01/24/2024:

* Went through a rough architecture/system design for the software
* Discussed what a package should carry, what will be on the front end, where to put the ML model, and how to get logging information from the backend and display it on the front end in real time

01/29/2024:

* Learned and used WebSocket to get logging information from the ML script and display it on the front end in real time
* Refactored ML code so that every function is in a designated class
* Refactored manager.py and mnist\_model.py to enable manager to update progress, allowing mnist\_model to focus on running ML tasks
* Setup a websocket in server’s app.py, pass it to manager, and let manager.py emit updates to the client app.js
* Currently debugging why socket.io is emitting the same information multiple times (id 0 emits 1 time, id 1 emits 2 times, id 2 emits 3 times, …) 🡪 Used socketio.once() to solve the problem
* Learned how to close/disconnect a WebSocket to avoid memory leaks/security issues.
* Used ChatGPT to refactor code and make it cleaner.
* Q: Don’t know how to check if the websocket is properly closed.

02/07/2024

* Carrying out issue #1: minimal front to back
* Just a simple front-to-back integration that lets users pick the params and train the model immediately. No Producer yet.
* Currently debugging this error: Access to fetch at 'http://localhost:9000/start\_experiment' from origin 'http://localhost:3000' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled. 🡪 Here’s what I found: The error stems from a security mechanism that browsers implement called the same-origin policy. The same-origin policy fights one of the most common cyber attacks out there: cross-site request forgery. In this maneuver, a malicious website attempts to take advantage of the browser’s cookie storage system. 🡪 Identified that it is mostly a **backend** problem, simply created a key ‘Access-Control-Allow-Origin’ – value ‘http://localhost:3000’ for the backend response object (which is a dict/json)
* Also modified the code a bit to remove proxy in package.json, set the server to use port 9000 instead of 5000.

02/08/2024

* Starts learning about **“Design job scheduling system”**. Some sources that I read:
  + [Job Scheduling Design: Behind the Scenes of a Distributed Job Scheduler (redwood.com)](https://www.redwood.com/article/job-scheduling-design/)
    - Desired job scheduling workflow -> API
    - Job metadata management -> Job ID, timestamp, execution time, dependencies
    - How to implement a task scheduler
    - Defining job execution
  + [Ace the System Design Interview: Job Scheduling System | by Zixuan Zhang | Towards Data Science](https://towardsdatascience.com/ace-the-system-design-interview-job-scheduling-system-b25693817950)
    - Create a new job with its schedule
  + [How we designed Dropbox ATF: an async task framework - Dropbox](https://dropbox.tech/infrastructure/asynchronous-task-scheduling-at-dropbox)
* Realized it’s such a big and complex topic, not something doable in 1 day 😊
* Discussed with pda, and agreed that we should start by designing a **linear system** first
* High level design:
  + Database design:
    - Read operations:
      * Given a userID, retrieve all jobs that belong to it (by client)
      * Find all jobs that are scheduled to run right now (by internal servers)
    - Write operations:
      * A user can create a new job schedule (by client)
      * The workers will add execution histories to the database (by internal servers)
      * The system updates the next execution timestamp of a job after running it (by internal servers)
    - Schema
      * Job table: UserID, JobID, timeCreated (UNIX timestamp)
      * History table: JobID, startTime (UNIX timestamp), runtime (seconds), epochs, lr, size, accuracy