```python
from pathlib import Path
from importlib.resources import files
from typing import Tuple, List

from pydantic import BaseModel, field_validator
from pydantic_settings import (
    BaseSettings,
    PydanticBaseSettingsSource,
    SettingsConfigDict,
    TomlConfigSettingsSource,
)


class DataSettings(BaseModel):
    """Settings for data generation."""
    batch_size: int = 128
    val_split: float = 0.1


class TrainingSettings(BaseModel):
    """Settings for model training."""
    batch_size: int = 128
    num_iters: int = 500
    learning_rate: float = 0.001


class ModelSettings(BaseModel):
    """Settings for model configuration."""
    input_depth: int = 1
    layer_depths: List[int] = [32, 64]
    layer_kernel_sizes: List[List[int]] = [[3, 3], [3, 3]]  #cast to list of lis
ts
    num_classes: int = 10
    dropout: float = 0.5
    l2_reg: float = 1e-4

    #validator to convert list of lists to list of tuples
    @field_validator('layer_kernel_sizes')
    @classmethod
    def convert_to_tuples(cls, v):
        """Convert list of lists to list of tuples for the model."""
        return [tuple(kernel) for kernel in v]


class AppSettings(BaseSettings):
    """Main application settings."""
    debug: bool = False
    random_seed: int = 31415
    data: DataSettings = DataSettings()
    model: ModelSettings = ModelSettings()
    training: TrainingSettings = TrainingSettings()

    model_config = SettingsConfigDict(
        toml_file=files("hw03").joinpath("config.toml"),
        env_nested_delimiter="__",
    )

    @classmethod
    def settings_customise_sources(
        cls,
        settings_cls: type[BaseSettings],
        init_settings: PydanticBaseSettingsSource,
        env_settings: PydanticBaseSettingsSource,
        dotenv_settings: PydanticBaseSettingsSource,
        file_secret_settings: PydanticBaseSettingsSource,
    ) -> tuple[PydanticBaseSettingsSource, ...]:
        """
    Set the priority of settings sources.
    We use a TOML file for configuration.
    """
        return (
            init_settings,
            TomlConfigSettingsSource(settings_cls),
            env_settings,
            dotenv_settings,
            file_secret_settings,
        )


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

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

log = structlog.get_logger()

class Conv2d(nnx.Module):
    """
  A 2D convolutional layer wrapper.
  """

    def __init__(
            self,
            in_features: int,
            out_features: int,
            kernel_size: tuple[int, int] = (3, 3),
            padding: str = 'SAME',  #output size is same as input
            strides: tuple[int, int] = (1, 1),      #stride = 1
            rngs=None,
    ):
        super().__init__()
        self.conv = nnx.Conv(
            in_features=in_features,
            out_features=out_features,
            kernel_size=kernel_size,
            padding=padding,
            strides=strides,
            rngs=rngs,
        )

    def __call__(self, x: jax.Array) -> jax.Array:
        return self.conv(x)


class Classifier_mnist(nnx.Module):
    """
  A simple CNN model for MNIST classification.
  """
    def __init__(
            self,
            input_depth: int,
            layer_depths: list[int],
            layer_kernel_sizes: list[tuple[int, int]],
            num_classes: int,
            dropout: float  = 0.5,
            l2_reg: float = 1e-4,
            rngs: nnx.Rngs = None,
    ):
        super().__init__()
        self.input_depth = input_depth
        self.layer_depths = layer_depths
        self.layer_kernel_sizes = layer_kernel_sizes
        self.num_classes = num_classes
        self.dropout = dropout
        self.l2_reg = l2_reg

        # Create convolutional layers
        self.conv_layers = []
        in_ch = input_depth
        for i, out_ch in enumerate(layer_depths):
            conv = Conv2d(
                in_features=in_ch,
                out_features=int(out_ch),
                kernel_size=tuple(self.layer_kernel_sizes[i]),
                padding='SAME',
                strides=(1, 1),
                rngs=None,
            )
            self.conv_layers.append(conv)
            in_ch = out_ch

        # Maxpool, dropout, flatten layers
        self.maxpool = nnx.MaxPool(window_shape=(2, 2), strides=(2, 2))
        self.dropout_layer = nnx.Dropout(rate=dropout)
        self.flatten = nnx.Flatten()

        # Calculate dense layer input size - MNIST: 28x28 -> 14x14 -> 7x7 after 2 maxpools
        final_size = 28 // (2 ** len(layer_depths))
        final_size = max(1, final_size)
        dense_input_size = final_size * final_size * self.layer_depths[-1]

        self.dense = nnx.Linear(dense_input_size, self.num_classes)

    def l2_loss(self, params) -> jax.Array:
        """
    Compute L2 penalty from a params PyTree (returned by model.init).
    Skips 0–D/1–D arrays (bias–like) to match common L2 usage.
    """
        leaves = jax.tree_util.tree_leaves(params)
        acc = 0.0
        for p in leaves:
            arr = jnp.asarray(p)
            if arr.ndim > 1:
                acc += jnp.sum(arr ** 2)
        return self.l2_reg * acc

    def __call__(self, x: jax.Array, training: bool = False, rng: jax.Array | None = None) -> jax.Array:
        """
    Forward pass of the classifier.
    """
        if x.ndim == 3:
            x = x[..., None]
        if x.ndim != 4:
            raise ValueError("Input must be (N,H,W) or (N,H,W,C)")

        # Prepare dropout keys: one per dropout call (one after each conv, plus one after fc)
        num_dropout_uses = len(self.conv_layers) + 1
        if training:
            if rng is None:
                raise ValueError("When training=True you must pass a PRNGKey via 'rng'.")
            drop_keys = jax.random.split(rng, num=num_dropout_uses)
        else:
            drop_keys = [None] * num_dropout_uses

        # conv -> relu -> pool -> dropout for each conv layer
        for i, conv_layer in enumerate(self.conv_layers):
            x = conv_layer(x)
            x = jax.nn.relu(x)
            x = self.maxpool(x)

            if training:
                # wrap the single key for this dropout call in nnx.Rngs
                x = self.dropout_layer(x, rngs=nnx.Rngs(dropout=drop_keys[i]))
            else:
                # deterministic path
                x = self.dropout_layer(x, deterministic=True)

        # flatten -> dropout -> dense
        x = self.flatten(x)

        if training:
```

```
            x = self.dropout_layer(x, rngs=nnx.Rngs(dropout=drop_keys[-1]))
        else:
            x = self.dropout_layer(x, deterministic=True)

        logits = self.dense(x)
        return logits
```

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

from .data import Data
from .model import Classifier_mnist
from .config import load_settings
from .training import train_mnist
from .logging import configure_logging


def main() -> None:
    """CLI entry point."""
    settings = load_settings()
    configure_logging()
    log = structlog.get_logger()
    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))

    # Load data
    data = Data(
        model=None,
        rng=np_rng,
        batch_size=int(settings.data.batch_size),
        val_split=float(settings.data.val_split),
    )
    log.debug("Data loaded")

    # Create model and optimizer
    model_rngs = nnx.Rngs(params=model_key)

    model = Classifier_mnist(
        input_depth=int(settings.model.input_depth),
        layer_depths=list(settings.model.layer_depths),
        layer_kernel_sizes=list(settings.model.layer_kernel_sizes),     #I caste
d and change config.py for this to work but I don't think this is good practice,
 is there a better way?
        num_classes=int(settings.model.num_classes),
        dropout=float(settings.model.dropout),
        l2_reg=float(settings.model.l2_reg),
        rngs=model_rngs,
    )
    log.debug("Model created")

    # Initialize optimizer
    optimizer = nnx.Optimizer(
        model, optax.adam(settings.training.learning_rate), wrt=nnx.Param
    )

    # Train the model
    train_mnist(model, optimizer, data, settings.training, np_rng)
    log.info("Training completed")
```

```
debug = false
random_seed = 0xc61dd8fc

[data]
batch_size = 128
val_split = 0.1

[model]
input_depth = 1
layer_depths = [32, 64]
layer_kernel_sizes = [[3, 3], [3, 3]]
num_classes = 10
dropout = 0.5
l2_reg = 1e-4

[training]
batch_size = 128
num_iters = 500
learning_rate = 0.001
```

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

log = structlog.get_logger()

class Conv2d(nnx.Module):
    """A 2D convolutional layer wrapper."""

    def __init__(
        self,
        in_features: int,
        out_features: int,
        kernel_size: tuple[int, int] = (3, 3),
        padding: str = 'SAME',
        strides: tuple[int, int] = (1, 1),
        rngs: nnx.Rngs = None,
    ):
        super().__init__()
        self.conv = nnx.Conv(
            in_features=in_features,
            out_features=out_features,
            kernel_size=kernel_size,
            padding=padding,
            strides=strides,
            rngs=rngs,
        )

    def __call__(self, x: jax.Array) -> jax.Array:
        return self.conv(x)


class Classifier_mnist(nnx.Module):
    """A simple CNN model for MNIST classification."""

    def __init__(
        self,
        input_depth: int,
        layer_depths: list[int],
        layer_kernel_sizes: list[tuple[int, int]],
        num_classes: int,
        dropout: float,
        l2_reg: float,
        rngs: nnx.Rngs = None,
    ):
        super().__init__()
        self.input_depth = input_depth
        self.layer_depths = layer_depths
        self.layer_kernel_sizes = layer_kernel_sizes
        self.num_classes = num_classes
        self.dropout = dropout
        self.l2_reg = l2_reg

        # Create convolutional layers
        self.conv_layers = []
        in_ch = input_depth

        for i, out_ch in enumerate(layer_depths):
            self.conv_layers.append(
                Conv2d(
                    in_features=in_ch,
                    out_features=out_ch,
                    kernel_size=layer_kernel_sizes[i],
                    rngs=rngs,
                )
            )
            in_ch = out_ch
```



```python
        # Calculate dense layer input size, 28x28 to 7x7 after 2 poolings
        final_size = 7 * 7 * layer_depths[-1]  # I should change this later to f
unction for more flexibility

        self.dense = nnx.Linear(final_size, num_classes, rngs=rngs)

    def max_pool(self, x: jax.Array, window_size: int = 2, stride: int = 2) -> j
ax.Array:
        """Max Pooling function to reduce overfitting and downsample for efficiency."""
        return jax.lax.reduce_window(
            x,
            -jnp.inf,   # Identity for max, -infinity
            jax.lax.max,
            window_dimensions=(1, window_size, window_size, 1),
            window_strides=(1, stride, stride, 1),
            padding='SAME'
        )

    def __call__(self, x: jax.Array, training: bool = False, rngs: nnx.Rngs = No
ne) -> jax.Array:
        """Forward pass of the classifier."""
        # Apply convolutional layers with ReLU and pooling
        for conv_layer in self.conv_layers:
            x = conv_layer(x)
            x = jax.nn.relu(x)
            x = self.max_pool(x, window_size=2, stride=2)

        # Flatten
        batch_size = x.shape[0]
        x = x.reshape((batch_size, -1))

        # Apply dropout for training data
        if training:
            if rngs is not None:
                # Use dropout with RNGs
                x = nnx.Dropout(rate=self.dropout)(x, rngs=rngs)
            else:
                # If no RNGs provided, create a temporary one (for non-JIT cases
)
                temp_rngs = nnx.Rngs(42)  # arbitrary seed
                x = nnx.Dropout(rate=self.dropout)(x, rngs=temp_rngs)

        x = self.dense(x)
        return x

    def l2_loss(self) -> jax.Array:
        """Calculate L2 regularization loss."""
        l2_loss = 0.0
        # For convolutional layers
        for conv_layer in self.conv_layers:
            if hasattr(conv_layer.conv, 'kernel'):
                l2_loss += self.l2_reg * jnp.sum(conv_layer.conv.kernel ** 2)

        # For dense layer
        if hasattr(self.dense, 'kernel'):
            l2_loss += self.l2_reg * jnp.sum(self.dense.kernel ** 2)

        return l2_loss
```

```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_mnist as Classifier

log = structlog.get_logger()

def calculate_accuracy(model: Classifier, data: Data, batch_size: int, validatio
n_set: bool = True) -> float:
    """Calculate classification accuracy."""
    if validation_set:
        x_np, y_np = data.get_val_data()
    else:
        x_np, y_np = data.get_test_data()

    n = len(y_np)
    correct = 0
    total = 0

    for i in range(0, n, batch_size):
        x_batch = jnp.asarray(x_np[i:i + batch_size])
        y_batch = jnp.asarray(y_np[i:i + batch_size])

        logits = model(x_batch, training=False)
        predictions = jnp.argmax(logits, axis=1)

        equal = predictions == y_batch
        total += len(equal)
        correct += jnp.sum(equal)

    accuracy = float(correct / total)
    log.info('Accuracy calculated',
             correct=int(correct),
             total=int(total),
             accuracy=accuracy,
             dataset='validation' if validation_set else 'test')

    return accuracy

def train_step(
    model: Classifier,
    optimizer: nnx.Optimizer,
    x: jnp.ndarray,
    y: jnp.ndarray,
    dropout_key: jax.Array
) -> float:
    """Single training step with cross-entropy loss and L2 regularization."""

    def loss_fn(model: Classifier):
        # Create RNGs from the key for this specific call
        rngs = nnx.Rngs(dropout=dropout_key)
        logits = model(x, training=True, rngs=rngs)
        loss_ce = optax.softmax_cross_entropy_with_integer_labels(logits, y)
        loss_ce = jnp.mean(loss_ce)
        l2_loss = model.l2_loss()
        return loss_ce + l2_loss

    loss, grads = nnx.value_and_grad(loss_fn)(model)

    # Update optimizer with both model and grads (new API)
    optimizer.update(model, grads)
```



```python
    return loss

# JIT compile the training step
train_step_jitted = nnx.jit(train_step)

def train_mnist(
    model: Classifier,
    optimizer: nnx.Optimizer,
    data: Data,
    settings: TrainingSettings,
    np_rng: np.random.Generator,
) -> float:
    """Train the model and return final accuracy."""
    log.info("Starting training", **settings.model_dump())

    # Convert numpy RNG to JAX RNG
    key = jax.random.PRNGKey(np_rng.integers(0, 2**32))

    bar = trange(settings.num_iters)
    for i in bar:
        x_np, y_np = data.get_batch(np_rng, settings.batch_size)
        x, y = jnp.asarray(x_np), jnp.asarray(y_np)

        # Split key for this training step
        key, dropout_key = jax.random.split(key)

        # Use JIT-compiled training step
        loss = train_step_jitted(model, optimizer, x, y, dropout_key)

        if i % 100 == 0:
            bar.set_description(f"Loss @ {i} => {loss:.6f}")

    log.info("Training finished")

    # Evaluate on validation set
    val_accuracy = calculate_accuracy(model, data, settings.batch_size, validati
on_set=True)

    if val_accuracy > 0.955:
        test_accuracy = calculate_accuracy(model, data, settings.batch_size, val
idation_set=False)
        log.info("Final test result:", accuracy=test_accuracy)
        return test_accuracy
    else:
        log.info("Validation accuracy too low, skipping test set")
        return val_accuracy
```

```python
import logging
import os
import sys
from pathlib import Path

import jax
import numpy as np
import structlog


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."""
    logging.basicConfig(
        format="%(message)s",
        filename="hw03_results.txt",
        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.dev.ConsoleRenderer(
                colors=True, exception_formatter=structlog.dev.RichTracebackFormatter()
            ),
        ],
        context_class=dict,
        logger_factory=structlog.stdlib.LoggerFactory(),
        wrapper_class=structlog.stdlib.BoundLogger,
        cache_logger_on_first_use=True,
    )
```

```python
from dataclasses import InitVar, dataclass, field

import structlog
import numpy as np
import tensorflow as tf

log = structlog.get_logger()

@dataclass
class Data:
    """Handles loading and batching of the MNIST dataset."""

    model: object
    rng: InitVar[np.random.Generator]
    batch_size: int
    val_split: float
    x_train: np.ndarray = field(init=False)
    y_train: np.ndarray = field(init=False)
    x_val: np.ndarray = field(init=False)
    y_val: np.ndarray = field(init=False)
    x_test: np.ndarray = field(init=False)
    y_test: np.ndarray = field(init=False)
    index: np.ndarray = field(init=False)

    def __post_init__(self, rng: np.random.Generator):
        """Load MNIST dataset using TensorFlow Datasets."""

        (x_train, y_train), (self.x_test, self.y_test) = tf.keras.datasets.mnist
.load_data()

        # Normalize and reshape to (N, 28, 28), add channel dimension so (N, 28,
 28, 1)
        x_train = x_train.astype(np.float32) / 255.0
        self.x_test = self.x_test.astype(np.float32) / 255.0
        x_train = np.expand_dims(x_train, axis=-1)
        self.x_test = np.expand_dims(self.x_test, axis=-1)

        # Split training data into training and validation sets
        index = np.arange(len(x_train))
        rng.shuffle(index)
        split_idx = int(len(x_train) * (1 - self.val_split))
        train_idx, val_idx = index[:split_idx], index[split_idx:]

        # Index for batching
        self.index = np.arange(len(train_idx))
        self.x_train, self.y_train = x_train[train_idx], y_train[train_idx]
        self.x_val, self.y_val = x_train[val_idx], y_train[val_idx]


    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.index, size=batch_size)

        return self.x_train[choices], self.y_train[choices].flatten()

    def get_val_data(self) -> tuple[np.ndarray, np.ndarray]:
        """Return the entire validation dataset."""
        return self.x_val, self.y_val.flatten()

    def get_test_data(self) -> tuple[np.ndarray, np.ndarray]:
        """Return the entire test dataset."""
        return self.x_test, self.y_test.flatten()
```

```
# pyproject.toml.jinja

[project]
name = "hw03"
version = "0.1.0"
description = "Classify MNIST digits with a convolutional neural network."
readme = "README.md"
authors = [
    { name = "Yunwen Zhu", email = "yunwen.zhu@cooper.edu" }
]
requires-python = ">=3.13"
dependencies = [
    "structlog",
    "numpy",
    "pydantic-settings>=2.10.1",
    "matplotlib>=3.10.5",
    "tqdm>=4.67.1",
    "jax[cpu]>=0.7.1",
    "flax>=0.11.2",
    "optax>=0.2.4",
    "tensorflow",
]

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

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