Обучение модели на успешных логах

Подготовка логов для Drain3 процесса.

Получение логов Azure DevOps.

Процесс получения логов начинается с того, что скрипт `01-get-pipelines.py` собирает полные сведения о всех доступных пайплайнах CI/CD в выбранной организации.

Здесь и далее для подключения к Azure DevOps нужно указать набор параметров подключения:

```
export TFS_PAT_TOKEN='111222333444555666777888999000'
export TFS_ACCOUNT='Username'
export TFS_ORGANIZATION='Organization'
export TFS_PROJECT='Project'
export TFS_API_URL='https://tfs.org.dom/tfs'
```

По месту расположения пайплайны в дереве исходных текстов можно выбрать те, что относятся к проектам, CI/CD которых будут далее обрабатываться. Такой выбор производится элементарным парсингом по строкам с именами проектов. В результате получаем списко номеров `pipeline_id`, которые далее передаются в скрипт `03-get-builds.sh` чтобы собрать все возможные запуски.

Обучно запусков очень много, как правило, то берем, например, первые 50, и поскольку для обучения нужны логи успешных выполнений, следовательно отбираем билды с признаком `successful`, и так получаем список `builds_id` для ранее найденных пайплайнов.

Затем скрипт `04-get-logs.py` для успешных исполнений собирает логи, причем, сначала просто логи как множество объектов TFS, а уже потом все их скачивает в единый файл. Это происходит потому что логи разбиты на секции, каждая из которых отвечает отдельному шагу исполнения или смены состояния процесса обработчика пайплайны, так называемого агента.

Таким путем для обучения модели были подготовлены 2947 файлов, содержащих логи успешного прогона 114 пайплайнов общим объемом 4.7G. В сырых логах исходно было практически 29 миллионов строк:

```
❯ pwd
dataset/original
❯ ls -1 | wc -l
2947
❯ ls -1 | awk -F- '{print $1}' | sort -u | wc -l
114
❯ du -sh
4.7G    .
❯ cat * | wc -l
28910262
```

Каждый файл в полученных данных размечен и именован по шаблону `pipeline_id-build_id-log_content.txt`.

Очистка логов перед Drain3.

Логи, собранные выше содержат служебные разделители секций и прочие вспомогательные данные. зашумляющие актуальную информацию. И чтобы далее их подготовить для обработки они проходят две стадии очистки.

Очистка от служебных данных Azure DevOps

Очистка файлов от служебных строк Azure DevOps производится по правилам, описанным в ключе dataset.clean_patterns и это позволяет не меняя число файлов уменьшить общий объем более чем на 1G и более 1 миллиона строк:

```
❯ pwd
dataset/cleaned
❯ ls -1 | wc -l
2947
❯ du -sh
3.4G    .
❯ cat * | wc -l
27611872
```

Всю работу исполняет скрипт `07-clean.py` и в результате формируются файлы, именованные как `pipeline_id-build_id-log_content.raw`.

Эти файлы фактически содержат всю полностью актуальную информацию по реальном исполнении заданий CI/CD с полным сохранением контекста, за исклбючением служебных логов и тамстампов TFS.

Замечу, от исходных логов до данного представления было произмедено необратимое преобразование, но далее все данные можно будет восстанавливать до настоящего состояния строк логов.

Подготовка логов к Drain3

На следующем этапе логи очищаются от данных, имеющих отношение к локальным данным, к окружению и описанию инфраструктуры. Здесь используются ключи dataset.filter_patterns , которые не только удаляют, но группируют данные и используют теги заменители характерных данных. Такая обработка не меняет число строк, но уменьшает объем еще на 1G.

```
❯ pwd
dataset/prepared
❯ ls -1 | wc -l
2947
❯ du -sh
2.4G    .
```

```
❯ cat * | wc -l
27611872
```

Зачем нужно использовать свой набор темплейтов подстановки, если Drain3 и так это делает далее.

Дело в том, что Drain3 использует собственный алгоритм темлейтизации, помечаяя все что не попадает в набор его шаблонов как `<*>`. Это может замаскировать важную для нас информацию. Очевидно, что Drain3 универсальные обработчик, поэтому он не понимает специфику TFS логов, и уж тем более не рассчитан для парсинга кастомных корпоративных процедур.

Для того чтобы не дать Drain3 все замаскировать астерисками, специальный скрипт `08-prepare.py` прячет все важно за набором шаблонов. Особенно нужно отметить, что строки не удаляются, а все шаблоны составлены так, что можно с высокой вероятностью вернуть строки к состоянию очищенных логов. Это работает потому, что что точно также как происходит поиск совпадения для замены на заглушку в процессе подготовки, можно искать и строки соответствующие заглушкам в очищенных логах, тем более что строки не удаляются, а файлы и там и там обрабатываются последовательно.

В результате получается набор файлов с именами `pipeline_id-build_id-log_content.lst`, которые и являются входными данными для Drain3.

Drain3 обработка.

Коротко: **Drain3** — это онлайн-парсер логов, который автоматически выделяет **шаблоны** (templates) строк и присваивает каждой строке **id события** (кластер).

Что делает Drain3

Он идёт слева направо по «префиксному дереву» (DAG), сравнивает токены новой строки с уже найденными кластерами по порогу похожести, и либо относит строку к существующему шаблону, либо создаёт новый. Переменные части заменяются на подстановку `*` (или `<*>`), а стабильные токены формируют «скелет» события.

Как это работает по шагам

1. **Предобработка**: нормализация/маскирование (таймстемпы, UUID/sha/IP, числа, пути и т.п. по вашим правилам).
2. **Разбор**: токенизация и спуск по дереву Drain (узлы организованы по длине строки и префиксам).
3. **Сопоставление**: вычисление похожести с кандидатными кластерами (порог `sim_threshold`).
4. **Кластеризация**: присвоение `cluster_id`/`template_id` либо создание нового кластера.
5. **Обновление статистики**: счётчики, последние встреченные параметры, примеры строк.
6. **Персистентность**: сериализация внутреннего состояния, чтобы **дополнять** шаблоны на следующих прогонах без «забывания».

Настройка Drain3 определяется в общем конфигурационном файле следующим образом

```
  # Настройки Drain3
  drain3_config:
    # Параметры для TemplateMiner
    depth: 4
    sim_threshold: 0.6 ## похожесть строк, можно увеличить до 0.7-0.8 для
более точных результатов
    max_children: 200
    max_clusters: 100000 # расход памяти примерно 5-10kb на кластер
    # Дополнительные параметры Drain3
    extra_delimiters: []
    param_str: "<*>"
    param_regex: "[\\w\\-\\.:/\\\\]+"   # расширить, если много путей/URL
    store_parameters: false  # не сохранять параметры и примеры сообщений,
так как применяем drain_mapping.jsonl
```

Здесь важное это max_clusters: 100000 . Это фактически то возможное разнообразие паттернов, которое допустимо в исходных логах. Из опыта 20000 исчерпываются крайне быстро! Возможно смущает тот факт, что это по сути емкость словаря для обучения модели, но чтож, к этому надо относится как к набору иероглифического письма.

И это практически единственный параметр, который можно менять "на лету". Изменение всех остальных потребует повторного запуска Drain3 на тех же входных данных.

Особенности процесса

В отличие от предыдущих стадий работы, это по сути самая трудоемкая и она не ускоряется даже за счет GPU. Очень важно подготовить реально большой набор данных для обучения. И если получить все пайплайны и их логи из Azure DevOps дело получаса, а их очистка вообще несколько минут, но Drain3 на указанном объеме занимает не один десяток часов!

Для успешной работы пришлось не просто протоколировать активность, обработывать Ctrl-C и сохранять состояние для возобновления работы после прерывания. Пришлось разделить всю работу на фазы и подготовленные логи из `dataset/prepared` на порции примерно по 500 файлов или менее, если файлы большие, так чтобы в среднем фаза исполнения было примерно от 1 часа до 3, но как оказалось, порой доходило до 8 часов на одной порции. Результат работы помещается в `dataset/drain`.

Первая фаза это построение дерева Drain.

На этой фазе не производится построение преобразованных в последовательность шаблонов логов. Здесь только индекс Drain. В начале построение индекса запускается командой `09-drain_dataset.py --init-drain` . В результате создается файл `drain_state.json`, который совсем не json , а base64 закодированное сериализованное **внутреннее состояние алгоритма Drain3**: дерево/лес узлов, хеш-индексы, счётчики, текущие `cluster_id`, пороги и служебные структуры. И нужно оно чтобы *продолжать парсинг инкрементально* (extend-drain) и сохранять стабильность присвоенных шаблонов/идентификаторов между запусками. Тут и весь секрет: обработали одну порцию логов, можно перейти к следующей и вызвать `09-drain_dataset.py --extend-drain`.

Это очень важный файл. При его повреждении надо будет все начинать сначала. Поэтому скрипт `09-drain_dataset.py` в процессе построения индекса делает снапшоты через каждые 300000 строк, так чтобы можно было возобновить работу с последнего снапшота.

При нармальном завершении фазы построения или расширения индекса, создается второй файл `drain_templates.json`, каталог **всех найденных шаблонов** (кластеров). Каждая запись — агрегированная «маска» строки вида `"##[section]Async Command <*> Update Build Number"` . Фактически это словарь событий с описанием.

Вторая фаза - построение набора данных.

В этой фазе скрипт вызывается в форме `09-drain_dataset.py --encode` для первого запуска и дажее как `09-drain_dataset.py --append` для добавления логов.

Обработку можно смешивать. Например сначала по всем логам собрать словарь событий, а потом также по всем логам построить цепочки. Или и то и то выполнять по каждой пачке логов. Главное, не допускать, чтобы логи, предназначенные для обучения, обрабатывались раньше, чем по ним пройдет сканет событий, так как это наполнит их неизвестными событиями, которые по идее и должны указывать на аномалию в первую очередь!

Основной файл это `drain_sequences.jsonl`, он представляет как раз то, во что превращаются логи. Внутри вот такие элементы:

```
{
  "pipeline_id": "204923",
  "build_id": "26492182",
  "section_name": "Build",
  "events_seq": ["E20","E21","E22","E23","E24"],
  "status": "success"
}
```

Далее `drain_chunks.jsonl` это разметка **чанков** — логических сегментов лога (например, секции `##[section]Start/End`, шаги пайплайна). Внутри:

- `chunk_id`, `pipeline_id`, `build_id`, `section_name`/`stage`,
- границы `start_line`/`end_line`,
- статистика по событиям внутри чанка (часто — список `template_id` в порядке встречи или их частоты). **Зачем нужно:**
- нарезка логов на последовательности для последующей агрегации;
- быстрый доступ к фрагментам без чтения гигантского `mapping`;
- удобная единица «окна» для LogBERT (N событий в одном шаге/секции).

Именно эти данные потом разделяются на train/val/test наборы данных.

Кроме этого создается `drain_mapping.jsonl` - **построчная карта** «оригинальная строка → template_id + параметры». Запись примерно:

```
{
  "pipeline_id": "…",
  "build_id": "…",
  "log_fname": "...",
  "line_no": 12873,
  "raw": "Update build number to 0.0.1.0 for build 5813854",
  "template_id": "E31",
  "params": ["0.0.1.0", "5813854"],
  "ts": "2025-11-14T21:13:07Z"
}
```

Этот файл в дальнейшем используется для восстановления **исходных строк** по событиям и позициям. Технически на этапе обучения это не используется, но так как скрипты подготовки данных и для обучения и для инференса практически одинаковые, то этот файл также создается. Кроме того, его можно использовать для контроля работы Drain3.

Результаты преобразования Drain3.

Как уже сказано выше, этот шаг весьма длительный и не уступает по нагрузке на вычислительную систему первому. Например вот так выглядит протокол процесса преобразования порции логов:

```
2025-11-16 05:00:04,582 — drain_dataset — INFO — [Drain] 4,250,000 строк,
46221 шаблонов (+6723), 497.7с с последнего сохранения
2025-11-16 05:09:01,532 — drain_dataset — INFO — [Drain] 4,300,000 строк,
46221 шаблонов (+6723), 1034.7с с последнего сохранения
Drain update: 100%|██████████████████████████████| 8/8
[10:40:41<00:00, 4805.19s/it]
2025-11-16 05:12:18,213 — drain_dataset — INFO — Финальное состояние
сохранено в drain_state.json
2025-11-16 05:12:18,981 — drain_dataset — INFO — Шаблоны обновлены после
сохранения: 46221 шаблонов
2025-11-16 05:12:18,981 — drain_dataset — INFO — Drain расширение
завершено: обработано 4,320,578 строк, всего шаблонов 46221
✅ Drain расширение завершено (46221 шаблонов, 4,320,578 строк).
✅ Этап завершён успешно.
```

Выше логи добавления 7-й порции, а ниже состояние папки `dataset/drain`:

- `drain_state.json` — **продолжает** знание Drain3 о шаблонах.
- `drain_templates.json` — **словарь шаблонов** (источник `E###`).
- `drain_mapping.jsonl` — **строка↔событие** (для декодирования и контекста).
- `drain_chunks.jsonl` — **нарезка** сырых логов в осмысленные куски.
- `drain_sequences.jsonl` — **последовательности E###** для LogBERT.

Создание датасета в формате pytorch.

Здесь также обработка состоит из двух этапов.

Создание словаря и разделение исходного датасета на сплиты

Эта задача полностью исполняется скриптом `10-build_vocab.py`. Сначала он из файла `drain_templates.json` формирует словарь, размещая внутри `dataset/vocab` файл `event_vocab.json` в котором собраны все события из словаря Drain3 и добавлены служебные токены:

```
❯ head -n 10 event_vocab.json
{
  "[PAD]": 0,
  "[UNK]": 1,
  "E20020": 2,
  "E20019": 3,
  "E27951": 4,
  "E12344": 5,
  "E20021": 6,
  "E12811": 7,
  "E20049": 8,
❯ wc -l event_vocab.json
43066 event_vocab.json
```

Эти события не повторяют полностью все что было собрано ранее в словаре Drain3. Точно также как Drain3 удаляет повторяющиеся строки, так и в данном процессе удаляются редко встречаемые события:

```
2025-11-16 19:29:28,678 - modules.vocab - INFO - [Vocab] Scanning 1 files…
2025-11-16 19:29:35,528 - modules.vocab - INFO - [Vocab] Built 43065
tokens (incl specials). Used templates=43063/46221. Min_support=1. Saved:
dataset/vocab/event_vocab.json
2025-11-16 19:29:35,528 - modules.vocab - WARNING - [Vocab] Filtered 2885
by pattern, 273 by support. See dataset/vocab/event_vocab_stats.json
```

Затем из файла `drain_chunks.jsonl` создаются наборы данных согласно политике описанной в конфигурационном файле:

```
  # Параметры разбиения данных для LogBERT
  train_split: 0.75          # Доля данных для обучения
  val_split: 0.15            # Доля данных для валидации (test = 1 - train
- val)
```

Указанные здесь доли несколько отличаются от принятого разбиения на 8:1:1. Дело в том, что природа исходных данных такова, что среди выборки пайплайнов есть те, что при запрошенном списке из 50 исполнений, фактически возвращают менее полудюжины. Очевидно, нельзя допустить чтобы какие-либо данные из наборов val/test не оказались в наборе train. Тогда они фактически сразу составят множество аномалий. Этого нельзя допускать в процессе обучения на успешных

логах, и по этой причине все подобные данные в обязательном порядке пополняют набор train, и тем самым увеличивают его долю выше заданных коэффициентов, без учета возможности выделения указанных долей в другие наборы. Желаемое соотношение примерно 0.85:0.1:0.5, однако реальное порой складывается иначе. Например так:

```
2025-11-16 19:29:28,532 - __main__ - INFO -   Всего чанков: 110,230
2025-11-16 19:29:28,532 - __main__ - INFO -   Train: 102,309 чанков
(92.8%) | Размер: 187.29 MB
2025-11-16 19:29:28,533 - __main__ - INFO -   Val:   5,891 чанков (5.3%) |
Размер: 8.86 MB
2025-11-16 19:29:28,533 - __main__ - INFO -   Test:  2,030 чанков (1.8%) |
Размер: 2.31 MB
```

Полученные файлы, размещаются в там же, где и весь Drain набор в `dataset/drain`:

- `train.jsonl`
- `val.jsonl`
- `test.jsonl`

Подготовка датасета в формате для pytorch

Это этап исполняется скриптм `11-prepare_windows.py`. Он решает задачу **разбиения последовательностей событий (event sequences) на окна фиксированной длины** и создания обучающих примеров для BERT-подобного обучения (Masked Language Modeling, Next Event Prediction и др.).

Главная цель — преобразовать набор последовательностей `E###` из train/val/test.jsonl в тензоропригодные окна одинаковой длины (`window_size`), с паддингом и метками для mask-learning.

Настройки, задающие режим работы.

Настройки задаются в конфигурационном файле

```
  windows: ## Настройки окна для LogBERT, Для RTX 4080 оптимально
    size: 64 ## Длина одного окна событий. Определяет, сколько токенов
(E###) модель видит одновременно. Это фактическая длина
последовательности, подаваемой в Transformer Encoder.
    d_model: 512 ## Размерность скрытого пространства токенов и attention-
голов. Каждая позиция окна представляется вектором длины d_model.
Определяет число параметров и расход GPU-памяти.
    max_len: 512 ## Максимальная длина, для которой заранее вычисляются
позиционные векторы. Должна быть не меньше size, иначе позиционные
эмбеддинги «обрежут» конец окна. Обычно max_len >= size
    batch_size: 256 ## Сколько окон одновременно обрабатывает модель. При
фиксированных size и d_model, именно batch_size задаёт основную нагрузку
на GPU память.
```

Критерии выбора:

1️⃣ Размер окна (`size`)

- Используется как **длина входной последовательности** `seq_len`.
- При обучении внимание (`Self-Attention`) строит матрицу весов `A` размером `[batch_size, n_heads, size, size]`. → память растёт квадратично с $size^2$.
- Для RTX 4080 (16 GB VRAM) оптимальное окно `size=64` даёт баланс контекста и производительности.
- Если увеличить `size` до 128, VRAM вырастет примерно в 4 раза.

2️⃣ Размерность скрытых представлений (`d_model`)

- Определяет ширину слоёв трансформера и эмбеддингов (`token_emb`, `section_emb`).
- Сложность и память линейно зависят от `d_model`, но внимание — квадратично по `size`.
- При `d_model=512` модель имеет около ~40–50 M параметров (зависит от числа слоёв), что комфортно для RTX 4080 в FP16.

3️⃣ Максимальная длина (`max_len`)

- Должна быть ≥ `size`, чтобы в `PositionalEncoding` были доступны все позиции.
- Иногда берут с запасом (`max_len=512`) даже при окнах по 64 событий, чтобы модель можно было обучать и на других длинах.

4️⃣ Размер батча (`batch_size`)

- Прямо масштабирует использование GPU: память ~ `batch_size × size × d_model`.
- Для RTX 4080 комфортно `batch_size=256` при `size=64`, `d_model=512`, FP16. Это даёт стабильную загрузку без OOM.

Процесс обработки.

На данном этапе происходит следующее:

1. Считываются сформированные чанки из `dataset/drain/train.jsonl`, `val.jsonl`, `test.jsonl`, представляющие собой последовательность событий.

2. Загружается словарь `dataset/vocab/vocab.json` чтобы каждое `E###` заменить на числовой индекс (`token_id`), пригодный для `nn.Embedding`.

3. Каждая последовательность разбивается на скользящие окна `windows`.

    - Размер окна задаётся `config.windows.size: 64`.

    - Для каждой последовательности создаются фрагменты:

    ```
    [E1 ... E64]
    [E2 ... E65]
    [E3 ... E66]
    ...
    ```

    - Все окна паддятся до одинаковой длины (`max_len: 512`).

4. Добавляются метки для задач LogBERT
    - маскирует случайные токены (`mask_prob`, как у BERT),
    - формирует `input_ids`, `labels`, `attention_mask`.
5. Сохраняются подготовленные примеры в отдельные JSONL-файлы, готовые к загрузке через `torch.utils.data.Dataset`.

Протокол обработки:

```
2025-11-16 19:48:34,618 - __main__ - INFO - Vocab path from config:
dataset/vocab/event_vocab.json
2025-11-16 19:48:34,618 - __main__ - INFO - Output dir from config:
dataset/train
2025-11-16 19:48:34,618 - __main__ - INFO - Window size from config: 64
2025-11-16 19:48:34,624 - __main__ - INFO - Loaded vocab with 43065 tokens
2025-11-16 19:48:34,624 - __main__ - INFO - Processing
dataset/drain/test.jsonl -> dataset/train/test.pt
2025-11-16 19:48:35,284 - modules.windowing - INFO - [Prepare] Saved
103669 windows -> dataset/train/test.pt
2025-11-16 19:48:35,296 - __main__ - INFO - Processing
dataset/drain/train.jsonl -> dataset/train/train.pt
2025-11-16 19:49:40,520 - modules.windowing - INFO - [Prepare] Saved
9454020 windows -> dataset/train/train.pt
2025-11-16 19:49:41,557 - __main__ - INFO - Processing
dataset/drain/val.jsonl -> dataset/train/val.pt
2025-11-16 19:49:44,425 - modules.windowing - INFO - [Prepare] Saved
497063 windows -> dataset/train/val.pt
2025-11-16 19:49:44,485 - __main__ - INFO - Done prepare_windows
```

Если проследить работу на примере датасета train, то можно заметить, что исходно было 102,309 чанков, затем получилось 9454020 windows, а в процессе обучения в логе будет уже упоминаться число батчей 36930. Один батч объединяет несколько окон.

По итогу выполнения данного шага создаются файлы датасета, пригодные для загрузки в процессе обучения.

```
❯ pwd
dataset/train
❯ ll
total 6.6G
-rw-rw-r-- 1 alekseybb alekseybb  18M Nov 16 19:48 test_meta.jsonl
-rw-rw-r-- 1 alekseybb alekseybb  53M Nov 16 19:48 test.pt
-rw-rw-r-- 1 alekseybb alekseybb 1.5G Nov 16 19:49 train_meta.jsonl
-rw-rw-r-- 1 alekseybb alekseybb 4.7G Nov 16 19:49 train.pt
-rw-rw-r-- 1 alekseybb alekseybb  78M Nov 16 19:49 val_meta.jsonl
-rw-rw-r-- 1 alekseybb alekseybb 251M Nov 16 19:49 val.pt
```

Здесь важно оговорить одно условие. В наборах train/val/test не должно встречаться ни одного события [UNK]. Недопустимо поизводить какие-либо сокращения или упрощения, в них должно оказаться ровно то, что было создано Drain3 и описано в словаре, соответствующему индексу кластеров.

Обучение модели.

Обучение модели представляет собой циклический процесс. Предстазать заранее его успех и длительность сложно. Поэтому обучение реализовано как многократный вызов скрипта `12-train_one_epoch.py` , который прогоняет train датасет для обучения следующей эпохи, переодически мониторит загрузку оборудования и в конце работы оценивает успешность и дает рекомендации о продолжении или завершении обучения. Модель сохраняется в `dataset/model/checkpoints` по завершении каждой эпохи, и загружается оттуда для обучения следующей:

```
❯ pwd
dataset/model/checkpoints
❯ ll
total 1.5G
-rw-rw-r-- 1 alekseybb alekseybb 246M Nov 16 20:35 model_epoch_1.pt
-rw-rw-r-- 1 alekseybb alekseybb 246M Nov 16 21:22 model_epoch_2.pt
-rw-rw-r-- 1 alekseybb alekseybb 246M Nov 16 22:07 model_epoch_3.pt
-rw-rw-r-- 1 alekseybb alekseybb 246M Nov 16 22:07 model.pt
-rw-rw-r-- 1 alekseybb alekseybb 490M Nov 16 22:07 optimizer.pt
-rw-rw-r-- 1 alekseybb alekseybb 1.4K Nov 16 22:07 scaler.pt
-rw-rw-r-- 1 alekseybb alekseybb  829 Nov 16 22:14 training_state.json
```

Описание модели

Для обучения выбрана модель LogBERT, которая по параметрам является вариантм TinyBERT. Далее её описание в коде:

```
class UnifiedLogBERT(nn.Module):                    # Определяем класс
нейросети, наследуемый от базового torch.nn.Module
    def __init__(                                    # Конструктор модели —
принимает размеры и гиперпараметры
        self,
        vocab_size: int,                             # Размер словаря
событий (кол-во уникальных токенов E###)
        section_buckets: int = 2048,                 # Кол-во возможных
"секции/шагов" пайплайна — для отдельной embedding-таблицы
        d_model: int = 512,                          # Размерность скрытого
представления (векторов токенов)
        n_heads: int = 8,                            # Количество голов в
multi-head attention
        n_layers: int = 6,                           # Глубина трансформера
— сколько слоёв encoder'a
        dim_ff: int = 2048,                          # Размер внутреннего
```

```
слоя в feed-forward блоке
        dropout: float = 0.1,                       # Вероятность дропаута
для регуляризации
        max_len: int = 256,                         # Максимальная длина
входной последовательности (число событий)
    ):
        super().__init__()                          # Инициализируем
базовый класс nn.Module

        # --- Эмбеддинги токенов ---
        self.token_emb = nn.Embedding(              # Таблица эмбеддингов
для событий E1, E2, ..., En
            vocab_size,                             # количество строк =
число событий в словаре
            d_model,                                # размер векторного
представления каждого события
            padding_idx=PAD_ID                      # специальный индекс
для паддинга (градиенты не обновляются)
        )

        # --- Эмбеддинги секций/пайплайнов ---
        self.section_emb = nn.Embedding(            # Дополнительные
эмбеддинги для "bucket'ов" — групп логов (например, стадия сборки)
            section_buckets,                        # количество возможных
секций (в train задаётся по числу уникальных section_name)
            d_model                                 # размерность
совпадает с d_model, чтобы можно было суммировать
        )

        # --- Позиционное кодирование ---
        self.pos = PositionalEncoding(              # Модуль добавляет
позиционные смещения (sin/cos или learnable)
            d_model,                                # размер векторного
пространства
            max_len=max_len                         # длина, на которую
заранее генерируются позиции
        )

        # --- Базовый слой Transformer Encoder ---
        layer = nn.TransformerEncoderLayer(         # Один слой энкодера
(Attention + FeedForward)
            d_model=d_model,                        # размер входных/
выходных векторов
            nhead=n_heads,                          # количество
attention-голов
            dim_feedforward=dim_ff,                 # размер скрытого слоя
внутри FFN
            dropout=dropout,                        # дропаут на attention
и FFN
            activation='gelu',                      # активация в FFN
(обычно GELU для BERT-подобных моделей)
            batch_first=True                        # формат входа:
(batch, seq_len, hidden)
        )
```

```
        # --- Стек энкодеров ---
        self.encoder = nn.TransformerEncoder(       # Собираем несколько
таких слоёв в стек
            layer,                                  # шаблон слоя,
определённый выше
            num_layers=n_layers                     # количество слоёв в
глубину (6 по умолчанию)
        )

        # --- Нормализация выходных признаков ---
        self.norm = nn.LayerNorm(d_model)           # LayerNorm
стабилизирует распределение активаций после энкодера

        # --- Голова предсказания токена (Language Modeling Head) ---
        self.lm_head = nn.Linear(                   # Линейный слой:
скрытое представление → вероятности по словарю событий
            d_model,                                # входная размерность
(из трансформера)
            vocab_size                              # выходная размерность
= число событий (E###)
        )
```

Вычисление loss.

В процессе обучения модели производится рассчет показателя loss, который определяет успех процесса обучения.

1. Инициализируется функция потерь `criterion = nn.CrossEntropyLoss()` . Используется `CrossEntropyLoss` для задачи предсказания следующего события, принимает логиты (сырые оценки) и индексы правильных классов и вычисляет отрицательное логарифмическое правдоподобие (NLL).

2. Производится Forward Pass через модель. Функция `forward` определяется в модели.

3. Расчета CrossEntropyLoss

Расчет loss для одного события считается по формуле:

```
loss_i = -log(softmax(logits[i])[target[i]])
       = -log(exp(logits[i, target[i]]) / Σ_j exp(logits[i, j]))
       = -logits[i, target[i]] + log(Σ_j exp(logits[i, j]))
```

CrossEntropyLoss автоматически применяет softmax и вычисляет NLL

Для всего батча:

```
batch_loss = mean([loss_0, loss_1, ..., loss_B-1])
```

4. Накопление Loss по батчам. Считается взвешенное среднее с учетом размера батча.

Loss уже усреднен по батчу — не нужно делить на batch_size, однако взвешенное суммирование нужно для корректного усреднения при разных размерах батчей.

5. Вычисление финального Loss за эпоху

Формула:

```
epoch_loss = Σ(batch_loss_i * batch_size_i) / Σ(batch_size_i)
```

Визуализация потока данных

```
┌──────────────────────────────────────────────────────────────────┐
│                        РАСЧЕТ LOSS                               │
└──────────────────────────────────────────────────────────────────┘

1. ВХОДНЫЕ ДАННЫЕ
   input_ids: (B, L) ──┐
   target_ids: (B,) ───┼──> .to(device)
   sec_ids: (B,) ──────┘

2. FORWARD PASS
    ┌──────────────────────────────────────────┐
    │ model(input_ids, sec_ids)                │
    │                                          │
    │ token_emb(input_ids) ──> (B, L, D)      │
    │ section_emb(sec_ids) ──> (B, 1, D)      │
    │ x = token_emb + section_emb              │
    │ x = pos_encoding(x)                      │
    │ h = encoder(x) ─────────> (B, L, D)     │
    │ h = norm(h[:, -1, :]) ──> (B, D)        │
    │ logits = lm_head(h) ────> (B, V)        │
    └──────────────────────────────────────────┘
                     │
                     ▼
    ┌──────────────────────────────────────────┐
    │ criterion(logits, target_ids)            │
    │ CrossEntropyLoss                         │
    │                                          │
    │ Для каждого примера i:                   │
    │   loss_i = -log(softmax(logits[i])[target[i]])
    │                                          │
    │ batch_loss = mean([loss_0, ..., loss_B-1])
    └──────────────────────────────────────────┘
                     │
                     ▼
                loss: scalar

3. НАКОПЛЕНИЕ
```

```
    batch_loss = loss.item()  # Скалярное значение
    total_loss += batch_loss * batch_size  # Взвешенное суммирование
    n += batch_size                        # Счетчик примеров

4. ФИНАЛЬНЫЙ LOSS
    epoch_loss = total_loss / n  # Взвешенное среднее
```

Оценка успешности обучения

В процессе обучения и сразу после него считается train_loss и val_loss. Второй на специальном наборе val, который не используется в процессе обучения и может определять реакцию модели на подобные, но незнакомые данные.

Логика принятия решения:

- Если это первая эпоха → продолжить
- Если validation loss улучшился более чем на 0.01 → продолжить
- Если val/train ratio > 1.2 → остановить (переобучение)
- Иначе → остановить (нет дальнейшего улучшения)

Процесс обучения.

Итак процесс обучения строится прогонами по эпохам. Это позволяет наблюдать за процессом, его успехом или возникающими проблемами.

Каждый прогон исполняется скриптом `12-train_one_epoch.py`. Обучение производится на наборе train, оценка на наборе val. Расчетный loss на train стадии и на val стадии должен сблищаться с каждой эпохой. В противном случае стоит задуматься о правильности составления наборов датасета. Как это обсуждалось выше, в наборах val и test не должно быть "незнакомых" событий и цепочек.

В конце каждого прогона оценивается необходимость продолжения обучения. Например так, далее в логе контрольная точка и завершение обучения второй эпохи:

```
2025-11-16 21:21:01,622 - __main__ - INFO - 📈 Статус обучения (Batch
36900/36930, 99.9%)
2025-11-16 21:21:01,622 - __main__ - INFO -   ⏱️  Время: 41.8 мин | ETA:
0.0 мин
2025-11-16 21:21:01,622 - __main__ - INFO -   📊 Loss: текущий=0.2306,
средний=0.2077
2025-11-16 21:21:01,622 - __main__ - INFO -   🚀 Скорость: 14.70
batches/sec
2025-11-16 21:21:01,622 - __main__ - INFO - 📊 Ресурсы системы:
  💾 RAM: 10.1 GB / 62.1 GB (16.3%)
  🖥️  CPU: 0.0% (16 cores)
  🎮 GPU (NVIDIA GeForce RTX 4080 SUPER):
     - Выделено: 1.00 GB
     - Зарезервировано: 3.42 GB
     - Всего: 15.57 GB (22.0%)
2025-11-16 21:21:01,622 - __main__ - INFO -
```

```
==========================================
Epoch 2 [Train]: 100%|██████████████████████████████████████|
36930/36930 [41:52<00:00, 14.70batch/s, loss=0.2156]
Epoch 2 [Val]: 100%|└└└
```

только для продолжения обучения.

- `training_state.json` Хранит *метаданные о процессе обучения*, которые не входят в бинарные состояния, и очевидно, только для обучения и нужен.

Получается, что для инференса нужен только файл `model.pt`.

Индекс Drain3.

Индекс Drain3 нужен для правильно преобразования логов, выбранных для инференса, так как они должны быть подготовлены в точности с тем словарем событий, что и логи, на которых производилась тренировка.

Поэтому важными артефактами, которые далее передаются в стадию инференса будут:

- `drain_state.json` Состояние Drain-индекса (дерево шаблонов).
- `drain_templates.json` Список уникальных шаблонов.
- `event_vocab.json` Словарь `{E###: index}`