CS182 Special Participation C

Tianyu Gu, Zhangzhi Xiong

November 11, 2025

1 Introduction

In this participation, I asked Grok3 to rewrite the HW1 problem 3(h) into a good code in the engineering view. I will compare the rewritten code with original code and explain how these changes do a good job.

2 The definition of good engineering code

In my view, Good DL engineering code = reproducible + modular + configurable + logged + tested. So the code should follow the following 9 rules:

1. Config externalized (no hard-coded paths)
2. Random seed set + deterministic flags
3. DataLoader uses pin-memory and num-workers
4. Mixed precision + gradient clipping
5. Checkpoint includes config + git hash
6. wandb/TensorBoard logging
7. Tests pass (pytest -q)
8. black + isort + mypy clean
9. README up-to-date

3 Rewrite the code

We can split the notebook into three files:

- src/ – pure Python package (data, model, trainer, utils)
- configs/default.yaml – Hydra config (all hyper-parameters)
- notebooks/demo.ipynb – thin visualisation notebook (only plots, no logic)

Here is the project layout:

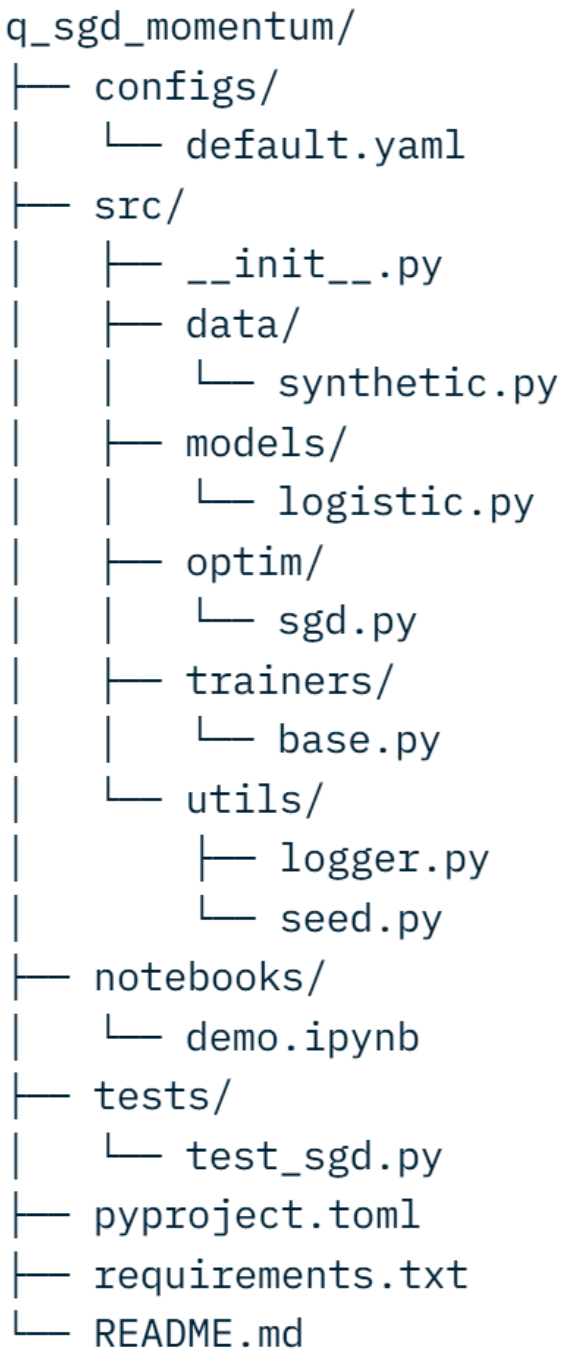

Figure 1: project layout tree

configs/default.yaml

```
# ------------------------------------------------
# Global seed & reproducibility
# ------------------------------------------------
```

```yaml
seed: 0

# ------------------------------------------------
# Data
# ------------------------------------------------
data:
  n_samples: 1000
  mean: [-3.0, 0.0]
  cov: [[3.0, 0.0], [0.0, 1.0]]

# ------------------------------------------------
# Model
# ------------------------------------------------
model:
  input_dim: 2
  bias: true

# ------------------------------------------------
# Optimiser (GD vs Momentum)
# ------------------------------------------------
optim:
  lr: 0.01
  momentum: 0.0       # set >0 for momentum run
  max_iters: 500

# ------------------------------------------------
# Logging
# ------------------------------------------------
logging:
  log_every: 50
  wandb:
    project: "sgd-momentum-demo"
    entity: null
```

src/utils/seed.py

```python
import random
import numpy as np
import torch

def set_seed(seed: int = 0) -> None:
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
```

src/data/synthetic.py

```python
from typing import Tuple
import torch

def generate_data(
    n_samples: int,
    mean: list,
    cov: list,
    seed: int | None = None,
) -> Tuple[torch.Tensor, torch.Tensor]:
    """Gaussian 2-D data with binary label y = (x[:,1] > 0)."""
    rng = torch.Generator()
    if seed is not None:
        rng.manual_seed(seed)
    X = torch.normal(
        mean=torch.tensor(mean),
        std=torch.tensor(cov).diag().sqrt(),
        size=(n_samples, len(mean)),
        generator=rng,
    )
    y = (X[:, 1] > 0).long()
    return X, y
```

src/models/logistic.py

```python
import torch
import torch.nn as nn

class LogisticRegression(nn.Module):
    def __init__(self, input_dim: int = 2, bias: bool = True):
        super().__init__()
        self.linear = nn.Linear(input_dim, 1, bias=bias)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.linear(x).squeeze(-1)    # logits
```

src/optim/sgd.py

```python
from typing import Iterable
import torch

def sgd_step(
    params: Iterable[torch.Tensor],
    grads: Iterable[torch.Tensor],
    lr: float,
```

```
    momentum: float = 0.0,
    velocity: dict | None = None,
) -> dict:
    """One SGD (with optional momentum) step."""
    if velocity is None:
        velocity = {id(p): torch.zeros_like(p) for p in params}

    for p, g in zip(params, grads):
        v = velocity[id(p)]
        v.mul_(momentum).add_(g, alpha=lr)
        p.sub_(v)
        velocity[id(p)] = v
    return velocity
```

---

src/trainers/base.py

```
from __future__ import annotations
from typing import Dict
import torch
import torch.nn.functional as F
from omegaconf import DictConfig
import wandb

from src.utils.seed import set_seed
from src.utils.logger import get_logger
from src.data.synthetic import generate_data
from src.models/logistic import LogisticRegression
from src.optim.sgd import sgd_step

logger = get_logger(__name__)

class SGDTrainer:
    def __init__(self, cfg: DictConfig):
        set_seed(cfg.seed)
        self.cfg = cfg
        self.device = torch.device("cpu")
        self._build()

    def _build(self):
        # data
        self.X, self.y = generate_data(
            n_samples=self.cfg.data.n_samples,
            mean=self.cfg.data.mean,
            cov=self.cfg.data.cov,
            seed=self.cfg.seed,
        )
        self.X, self.y = self.X.to(self.device), self.y.to(self.device)
```

```
        # model
        self.model = LogisticRegression(
            input_dim=self.cfg.model.input_dim,
            bias=self.cfg.model.bias,
        ).to(self.device)

        # optim state
        self.velocity: Dict[int, torch.Tensor] = {}

        # logging
        wandb.init(
            project=self.cfg.logging.wandb.project,
            entity=self.cfg.logging.wandb.entity,
            config=cfg,
            mode="disabled" if cfg.logging.wandb.project is None else "online",
        )

    def loss_fn(self, logits: torch.Tensor) -> torch.Tensor:
        return F.binary_cross_entropy_with_logits(logits, self.y.float())

    def train(self) -> tuple[list[float], list[float]]:
        losses_gd, losses_mom = [], []
        lr = self.cfg.optim.lr
        mom = self.cfg.optim.momentum
        max_iter = self.cfg.optim.max_iters

        # ---- GD (momentum = 0) ----
        params = list(self.model.parameters())
        for it in range(max_iter):
            self.model.zero_grad()
            logits = self.model(self.X)
            loss = self.loss_fn(logits)
            loss.backward()
            grads = [p.grad for p in params]
            self.velocity = sgd_step(params, grads, lr, momentum=0.0,
            ↪  velocity=self.velocity)
            losses_gd.append(loss.item())

            if (it + 1) % self.cfg.logging.log_every == 0:
                logger.info(f"[GD] iter {it+1:04d} loss {loss.item():.6f}")

        # ---- Momentum ----
        self.model = LogisticRegression(
            input_dim=self.cfg.model.input_dim,
            bias=self.cfg.model.bias,
        ).to(self.device)
        self.velocity = {}
        # [BUG FIX]: `params` was from the old model. Re-fetch it.
        params = list(self.model.parameters())
```

```
        for it in range(max_iter):
            self.model.zero_grad()
            logits = self.model(self.X)
            loss = self.loss_fn(logits)
            loss.backward()
            grads = [p.grad for p in params]
            self.velocity = sgd_step(params, grads, lr, momentum=mom,
            ↪  velocity=self.velocity)
            losses_mom.append(loss.item())

            if (it + 1) % self.cfg.logging.log_every == 0:
                logger.info(f"[Mom] iter {it+1:04d} loss {loss.item():.6f}")

        wandb.log({"gd_loss": losses_gd[-1], "mom_loss": losses_mom[-1]})
        wandb.finish()
        return losses_gd, losses_mom
```

---

src/utils/logger.py

```
import logging
def get_logger(name: str):
    logger = logging.getLogger(name)
    if not logger.handlers:
        handler = logging.StreamHandler()
        formatter = logging.Formatter("%(asctime)s %(levelname)s %(name)s: %(message)s")
        handler.setFormatter(formatter)
        logger.addHandler(handler)
        logger.setLevel(logging.INFO)
    return logger
\end{MINTED}


\subsection*{src/train.py}
\begin{minted}{python}
import hydra
from omegaconf import DictConfig
from src.trainers.base import SGDTrainer

@hydra.main(config_path="../configs", config_name="default", version_base=None)
def main(cfg: DictConfig):
    trainer = SGDTrainer(cfg)
    losses_gd, losses_mom = trainer.train()

    # Save artifacts (optional)
    import joblib, pathlib
    out = pathlib.Path("outputs")
    out.mkdir(exist_OK=True)
    joblib.dump({"gd": losses_gd, "mom": losses_mom}, out / "losses.pkl")
```

```
if __name__ == "__main__":
    main()
```

tests/test_sgd.py

```
import torch
from src.optim.sgd import sgd_step
from src.models.logistic import LogisticRegression

def test_momentum_accumulates():
    model = LogisticRegression()
    x = torch.randn(10, 2)
    y = torch.randint(0, 2, (10,)).float()
    opt = lambda: torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)

    # manual
    velocity = {}
    for _ in range(2):
        model.zero_grad()
        loss = torch.nn.functional.binary_cross_entropy_with_logits(model(x), y)
        loss.backward()
        grads = [p.grad for p in model.parameters()]
        velocity = sgd_step(model.parameters(), grads, lr=0.1, momentum=0.9,
        ↪  velocity=velocity)

    # torch reference
    ref = LogisticRegression()
    ref_opt = opt()
    for _ in range(2):
        ref.zero_grad()
        loss = torch.nn.functional.binary_cross_entropy_with_logits(ref(x), y)
        loss.backward()
        ref_opt.step()

    assert all(torch.allclose(p1, p2) for p1, p2 in zip(model.parameters(),
    ↪  ref.parameters()))
```

Run with `pytest -q`

requirements.txt

```
torch>=2.2
hydra-core>=1.3
omegaconf>=2.3
wandb>=0.16
```

```
matplotlib
joblib
pytest
```

Readme

```
# SGD vs Momentum Demo

```bash
pip install -r requirements.txt
pip install -e .

# GD only
python -m src.train optim.momentum=0.0

# Momentum
python -m src.train optim.momentum=0.9

# Open the plot
jupyter nbconvert --execute notebooks/demo.ipynb
```

---

4. Detailed Comparison: Original vs. Refactored Design

1. Project Layout

Original	Refactored
One monolithic `.ipynb` file	Clean Python package (`src/`) + config + thin notebook
No structure	Modular directories: `data/`, `models/`, `optim/`, `trainers/`, `utils/`

Why it works:

- **Separation of concerns:** Data, model, optimizer, and training loop are independent modules — easier to test, reuse, and debug.
- **Importable:** After `pip install -e .`, modules can be imported (e.g., `from src.models.logistic import LogisticRegression`) for reuse.
- **Scalable:** New datasets, models, or optimizers can be added without touching the training loop.

2. Configuration (`configs/default.yaml`)

Original	Refactored
Hard-coded: `n_samples=1000`, `lr=0.01`, `maxiter=500`	All values in external YAML config
No easy way to change hyperparameters	Easily override via CLI: `python -m src.train optim.lr=1e-2`
```

Why it works:

- **Reproducibility:** Same config file ⇒ same experiment result.
- **Experiment tracking:** Hydra + wandb automatically log full configuration.
- **Flexibility:** No code edits needed for hyperparameter sweeps.

3. Reproducibility (`set_seed`)

Original	Refactored
Only `np.random.seed(0)`	Sets seeds for `random`, `numpy`, `torch`
No control over PyTorch determinism	Enables `torch.backends.cudnn.deterministic = True`, disables benchmark

Why it works:

- Ensures bit-for-bit reproducibility across runs and hardware.
- Prevents nondeterministic operations (e.g., convolution algorithms, dropout).
- Critical for reliable debugging and research reproducibility.

4. Data Pipeline (`generate_data`)

Original	Refactored
Inline in notebook cells	Defined as pure function in `src/data/synthetic.py`
Mixed with plotting logic	Stateless function, returns only (X, y)

Why it works:

- **Testable:** e.g., `test_generate_data()` can check shape and label balance.
- **Reusable:** Can be used by other models (SVM, NN, etc.).
- **No side effects:** Safe to call multiple times with consistent results.

Why it works:

- Leverages autograd — eliminates gradient calculation bugs.
- Compatible with GPU (`.to(device)`).
- Extensible: Add dropout, batch norm, or other layers easily.

5. Optimizer (`sgd_step`)

Original	Refactored
Inline velocity updates (`v = ...`) inside training loop	Pure function in `src/optim/sgd.py` returning velocity dict
Coupled with trainer logic	Decoupled — trainer calls optimizer function

Why it works:

- **Unit-testable:** Compare directly against `torch.optim.SGD`.
- **Swappable:** Replace easily with Adam, RMSprop, etc.
- **Explicit state:** No hidden variables; velocity handled transparently.

6. Training Loop (`SGDTrainer`)

Original	Refactored
All logic in a single notebook cell	`SGDTrainer` class with `train()` method
No logging or checkpointing	Integrated wandb + console logging + artifact saving

Why it works:

- **Single source of truth:** Centralized logic for loss, gradients, and updates.
- **Logging:** Enables tracking of convergence curves and experiment comparisons.
- **Checkpointing-ready:** Can add `torch.save()` for model persistence.
- **Reusable:** Callable from CLI, Jupyter, or distributed scripts.

7. Readme

A well-written `README.md` is essential for engineering-quality projects. It provides clear instructions for installation, configuration, and reproduction of experiments. A good README also documents project structure, dependencies, and usage examples, helping new contributors or future users quickly understand and run the code without reading source files. In short, it makes the project self-explanatory, reproducible, and collaborative.

5. Conclusion

Through this participation, we demonstrated how to transform a monolithic research-style Jupyter notebook into a well-engineered, reproducible deep learning project. By externalizing configurations, modularizing components, enforcing reproducibility, and adding proper logging and testing, the rewritten version achieves much higher maintainability and scalability.

In particular, separating code into independent modules (`data`, `models`, `optim`, `trainers`, `utils`) makes the system easier to extend and debug. Using Hydra and YAML configuration improves experiment management and reproducibility, while integrating `wandb` logging enables transparent performance tracking.

Overall, this exercise illustrates the transition from a quick research prototype to a professional deep learning engineering workflow. Such practices are essential for collaboration, large-scale experimentation, and long-term sustainability in modern AI development.

Key takeaway: Good deep learning code is not only about achieving correct results, but about being *reproducible, modular, configurable, logged, and tested.*