```toml
# pyproject.toml.jinja

[project]
name = "hw03"
version = "0.1.0"
description = "MNIST CNN"
readme = "README.md"
authors = [
    { name = "Josh Miao", email = "joshuamiao03@gmail.com" }
]
requires-python = ">=3.13"
dependencies = [
    "structlog",
    "rich",
    "numpy",
    "jax",
    "flax",
    "tensorflow>=2.20.0",
    "tqdm>=4.67.1",
    "scikit-learn>=1.7.2",
    "pydantic>=2.11.9",
    "pydantic-settings>=2.10.1",
    "tensorflow-datasets>=4.9.9",
    "optax>=0.2.6",
    "jaxlib>=0.7.2",
    "tfds>=0.3",
    "typing>=3.10.0.0",
    "dataclasses>=0.8",
]

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

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

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

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


class DataSettings(BaseModel):
    """Settings for data generation."""
    train_ratio: float = 0.8

class ModelSettings(BaseModel):
    input_depth: int = 1
    num_classes: int = 10
    layer_depth: List[int] = [32, 64]
    layer_kernel_sizes: List[List[int]] = [[3, 3], [3,3]]
    strides: List[int] = [2, 2]
#increased spatial size for faster training bc it literally took 10 min before t
his
    dropout_rate: float = 0.1
    shape: List[int] = [28, 28]

class TrainingSettings(BaseModel):
    """Settings for model training."""
    batch_size: int = 128
    num_iters: int = 1000
    learning_rate: float = 0.001
    L2_weight: float = 1e-3

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

    debug: bool = False
    random_seed: int = 427 #for ece 427 ofc
    data: DataSettings = DataSettings()
    training: TrainingSettings = TrainingSettings()
    model: ModelSettings = ModelSettings()

    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,
```

```python
        )

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

```
from dataclasses import dataclass, InitVar, field
import numpy as np
import structlog
import tensorflow as tf

log = structlog.get_logger()

@dataclass
class MNIST:
    """generate spiral data"""
    rng: InitVar[np.random.Generator]
    train_ratio: float = 0.8

    # Data arrays (initialized post-construction)
    x_train: np.ndarray = field(init=False)
    x_val: np.ndarray = field(init=False)
    x_test: np.ndarray = field(init=False)
    y_train: np.ndarray = field(init=False)
    y_val: 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 and normalize MNIST data
        (x_train_raw, y_train_raw), (x_test_raw, y_test_raw) = tf.keras.datasets
.mnist.load_data()
        x_train_raw = self.normalize_and_expand(x_train_raw)
        x_test_raw = self.normalize_and_expand(x_test_raw)

        # Shuffle and split training data into train/val
        train_idx, val_idx = self.split_indices(len(y_train_raw), rng)

        self.x_train = x_train_raw[train_idx]
        self.y_train = y_train_raw[train_idx]
        self.x_val = x_train_raw[val_idx]
        self.y_val = y_train_raw[val_idx]
        self.x_test = x_test_raw
        self.y_test = y_test_raw
        self.index = np.arange(len(self.y_train))

    def normalize_and_expand(self, x: np.ndarray) -> np.ndarray:
        return np.expand_dims(x / 255.0, axis=-1)

    def split_indices(self, size: int, rng:np.random.Generator) -> tuple[np.ndar
ray, np.ndarray]:
        indices = np.arange(size)
        rng.shuffle(indices)
        split = int(self.train_ratio * size)
        return indices[:split], indices[split:]

    def get_batch(self, rng:np.random.Generator, batch_size: int) -> tuple[np.nd
array, np.ndarray]:
        """get random batch"""
        choices = rng.choice(self.index, size=batch_size)
        return self.x_train[choices], self.y_train[choices] #x_train has dimensi
ons 28x28x1

    def get_validation(self):
        return self.x_val, self.y_val

    def get_test(self):
        return self.x_test, self.y_test
```

```
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",
        stream=sys.stdout,
    )

    # 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.RichTracebackForm
atter()
            ),
        ],
        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
from flax import nnx
from typing import Union, Tuple, List
import structlog
import math

log = structlog.get_logger()

class Convolution(nnx.Module):
    """A single convolutional layer with dropout and optional L2 regularization."""

    def __init__(
        self,
        *,
        keys,
        L2_weight: float,
        dropout_rate: float,
        in_features: int,
        out_features: int,
        kernel_size: Union[int, Tuple[int, int]],
        strides: Union[int, Tuple[int, int]],
        padding: str = "SAME",
    ):
        self.rngs = nnx.Rngs(params=keys, dropout=keys)
        self.L2_weight = L2_weight
        self.padding = padding

        self.layer = nnx.Conv(
            in_features=in_features,
            out_features=out_features,
            kernel_size=kernel_size,
            strides=strides,
            padding=self.padding,
            rngs=self.rngs,
        )
        self.dropout = nnx.Dropout(rate=dropout_rate, rngs=self.rngs)

    def __call__(self, x: jnp.ndarray, train: bool):
        x = self.layer(x)
        x = self.dropout(x, deterministic=not train)
        return x

    def L2_loss(self) -> jnp.ndarray:
        return jnp.sum(jnp.square(self.layer.kernel.value)) * self.L2_weight


class Classify(nnx.Module):
    """CNN classifier for MNIST."""

    def __init__(
        self,
        *,
        input_depth: int,
        layer_depths: List[int],
        layer_kernel_sizes: List[Tuple[int, int]],
        strides: List[Union[int, Tuple[int, int]]],
        num_classes: int,
        dropout_rate: float,
        L2_weight: float,
        rngs: nnx.Rngs,
        shape: List[int],
    ):
        keys = rngs.params()

        self.input_depth = input_depth
        self.layer_depths = layer_depths
        self.layer_kernel_sizes = layer_kernel_sizes
```

```python
        self.strides = strides
        self.num_classes = num_classes
        self.dropout_rate = dropout_rate
        self.L2_weight = L2_weight

        # Depth of inputs to each layer
        input_features = [input_depth] + layer_depths[:-1]
        layer_keys = jax.random.split(keys, len(layer_depths))

        log.debug(
            "conv layers dimentinos:",
            layer_depthsdepths=layer_depths,
            input_features=input_features,
            layer_kernel_sizes=layer_kernel_sizes,
            strides=strides,
        )

        # Build convolutional layers
        self.layers = [
            Convolution(
                in_features=input_features[i],
                out_features=layer_depths[i],
                kernel_size=layer_kernel_sizes[i],
                strides=strides[i],
                keys=layer_keys[i],
                L2_weight=L2_weight,
                dropout_rate=dropout_rate,
            )
            for i in range(len(layer_depths))
        ]

        #flatten and account for downsampling caused by larger stride
        # normalize strides to a (n, 2) array of ints
        stride_arr = jnp.array(
            [[s, s] if isinstance(s, (int, np.integer)) else s for s in strides]
,
            dtype=int
        )
        # product of strides along each dimension
        stride_prod = jnp.prod(stride_arr, axis=0)   # shape (2,whatever)
        h, w = shape
        h = math.ceil(h / stride_prod[0])
        w = math.ceil(w / stride_prod[1])
        log.debug("h and w:", h = h, w =w)

        flatten_size = int(h * w * layer_depths[-1])

        self.final_layer = nnx.Linear(
            in_features=flatten_size,
            out_features=num_classes,
            rngs=rngs,
        )

    def __call__(self, x: jax.Array, train: bool) -> jax.Array:
        val = x
        for layer in self.layers:
            val = layer(val, train)
            val = jax.nn.leaky_relu(val)

        val = val.reshape((val.shape[0], -1))  # (batch, h*w*d)
        return self.final_layer(val)

    def L2_loss(self) -> jnp.ndarray:
        return sum(layer.L2_loss() for layer in self.layers)
```

```
josh@JoshsGalaxyBook:/mnt/c/Users/joshu/OneDrive/Cooper Union/DL_laptop/hw03$ ju
st run hw03
==> Running homework 'hw03'...
2025-09-24T23:49:07.103642Z [info     ] Settings loaded                [hw03] se
ttings={'debug': False, 'random_seed': 427, 'data': {'train_ratio': 0.8}, 'train
ing': {'batch_size': 128, 'num_iters': 1000, 'learning_rate': 0.001, 'L2_weight'
: 0.001}, 'model': {'input_depth': 1, 'num_classes': 10, 'layer_depth': [32, 64]
, 'layer_kernel_sizes': [[3, 3], [3, 3]], 'strides': [2, 2], 'dropout_rate': 0.1
, 'shape': [28, 28]}}
WARNING:2025-09-24 19:49:07,355:jax._src.xla_bridge:864: An NVIDIA GPU may be pr
esent on this machine, but a CUDA-enabled jaxlib is not installed. Falling back
to cpu.
An NVIDIA GPU may be present on this machine, but a CUDA-enabled jaxlib is not i
nstalled. Falling back to cpu.
2025-09-24T23:49:09.105202Z [info     ] MNIST data retrieved           [hw03] sh
ape=(48000, 28, 28, 1)
2025-09-24T23:49:10.434408Z [info     ] spiral model generated!        [hw03] pa
rams=Classify( # Param: 50,186 (200.7 KB), RngState: 12 (72 B), Total: 50,198 (2
00.8 KB)
  input_depth=1,
  layer_depths=[32, 64],
  layer_kernel_sizes=[[3, 3], [3, 3]],
  strides=[2, 2],
  num_classes=10,
  dropout_rate=0.1,
  L2_weight=0.001,
  layers=[Convolution( # RngState: 6 (36 B), Param: 320 (1.3 KB), Total: 326 (1.
3 KB)
    rngs=Rngs( # RngState: 4 (24 B)
      params=RngStream( # RngState: 2 (12 B)
        tag='params',
        key=RngKey( # 1 (8 B)
          value=Array((), dtype=key<fry>) overlaying:
          [2379901428 2398724654],
          tag='params'
        ),
        count=RngCount( # 1 (4 B)
          value=Array(2, dtype=uint32),
          tag='params'
        )
      ),
      dropout=RngStream( # RngState: 2 (12 B)
        tag='dropout',
        key=RngKey( # 1 (8 B)
          value=Array((), dtype=key<fry>) overlaying:
          [2379901428 2398724654],
          tag='dropout'
        ),
        count=RngCount( # 1 (4 B)
          value=Array(1, dtype=uint32),
          tag='dropout'
        )
      )
    ),
    L2_weight=0.001,
    padding='SAME',
    layer=Conv( # Param: 320 (1.3 KB)
      kernel_shape=(3, 3, 1, 32),
      kernel=Param( # 288 (1.2 KB)
        value=Array(shape=(3, 3, 1, 32), dtype=dtype('float32'))
      ),
      bias=Param( # 32 (128 B)
        value=Array(shape=(32,), dtype=dtype('float32'))
      ),
      in_features=1,
      out_features=32,
      kernel_size=(3, 3),
      strides=2,
      padding='SAME',
```

```
      input_dilation=1,
      kernel_dilation=1,
      feature_group_count=1,
      use_bias=True,
      mask=None,
      dtype=None,
      param_dtype=float32,
      precision=None,
      kernel_init=<function variance_scaling.<locals>.init at 0x7fcc88a86520>,
      bias_init=<function zeros at 0x7fcc89933c40>,
      conv_general_dilated=<function conv_general_dilated at 0x7fcc89fd4fe0>,
      promote_dtype=<function promote_dtype at 0x7fcc88a86840>
    ),
    dropout=Dropout( # RngState: 2 (12 B)
      rate=0.1,
      broadcast_dims=(),
      deterministic=False,
      rng_collection='dropout',
      rngs=RngStream( # RngState: 2 (12 B)
        tag='dropout',
        key=RngKey( # 1 (8 B)
          value=Array((), dtype=key<fry>) overlaying:
          [2716628393 3028672913],
          tag='dropout'
        ),
        count=RngCount( # 1 (4 B)
          value=Array(0, dtype=uint32),
          tag='dropout'
        )
      )
    )
  ), Convolution( # RngState: 6 (36 B), Param: 18,496 (74.0 KB), Total: 18,502 (
74.0 KB)
    rngs=Rngs( # RngState: 4 (24 B)
      params=RngStream( # RngState: 2 (12 B)
        tag='params',
        key=RngKey( # 1 (8 B)
          value=Array((), dtype=key<fry>) overlaying:
          [ 714023341 1551763251],
          tag='params'
        ),
        count=RngCount( # 1 (4 B)
          value=Array(2, dtype=uint32),
          tag='params'
        )
      ),
      dropout=RngStream( # RngState: 2 (12 B)
        tag='dropout',
        key=RngKey( # 1 (8 B)
          value=Array((), dtype=key<fry>) overlaying:
          [ 714023341 1551763251],
          tag='dropout'
        ),
        count=RngCount( # 1 (4 B)
          value=Array(1, dtype=uint32),
          tag='dropout'
        )
      )
    ),
    L2_weight=0.001,
    padding='SAME',
    layer=Conv( # Param: 18,496 (74.0 KB)
      kernel_shape=(3, 3, 32, 64),
      kernel=Param( # 18,432 (73.7 KB)
        value=Array(shape=(3, 3, 32, 64), dtype=dtype('float32'))
      ),
      bias=Param( # 64 (256 B)
        value=Array(shape=(64,), dtype=dtype('float32'))
      ),
      in_features=32,
      out_features=64,
      kernel_size=(3, 3),
      strides=2,
      padding='SAME',
      input_dilation=1,
      kernel_dilation=1,
      feature_group_count=1,
      use_bias=True,
      mask=None,
      dtype=None,
      param_dtype=float32,
      precision=None,
      kernel_init=<function variance_scaling.<locals>.init at 0x7fcc88a86520>,
      bias_init=<function zeros at 0x7fcc89933c40>,
      conv_general_dilated=<function conv_general_dilated at 0x7fcc89fd4fe0>,
      promote_dtype=<function promote_dtype at 0x7fcc88a86840>
    ),
    dropout=Dropout( # RngState: 2 (12 B)
      rate=0.1,
      broadcast_dims=(),
      deterministic=False,
      rng_collection='dropout',
      rngs=RngStream( # RngState: 2 (12 B)
        tag='dropout',
        key=RngKey( # 1 (8 B)
          value=Array((), dtype=key<fry>) overlaying:
          [1708850055 3551380339],
          tag='dropout'
        ),
        count=RngCount( # 1 (4 B)
          value=Array(0, dtype=uint32),
          tag='dropout'
        )
      )
    )
  )],
  final_layer=Linear( # Param: 31,370 (125.5 KB)
    kernel=Param( # 31,360 (125.4 KB)
      value=Array(shape=(3136, 10), dtype=dtype('float32'))
    ),
    bias=Param( # 10 (40 B)
      value=Array(shape=(10,), dtype=dtype('float32'))
    ),
    in_features=3136,
    out_features=10,
    use_bias=True,
    dtype=None,
    param_dtype=float32,
    precision=None,
    kernel_init=<function variance_scaling.<locals>.init at 0x7fcc88a86520>,
    bias_init=<function zeros at 0x7fcc89933c40>,
    dot_general=<function dot_general at 0x7fcc8a15fec0>,
    promote_dtype=<function promote_dtype at 0x7fcc88a86840>
  )
)
2025-09-24T23:49:10.504129Z [info     ] started training               [hw03.tra
ining]
loss at 999 => 0.218302: 100%|âM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^V
M-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^Hâ
M-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^H
âM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^V
M-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^Hâ
M-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^H
âM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^V
M-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^Hâ
M-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^H
âM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^H| 1000/1000 [00:19<00:00, 51.89it/s
]
```

```
2025-09-24T23:49:29.776814Z [info     ] done training                  [hw03.tra
ining]
2025-09-24T23:49:29.777422Z [info     ] training concluded             [hw03]
2025-09-24T23:49:30.673084Z [info     ] Evaluated validation set: 12000 samples,
 11613 correct, Accuracy=96.78% [hw03.test]
2025-09-24T23:49:30.673745Z [info     ] passed validation w/ acc > 0.96 [hw03]
2025-09-24T23:49:31.237287Z [info     ] Evaluated test set: 10000 samples, 9739
correct, Accuracy=97.39% [hw03.test]
2025-09-24T23:49:31.237664Z [info     ] real test accuracy:            [hw03] ac
curacy=0.9739
2025-09-24T23:49:31.237775Z [info     ] yippeeeee                      [hw03]
josh@JoshsGalaxyBook:/mnt/c/Users/joshu/OneDrive/Cooper Union/DL_laptop/hw03$
```

```python
import jax.numpy as jnp
import numpy as np
import structlog
import optax

from .model import Classify
from .data import MNIST

log = structlog.get_logger()

def test_accuracy(
    model: Classify,
    data: MNIST,
    batch_size: int,
    validation: bool = True
) -> float:

    if validation == True:
        x_np, y_np = data.get_validation()
    else:
        x_np, y_np = data.get_test()

    num_samples = x_np.shape[0]

    #batch predictions
    preds = []
    for start in range(0, num_samples, batch_size):
        end = start + batch_size
        xb = jnp.asarray(x_np[start:end])
        logits = model(xb, train=False)
        batch_pred = jnp.argmax(logits, axis=-1)
        preds.append(batch_pred)

    #compute accuracy
    all_preds = jnp.concatenate(preds, axis=0)
    all_labels = jnp.asarray(y_np)
    correct = jnp.sum(all_preds == all_labels)
    accuracy = correct / num_samples

    log.info(
        "Evaluated %s set: %d samples, %d correct, Accuracy=%.2f%%",
        "validation" if validation else "test",
        int(num_samples),
        int(correct),
        float(accuracy * 100),
    )

    return float(accuracy)
```

```
import jax.numpy as jnp
import numpy as np
import optax
import structlog

from flax import nnx
from tqdm import trange
from .config import TrainingSettings
from .data import MNIST
from .model import Classify

log = structlog.get_logger()

@nnx.jit
def train_step_mnist(
    model: Classify,
    optimizer: nnx.Optimizer,
    x: jnp.ndarray,
    y: jnp.ndarray,
 ):

    """perform on training step"""

    def loss_function(model: Classify):
        logits = model(x, True)
        ce_loss = jnp.mean(optax.softmax_cross_entropy_with_integer_labels(logit
s, y))
        L2_loss = model.L2_loss()

        return ce_loss + L2_loss

    loss, grads = nnx.value_and_grad(loss_function)(model)
    optimizer.update(model, grads)
    return loss

def MNIST_train(
    model: Classify,
    optimizer: nnx.Optimizer,
    data: MNIST,
    settings: TrainingSettings,
    np_rng: np.random.Generator,
 ) -> None:
    """train with grad descend"""
    log.info("started training")
    bar = trange(settings.num_iters)
    for i in bar:
        x_np, y_np = data.get_batch(np_rng, settings.batch_size)
        log.debug("y_np", y_np=y_np)
        x, y = jnp.asarray(x_np), jnp.asarray(y_np)
        log.debug("y here", y=y)
        # x is (batch_size, 28, 28, 1)
        loss = train_step_mnist(model, optimizer, x, y)
        bar.set_description(f"loss at {i} => {loss:.6f}")
        bar.refresh()
    log.info("done training")
```

```
import jax
import numpy as np
import structlog
import optax

from flax import nnx

from .config import load_settings
from .data import MNIST
from .model import Classify
from .logging import configure_logging
from .training import MNIST_train
from .test import test_accuracy

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(472) #472 for ECE 472
    data_key, model_key = jax.random.split(key)
    np_rng = np.random.default_rng(np.array(data_key))

    data = MNIST(
        rng = np_rng,
        train_ratio = settings.data.train_ratio,
    )
    log.info("MNIST data retrieved", shape = data.x_train.shape)

    model = Classify(
        rngs = nnx.Rngs(params=model_key),
        num_classes = settings.model.num_classes,
        shape = settings.model.shape,
        dropout_rate = settings.model.dropout_rate,
        input_depth = settings.model.input_depth,
        layer_depths = settings.model.layer_depth,
        layer_kernel_sizes = settings.model.layer_kernel_sizes,
        strides = settings.model.strides,
        L2_weight = settings.training.L2_weight,
    )

    log.info("spiral model generated!", params = model)

    optimizer_schedule = optax.cosine_decay_schedule(settings.training.learning_
rate, settings.training.num_iters,) #same schedule as you
    optimizer = nnx.Optimizer(
        model, optax.adam(optimizer_schedule), wrt=nnx.Param
    )

    MNIST_train(model, optimizer, data, settings.training, np_rng)
    log.info("training concluded")

    validation_accuracy = test_accuracy(
        data = data,
        batch_size = settings.training.batch_size,
        model = model,
        validation = True,
    )

    if validation_accuracy >= 0.96:
        log.info("passed validation w/ acc > 0.96")
        real_test_accuracy = test_accuracy(
            data = data,
            batch_size = settings.training.batch_size,
            model = model,
            validation = False,
```

```
        )
        log.info("real test accuracy:", accuracy = real_test_accuracy)
        if real_test_accuracy >= 0.955:
            log.info("yippeeeee")
        else:
            log.info("you failed the test bruh")
    else:
        log.info("failed validation noooooooooooooooooooooo")
```