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

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


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

    num_features: int = 1
    num_basis_ftn: int = 6
    num_samples: int = 50
    sigma_noise: float = 0.1


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

    batch_size: int = 20
    num_iters: int = 300
    learning_rate: float = 0.1


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

    figsize: Tuple[int, int] = (5, 5)
    dpi: int = 200
    output_dir: Path = Path("artifacts")


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

    input_dim: int = 2
    output_dim: int = 1
    hidden_layer_width: int = 64
    num_hidden_layers: int = 2


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()

    model_config = SettingsConfigDict(
        toml_file=files("hw02").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 numpy as np
import optax
import structlog
from flax import nnx

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


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))

    data = Data(
        model=None,
        rng=np_rng,
        num_features=settings.data.num_features,
        sigma_noise=settings.data.sigma_noise,
        num_samples=settings.data.num_samples,
    )
    log.debug("Generating data points", model=data)

    # create the MLP model using NNX
    rngs = nnx.Rngs(settings.random_seed)
    model = NNXMLPModel(
        rngs=rngs,
        input_dim=int(settings.model.input_dim),
        output_dim=int(settings.model.output_dim),
        hidden_layer_width=int(settings.model.hidden_layer_width),
        num_hidden_layers=int(settings.model.num_hidden_layers),
        hidden_activation=jax.nn.relu,
        output_activation=jax.nn.sigmoid,
    )
    log.debug("Initial model created", model=model)

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

    train(model, optimizer, data, settings.training, np_rng)
    log.info("finish training")

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

```
debug = false
random_seed = 0xc61dd8fc

[data]
num_features = 2
num_samples = 500
sigma_noise = 0.08

[model]
input_dim = 2
output_dim = 1
hidden_layer_width = 128
num_hidden_layers = 5

[training]
batch_size = 256
num_iters = 800
learning_rate = 0.001

[plotting]
output_dir = "artifacts"
figsize = [5, 5]
dpi = 200
```

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

log = structlog.get_logger()


class Block(nnx.Module):
    """
  Compute one instance of hidden layer, linear and activation.
  maps (batch, hidden_dim) to (batch, hidden_dim)
  """

    def __init__(
        self,
        rngs: nnx.Rngs,
        hidden_layer_width: int,
        hidden_activation=jax.nn.relu,
    ):
        # create linear mapping
        self.linear = nnx.Linear(hidden_layer_width, hidden_layer_width, rngs=rn
gs)
        self.hidden_activation = hidden_activation

    # runs the forward pass
    def __call__(self, x: jax.Array) -> jax.Array:
        # x@w + b, x is now (batch, hidden layer width)
        h = self.linear(x)
        return self.hidden_activation(h)


class NNXMLPModel(nnx.Module):
    """
  input linear (input_dim) to (hidden_layer_width)
  Block modules (hidden_layer_width) to (hidden_layer_width)
  output linear (hidden_layer_width) to (output_dim)
  """

    def __init__(
        self,
        *,
        rngs: nnx.Rngs,
        input_dim: int,
        output_dim: int,
        hidden_layer_width: int,
        num_hidden_layers: int,
        hidden_activation=jax.nn.relu,
        output_activation=jax.nn.sigmoid,
    ):
        self.input_dim = int(input_dim)
        self.output_dim = int(output_dim)
        self.hidden_layer_width = int(hidden_layer_width)
        self.num_hidden_layers = int(num_hidden_layers)
        self.hidden_activation = hidden_activation
        self.output_activation = output_activation

        # input layer
        self.input_linear = nnx.Linear(
            self.input_dim, self.hidden_layer_width, rngs=rngs
        )

        """hidden layers, tried to split the RNGs by indexing rngs_split but run into
    error as using split on RngStream object. Couldn't figure out how to use @nnx.vmap
    for my written function... So used a list and loop instead.
    """
        self.hidden_layers = []
        for i in range(self.num_hidden_layers):
            # new Rngs object for each hidden layer
```



```
            layer_rngs = nnx.Rngs(rngs.params())
            self.hidden_layers.append(
                Block(
                    rngs=layer_rngs,
                    hidden_layer_width=self.hidden_layer_width,
                    hidden_activation=hidden_activation,
                )
            )

        # output layer
        self.output_linear = nnx.Linear(
            self.hidden_layer_width, self.output_dim, rngs=rngs
        )

    def __call__(self, x: jax.Array) -> jax.Array:
        """
    pass x, size (batch, input_dim)
    returns (batch,) if output layer, else (batch, 1)
    """
        # input layer, (batch, hidden_layer_width)
        h = self.input_linear(x)
        h = self.hidden_activation(h)

        # hidden layers
        for layer in self.hidden_layers:
            h = layer(h)

        # (batch, output_dim)
        logits = self.output_linear(h)
        logits = self.output_activation(logits)

        if logits.shape[-1] == 1:
            return jnp.ravel(logits)
        return logits
```

```python
import jax.numpy as jnp
import numpy as np
import structlog
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 0.5 * jnp.mean((y_hat - y) ** 2)

    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:
        x_np, y_np = data.get_batch(np_rng, settings.batch_size)
        x, y = jnp.asarray(x_np), jnp.asarray(y_np)

        loss = train_step(model, optimizer, x, y)

        bar.set_description(f"Loss @ {i} => {loss:.6f}")
        bar.refresh()
    log.info("Training finished")
```

```python
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import structlog


from .config import PlottingSettings
from .data import Data
from .model import NNXMLPModel
from sklearn.inspection import DecisionBoundaryDisplay

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 synthetic spiral data with decision boundary."""
    log.info("Spiral Data with decision boundary.")
    fig, ax = plt.subplots(1, 1, figsize=settings.figsize, dpi=settings.dpi)

    labels = data.y.flatten()
    # plot data points
    scatter = ax.scatter(
        data.x[:, 0], data.x[:, 1], c=labels, cmap="summer", edgecolor="k"
    )
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_title("MLP Decision Boundary on Spiral Dataset")

    # mesh grid for decision boundary, add padding
    x_min, x_max = data.x[:, 0].min(), data.x[:, 0].max()
    y_min, y_max = data.x[:, 1].min(), data.x[:, 1].max()
    pad_x = (x_max - x_min) * 0.15
    pad_y = (y_max - y_min) * 0.15
    x_min -= pad_x
    x_max += pad_x
    y_min -= pad_y
    y_max += pad_y
    ax.set_xlim(-10, 10)
    ax.set_ylim(-10, 10)

    grid_res = 1000
    xx, yy = np.meshgrid(
        np.linspace(x_min, x_max, grid_res), np.linspace(y_min, y_max, grid_res)
    )

    # Plot decision boundary
    grid = np.vstack([xx.ravel(), yy.ravel()]).T
    probs = model(grid)
    probs = probs.ravel().reshape(xx.shape)
    probs = (probs > 0.5).astype(int)
    display = DecisionBoundaryDisplay(xx0=xx, xx1=yy, response=probs)
    display.plot(ax=ax, cmap="RdYlGn", alpha=0.35)

    plt.tight_layout()

    settings.output_dir.mkdir(parents=True, exist_ok=True)
    output_path = settings.output_dir / "spiral.pdf"
```

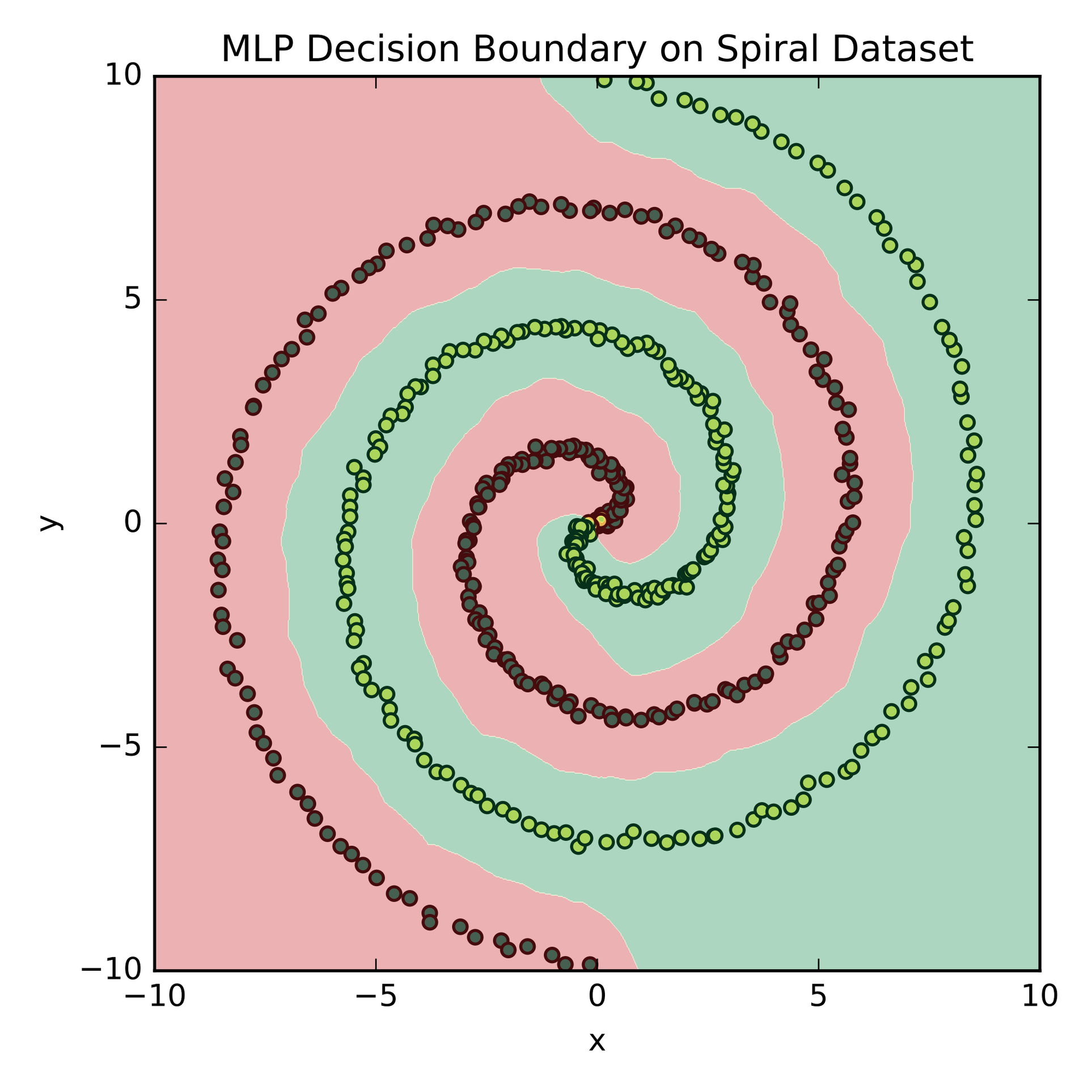


```python
    plt.savefig(output_path)
    log.info("Saved plot", path=str(output_path))
```

```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 spiral synthetic data for MLP."""

    model: object
    rng: InitVar[np.random.Generator]
    num_features: int
    num_samples: int
    sigma_noise: float
    x: np.ndarray = field(init=False)
    y: np.ndarray = field(init=False)
    index: np.ndarray = field(init=False)

    def __post_init__(self, rng: np.random.Generator):
        """Generate synthetic data(spiral) based on the model.
    Uses Archimedean Spiral Formula and Desmos for formula
    r=a+b*theta and r=a-b*theta, a=0, b=0.9, theta=[0, 3.5*pi].
    Generates data in polar coordinate, then converted to cartesian.
    """
        self.index = np.arange(self.num_samples)
        a = 0
        b = 0.9
        theta = np.linspace(0, 7 * np.pi, self.num_samples)

        r1 = a + b * theta
        r2 = a - b * theta

        # Split samples into 2 arms of the spiral
        half_data = self.num_samples // 2
        clean_r = np.concatenate([r1[:half_data], r2[: (self.num_samples - half_
data)]])
        clean_theta = np.concatenate(
            [theta[:half_data], theta[: (self.num_samples - half_data)]]
        )

        # cartesian and [N, 2] data points
        x_clean = clean_r * np.cos(clean_theta)
        y_clean = clean_r * np.sin(clean_theta)

        self.x = np.stack([x_clean, y_clean], axis=1)
        self.x += rng.normal(loc=0.0, scale=float(self.sigma_noise), size=self.x
.shape)

        # label into group 0 and 1
        self.y = np.concatenate(
            [np.zeros(half_data), np.ones(self.num_samples - half_data)]
        )

    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[choices], self.y[choices].flatten()
```

```
# pyproject.toml.jinja

[project]
name = "hw02"
version = "0.1.0"
description = "Perform binary classification on spirals dataset using multi-laye
r perceptron."
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",
    "scikit-learn>=1.0",
]

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

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


MLP Decision Boundary on Spiral Dataset
10
5
0
−5
−10
y
−10
−5
0
5
10
x