ChatGPT

Distributed Training with Blockchain – Initial README

Описание проекта

Этот репозиторий содержит **прототип** системы распределённого обучения нейронных сетей с использованием блокчейн-технологий. Целью является создание открытого рынка, на котором заказчики могут размещать задания по обучению моделей, а участники с видеокартами – выполнять микрозадачи обучения, получая вознаграждение через умный контракт. Система ориентирована на обучение или дообучение нейросетей на нерегламентированных данных, имеет ролевую модель (тренер, валидатор, лидер задания) и предоставляет простое подключение для узлов с разной вычислительной мощностью.

Для управления заданиями и выплатами используется смарт-контракт, написанный на **Solidity** и развёртываемый через **Hardhat**. Off-chain логика, распределяющая задачи и собирающая обновления, реализована на **Python** с использованием библиотеки **web3.py** (рекомендуется стабильная версия v7.14.0 от 16 октября 2025 года [1]). Тренеры и валидаторы также написаны на Python и используют **PyTorch** для выполнения шагов обучения. Для демонстрации подходит небольшой датасет (MNIST/ Fashion-MNIST), поскольку модель и объём данных легко обрабатываются на обычных GPU.

Основные компоненты

1. **Смарт-контракт JobManager** – хранит информацию о заданиях, бюджетах, зарегистрированных тренерах и валидаторах, принимает хэши обновлений и распределяет выплаты. Контракт реализован на Solidity 0.8.x и развёртывается с помощью Hardhat. Функции контракта:
2. `createJob(jobSpec)`: создание задания и фиксация депозита заказчика;
3. `registerTrainer()` / `registerValidator()`: регистрация участников;
4. `submitUpdate(jobId, updateHash)`: запись хэша обновления в блокчейн;
5. `validateUpdate(jobId, trainerAddress, isValid)`: фиксация результата проверки;
6. `payout(jobId, trainerAddress)`: перечисление вознаграждения после подтверждения апдейта.
7. **Оркестратор (Job Leader)** – off-chain сервис на Python (использует web3.py), который:
8. слушает события смарт-контракта (создание задания, регистрация, отправка обновлений);
9. выдаёт тренерам микрозадачи (выбирая подходящий шард данных и количество локальных шагов);
10. собирает `delta`-обновления от тренеров, вызывает валидаторов для проверки и агрегирует корректные обновления;
11. отправляет в контракт хэши обновлений и подтверждения валидаторов, инициирует выплаты.

12. **Клиент-тренер** – скрипт на Python, загружающий модель и датасет, выполняющий K локальных шагов обучения и отправляющий обновление (`delta` весов) оркестратору. Тренер публикует хэш обновления в контракт, а сам `delta` отправляет off-chain. Дополнительно тренер сообщает базовые метрики (loss до и после обучения).

13. **Клиент-валидатор** – скрипт на Python, который перепроверяет задачи (spot-check) или повторяет обучение с теми же параметрами, сравнивает результат с присланным обновлением и передаёт смарт-контракту вердикт. При несоответствии валидатор отклоняет обновление, и вознаграждение не начисляется.

14. **Данные и модели** – для демонстрации используются открытые датасеты (например, MNIST). Весы модели и сами данные не хранятся в блокчейне; в контракт записываются только хэши обновлений и результатов проверки. Заказчик хранит канонический чекпойнт модели и получает итоговые веса в конце задания.

Установка и подготовка

Предварительные требования

- Node.js $\geq$ 16 и npm (для работы с Hardhat);
- Python $\geq$ 3.9;
- Docker (желательно, для запуска изолированных окружений);
- Git для контроля версий;
- Среда разработки для Solidity (Hardhat).

Шаг 1. Клонирование репозитория

```
git clone <URL_репозитория>
cd <имя_репозитория>
```

Шаг 2. Смарт-контракт

1. Перейдите в каталог `contracts/` и установите зависимости:

   ```
   cd contracts
   npm install
   ```

2. Скомпилируйте контракт:

   ```
   npx hardhat compile
   ```

3. Разверните контракт на локальной сети Hardhat (или тестнете Sepolia):

   ```
   npx hardhat run scripts/deploy_contract.js --network localhost
   ```

   Запомните адрес развернутого контракта и добавьте его в конфигурационный файл `.env` или `config.js`.

Шаг 3. Оркестратор

1. Перейдите в каталог `orchestrator/`:

   ```
   cd orchestrator
   pip install -r requirements.txt  # web3.py, fastapi (если нужен REST), т.п.
   ```

2. Укажите адрес смарт-контракта, сеть и приватные ключи для подписания транзакций в файле конфигурации (например, `config.py`). Приватные ключи **не** коммитятся в репозиторий.
3. Запустите оркестратор:

   ```
   python src/orchestrator.py
   ```

   Оркестратор будет слушать события контракта и назначать задачи зарегистрированным тренерам.

Шаг 4. Тренеры и валидаторы

1. В каталоге `trainer/` установите зависимости (PyTorch, web3.py):

   ```
   cd trainer
   pip install -r requirements.txt
   ```

2. Запустите тренеров (можно несколько экземпляров):

   ```
   python trainer.py --registry http://<orchestrator_host>:<port> --job <jobId>
   ```

3. Аналогично установите зависимости для валидаторов в каталоге `validator/` и запустите их:

   ```
   cd validator
   pip install -r requirements.txt
   python validator.py --registry http://<orchestrator_host>:<port>
   ```

4. Каждый тренер будет получать небольшие задачи (количество шагов K и шард данных), выполнять обучение и отправлять обновление. Валидаторы будут подтверждать или отклонять эти обновления.

Шаг 5. Создание задания (для заказчика)

Заказчик создаёт новое задание командой (CLI или отдельный frontend):

```
npx hardhat run scripts/create_job.js --network localhost --jobSpec job.json
```

Параметры задания включают:

- адрес заказчика;
- депозит (средства для выплат участникам);
- число раундов и шагов K;
- модель и датасет (ссылки/идентификаторы);
- базовая и бонусная ставки за задачу.

После создания задания оркестратор получит событие `JobCreated` и начнёт распределять задачи тренерам.

Структура проекта

```
repo-root/
├── contracts/             # Solidity-контракт JobManager и скрипты
развертывания
│   ├── JobManager.sol
│   ├── scripts/
│   │   ├── deploy_contract.js
│   │   └── create_job.js
│   ├── test/               # Тесты для контракта
│   └── README.md
├── orchestrator/
│   ├── src/
│   │   ├── orchestrator.py  # Основная логика распределения задач
│   │   ├── contract_client.py  # Работа с контрактом через web3.py
│   │   └── aggregator.py    # Агрегация delta-обновлений
│   ├── requirements.txt
│   └── README.md
├── trainer/
│   ├── trainer.py
│   ├── model.py
│   ├── requirements.txt
│   └── README.md
├── validator/
│   ├── validator.py
│   ├── requirements.txt
│   └── README.md
├── models/                # Чекпойнты и датасеты (не хранятся в блокчейне)
├── docs/
│   ├── whitepaper_blockchain_distr_ml.md
│   └── API_SPEC.md
├── init_readme.md         # Этот файл с описанием проекта
└── README.md              # Краткий обзор и ссылка на init_readme.md
```

Архитектурные решения и паттерны

- **Событийно-ориентированная архитектура** – оркестратор и валидаторы подписываются на события смарт-контракта (создание задания, отправка обновления, подтверждение). Это уменьшает накладные расходы и позволяет реагировать на изменения в реальном времени.
- **Микросервисы** – каждая роль (контракт, оркестратор, тренеры, валидаторы) реализована как отдельный сервис. Это упрощает масштабирование, тестирование и замену компонентов.
- **Strategy Pattern для агрегации** – в `aggregator.py` можно реализовать несколько стратегий (усреднение, медиана, усечение выбросов) и выбирать их в зависимости от характера модели или количества вредоносных обновлений.
- **Factory Method** – генерация микрозадач (заданий) осуществляется через фабричный метод, который учитывает классификацию GPU-узлов (S/M/L) и выдаёт соответствующий объём данных.
- **Proxy/Adapter для web3** – `contract_client.py` инкапсулирует всю работу с web3.py (создание транзакций, обработка событий), скрывая детали реализации от оркестратора и клиентов.
- **Robust Aggregation** – при агрегации delta-обновлений используются устойчивые методы (например, trimmed mean, coordinate-wise median). Это снижает влияние вредных апдейтов на итоговый чекпойнт.

Следующие шаги и возможные улучшения

После того как базовый прототип заработает, можно рассмотреть:

- **Токенизированные стимулы и владение моделью** – внедрение ERC-20 токена для выплат и ERC-721/1155 для долей модели, чтобы участники могли владеть частью модели и получать долю от будущих лицензий.
- **DAO и голосование** – владельцы токенов смогут голосовать за параметры протокола (размер бонусов, выбор модели, правила валидации).
- **Подключение IPFS для хранения данных** – чтобы полностью уйти от централизации хранилищ.
- **Zero-Knowledge Proofs** – для доказательства корректности вычислений без раскрытия содержимого; подходит для приватных датасетов.
- **Web-интерфейс** для удобной работы заказчиков и участников (React/Vue + web3.js). С помощью фронтенда можно отображать статус заданий, выплаты и статистику.

Ссылки и цитаты

- Подробное руководство по библиотеке web3.py доступно в официальной документации. Актуальная стабильная версия 7.14.0 выпущена 16 октября 2025 года [1].
- Версия web3.py 7.14.0 доступна на PyPI с датой релиза 16 октября 2025 года [2]. Рекомендуется использовать эту версию для совместимости и поддержки новых возможностей.

Этот документ предназначен для начальной ориентации команды. Подробности реализации (форматы `JobSpec`, `TaskSpec`, а также API оркестратора) приведены в `docs/API_SPEC.md` и whitepaper. Обязательно ознакомьтесь с ними перед началом разработки.

1 Release Notes — web3.py 7.14.0 documentation
https://web3py.readthedocs.io/en/stable/release_notes.html

2 web3 · PyPI
https://pypi.org/project/web3/