```python
import structlog
import jax
import numpy as np
import optax
import tensorflow_datasets as tfds
from flax import nnx

from .logging import configure_logging
from .config import load_settings
from .data import Data
from .model import Classifier
from .training_testing import train, test


def main() -> None:
    """CLI entry point."""
    configure_logging()
    log = structlog.get_logger()
    log.info("Hello from hw03!")

    settings = load_settings()
    log.info("Settings loaded", settings=settings.model_dump())

    # JAX PRNG
    key = jax.random.PRNGKey(settings.random_seed)
    data_key, model_key = jax.random.split(key)
    np_rng = np.random.default_rng(np.array(data_key))

    mnist_train_val = tfds.load(
        "mnist",
        split=[
            "train[:{percent}%]".format(percent=settings.data.percent_train),
            "train[{percent}%:]".format(percent=settings.data.percent_train),
        ],
        shuffle_files=True,
        as_supervised=True,
    )
    mnist_train, mnist_val = tfds.as_numpy(mnist_train_val)
    mnist_test = tfds.load(
        "mnist", split="test", shuffle_files=True, as_supervised=True
    )
    mnist_test = tfds.as_numpy(mnist_test)

    data = Data(
        rng=np_rng,
        train_set=mnist_train,
        val_set=mnist_val,
        test_set=mnist_test,
    )

    model = Classifier(
        key=model_key,
        input_height=settings.model.input_height,
        input_width=settings.model.input_width,
        input_depth=settings.model.input_depth,
        layer_depths=settings.model.layer_depths,
        layer_kernel_sizes=settings.model.layer_kernel_sizes,
        num_classes=settings.model.num_classes,
    )

    learning_rate_schedule = optax.exponential_decay(
        init_value=settings.training.learning_rate,
        transition_steps=5000,
        decay_rate=0.1,
    )

    optimizer = nnx.Optimizer(
        model,
        optax.adamw(
```

```python
            learning_rate=learning_rate_schedule,
            weight_decay=settings.training.l2_reg,
        ),
        wrt=nnx.Param,
    )

    val_accuracy = train(model, optimizer, data, settings.training, np_rng)
    log.debug("Finished Training model")

    if val_accuracy > 0.955:
        log.info("Achieved target accuracy on validation set", accuracy=val_accuracy)
        test(model, data)
    else:
        log.info("Did not achieve target accuracy on validation set", accuracy=val_accuracy)
```

```python
import jax
import structlog
from flax import nnx
from jax import numpy as jnp
import numpy as np

log = structlog.get_logger()


class Conv2d(nnx.Module):
    """A Flax NNX module for a 2D convolutional layer."""

    def __init__(
        self,
        *,
        rngs: nnx.Rngs,
        in_features: int,
        out_features: int,
        kernel_size: tuple[int, int],
    ):
        self.conv = nnx.Conv(
            in_features=in_features,
            out_features=out_features,
            kernel_size=kernel_size,
            rngs=rngs,
        )

    def __call__(self, x: jax.Array) -> jax.Array:
        return nnx.avg_pool(nnx.relu(self.conv(x)), window_shape=(2, 2), strides
=(2, 2))


class Classifier(nnx.Module):
    """A Flax NNX module for an CNN model."""

    def __init__(
        self,
        *,
        key: jax.random.PRNGKey,
        input_height: int,
        input_width: int,
        input_depth: int,
        layer_depths: list[int],
        layer_kernel_sizes: list[tuple[int, int]],
        num_classes: int,
    ):
        keys = jax.random.split(key, len(layer_depths) + 2)
        # layer_depths and layer_kernel_sizes must have the same length
        assert len(layer_depths) == len(layer_kernel_sizes)
        self.conv_layers = []
        current_depth = input_depth

        for depth, kernel_size, rng_key in zip(
            layer_depths, layer_kernel_sizes, keys[:-2]
        ):
            self.conv_layers.append(
                Conv2d(
                    rngs=nnx.Rngs(params=rng_key),
                    in_features=current_depth,
                    out_features=depth,
                    kernel_size=kernel_size,
                )
            )
            current_depth = depth

        self.dropout = nnx.Dropout(rate=0.5, rngs=nnx.Rngs(dropout=keys[-2]))

        # Compute the number of features after the conv layers
        dummy_input = jnp.ones((1, input_height, input_width, input_depth))
```

```python
        dummy_output = dummy_input
        for layer in self.conv_layers:
            dummy_output = layer(dummy_output)
        dense_in_features = np.prod(dummy_output.shape[1:])

        self.dense = nnx.Linear(
            in_features=dense_in_features,
            out_features=num_classes,
            rngs=nnx.Rngs(params=keys[-1]),
        )

    def __call__(self, x: jax.Array) -> jax.Array:
        # """Iterates through the CNN layers."""
        for layer in self.conv_layers:
            x = layer(x)

        x = x.reshape((x.shape[0], -1))
        x = self.dropout(x)
        x = self.dense(x)

        return x
```

```
from dataclasses import InitVar, dataclass, field

import numpy as np
import structlog


log = structlog.get_logger()


@dataclass
class Data:
    """Handles generation of synthetic data for CNN."""

    rng: InitVar[np.random.Generator]
    train_set: np.ndarray
    val_set: np.ndarray
    test_set: np.ndarray
    train_image_set: np.ndarray = field(init=False)
    train_label_set: np.ndarray = field(init=False)
    val_image_set: np.ndarray = field(init=False)
    val_label_set: np.ndarray = field(init=False)

    def __post_init__(self, rng: np.random.Generator) -> None:
        """Generate training and validating data."""

        self.train_image_set = np.stack([x for x, _ in self.train_set])
        self.train_label_set = np.stack([y for _, y in self.train_set])
        self.val_image_set = np.stack([x for x, _ in self.val_set])
        self.val_label_set = np.stack([y for _, y in self.val_set])
        self.test_image_set = np.stack([x for x, _ in self.test_set])
        self.test_label_set = np.stack([y for _, y in self.test_set])
        self.train_index = np.arange(self.train_image_set.shape[0])

        log.debug(
            "Data initialized",
            train_images=self.train_image_set.shape,
            train_labels=self.train_label_set.shape,
            val_images=self.val_image_set.shape,
            val_labels=self.val_label_set.shape,
        )

    def get_batch(
        self, rng: np.random.Generator, batch_size: int
    ) -> tuple[np.ndarray, np.ndarray]:
        """Select random subset of examples for training batch."""
        choices = rng.choice(self.train_index, size=batch_size)

        return self.train_image_set[choices], self.train_label_set[choices].flat
ten()
```

```
import logging
import os
from pathlib import Path

import jax
import numpy as np
import structlog

from .config import LoggingSettings


class FormattedFloat(float):
    def __repr__(self) -> str:
        return f"{self:.4g}"


def custom_serializer_processor(logger, method_name, event_dict):
    for key, value in event_dict.items():
        # Handle JAX arrays in addition to TF tensors
        if hasattr(value, "numpy"):  # Covers TF tensors
            value = value.numpy()
        if isinstance(value, jax.Array):
            value = np.array(value)
        if isinstance(value, (np.generic, np.ndarray)):
            value = value.item() if value.size == 1 else value.tolist()
        if isinstance(value, float):
            value = FormattedFloat(value)
        if isinstance(value, Path):
            value = str(value)
        event_dict[key] = value
    return event_dict


def configure_logging():
    """Configure logging for the application."""
    log_settings = LoggingSettings()
    log_settings.output_dir.mkdir(parents=True, exist_ok=True)
    log_path = log_settings.output_dir / "hw03.log"
    logging.basicConfig(
        format="%(message)s",
        level=logging.INFO,
        filename=log_path,
        filemode="w",
    )

    # Set the level for the application’s logger
    log_level = os.environ.get("LOG_LEVEL", "INFO").upper()
    logging.getLogger("hw03").setLevel(log_level)

    structlog.configure(
        processors=[
            structlog.stdlib.filter_by_level,
            structlog.stdlib.add_logger_name,
            structlog.stdlib.add_log_level,
            structlog.stdlib.PositionalArgumentsFormatter(),
            structlog.processors.TimeStamper(fmt="iso"),
            structlog.processors.StackInfoRenderer(),
            structlog.processors.format_exc_info,
            structlog.processors.UnicodeDecoder(),
            custom_serializer_processor,
            structlog.processors.JSONRenderer(),
        ],
        context_class=dict,
        logger_factory=structlog.stdlib.LoggerFactory(),
        wrapper_class=structlog.stdlib.BoundLogger,
        cache_logger_on_first_use=True,
    )
```

```python
import jax
import jax.numpy as jnp
import numpy as np
import structlog
import optax
from flax import nnx
from tqdm import trange

from .config import TrainingSettings
from .data import Data
from .model import Classifier

log = structlog.get_logger()


def calc_values(x, y):
    one_hot_labels = jax.nn.one_hot(y, num_classes=10)
    loss = optax.softmax_cross_entropy(x, one_hot_labels).mean()
    accuracy = jnp.mean(jnp.argmax(x, axis=-1) == y)
    return [loss, accuracy]


@nnx.jit
def train_step(
    model: Classifier, optimizer: nnx.Optimizer, x: jnp.ndarray, y: jnp.ndarray
):
    """Performs a single training step."""

    def loss_fn(model: Classifier):
        y_hat = model(x)
        return calc_values(y_hat, y)[0]

    model.train()
    loss, grads = nnx.value_and_grad(loss_fn)(model)
    optimizer.update(model, grads)  # In-place update of model parameters
    model.eval()
    accuracy = calc_values(model(x), y)[1]
    return loss, accuracy


def train(
    model: Classifier,
    optimizer: nnx.Optimizer,
    data: Data,
    settings: TrainingSettings,
    np_rng: np.random.Generator,
) -> float:
    """Train the model using SGD."""
    log.info("Starting training", **settings.model_dump())
    bar = trange(settings.epochs)
    for i in bar:
        train_image_batch, train_label_batch = data.get_batch(
            np_rng, settings.batch_size
        )

        loss, accuracy = train_step(
            model, optimizer, train_image_batch, train_label_batch
        )
        # log.debug("Training step", step=i, loss=loss)
        bar.set_description(f"Loss @ {i} => {loss:.6f}, Acc @ {accuracy:.6f}")
        bar.refresh()

        if i % 100 == 0:
            log.info("Training step", step=i, loss=loss, accuracy=accuracy)

    log.info("Training step", step=settings.epochs, loss=loss, accuracy=accuracy)
    log.info("Training finished")

    # test on validation set
```

```python
    val_accuracy = calc_values(model(data.val_image_set), data.val_label_set)[1]
    log.info("Validation accuracy", accuracy=val_accuracy)

    return val_accuracy


def test(
    model: Classifier,
    data: Data,
) -> None:
    """Test the model using test dataset."""
    test_accuracy = calc_values(model(data.test_image_set), data.test_label_set)
[1]
    log.info("Test accuracy", accuracy=test_accuracy)
```

```python
from pathlib import Path
from typing import Tuple

from pydantic import BaseModel
from pydantic_settings import BaseSettings


class DataSettings(BaseModel):
    """Settings for data generation."""

    percent_train: int = 90


class ModelSettings(BaseModel):
    """Settings for model architecture."""

    input_height: int = 28
    input_width: int = 28
    input_depth: int = 1
    layer_depths: list[int] = [32, 64, 128]
    layer_kernel_sizes: list[tuple[int, int]] = [(7, 7), (5, 5), (3, 3)]
    num_classes: int = 10


class TrainingSettings(BaseModel):
    """Settings for model training."""

    batch_size: int = 64
    epochs: int = 500
    learning_rate: float = 0.01
    l2_reg: float = 0.01


class PlottingSettings(BaseModel):
    """Settings for plotting."""

    figsize: Tuple[int, int] = (10, 10)
    dpi: int = 200
    output_dir: Path = Path("hw03/artifacts")


class LoggingSettings(BaseModel):
    """Settings for logging."""

    log_level: str = "INFO"
    log_format: str = "plain"  # "json" or "plain"
    log_output: str = "stdout"  # "stdout" or "file"
    output_dir: Path = Path("hw03/artifacts")


class AppSettings(BaseSettings):
    """Main application settings."""

    debug: bool = False
    random_seed: int = 31415
    data: DataSettings = DataSettings()
    model: ModelSettings = ModelSettings()
    training: TrainingSettings = TrainingSettings()
    plotting: PlottingSettings = PlottingSettings()
    logging: LoggingSettings = LoggingSettings()


def load_settings() -> AppSettings:
    """Load application settings."""
    return AppSettings()
```

```
# pyproject.toml.jinja

[project]
name = "hw03"
version = "0.1.0"
description = "Classify MNIST digits with a convolutional neural network with at
 least 95.5% accuracy."
readme = "README.md"
authors = [
    { name = "Donghyun Park", email = "donghyun.park@cooper.edu" }
]
requires-python = ">=3.13"
dependencies = [
    "structlog",
    "rich",
    "numpy",
    "jax",
    "flax",
    "pydantic>=2.11.9",
    "pydantic-settings>=2.10.1",
    "tensorflow-datasets>=4.9.9",
    "tensorflow>=2.20.0",
]

[project.scripts]
hw03 = "hw03:main"

[build-system]
requires = ["uv_build>=0.8.3,<0.9.0"]
build-backend = "uv_build"
```