ChatGPT

Modular OOP Framework for Machine Learning Experiments

We propose a clean, object-oriented design where each experiment is a self-contained class and a **Runner** orchestrates them sequentially. A **ConfigManager** reads JSON files into Python dataclasses, providing typed experiment settings. Each **Experiment** class handles its model, data loading, training loops, and metrics logging. A centralized **Logger** (using Python's `logging` module) writes all events to console and a log file [1] [2]. A **StateManager** ensures checkpoints and intermediate state are saved safely, enabling resume after interruptions [3]. Key design goals are dynamic configuration (any number of experiments via JSON), extensibility (new models or techniques by subclassing), and full recoverability on failure.

Figure: A standard ML pipeline runs stages sequentially (data ingestion → preprocessing → training → validation) [4] *. Our framework runs each experiment in such a pipeline, one after another.*

Key Classes and Responsibilities

- **ConfigManager**: Reads JSON config into a typed dataclass. For example, one can define:

```
@dataclass
class ExperimentConfig:
    name: str
    model_type: str
    dataset_path: str
    hyperparams: dict
    # ... other fields
```

The manager loads JSON into this dataclass (`cfg = ExperimentConfig(**json.load(f))` [5]). This enforces schema, default values, and easy access to parameters. New config fields automatically become class attributes, aiding clarity.

- **Experiment (base class)**: An abstract class defining the interface for any experiment. It holds its `config` and implements:

- `run()`: Executes the experiment (data load, training, evaluation).
- `save_results()`: Writes metrics to a `result.json`.
- `load_state()/save_state()`: Uses **StateManager** to persist progress (e.g., last epoch, best metrics).
  Subclasses (e.g. `ImageClassificationExperiment`) override these methods to handle specific model and data. Using an abstract base or simply raising `NotImplementedError` ensures a consistent interface.

- **Runner**: Reads multiple JSON configs (via ConfigManager) and runs them sequentially. Pseudocode:

```
for cfg_file in config_files:
    config = ConfigManager.load(cfg_file)
    exp = ExperimentFactory.create(config)  # returns an Experiment
instance
    try:
        exp.run()
    except Exception as e:
        Logger.error(f"Experiment {config.name} failed: {e}")
        exp.save_state()  # ensure last state is saved
    else:
        exp.save_results()
```

The Runner checks for existing state at start of each experiment; if found and incomplete, it resumes from the last checkpoint. This "foreach" pattern ensures each experiment runs to completion before the next [4].

- **Logger**: Wraps Python's `logging` to direct output to both console and file. We configure a `StreamHandler` (console) and `FileHandler` pointing to `log.txt`. By creating a module-level logger (`logger = logging.getLogger(__name__)` [1]) and using levels (`info()`, `error()`, etc.), every step (start time, parameters, metrics) can be logged. For example:

```
logger.info(f"Starting experiment: {config.name}")
logger.debug(f"Config: {config}")
try:
    result = model.train(...)
except Exception as e:
    logger.error("Training failed", exc_info=True)
    raise
```

A rotating file handler can manage log size, ensuring old logs rotate out. The Logger class may be a singleton or simply a module to centralize formatting.

- **StateManager**: Handles checkpointing and resume. It saves experiment state (e.g. current epoch, random seeds, best model path) to disk (e.g. `state.pkl` or JSON). Following best practices, it writes to a temp file then atomically renames (as in [3]) to avoid corruption. On start, it checks for an existing state file:

```
state = StateManager.load_state(exp.name)
if state:
    exp.restore_from(state)  # set model weights, epoch, etc.
```

After each training epoch or critical step, `StateManager.save_state(exp.name, exp.get_state())` is called. This way, if the program is interrupted (error or manual stop), no progress is lost [3]. Upon successful completion, the state file can be deleted or archived.

Configuration Management

All experiment parameters live in JSON config files (one per experiment). Using **dataclasses** provides clarity and type checking:

```
// exp1.json
{
  "name": "exp1",
  "model_type": "ResNet",
  "dataset_path": "/data/images",
  "hyperparams": {"lr": 0.01, "batch_size": 32}
}
```

The `ConfigManager` simply loads this into a dataclass:

```
config_dict = json.load(open(path))
config = ExperimentConfig(**config_dict)  # e.g. name, model_type, etc.
```

This approach (dataclass + JSON) ensures defaults and validation at load time [5]. If new config formats arise (XML, YAML), one could extend ConfigManager, but JSON is straightforward and avoids extra deps.

The Runner can accept a list of config filenames or a directory. Using a config manager means adding a new experiment is as simple as creating a new JSON and Experiment subclass; the rest of the system "just works."

Execution Pipeline

Experiments execute in a strict sequence (no parallelism). Before running, the Runner can log a **start timestamp**. Each experiment's `run()` might look like: 1. **Load data** (using paths from config). 2. **Initialize model** (based on `model_type` and hyperparameters). 3. **Training loop**: for each epoch, train and evaluate on validation set. 4. **Checkpoint**: After each epoch, use `StateManager.save_state()`. If a new "best" model is found, save a checkpoint artifact. 5. **Logging**: After each epoch or major event, log metrics (loss, accuracy) via the Logger. 6. **Finish**: Save final model and metrics (in `result.json`). Log end time.

This design follows the *foreach* pattern of pipelines [4], treating each experiment as one node. The Runner's loop ensures Stage N starts only after Stage N–1 is done.

Logging and Monitoring

Every important event is logged. Using Python's logging module is recommended [1]. For example, at experiment start: `logger.info("Experiment start...")`. Within training, debug logs might record loss values, and exceptions are logged with `logger.error(..., exc_info=True)` to capture stack traces. A `Logger` class can configure both console and file handlers:

- **Console (`StreamHandler`)** for real-time status.
- **File (`FileHandler` or `RotatingFileHandler`)** writing to `log.txt` [2].

A typical setup in code:

```
logger = logging.getLogger("MLRunner")
logger.setLevel(logging.DEBUG)
console = logging.StreamHandler()
file = logging.FileHandler("log.txt")
formatter = logging.Formatter("%(asctime)s - %(name)s - %(levelname)s - %
(message)s")
console.setFormatter(formatter)
file.setFormatter(formatter)
logger.addHandler(console)
logger.addHandler(file)
```

Using the logger this way avoids scattering `print()` calls, and ensures every message (including errors) goes to both console and disk [1] [2].

State Persistence and Recovery

Resilience to crashes/stops is critical. The **StateManager** abstracts saving/loading. It might use Python's `pickle` or `json` for serialization. Key practices (from StackOverflow advice [3]) include:

- **Atomic write**: Write state to a temporary file, then rename to `state.pkl`. If the program dies mid-save, the old state file remains safe.
- **Backup file**: Optionally keep a `.old` copy during renaming, recovering it if needed [3].
- **Periodic saving**: Call `save_state()` at checkpoints (e.g., end of each epoch) or on catching exceptions/interrupts. Use a try/finally block or signal handler so that even on `KeyboardInterrupt`, the state is saved.
- **Resume logic**: On experiment start, check `StateManager.exists()`. If so, load and restore state (model weights, epoch counter, metrics). The experiment's run method then continues from the next epoch.

Example snippet in Experiment:

```
state = StateManager.load(config.name)
if state:
```

```
    self.model.load_state_dict(state['model_weights'])
    start_epoch = state['epoch'] + 1
else:
    start_epoch = 1

for epoch in range(start_epoch, config.epochs+1):
    train_one_epoch()
    if epoch % save_every == 0:
        StateManager.save(config.name, {'epoch': epoch, 'model_weights':
self.model.state_dict(), ...})
```

This guarantees no progress is lost on failure [3]. After successful completion, the final state can be cleared or archived.

Error Handling and Robustness

Exceptions are caught at multiple levels. Within an Experiment, errors (e.g. data loading failure) can raise, but are logged and then propagated to the Runner. In the Runner loop, we use:

```
try:
    exp.run()
except Exception as e:
    Logger.error(f"Experiment {config.name} failed: {e}", exc_info=True)
    exp.save_state()
```

This ensures any unexpected error is logged with traceback, and the state is saved before halting that experiment. We can then decide whether to retry, skip, or abort the pipeline. For recoverable errors (e.g. flaky data), one might implement retries or fallbacks within Experiment.

Using Python **context managers** (the `with` statement) enhances safety. For example, opening datasets or models in a `with` block ensures they close properly even on errors. The StateManager's file operations should also use `with open(...)` to auto-close. Similarly, the Logger setup can be wrapped in a context or just rely on Python's shutdown handlers.

Extensibility and Best Practices

- **OOP Principles**: Each class has a single responsibility (SRP). The Experiment hierarchy can use inheritance (common code in base, specifics in subclasses) and even a Factory pattern to instantiate by name/type. Adding a new experiment means creating a new config and subclass only; Runner and Logger need no changes.
- **Dataclasses for Configs**: Using `@dataclass` avoids writing boilerplate `__init__` for config structures, and makes the code more maintainable [5].
- **Logging Levels**: Leverage logging levels (`INFO`, `DEBUG`, `ERROR`) to control verbosity. For routine runs use `INFO`, and allow switching to `DEBUG` for detailed tracing.

- **Unit Testing**: Since classes are decoupled, we can write unit tests for each: feed a mock config to ConfigManager, simulate a small dataset in Experiment, or a fake state to StateManager. Dependency injection (e.g. passing file paths or random seeds) aids testing.
- **Code Clarity**: Use clear naming (e.g. `Experiment.run()`, `StateManager.save()`), and include docstrings. Avoid deeply nested code; break experiment run into helper methods.

File Structure

A clean project layout aids navigation:

```
my_ml_pipeline/
├── config/                # JSON configs for each experiment
│   ├── exp1.json
│   ├── exp2.json
│   └── exp3.json
├── experiments/
│   ├── __init__.py
│   ├── base_experiment.py  # defines Experiment base class
│   ├── exp1.py            # Experiment subclass for exp1
│   ├── exp2.py
│   └── exp3.py
├── runner.py              # orchestrates experiments
├── config_manager.py      # loads configs into dataclasses
├── state_manager.py       # handles checkpoint persistence
├── logger.py              # configures Python logging
├── utils.py               # (optional) helper functions
└── results/               # output directory
    ├── exp1/
    │   ├── log.txt
    │   ├── result.json
    │   ├── model_best.pth
    │   └── model_last.pth
    └── exp2/  ...
```

Each experiment has its own subfolder under `results/`, where `log.txt`, `result.json` and model files are stored. This keeps outputs organized and separate.

Example Snippets

ConfigManager (simplified):

```
from dataclasses import dataclass
import json
```

```
@dataclass
class ExperimentConfig:
    name: str
    model_type: str
    dataset_path: str
    hyperparams: dict

class ConfigManager:
    @staticmethod
    def load(path) -> ExperimentConfig:
        data = json.load(open(path))
        return ExperimentConfig(**data)  # uses dataclass constructor
```

(**Note:** As recommended, load JSON into a dict then unpack into dataclass [5].)

Experiment Base Class:

```
class Experiment:
    def __init__(self, config: ExperimentConfig, logger):
        self.config = config
        self.logger = logger
        self.state = {}
        # initialize model, etc.

    def run(self):
        self.logger.info(f"Running {self.config.name}")
        for epoch in range(1, self.config.hyperparams["epochs"] + 1):
            # Training step...
            loss, acc = self.train_epoch(epoch)
            self.logger.info(f"Epoch {epoch}: loss={loss:.4f}, acc={acc:.2f}")
            StateManager.save(self.config.name, {
                "epoch": epoch,
                "model_state": self.model.state_dict(),
                # ... other state
            })
        self.logger.info(f"Finished {self.config.name}")

    def train_epoch(self, epoch):
        # Implement training logic or abstractmethod
        raise NotImplementedError
```

Each subclass overrides `train_epoch()` and possibly other hooks. The base `run()` handles the loop and state saving.

StateManager (simplified):

```python
import pickle, os

class StateManager:
    @staticmethod
    def save(exp_name, state):
        tmp = f"{exp_name}_state.tmp"
        final = f"{exp_name}_state.pkl"
        with open(tmp, 'wb') as f:
            pickle.dump(state, f)
        os.replace(tmp, final)  # atomic replace

    @staticmethod
    def load(exp_name):
        path = f"{exp_name}_state.pkl"
        if os.path.exists(path):
            with open(path, 'rb') as f:
                return pickle.load(f)
        return None
```

This ensures safe writes (writing to `.tmp` then `replace`) and easy load logic.

Runner Skeleton:

```python
if __name__ == "__main__":
    config_files = ["config/exp1.json", "config/exp2.json", "config/exp3.json"]
    for cfg_path in config_files:
        config = ConfigManager.load(cfg_path)
        logger = Logger.get(config.name)  # each experiment may have its own
logger
        logger.info("Starting experiment")
        # Instantiate appropriate Experiment subclass (e.g., via a factory)
        exp = ExperimentFactory.create(config, logger)
        state = StateManager.load(config.name)
        if state:
            exp.restore(state)
            logger.info("Resuming from saved state")
        try:
            exp.run()
        except Exception as e:
            logger.error(f"Error: {e}", exc_info=True)
            exp.save_state()  # persist before abort
        else:
            exp.save_state()  # final state or cleanup
            exp.save_results()  # write result.json
        logger.info("Experiment finished")
```

This shows how **Runner** coordinates everything, handling exceptions and state.

Summary

This design uses Python OOP best practices—**dataclasses** for config clarity, **context managers** for resource safety, and structured exception handling—to build a robust, maintainable experiment pipeline. Each component (Experiment, Runner, ConfigManager, Logger, StateManager) has a clear role and can be tested in isolation. Configuration is dynamic (just edit a JSON), and new experiments plug in easily. Detailed logging and checkpointing (inspired by industry advice [3] [1]) ensure full observability and fault tolerance. The result is a modular framework where ML experiments run in sequence, each fully monitored and recoverable.

Sources: We leveraged Python's logging best practices [1] [2], JSON+dataclass configuration loading [5], and safe checkpointing guidance [3] to inform this design.

---

[1] [2] Logging HOWTO — Python 3.13.3 documentation
https://docs.python.org/3/howto/logging.html

[3] python - How to checkpoint a long-running function pythonically? - Stack Overflow
https://stackoverflow.com/questions/34155841/how-to-checkpoint-a-long-running-function-pythonically

[4] ML Pipeline Architecture Design Patterns (With Examples)
https://neptune.ai/blog/ml-pipeline-architecture-design-patterns

[5] Make the Python json encoder support Python's new dataclasses - Stack Overflow
https://stackoverflow.com/questions/51286748/make-the-python-json-encoder-support-pythons-new-dataclasses