softmax(next_logits, dim=-1)
            next_token = torch.multinomial(probs, num_samples=1)  # сэмплируем

        tokens = torch.cat([tokens, next_token], dim=1)    # добавляем

    return tokens[0].tolist()
```

Пример вызова с условным словарём (индексы):

```
# Предположим, у нас есть токенизатор:
prompt_indices = [1205, 3807, 1681]  # "кот сидит на"
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = MiniGPT(vocab_size=10000, d_model=256, n_heads=8, d_ff=1024, n_layers=6, max_len=5
model.eval()

generated = generate_text(model, prompt_indices, max_new_tokens=10, max_len=512, device=de
# Полученный список индексов можно декодировать обратно в текст (не реализовано здесь)
print(generated)
```

---

7. Краткое резюме того, как слои идут по очереди

1. Входные индексы → эмбеддинги токенов + позиционные эмбеддинги.
2. **Первый блок декодера:**
   LayerNorm → Causal Multi-Head Attention → остаточное сложение → LayerNorm → FFN →

остаточное сложение.
3. Выход первого блока подаётся на **второй блок** (той же структуры).
4. И так N раз.
5. Финальный LayerNorm.
6. Линейный слой → `vocab_size` — предсказание вероятностей для каждого токена последовательности.

Именно так внутри устроен GPT.