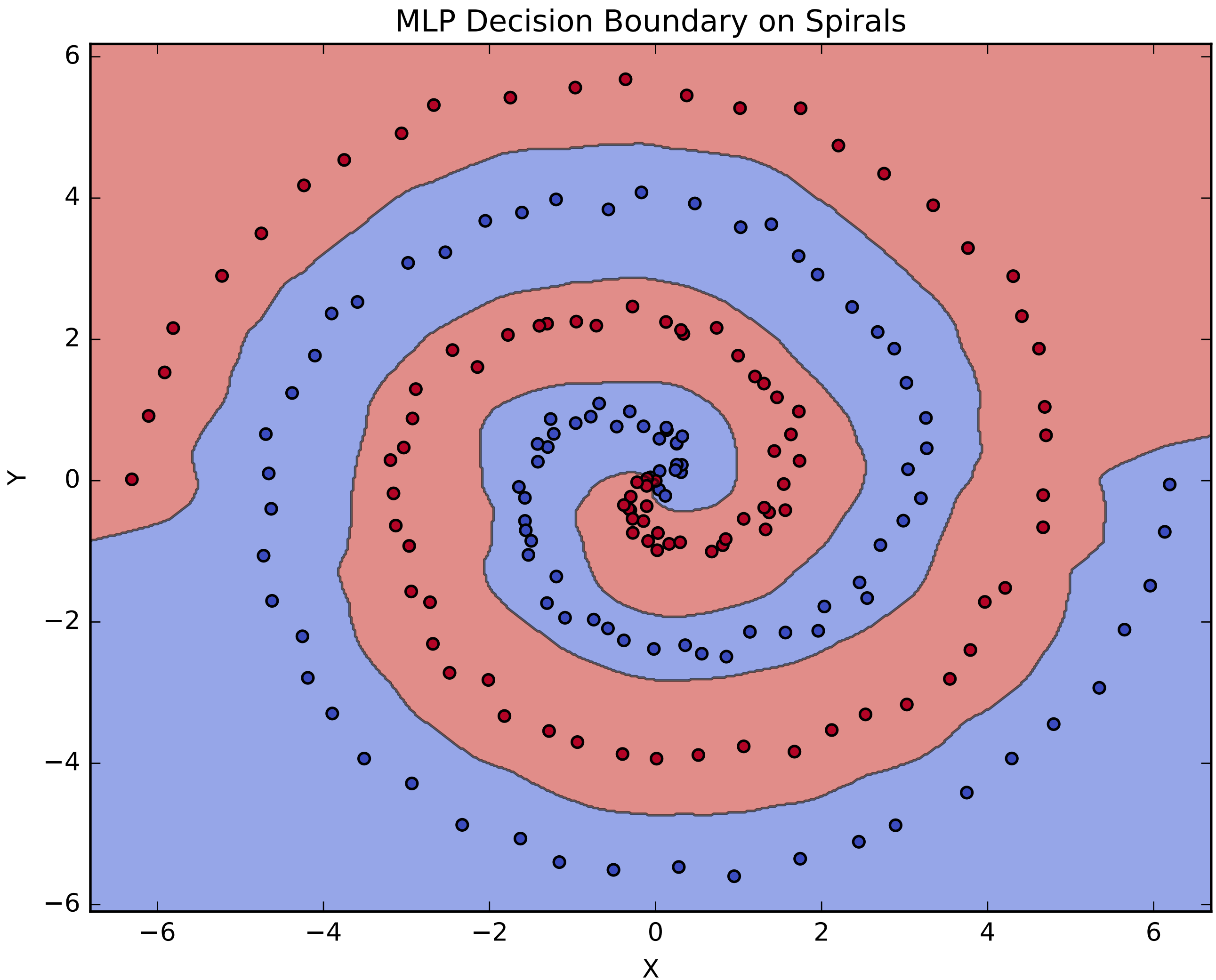
MLP Decision Boundary on Spirals
X
Y
−6
−4
−2
0
2
4
6
−6
−4
−2
0
2
4
6

```
# pyproject.toml.jinja

[project]
name = "hw02"
version = "0.1.0"
description = "spiral with MLP"
readme = "README.md"
authors = [
    { name = "wongee (freddy) hong", email = "wongee.hong@cooper.edu" }
]
requires-python = ">=3.13"
dependencies = [
    "structlog",
    "numpy",
    "pydantic-settings>=2.10.1",
    "matplotlib>=3.10.5",
    "tqdm>=4.67.1",
    "jax>=0.4.25",
    "jaxlib>=0.4.25",
    "flax>=0.11.2",
    "optax>=0.2.4",
    "scikit-learn>=1.5.0",
]


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

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

```
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.RichTracebackForm
atter()
            ),
        ],
        context_class=dict,
        logger_factory=structlog.stdlib.LoggerFactory(),
        wrapper_class=structlog.stdlib.BoundLogger,
        cache_logger_on_first_use=True,
    )
```

```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 ModelSettings(BaseModel):
    """Settings for the MLP model."""

    num_input: int = 2
    num_output: int = 1
    hidden_layer_width: int = 1024
    num_hidden_layers: int = 3


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

    n_points: int = 100
    n_laps: int = 2
    noise: float = 0.1


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

    batch_size: int = 700
    num_iters: int = 2000
    learning_rate: float = 0.001


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

    figsize: Tuple[int, int] = (5, 3)
    dpi: int = 200
    output_dir: Path = Path("artifacts")


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

    debug: bool = False
    random_seed: int = 31415
    model: ModelSettings = ModelSettings()
    data: DataSettings = DataSettings()
    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,
```



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

from .config import PlottingSettings
from .data import SpiralData
from .model import NNXMLP

log = structlog.get_logger()

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

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


def plot_spiral(model: NNXMLP, data: SpiralData, settings: PlottingSettings):
    X, y = data.x, data.y

    # Creating a mesh grid to plot on
    x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
    y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
    xx, yy = np.meshgrid(
        np.linspace(x_min, x_max, 600),
        np.linspace(y_min, y_max, 600),
    )

    # Get model predictions
    grid_points = np.c_[xx.ravel(), yy.ravel()]

    # Get the raw logit output
    logits = model(grid_points)

    # Changing to probability with sigmoid
    preds = (jax.nn.sigmoid(logits) > 0.5).astype(int)
    response = np.array(preds).reshape(xx.shape)

    disp = DecisionBoundaryDisplay(
        xx0=xx,
        xx1=yy,
        response=response,
    )
    disp.plot(ax=plt.gca(), cmap=plt.cm.coolwarm, alpha=0.6)

    plt.contour(xx, yy, response, levels=[0.5], colors="k", linewidths=1, alpha=
0.6)

    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.coolwarm, edgecolors="k")

    plt.title("MLP Decision Boundary on Spirals")
    plt.xlabel("X")
    plt.ylabel("Y")

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

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

from .config import load_settings
from .data import SpiralData
from .logging import configure_logging
from .model import NNXMLP
from .plotting import plot_spiral
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())

    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 = SpiralData(
        rng=np_rng,
        n_points=settings.data.n_points,
        n_laps=settings.data.n_laps,
        noise=settings.data.noise,
    )

    model = NNXMLP(
        rngs=nnx.Rngs(params=model_key),
        num_input=settings.model.num_input,
        num_output=settings.model.num_output,
        hidden_layer_width=settings.model.hidden_layer_width,
        num_hidden_layers=settings.model.num_hidden_layers,
    )

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

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

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

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


"""
    First, I just applied same weight for all linear model. While the shape of the result was spiral, it was very noisy and s
piky.
    Changing loss function from MSE to BCE also helped softening the result.
    After doing some searching, I figured out that it is useful to apply Kaiming He initialization with ReLu activation.
    After applying Kaiming He initialization, I could see some improvements.
    Then from here, to soften the spiral, I changed the hyperparameters such as learning rate, hidden layer width, depth a
nd batch size.
    After few trials, following configuration gave me the best result.
    I also tried to apply learning weight schedular (linear and exponential decay) but I didn't see any big difference.
"""

class NNXLinearModel(nnx.Module):
    """Flax Linear Regression Model"""

    def __init__(self, *, rngs: nnx.Rngs, num_input: int, num_output: int):
        self.num_input = num_input
        self.num_output = num_output
        key = rngs.params()
        # Applying Kaiming He initialization
        stddev = jnp.sqrt(2.0 / self.num_input)
        self.weights = nnx.Param(
            jax.random.normal(key, (self.num_input, num_output)) * stddev
        )
        self.bias = nnx.Param(jnp.zeros((1, num_output)))

    def __call__(self, x: jax.Array) -> jax.Array:
        """Predicts the output for a given input."""
        return x @ self.weights.value + self.bias.value


class NNXMLP(nnx.Module):
    """A Flax NNX module for a MLP model"""

    def __init__(
        self,
        *,
        rngs: nnx.Rngs,
        num_input: int,
        num_output: int,
        hidden_layer_width: int,
        num_hidden_layers: int,
        hidden_activation=nnx.relu,
        output_activation=nnx.identity,
    ):
        @nnx.split_rngs(splits=num_hidden_layers + 2, only="params")
        def _split(rngs: nnx.Rngs):
            @nnx.vmap(in_axes=0, out_axes=0)
            def _one(r: nnx.Rngs):
                return r.params()

            return _one(rngs)

        keys = _split(rngs)
        self.num_input = num_input
        self.num_output = num_output
        self.hidden_layer_width = hidden_layer_width
        self.num_hidden_layers = num_hidden_layers

        self.hidden_activation = hidden_activation
        self.output_activation = output_activation

        self.input_layer = NNXLinearModel(
            rngs=nnx.Rngs(keys[0]), num_input=num_input, num_output=hidden_layer
```



```python
_width
        )

        @nnx.vmap(in_axes=0, out_axes=0)
        def make_hidden(key):
            return NNXLinearModel(
                rngs=nnx.Rngs(params=key),
                num_input=hidden_layer_width,
                num_output=hidden_layer_width,
            )

        hidden_models = make_hidden(keys[1:-1])

        @nnx.scan(in_axes=(0, nnx.Carry), out_axes=nnx.Carry)
        def apply_hidden(layer: NNXLinearModel, x):
            return self.hidden_activation(layer(x))

        self.hidden_layers = hidden_models
        self.apply_hidden = apply_hidden

        self.output_layer = NNXLinearModel(
            rngs=nnx.Rngs(keys[-1]), num_input=hidden_layer_width, num_output=nu
m_output
        )

    def __call__(self, x: jax.Array) -> jax.Array:
        x = self.hidden_activation(self.input_layer(x))
        x = self.apply_hidden(self.hidden_layers, x)
        x = self.output_activation(self.output_layer(x))
        return x
```

```toml
debug = false
random_seed = 31415

[model]
num_input = 2
num_output = 1
hidden_layer_width = 1024
num_hidden_layers = 3

[data]
n_points = 100
n_laps = 2
noise = 0.1

[training]
batch_size = 700
num_iters = 2000
learning_rate = 0.001

[plotting]
output_dir = "artifacts"
figsize = [5, 3]
dpi = 200
```

```python
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 SpiralData
from .model import NNXMLP

log = structlog.get_logger()

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

    def loss_fn(m: NNXMLP):
        logits = m(x)
        yb = jnp.asarray(y, jnp.float32).reshape(-1, 1)
        bce = optax.sigmoid_binary_cross_entropy(logits, yb).mean()
        return bce

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

def train(
    model: NNXMLP,
    optimizer: nnx.Optimizer,
    data: SpiralData,
    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)
        if i % 10 == 0:
            log.info(f"Training Loss @ {i}: {loss:.6f}")
        bar.set_description(f"Loss @ {i} => {loss:.6f}")
        bar.refresh()
    log.info("Training finished")
```

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

import numpy as np


@dataclass
class SpiralData:
    """Data generation"""

    rng: InitVar[np.random.Generator]
    n_points: int
    n_laps: int
    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):
        self.index = np.arange(2 * self.n_points)
        theta = np.linspace(0, self.n_laps * 2 * np.pi, self.n_points)
        r = theta / self.n_laps  # just setting it proportional to theta
        x1 = r * np.cos(theta)
        y1 = r * np.sin(theta)

        x2 = r * np.cos(theta + np.pi)
        y2 = r * np.sin(theta + np.pi)

        X = np.vstack(
            [
                np.stack([x1, y1], axis=1),
                np.stack([x2, y2], axis=1),
            ]
        )

        epsilon = rng.normal(0, self.noise, size=X.shape)

        X += epsilon

        y = np.concatenate(
            [
                np.zeros(self.n_points, dtype=int),
                np.ones(self.n_points, dtype=int),
            ]
        )

        self.x = X
        self.y = y

    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]
```