```
# pyproject.toml.jinja

[project]
name = "hw02"
version = "0.1.0"
description = "Binary classification on the spirals dataset using a multi-layer
perceptron"
readme = "README.md"
authors = [
    { name = "Donghyun Park", email = "donghyun.park@cooper.edu" }
]
requires-python = ">=3.13"
dependencies = [
    "structlog",
    "rich",
    "numpy",
    "jax",
    "flax",
    "tqdm>=4.67.1",
    "matplotlib>=3.10.6",
    "pydantic>=2.11.9",
    "pydantic-settings>=2.10.1",
    "scikit-learn>=1.7.2",
]

[project.scripts]
hw02 = "hw02:main"

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

```python
import matplotlib
import matplotlib.pyplot as plt
import jax
import jax.numpy as jnp
import numpy as np
import structlog
from sklearn.inspection import DecisionBoundaryDisplay

from .config import PlottingSettings
from .data import Data
from .model import NNXMLPModel

log = structlog.get_logger()

font = {
    # "family": "Adobe Caslon Pro",
    "size": 10,
}

matplotlib.style.use("classic")
matplotlib.rc("font", **font)


def plot_fit(
    model: NNXMLPModel,
    data: Data,
    settings: PlottingSettings,

):
    """Plots the graph and saves it to a file."""
    log.info("Plotting Graph")
    fig, ax = plt.subplots(figsize=settings.figsize, dpi=settings.dpi)

    x_min, x_max = data.spiral_coordinates[:, 0].min() - 1, data.spiral_coordina
tes[:, 0].max() + 1
    y_min, y_max = data.spiral_coordinates[:, 1].min() - 1, data.spiral_coordina
tes[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1), np.arange(y_min, y_max, 0
.1))

    z = model(jnp.column_stack([xx.ravel(), yy.ravel()]))
    predicts = (jax.nn.sigmoid(z) > 0.5).astype(int)
    predicts = predicts.reshape(xx.shape)

    display = DecisionBoundaryDisplay(xx0=xx, xx1=yy, response=predicts)
    display.plot(ax=ax, cmap=plt.cm.RdBu, alpha=0.5)

    ax.scatter(data.spiral_coordinates[:, 0], data.spiral_coordinates[:, 1], c=d
ata.spiral_labels, cmap=plt.cm.RdBu, edgecolors='k', s=100)

    ax.set_title("Decision Boundary of Spirals")
    ax.set_xlabel("x")
    ax.set_ylabel("y", labelpad=10).set_rotation(0)

    plt.tight_layout()

    settings.output_dir.mkdir(parents=True, exist_ok=True)
    output_path = settings.output_dir / "fit.pdf"
    plt.savefig(output_path)
    log.info("Saved plot", path=str(output_path))
```

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

from flax import nnx
from pydantic import BaseModel
from pydantic_settings import BaseSettings


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

    num_samples_per_spiral: int = 200
    sigma_noise: float = 0.1

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

    num_inputs: int = 2
    num_outputs: int = 1
    num_hidden_layers: int = 4
    hidden_layer_width: int = 128


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

    batch_size: int = 64
    num_iters: int = 20000
    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("hw02/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()


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

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

from .logging import configure_logging
from .config import load_settings
from .data import Data
from .model import NNXMLPModel
from .training import train
from .plotting import plot_fit

"""
Discussion:
various configurations for number of hidden layers and width of hidden layers were tried
moderate number of hidden layers (4–6) with width of 128 seemed to work well
going either too high or too low caused under/over fitting
optimizer with decay was also added to make sure the model was learning correctly

submission was late due to listeria infection
"""

def main() -> None:
    """CLI entry point."""
    settings = load_settings()
    configure_logging()
    log = structlog.get_logger()
    log.info("Hello from hw02!")
    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)
    log.debug("keys", key=key, data_key=data_key, model_key=model_key)
    np_rng = np.random.default_rng(np.array(data_key))

    data = Data(
        rng=np_rng,
        num_samples_per_spiral=settings.data.num_samples_per_spiral,
        sigma=settings.data.sigma_noise,
    )
    # log.debug("Data generated", x=data.x, y=data.y)

    model = NNXMLPModel(
        key = model_key,
        num_inputs=settings.model.num_inputs,
        num_outputs=settings.model.num_outputs,
        num_hidden_layers=settings.model.num_hidden_layers,
        hidden_layer_width=settings.model.hidden_layer_width,
        hidden_activation=nnx.relu,
        output_activation=nnx.identity,
    )

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



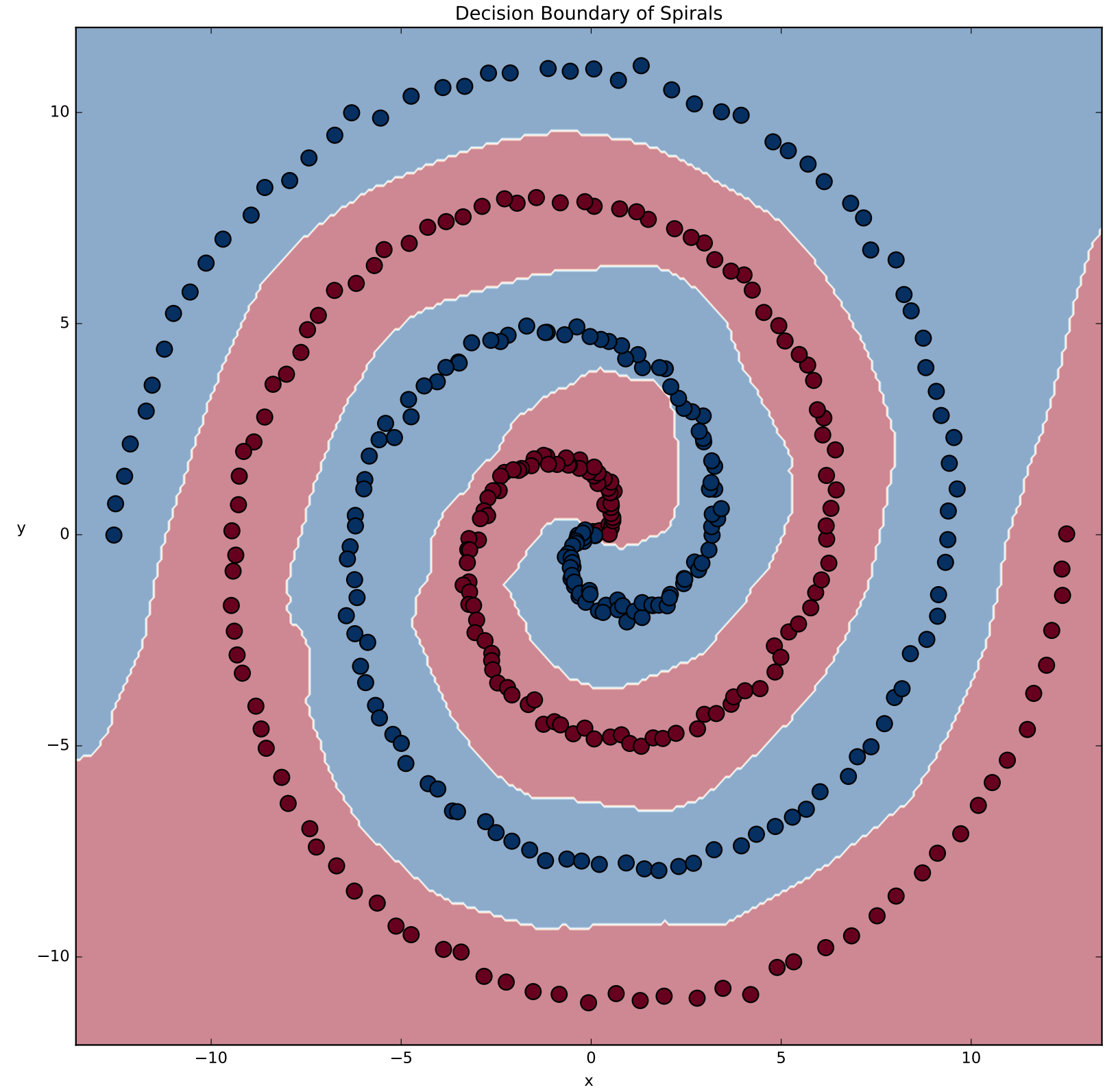
```python
    train(model, optimizer, data, settings.training, np_rng)

    plot_fit(model, data, settings.plotting)
```

```
import jax
import structlog
from flax import nnx


log = structlog.get_logger()


class NNXMLPModel(nnx.Module):
    """A Flax NNX module for an MLP model."""

    def __init__(self, *,
                 key: jax.random.PRNGKey,
                 num_inputs: int,
                 num_outputs: int,
                 num_hidden_layers: int,
                 hidden_layer_width: int,
                 hidden_activation = nnx.identity,
                 output_activation = nnx.identity):
        keys = jax.random.split(key, num_hidden_layers + 2)
        log.debug("RNG keys generated", original_key=key, keys=keys)
        input_rngs = nnx.Rngs(params=keys[0])
        output_rngs = nnx.Rngs(params=keys[-1])
        hidden_keys = keys[1:-1]

        self.input_layer = nnx.Linear(num_inputs, hidden_layer_width, rngs=input
_rngs)
        self.output_layer = nnx.Linear(hidden_layer_width, num_outputs, rngs=out
put_rngs)
        self.hidden_layers = []
        if num_hidden_layers > 0:
            def create_hidden_layer(key: jax.random.PRNGKey) -> nnx.Linear:
                hidden_rngs = nnx.Rngs(params=key)
                return nnx.Linear(hidden_layer_width, hidden_layer_width, rngs=h
idden_rngs)

            vmap_create_layer = jax.vmap(create_hidden_layer)
            self.hidden_layers = vmap_create_layer(hidden_keys)

        self.hidden_activation = hidden_activation
        self.output_activation = output_activation

    def __call__(self, coords: jax.Array) -> jax.Array:
        # """Iterates through the MLP layers."""
        coords = self.hidden_activation(self.input_layer(coords))

        def forward(carry, hidden_layer):
            output_coords = self.hidden_activation(hidden_layer(carry))
            return output_coords, None

        coords, _ = jax.lax.scan(f=forward,init=coords,xs=self.hidden_layers)

        return self.output_activation(self.output_layer(coords))
```

```python
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 NNXMLPModel

log = structlog.get_logger()


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

    def loss_fn(model: NNXMLPModel):
        y_hat = model(x)
        return optax.sigmoid_binary_cross_entropy(y_hat, y.reshape(y_hat.shape))
.mean()

    loss, grads = nnx.value_and_grad(loss_fn)(model)
    optimizer.update(model, grads)  # In-place update of model parameters
    return loss


def train(
    model: NNXMLPModel,
    optimizer: nnx.Optimizer,
    data: Data,
    settings: TrainingSettings,
    np_rng: np.random.Generator,
) -> None:
    """Train the model using SGD."""
    log.info("Starting training", **settings.model_dump())
    bar = trange(settings.num_iters)
    for i in bar:
        coords_np, labels_np = data.get_batch(np_rng, settings.batch_size)
        coords, labels = jnp.asarray(coords_np), jnp.asarray(labels_np)
        # log.debug("Y value", y=y)

        loss = train_step(model, optimizer, coords, labels)
        # log.debug("Training step", step=i, loss=loss)
        bar.set_description(f"Loss @ {i} => {loss:.6f}")
        bar.refresh()
    log.info("Training finished")
```

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

    # Set the level for the application's logger
    log_level = os.environ.get("LOG_LEVEL", "INFO").upper()
    logging.getLogger("hw02").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 numpy as np


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

    rng: InitVar[np.random.Generator]
    num_samples_per_spiral: int
    sigma: float
    x0: np.ndarray = field(init=False)
    y0: np.ndarray = field(init=False)
    x1: np.ndarray = field(init=False)
    y1: np.ndarray = field(init=False)
    index: np.ndarray = field(init=False)
    data: np.ndarray = field(init=False)
    spiral_coordinates: np.ndarray = field(init=False)
    spiral_labels: np.ndarray = field(init=False)

    def __post_init__(self, rng: np.random.Generator):
        """Generate two spiral data."""
        self.index = np.arange(self.num_samples_per_spiral*2)

        spiral = np.linspace(0, 4 * np.pi, self.num_samples_per_spiral) # 2 2pi
rotations

        # spiral * np.foo(spiral) makes the spiral radially grow outwards
        self.x0 = spiral * np.cos(spiral) + rng.normal(0, self.sigma, size=(self
.num_samples_per_spiral))
        self.y0 = spiral * np.sin(spiral) + rng.normal(0, self.sigma, size=(self
.num_samples_per_spiral))
        self.labels0 = np.zeros((self.num_samples_per_spiral,1))
        self.x1 = -spiral * np.cos(spiral) + rng.normal(0, self.sigma, size=(sel
f.num_samples_per_spiral))
        self.y1 = -spiral * np.sin(spiral) + rng.normal(0, self.sigma, size=(sel
f.num_samples_per_spiral))
        self.labels1 = np.ones((self.num_samples_per_spiral,1))

        spirals = np.vstack([
            np.stack([self.x0, self.y0], axis=1),
            np.stack([self.x1, self.y1], axis=1)
        ])
        labels = np.vstack([self.labels0, self.labels1])
        self.data = np.hstack([spirals, labels])

        self.spiral_coordinates = self.data[:,:2]
        self.spiral_labels = self.data[:,2]

    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.spiral_coordinates[choices], self.spiral_labels[choices]
```


Decision Boundary of Spirals
10
5
0
−5
−10
y
−10
−5
0
5
10
x