```
root@LegionT5:/mnt/c/Users/joshu/OneDrive/Cooper Union/Deep Learning/hw07# just
run hw07
==> Running homework 'hw07'...
2025-11-25T08:45:58.326936Z [info     ] Settings loaded                [hw07] se
ttings={'debug': False, 'random_seed': 427, 'data': {'num_features': 2, 'num_out
puts': 1, 'num_samples': 2000, 'noise': 0.1}, 'model': {'num_hl': 16, 'hl_width'
: 256, 'latent_dim': 2048}, 'mlp_training': {'batch_size': 64, 'num_iters': 2000
, 'learning_rate': 0.001, 'epsilon': 1e-06, 'lr_decay_alpha': 0.01, 'use_cosine_
decay': False}, 'sae_training': {'batch_size': 128, 'num_iters': 5000, 'learning
_rate': 0.01, 'lambda_sparsity': 0.001, 'lr_decay_alpha': 0.005, 'use_cosine_dec
ay': False}, 'plotting': {'figsize': (10, 10), 'dpi': 200, 'output_dir': PosixPa
th('hw07/artifacts'), 'linspace': 1000}}
W1125 03:46:06.283004   30010 cuda_executor.cc:1802] GPU interconnect informatio
n not available: INTERNAL: NVML doesn't support extracting fabric info or NVLink
 is not used by the device.
W1125 03:46:06.299571   29960 cuda_executor.cc:1802] GPU interconnect informatio
n not available: INTERNAL: NVML doesn't support extracting fabric info or NVLink
 is not used by the device.
2025-11-25T08:46:09.069130Z [info     ] spirals generated              [hw07] sh
ape=(2000, 2)
2025-11-25T08:46:17.822846Z [info     ] spiral model generated!        [hw07] hl
_width=256 num_hl=16
2025-11-25T08:46:19.530478Z [info     ] Start training MLP             [hw07.tra
ining] batch_size=64 learning_rate=0.001 num_iters=2000
MLP Loss @ 1999 => 0.002859: 100%|âM-^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-^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-^H| 2000/2000 [00:13<00:00, 146.77it/s]
2025-11-25T08:46:33.158075Z [info     ] MLP Training Finished          [hw07.tra
ining] final_loss=0.002859
2025-11-25T08:46:33.158258Z [info     ] finished training MLP          [hw07]
2025-11-25T08:46:33.158384Z [info     ] generating spiral_plot         [hw07.plo
tting]
2025-11-25T08:46:40.445408Z [info     ] Decision boundary plot generated! [hw07.
plotting] path=hw07/artifacts/classification.pdf
2025-11-25T08:46:40.445722Z [info     ] finished plotting decision boundary [hw0
7]
2025-11-25T08:46:42.065805Z [info     ] SAE model generated!           [hw07] ex
pansion_factor=8 hidden_width=256 latent_dim=2048
2025-11-25T08:46:43.026117Z [info     ] Start training SAE on frozen MLP [hw07.t
raining] batch_size=128 lambda_sparsity=0.001 learning_rate=0.01 num_iters=5000
SAE Loss @ 4999 => L_tot=0.0001 | L_recon=0.0001 | L_sparse=0.0175: 100%|âM-^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-^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-^H| 5000/5000 [00:38<00:00, 128.60it/s]
```



```
2025-11-25T08:47:23.495298Z [info     ] SAE Training Finished          [hw07.tra
ining] final_recon_loss=0.0001021 final_sparse_loss=0.0175 final_total_loss=0.00
01196
2025-11-25T08:47:23.495516Z [info     ] finished training SAE          [hw07]
2025-11-25T08:47:23.495623Z [info     ] Plotting Sparse Latent Features [hw07.pl
otting]
Generating feature plots: 100%|âM-^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-^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-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^V
M-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^H| 9/9 [00:06<00:00,  1.46it/s]
2025-11-25T08:48:11.253090Z [info     ] Latent features plot generated! [hw07.pl
otting] path=hw07/artifacts/latent_features.pdf
2025-11-25T08:48:11.253378Z [info     ] finished plotting latent features [hw07]
```

```
# pyproject.toml.jinja

[project]
name = "hw07"
version = "0.1.0"
description = "Sparse Autoencoder on Spiral Classifer"
readme = "README.md"
authors = [
    { name = "Josh Miao", email = "josh.miao@cooper.edu" }
]
requires-python = ">=3.13"
dependencies = [
    "structlog",
    "rich",
    "numpy",
    "jax[cuda12]",
    "flax",
    "matplotlib>=3.10.7",
    "tqdm>=4.67.1",
    "pydantic>=2.12.4",
    "pydantic-settings>=2.12.0",
    "typing>=3.10.0.0",
    "optax>=0.2.6",
    "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
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 = 2
    num_outputs: int = 1
    num_samples: int = 2000  # 1000 per spiral
    noise: float = 0.1


class ModelSettings(BaseModel):
    """Settings for model architecture."""
    num_hl: int = 16
    hl_width: int = 256
    latent_dim: int = 2048  # SAE latent dimension


class MLPTrainingSettings(BaseModel):
    """Settings for MLP classifier training."""
    batch_size: int = 64
    num_iters: int = 2000
    learning_rate: float = 0.001
    epsilon: float = 1e-6
    # Cosine decay settings
    lr_decay_alpha: float = 0.01
    use_cosine_decay: bool = False # the decay was detrimental to training MLP i
dk why and training the MLP isnt a big enough deal to warrant this additional ef
fort


class SAETrainingSettings(BaseModel):
    """Settings for SAE training."""
    batch_size: int = 128
    num_iters: int = 5000
    learning_rate: float = 0.01
    lambda_sparsity: float = 0.001
    # Cosine decay settings
    lr_decay_alpha: float = 0.005
    use_cosine_decay: bool = False #for some reason it either does a really good
 job of minimizing recon loss or sparse loss but not both


class PlottingSettings(BaseModel):
    """Settings for plotting."""
    figsize: Tuple[int, int] = (10, 10)
    dpi: int = 200
    output_dir: Path = Path("hw07/artifacts")
    linspace: int = 1000


class AppSettings(BaseSettings):
    """Main application settings."""
    debug: bool = False
    random_seed: int = 427
    data: DataSettings = DataSettings()
    model: ModelSettings = ModelSettings()
    mlp_training: MLPTrainingSettings = MLPTrainingSettings()
    sae_training: SAETrainingSettings = SAETrainingSettings()
```



```python
    plotting: PlottingSettings = PlottingSettings()

    model_config = SettingsConfigDict(
        toml_file=files("hw07").joinpath("config.toml") if files("hw07").joinpath("c
onfig.toml").is_file() else None,
        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."""
        return (
            init_settings,
            TomlConfigSettingsSource(settings_cls),
            env_settings,
            dotenv_settings,
            file_secret_settings,
        )


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

```python
from dataclasses import dataclass, InitVar, field
import numpy as np
from numpy import pi
from numpy.random import Generator

@dataclass
class Data:
    """generate spiral data"""
    rng: InitVar[Generator]
    num_points: int
    num_turns: float #number of convolutions in spiral
    scale: float
    noise: float
    x: np.ndarray = field(init=False)
    target: np.ndarray = field(init=False)

    def __post_init__(self, rng):
        #angular sweep, 2pi represents one full convolution
        theta = np.sqrt(rng.random(self.num_points)) * self.num_turns * pi

        #spiral tightness and offset
        r_a = self.scale * theta
        #spiral b will just mirror spiral a across origin
        r_b = -r_a

        #generate cartesian data
        data_a = np.array([np.cos(theta) * r_a, np.sin(theta) * r_a]).T
        data_b = np.array([np.cos(theta) * r_b, np.sin(theta) * r_b]).T
        #noise using rng
        x_a = data_a + rng.normal(0, self.noise, (self.num_points, 2))
        x_b = data_b + rng.normal(0, self.noise, (self.num_points, 2))

        self.x = np.vstack((x_a, x_b))
        self.target = np.concatenate((np.zeros(self.num_points), np.ones(self.nu
m_points)))

    def get_batch(self, rng: Generator, batch_size: int) -> tuple[np.ndarray, np
.ndarray]:
        """Select random subset of examples for training batch."""
        indices = rng.choice(len(self.x), size=batch_size, replace=False)
        return self.x[indices], self.target[indices]
```

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

log = structlog.get_logger()

def xavier_kernel_init(num_inputs: int, num_outputs: int):
    """Glorot/Xavier uniform initialization"""
    def init(rng, shape, dtype=jnp.float32):
        limit = jnp.sqrt(6.0 / (num_inputs + num_outputs))
        return jax.random.uniform(rng, shape, dtype, minval=-limit, maxval=limit
)
    return init

class Block(nnx.Module):
    """define each hidden layer block"""
    def __init__(self, num_inputs: int, hl_width: int, rngs: nnx.Rngs):
        super().__init__()
        self.linear = nnx.Linear(
            num_inputs,
            hl_width,
            rngs=rngs,
            kernel_init=xavier_kernel_init(num_inputs, hl_width),
        )

    def __call__(self, x: jax.Array):
        x = self.linear(x)
        return jax.nn.relu(x)

class MLP(nnx.Module):
    """build our hidden layers"""
    def __init__(
        self,
        *,
        rngs: nnx.Rngs,
        num_inputs: int,
        num_outputs: int,
        num_hl: int,
        hl_width: int,
    ):
        super().__init__()
        self.num_inputs = num_inputs
        self.num_outputs = num_outputs
        self.num_hl = num_hl
        self.hl_width = hl_width

        key = rngs.params()

        #initial layer
        k_first, key = jax.random.split(key)
        self.first = nnx.Linear(
            num_inputs,
            hl_width,
            rngs=nnx.Rngs(k_first),
            kernel_init=xavier_kernel_init(num_inputs, hl_width),
        )

        #hidden layers
        @nnx.split_rngs(splits=num_hl - 1)
        @nnx.vmap(in_axes=(0,), out_axes=0)
        def make_block(rngs_for_one_layer: nnx.Rngs):
            return Block(num_inputs=hl_width, hl_width=hl_width, rngs=rngs_for_o
ne_layer)

        self.hidden_blocks = make_block(nnx.Rngs(key))

        #output layer
        k_out = jax.random.split(key, 2)[1]
```



```python
        self.out = nnx.Linear(
            hl_width,
            num_outputs,
            rngs=nnx.Rngs(k_out),
            kernel_init=xavier_kernel_init(hl_width, num_outputs),
        )

    def extract_final_hidden_state(self, x: jax.Array) -> jax.Array:
        """Extract activations before the output layer for SAE training."""
        # First transform
        x = jax.nn.relu(self.first(x))

        #applies Block sequentially to our layers
        @nnx.scan(in_axes=(nnx.Carry, 0), out_axes=nnx.Carry)
        def forward(carry, block):
            return block(carry)

        x = forward(x, self.hidden_blocks)
        return x

    def __call__(self, x: jax.Array):
        # Extract final hidden state
        x = self.extract_final_hidden_state(x)

        #use logits as an input to the sigmoid function in our last matmul
        logits = self.out(x)
        probs = jax.nn.sigmoid(logits)
        return probs


class SparseAutoEncoder(nnx.Module):
    """Sparse Autoencoder for discovering interpretable features."""
    def __init__(
        self,
        *,
        rngs: nnx.Rngs,
        hidden_layer_width: int,
        latent_dim: int,
    ):
        super().__init__()
        self.hidden_layer_width = hidden_layer_width
        self.latent_dim = latent_dim

        key = rngs.params()
        k_encoder, k_decoder = jax.random.split(key, 2)

        # Encoder: compress hidden states to sparse latent space
        self.encoder = nnx.Linear(
            hidden_layer_width,
            latent_dim,
            rngs=nnx.Rngs(k_encoder),
            kernel_init=xavier_kernel_init(hidden_layer_width, latent_dim),
        )

        # Decoder: reconstruct hidden states from latent space
        self.decoder = nnx.Linear(
            latent_dim,
            hidden_layer_width,
            rngs=nnx.Rngs(k_decoder),
            kernel_init=xavier_kernel_init(latent_dim, hidden_layer_width),
        )

    def encode(self, z: jax.Array) -> jax.Array:
        """Encode hidden states to latent space with ReLU activation."""
        return jax.nn.relu(self.encoder(z))

    def decode(self, h: jax.Array) -> jax.Array:
        """Decode latent activations back to hidden state space."""
        return self.decoder(h)
```

```
    def __call__(self, z: jax.Array) -> tuple[jax.Array, jax.Array]:
        """Forward pass: returns (reconstruction, latent_activations)."""
        h = self.encode(z)
        z_hat = self.decode(h)
        return z_hat, h
```

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

from sklearn.inspection import DecisionBoundaryDisplay
from tqdm import tqdm
from .config import PlottingSettings
from .data import Data
from .model import MLP, SparseAutoEncoder

log = structlog.get_logger()

font = {
    # "family": "Adobe Caslon Pro",
    "size": 10,
}
matplotlib.style.use("classic")
matplotlib.rc("font", **font)

def spiral_plot(
    model: MLP,
    data: Data,
    settings: PlottingSettings,
):
    """Plot spiral decision boundary and save to a PDF."""
    log.info("generating spiral_plot")

    x_min, x_max = data.x[:, 0].min() * 1.1, data.x[:, 0].max() * 1.1
    y_min, y_max = data.x[:, 1].min() * 1.1, data.x[:, 1].max() * 1.1

    x_vals = np.linspace(x_min, x_max, settings.linspace)
    y_vals = np.linspace(y_min, y_max, settings.linspace)

    feature1, feature2 = np.meshgrid(x_vals, y_vals)

    log.debug("feature1 has dimensions:", shape=feature1.shape)
    log.debug("feature2 has dimensions:", shape=feature2.shape)

    grid = np.vstack([feature1.ravel(), feature2.ravel()]).T

    predicted_prob = model(grid)
    predicted_prob = predicted_prob.ravel().reshape(feature1.shape)
    log.debug("predicted_prob has dimensions:", shape=predicted_prob.shape)

    predicted_prob = np.where(predicted_prob > 0.5, 1, 0)

    fig, ax = plt.subplots(figsize=settings.figsize, dpi=settings.dpi)
    display = DecisionBoundaryDisplay(xx0 = feature1, xx1 = feature2, response =
 predicted_prob)
    display.plot(ax=ax, cmap=plt.cm.seismic, alpha=0.5)
    ax.scatter(data.x[:, 0], data.x[:, 1], c=data.target, cmap=plt.cm.coolwarm,
edgecolors='k', s=100)

    ax.set_xlabel('x-position')
    ax.set_ylabel('y-position')
    ax.set_title("Decision Boundary of Spirals")

    plt.tight_layout()

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

    plt.savefig(output_path)
    plt.close()

    log.info("Decision boundary plot generated!", path=str(output_path))
```



```python
def latent_features_plot(
    mlp: MLP,
    sae: SparseAutoEncoder,
    data: Data,
    settings: PlottingSettings,
    num_features_to_plot: int =9,
):
    """Plot activation maps of top sparse latent features."""
    log.info("Plotting Sparse Latent Features")

    x_min, x_max = data.x[:, 0].min() - 1, data.x[:, 0].max() + 1
    y_min, y_max = data.x[:, 1].min() - 1, data.x[:, 1].max() + 1

    mesh_step = 0.1
    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()])

    # Extract hidden states and encode to latent space
    dense_hidden_state_z = mlp.extract_final_hidden_state(coords)
    latent_activations = sae.encode(dense_hidden_state_z)

    # Find top features by mean activation
    mean_activations = jnp.mean(latent_activations, axis=0)
    top_indices = jnp.argsort(mean_activations)[::-1][:num_features_to_plot]

    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='plasma', shading='auto')

        ax.scatter(
            data.x[:, 0],
            data.x[:, 1],
            c=data.target,
            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)
    plt.close()
    log.info("Latent features plot generated!", path=str(output_path))
```

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

from flax import nnx
from tqdm import trange
from .config import MLPTrainingSettings, SAETrainingSettings
from .data import Data
from .model import MLP, SparseAutoEncoder

log = structlog.get_logger()

@nnx.jit
def train_step_spiral(
    model: MLP,
    optimizer: nnx.Optimizer,
    x: jnp.ndarray,
    target: jnp.ndarray,
    epsilon: float,
 ):
    """Single training step for MLP classifier."""
    def loss_function(model: MLP):
        probs = model(x)
        #limit the max and min value of value passed into logarithm to avoid nan
 loss
        probs = jnp.where(probs < epsilon, epsilon, probs)
        probs = jnp.where(probs > 1 - epsilon, 1 - epsilon, probs)

        #calculate average loss using binary cross entropy
        loss = -((jnp.log(probs) * target) + ((1 - target) * jnp.log(1 - probs))
)
        loss = jnp.mean(loss)
        return loss

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

def spiral_train(
    model: MLP,
    optimizer: nnx.Optimizer,
    data: Data,
    settings: MLPTrainingSettings,
    np_rng: np.random.Generator,
):
    """SGD training for MLP classifier."""
    log.info("Start training MLP",
             num_iters=settings.num_iters,
             batch_size=settings.batch_size,
             learning_rate=settings.learning_rate)
    bar = trange(settings.num_iters, desc="Training MLP")
    for i in bar:
        x_np, target_np = data.get_batch(np_rng, settings.batch_size)
        x, target = jnp.asarray(x_np), jnp.asarray(target_np)
        target = target.reshape(-1, 1)  #reshape target to correct output dimens
ions

        loss = train_step_spiral(model, optimizer, x, target, settings.epsilon)

        bar.set_description(f"MLP Loss @ {i} => {loss:.6f}")

        if i % 1000 == 0:
            log.debug("MLP training progress", iteration=i, loss=float(loss))

        bar.refresh()
    log.info("MLP Training Finished", final_loss=float(loss))


@nnx.jit
```



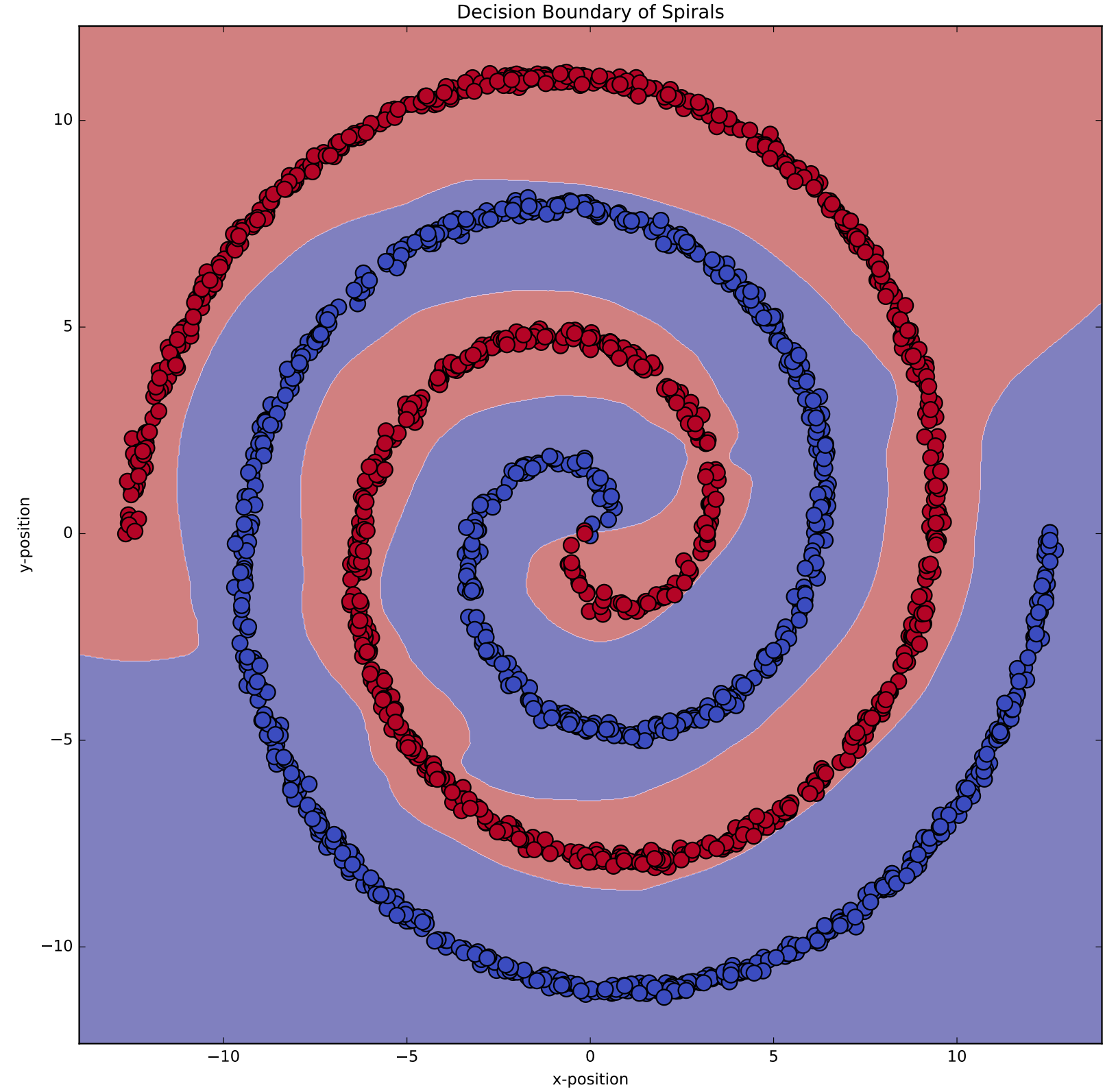
```python
def train_step_sae(
    sae: SparseAutoEncoder,
    optimizer: nnx.Optimizer,
    z: jnp.ndarray,
    lambda_sparsity: float,
):
    """Single training step for Sparse Autoencoder."""
    def loss_function(sae: SparseAutoEncoder):
        z_hat, h = sae(z)

        # Reconstruction loss: MSE between original and reconstructed hidden sta
tes
        reconstruction_loss = jnp.mean((z - z_hat) ** 2)

        # Sparsity loss: L1 penalty on latent activations
        sparsity_loss = jnp.mean(jnp.abs(h))

        # Combined loss
        total_loss = reconstruction_loss + lambda_sparsity * sparsity_loss

        return total_loss, (reconstruction_loss, sparsity_loss)

    (total_loss, (recon_loss, sparse_loss)), grads = nnx.value_and_grad(
        loss_function, has_aux=True
    )(sae)

    optimizer.update(sae, grads)
    return total_loss, recon_loss, sparse_loss


def sae_train(
    mlp: MLP,
    sae: SparseAutoEncoder,
    optimizer: nnx.Optimizer,
    data: Data,
    settings: SAETrainingSettings,
    np_rng: np.random.Generator,
):
    """Train Sparse Autoencoder on frozen MLP hidden states."""
    log.info("Start training SAE on frozen MLP",
             num_iters=settings.num_iters,
             batch_size=settings.batch_size,
             learning_rate=settings.learning_rate,
             lambda_sparsity=settings.lambda_sparsity)

    # Extract all hidden states from frozen MLP
    all_x = jnp.asarray(data.x)
    log.debug("Extracting hidden states from MLP", num_samples=all_x.shape[0])
    all_z = mlp.extract_final_hidden_state(all_x)
    log.debug("Hidden states extracted", shape=all_z.shape)

    bar = trange(settings.num_iters, desc="Training SAE")
    for i in bar:
        # Sample random batch of hidden states
        choices = np_rng.choice(all_z.shape[0], size=settings.batch_size, replac
e=False)
        z_batch = all_z[choices]

        total_loss, recon_loss, sparse_loss = train_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}"
        )

        if i % 1000 == 0:
            log.debug("SAE training progress",
```

```
                    iteration=i,
                    total_loss=float(total_loss),
                    recon_loss=float(recon_loss),
                    sparse_loss=float(sparse_loss))

        bar.refresh()

    log.info("SAE Training Finished",
             final_total_loss=float(total_loss),
             final_recon_loss=float(recon_loss),
             final_sparse_loss=float(sparse_loss))
```

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

from flax import nnx
from numpy.random import Generator, PCG64

from .config import load_settings
from .data import Data
from .model import MLP, SparseAutoEncoder
from .logging import configure_logging
from .plotting import spiral_plot, latent_features_plot
from .training import spiral_train, sae_train

"""
Discussion:
The Sparse Autoencoder has been trained to learn features about the Spirals MLP. The SAE was trained on a dimension
 of 2048.
The features learned by the SAE seem to strongly correspond with particular spirals, as shown in the graphs.
Features 1788 and 1047 strongly correlated with the blue spiral while features 1688and 1091 strongly correlated with th
e red spiral.
The other activated features corresponded most strongly to the corners of the graph. I feel like this could be explained b
ecause the
corners are the regions that are easiest to identify as belonging to a certain spiral and thus be tied to blue for the bottom
corners
and red to the top corners.its worth nothing that the magnitudes of the activation are signficantly stronger for the first 5
features
which highlight an entire decision boundary. it seems like everything in the spirals are just mapped to a handful of featu
res and the
rest of the features map to either the corners or very minutes parts of the spirals.
"""

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, sae_key = jax.random.split(key, 3)
    np_rng = np.random.default_rng(np.array(data_key))

    #Generate spiral data
    data = Data(
        rng = Generator(PCG64(seed=472)),
        num_points = settings.data.num_samples // 2,
        num_turns = 4,
        scale = 1.0,
        noise = settings.data.noise,
    )
    log.info("spirals generated", shape = data.x.shape)

    # Build and train MLP classifier
    model = MLP(
        rngs=nnx.Rngs(params=model_key),
        num_inputs = settings.data.num_features,
        num_outputs = settings.data.num_outputs,
        num_hl=settings.model.num_hl,
        hl_width=settings.model.hl_width,
    )
    log.info("spiral model generated!",
             num_hl=settings.model.num_hl,
             hl_width=settings.model.hl_width)

    # MLP optimizer with optional cosine decay
    if settings.mlp_training.use_cosine_decay:
```



```
        mlp_schedule = optax.cosine_decay_schedule(
            init_value=settings.mlp_training.learning_rate,
            decay_steps=settings.mlp_training.num_iters,
            alpha=settings.mlp_training.lr_decay_alpha
        )
        mlp_optimizer = nnx.Optimizer(model, optax.adam(mlp_schedule), wrt=nnx.P
aram)
    else:
        mlp_optimizer = nnx.Optimizer(
            model, optax.adam(settings.mlp_training.learning_rate), wrt=nnx.Para
m
        )

    spiral_train(model, mlp_optimizer, data, settings.mlp_training, np_rng)
    log.info("finished training MLP")

    spiral_plot(model, data, settings.plotting)
    log.info("finished plotting decision boundary")

    # Build and train Sparse Autoencoder
    sae = SparseAutoEncoder(
        rngs=nnx.Rngs(params=sae_key),
        hidden_layer_width=settings.model.hl_width,
        latent_dim=settings.model.latent_dim,
    )
    log.info("SAE model generated!",
             hidden_width=settings.model.hl_width,
             latent_dim=settings.model.latent_dim,
             expansion_factor=settings.model.latent_dim / settings.model.hl_widt
h)

    # SAE optimizer with optional cosine decay
    if settings.sae_training.use_cosine_decay:
        sae_schedule = optax.cosine_decay_schedule(
            init_value=settings.sae_training.learning_rate,
            decay_steps=settings.sae_training.num_iters,
            alpha=settings.sae_training.lr_decay_alpha
        )
        sae_optimizer = nnx.Optimizer(sae, optax.adam(sae_schedule), wrt=nnx.Par
am)
    else:
        sae_optimizer = nnx.Optimizer(
            sae, optax.adam(settings.sae_training.learning_rate), wrt=nnx.Param
        )

    sae_train(model, sae, sae_optimizer, data, settings.sae_training, np_rng)
    log.info("finished training SAE")

    latent_features_plot(model, sae, data, settings.plotting)
    log.info("finished plotting latent features")
```


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

Activation Maps of Top 9 Sparse Latent Features