```
# pyproject.toml.jinja

[project]
name = "hw07"
version = "0.1.0"
description = "Induce sparsity in an MLP classifier"
readme = "README.md"
authors = [
    { name = "My name", email = "donghyun.park@cooper.edu" }
]
requires-python = ">=3.13"
dependencies = [
    "structlog",
    "rich",
    "numpy",
    "jax",
    "flax",
    "tqdm>=4.67.1",
    "matplotlib>=3.10.7",
    "pydantic>=2.12.4",
    "pydantic-settings>=2.12.0",
    "scikit-learn>=1.7.2",
]

[project.scripts]
hw07 = "hw07: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 tqdm import tqdm

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

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_coordinates[:, 0].max() + 1,
    )
    y_min, y_max = (
        data.spiral_coordinates[:, 1].min() - 1,
        data.spiral_coordinates[:, 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=data.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)
```



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


def plot_latent_features(
    mlp: NNXMLPModel,
    sae: SparseAutoEncoder,
    data: Data,
    settings: PlottingSettings,
    num_features_to_plot: int = 9,
):
    log.info("Plotting Sparse Latent Features")

    x_min, x_max = (
        data.spiral_coordinates[:, 0].min() - 1,
        data.spiral_coordinates[:, 0].max() + 1,
    )
    y_min, y_max = (
        data.spiral_coordinates[:, 1].min() - 1,
        data.spiral_coordinates[:, 1].max() + 1,
    )
    mesh_step = 0.05
    xx, yy = np.meshgrid(
        np.arange(x_min, x_max, mesh_step), np.arange(y_min, y_max, mesh_step)
    )
    coords = jnp.column_stack([xx.ravel(), yy.ravel()])

    dense_hidden_state_z = mlp.extract_final_hidden_state(coords)

    latent_activations = sae.encode(dense_hidden_state_z)
    mean_activations = jnp.mean(latent_activations, axis=0)
    top_indices = jnp.argsort(mean_activations)[::-1][:num_features_to_plot]

    if jnp.all(mean_activations == 0):
        log.warning(
            "All latent features have zero mean activation. No features to plot."
        )
        return

    nrows_cols = int(np.ceil(np.sqrt(num_features_to_plot)))
    fig, axes = plt.subplots(
        nrows=nrows_cols, ncols=nrows_cols, figsize=(12, 12), dpi=settings.dpi
    )
    axes = axes.flatten()

    for i, ax in enumerate(tqdm(axes, desc="Generating feature plots")):
        if i >= len(top_indices):
            ax.axis("off")
            continue

        feature_index = top_indices[i]

        feature_activations_1d = latent_activations[:, feature_index]
        feature_map = feature_activations_1d.reshape(xx.shape)

        max_val = jnp.max(feature_activations_1d)
        max_idx = jnp.argmax(feature_activations_1d)
        max_coords = coords[max_idx, :]
        x_at_max = max_coords[0]
        y_at_max = max_coords[1]

        ax.set_title(
            f"N={feature_index} (Max={max_val:.2f} @ ({x_at_max:.1f}, {y_at_max:.1f}))",
            fontsize=10,
        )

        c = ax.pcolormesh(xx, yy, feature_map, cmap="viridis", shading="auto")

        ax.scatter(
            data.spiral_coordinates[:, 0],
```

```
            data.spiral_coordinates[:, 1],
            c=data.spiral_labels,
            cmap=plt.cm.RdBu,
            edgecolors="k",
            s=5,
            alpha=0.2,
        )

        ax.set_xticks([])
        ax.set_yticks([])

        plt.colorbar(c, ax=ax, orientation="vertical", shrink=0.6)

    plt.suptitle(
        f"Activation Maps of Top {num_features_to_plot} Sparse Latent Features",
        fontsize=14,
    )
    plt.tight_layout(rect=[0, 0.03, 1, 0.95])

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

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

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
    latent_dim: int = 2048


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
    lambda_sparsity: float = 0.001
    sae_learning_rate: float = 0.01


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

    figsize: Tuple[int, int] = (10, 10)
    dpi: int = 200
    output_dir: Path = Path("hw07/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, SparseAutoEncoder
from .training import train_mlp, train_sae
from .plotting import plot_fit, plot_latent_features  # Updated import

"""
Discussion:
The Sparse Autoencoder has been trained to learn features about the Spirals MLP.
Experiments were done with SAE of dimension 1024 and 2048.
The features learned by the SAE seem to strongly correspond with particular spirals, as shown in the graphs.
For example, in latent_features_1024.pdf, feature 129 seems to strongly activate on the red spiral's overall shape.
In latent_features_2048.pdf, feature 1177 activates strongly on the blue spiral's overall shape.
Other features activate on smaller subsections of sprials, indicating that they have learned localized features of spirals.
These localized features do not cross over to the opposite spiral, indicating that the localized features are also spiral dis
criminative.
"""


def main() -> None:
    """CLI entry point."""
    settings = load_settings()
    configure_logging()
    log = structlog.get_logger()
    log.info("Hello from hw07!")
    log.info("Settings loaded", settings=settings.model_dump())

    # JAX PRNG
    key = jax.random.PRNGKey(settings.random_seed)
    data_key, model_key, sae_key = jax.random.split(key, 3)
    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,
    )

    mlp_optimizer = nnx.Optimizer(
        model,
        optax.adamw(
            learning_rate=learning_rate_schedule,
            weight_decay=settings.training.l2_reg,
        ),
        wrt=nnx.Param,
    )

    train_mlp(model, mlp_optimizer, data, settings.training, np_rng)

    sae = SparseAutoEncoder(
        key=sae_key,
        hidden_layer_width=settings.model.hidden_layer_width,
        latent_dim=settings.model.latent_dim,
    )

    sae_optimizer = nnx.Optimizer(
        sae,
        optax.adamw(
            learning_rate=settings.training.sae_learning_rate,
            weight_decay=settings.training.l2_reg,
        ),
        wrt=nnx.Param,
    )

    train_sae(model, sae, sae_optimizer, data, settings.training, np_rng)

    plot_fit(model, data, settings.plotting)

    plot_latent_features(model, sae, data, settings.plotting)
```

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


log = structlog.get_logger()


class NNXMLPModel(nnx.Module):
    """A Flax NNX module for an MLP-based Sparse Autoencoder with a classification head."""

    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.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=hidden_rngs
                )

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

        self.output_layer = nnx.Linear(
            hidden_layer_width, num_outputs, rngs=output_rngs
        )
        self.hidden_activation = hidden_activation
        self.output_activation = output_activation

    def extract_final_hidden_state(self, coords: jax.Array) -> jnp.ndarray:
        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

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

    def __call__(self, coords: jax.Array) -> jax.Array:
        final_hidden_state = self.extract_final_hidden_state(coords)

        return self.output_activation(self.output_layer(final_hidden_state))


class SparseAutoEncoder(nnx.Module):
    def __init__(
```



```python
        self,
        *,
        key: jax.random.PRNGKey,
        hidden_layer_width: int,
        latent_dim: int,
    ):
        keys = jax.random.split(key, 3)

        self.encoder_layer = nnx.Linear(
            hidden_layer_width, latent_dim, rngs=nnx.Rngs(params=keys[0])
        )
        self.decoder_layer = nnx.Linear(
            latent_dim, hidden_layer_width, rngs=nnx.Rngs(params=keys[1])
        )
        self.latent_activation = nnx.relu
        self.latent_dim = latent_dim

    def encode(self, final_hidden_state: jnp.ndarray) -> jnp.ndarray:
        latent_vector = self.latent_activation(self.encoder_layer(final_hidden_s
tate))
        return latent_vector

    def decode(self, latent_vector: jnp.ndarray) -> jnp.ndarray:
        return self.decoder_layer(latent_vector)

    def __call__(
        self, final_hidden_state: jnp.ndarray
    ) -> tuple[jnp.ndarray, jnp.ndarray]:
        latent_vector = self.encode(final_hidden_state)
        reconstruction = self.decode(latent_vector)
        return reconstruction, latent_vector
```

```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, SparseAutoEncoder

log = structlog.get_logger()


@nnx.jit
def train_mlp_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_mlp(
    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)

        loss = train_mlp_step(model, optimizer, coords, labels)
        bar.set_description(f"MLP Loss @ {i} => {loss:.6f}")
        bar.refresh()
    log.info("Training finished")


@nnx.jit
def train_sae_step(
    sae: SparseAutoEncoder,
    optimizer: nnx.Optimizer,
    z: jnp.ndarray,
    lambda_sparsity: float,
):
    """Performs a single training step for the Sparse Autoencoder."""

    def loss_fn(sae: SparseAutoEncoder):
        z_hat, h = sae(z)

        reconstruction_loss = jnp.mean((z - z_hat) ** 2)

        sparsity_loss = jnp.mean(jnp.abs(h))

        total_loss = reconstruction_loss + lambda_sparsity * sparsity_loss

        return total_loss, (reconstruction_loss, sparsity_loss)
```

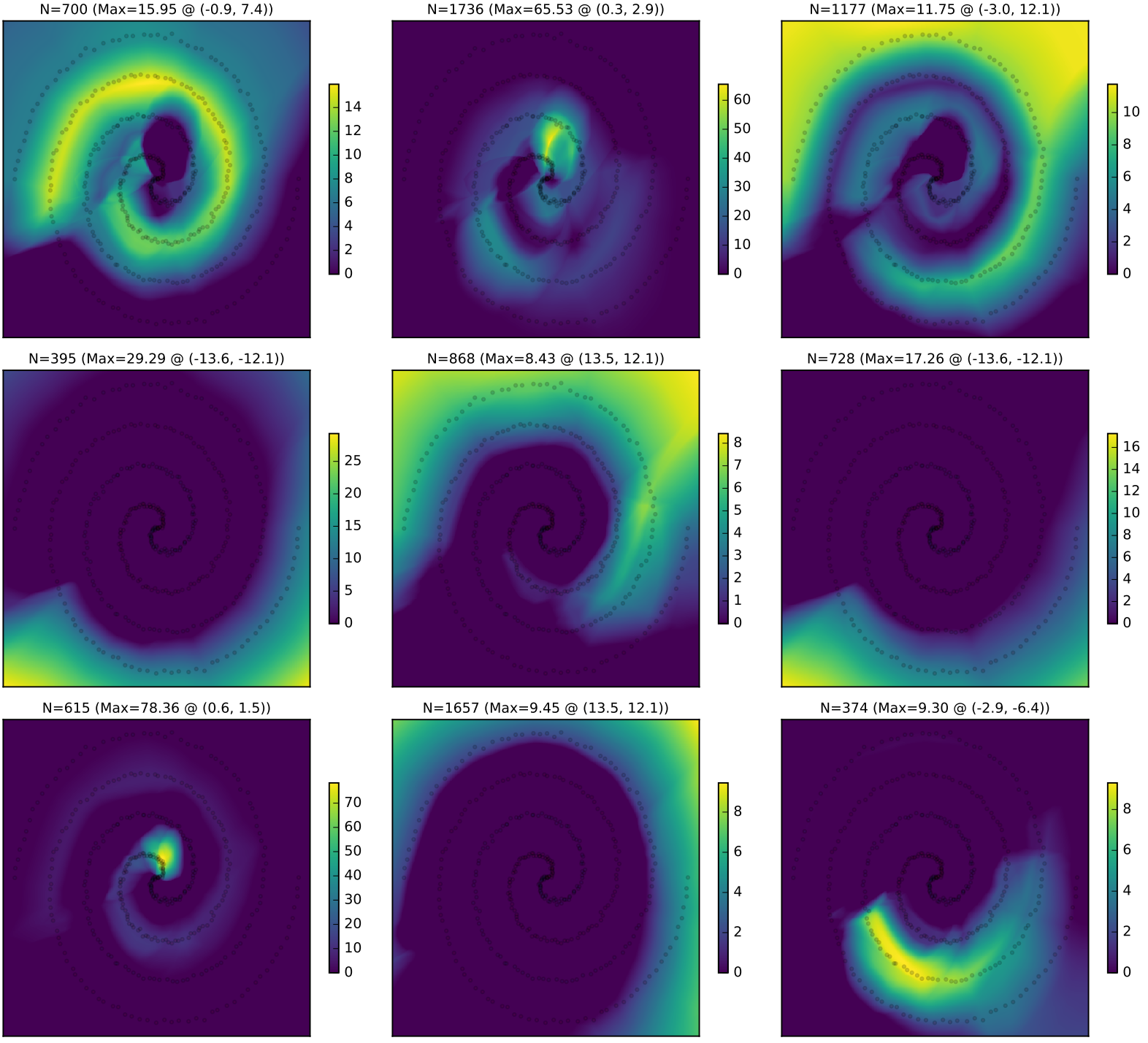


```python
    (total_loss, (recon_loss, sparse_loss)), grads = nnx.value_and_grad(
        loss_fn, has_aux=True
    )(sae)

    optimizer.update(sae, grads)

    return total_loss, recon_loss, sparse_loss


def train_sae(
    mlp: NNXMLPModel,
    sae: SparseAutoEncoder,
    sae_optimizer: nnx.Optimizer,
    data: Data,
    settings: TrainingSettings,
    np_rng: np.random.Generator,
) -> None:
    log.info("Starting SAE Training (Stage 1)", **settings.model_dump())

    all_coords = jnp.asarray(data.spiral_coordinates)
    all_z = mlp.extract_final_hidden_state(all_coords)

    bar = trange(settings.num_iters // 2)
    for i in bar:
        choices = np_rng.choice(all_z.shape[0], size=settings.batch_size)
        z_batch = all_z[choices]

        total_loss, recon_loss, sparse_loss = train_sae_step(
            sae, sae_optimizer, z_batch, settings.lambda_sparsity
        )

        bar.set_description(
            f"SAE Loss @ {i} => L_tot={total_loss:.4f} | L_recon={recon_loss:.4f} | L_sparse={sparse_loss:.4
f}"
        )
    log.info("SAE Training finished")
```

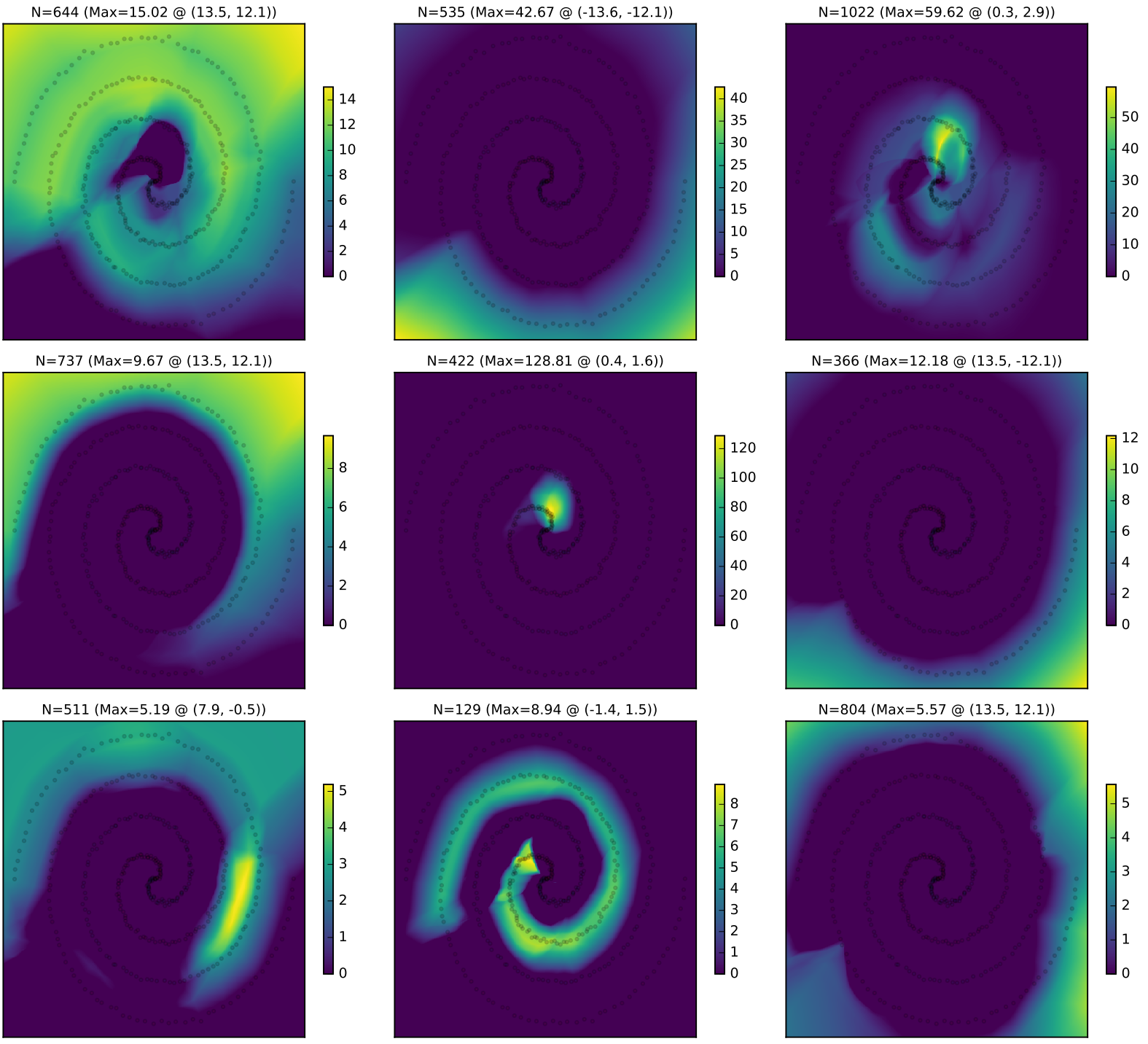
```python
import logging
import os
import sys
from pathlib import Path

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("hw07").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,
    )
```

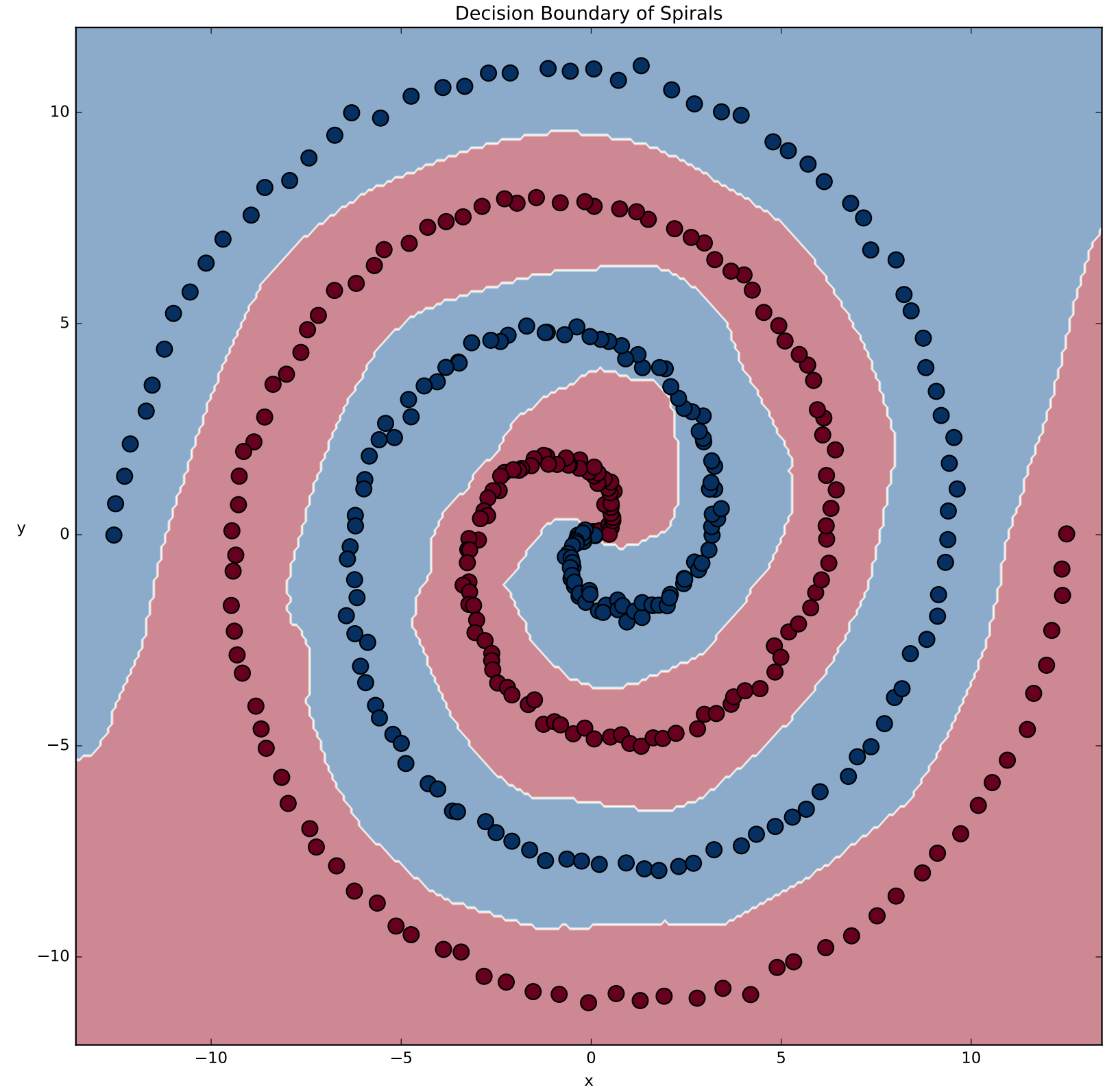
```python
from dataclasses import InitVar, dataclass, field

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=(self.num_samples_per_spiral)
        )
        self.y1 = -spiral * np.sin(spiral) + rng.normal(
            0, self.sigma, size=(self.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]
```

Activation Maps of Top 9 Sparse Latent Features



Activation Maps of Top 9 Sparse Latent Features




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