```
# pyproject.toml.jinja

[project]
name = "hw02"
version = "0.1.0"
description = "My new homework assignment."
readme = "README.md"
authors = [
    { name = "My name", email = "my@email" }
]
requires-python = ">=3.13"
dependencies = [
    "structlog",
    "rich",
    "numpy",
    "jax",
    "flax",
    "matplotlib>=3.10.6",
    "scikit-learn>=1.7.2",
    "pydantic>=2.11.9",
    "optax>=0.2.6",
    "pathlib>=1.0.1",
    "typing>=3.10.0.0",
    "tqdm>=4.67.1",
    "pydantic-settings>=2.10.1",
]

[project.scripts]
hw02 = "hw02: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 = 650
    noise: float = .3

class ModelSettings(BaseModel):
    num_hl: int = 8
    hl_width: int = 128

class TrainingSettings(BaseModel):
    """Settings for model training."""
    batch_size: int = 64
    num_iters: int = 3000
    learning_rate: float = 0.001
    epsilon: float = 1e-6

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

    figsize: Tuple[int, int] = (5, 3)
    dpi: int = 200
    output_dir: Path = Path("artifacts")
    linspace: int = 1000


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

    debug: bool = False
    random_seed: int = 427 #for ece 427 ofc
    data: DataSettings = DataSettings()
    training: TrainingSettings = TrainingSettings()
    plotting: PlottingSettings = PlottingSettings()
    model: ModelSettings = ModelSettings()

    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.


```python
    """
        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):

        #bitgenerator or smth, as required by you
        #rng = Generator(PCG64(seed=472))

        #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]:
        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("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,
    )
```

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

"""
 I HAD DINNER WITH ERIC ENG AND HE PROPOSED TANH AS MY ACTIVATION FUNCTION
 SO I SWITCHED TO THE XAVIER GLOROT KERNEL AND SWITHCED TO TANH AND IT WORKED PROFESSOR
"""


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)
        """
    performs the "mat-mul" as you would say. âM-^XM-^]ï¸M-^OðM-^_¤M-^SâM-^XM-^]ï¸M-^OðM-^_¤M-^SâM-^XM-^]ï¸M-^OðM-^_¤M-^S
    idk if just pdf will print the emojis but in case they dont, i pasted the nerd emoji with the
    finger pointing up because you embody the "ermmm actually" *pushes glasses up* persona.
   """
        return jax.nn.tanh(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__() #initialize nnx.Module internal parameters
        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_one_layer)

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

        #output layer
        k_out = jax.random.split(key, 2)[1]
        self.out = nnx.Linear(
            hl_width,
            num_outputs,
            rngs=nnx.Rngs(k_out),
            kernel_init=xavier_kernel_init(hl_width, num_outputs=1),
        )

    def __call__(self, x: jax.Array):
        # First transform
        x = jax.nn.tanh(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)

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

"""
my first attempt involved using ReLUs as my activation unit with only 2 hidden layers since He talks about how 2 is almost
always enough. However, i soon realized that i needed greater expressive power because i would get very jagged decision boundaries.
I then increased my num_hl and hl_width to be able to model the curves of the spirals. However, when my num_hl or width grew too much,
my plot would be either all red or all blue which led me to beleive that the more hidden layers, the more vulnerable i am to gradient
explosion or implosion. I found a happy medium with what i currently have in my config file.

I had dinner with Eric Eng and Isaac Rivera today and they suggested using tanh activation function which they claimed were good for
circular things. I also switched my kernal initiazliation from the kaiming he to the glorot xavier one because the kaiming one
was specific to ReLUs. They also told me to just crank up my num_hl and hl_width to increase my expressive power but all that did
was make my computer turn into a jet turbine.

In the end, im happy with tanh as my activation functin and my parameters in the config file.
"""
```

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

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

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=feature1.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)


    log.debug("target", target=predicted_prob)
    predicted_prob = np.where(predicted_prob > 0.5, 1, 0)

    display = DecisionBoundaryDisplay(xx0 = feature1, xx1 = feature2, response =
 predicted_prob)
    display.plot()
    display.ax_.scatter(data.x[:, 0], data.x[:, 1], c=data.target)

    plt.xlabel('x-position')
    plt.ylabel('y-position')

    settings.output_dir.mkdir(parents=True, exist_ok=True)
    output_path = settings.output_dir / "hw02_plt.pdf"

    plt.title("Binary Classification of 2 Archimedean Spirals")
    plt.savefig(output_path)

    log.info("Plot Generated! FInally done!!!!!!!", 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 TrainingSettings
from .data import Data
from .model import MLP

log = structlog.get_logger()

@nnx.jit
def train_step_spiral(
    model: MLP,
    optimizer: nnx.Optimizer,
    x: jnp.ndarray,
    target: jnp.ndarray,
    epsilon: float,
 ):

    def loss_function(model: MLP):
        """trianing step"""
        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, 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: TrainingSettings,
    np_rng: np.random.Generator,
):
    """SGD training"""
    log.info("Start training")
    bar = trange(settings.num_iters)
    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 diemns
ions

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

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

        bar.refresh()
    log.info("Training Finished")
```

```python
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
from .logging import configure_logging
from .plotting import spiral_plot
from .training import spiral_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(472) #472 for ECE 472
    data_key, model_key = jax.random.split(key)
    np_rng = np.random.default_rng(np.array(data_key))

    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)

    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!", params = model)

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

    spiral_train(model, optimizer, data, settings.training, np_rng)
    log.info("finished training")

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