```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 = 16
    num_iters: int = 300
    learning_rate: float = 0.1


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
    data: DataSettings = DataSettings()
    training: TrainingSettings = TrainingSettings()
    plotting: PlottingSettings = PlottingSettings()

    model_config = SettingsConfigDict(
        toml_file=files("hw01").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),
```

```python
            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 Basis_Model_param, BasisModel
from .plotting import compare_linear_models, plot_fit, base_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))

    # Generate 50 data points given parameters
    data_generating_model = Basis_Model_param(
        mu=np_rng.integers(low=0, high=5, size=(settings.data.num_features)),
        sigma=np_rng.integers(low=0, high=5, size=(settings.data.num_features)),
        w=np_rng.integers(low=0, high=5, size=(settings.data.num_features)),
        b=0,
    )
    log.debug("Data generating model", model=data_generating_model)

    data = Data(
        model=data_generating_model,
        rng=np_rng,
        num_features=settings.data.num_features,
        num_basis_ftn=settings.data.num_basis_ftn,
        num_samples=settings.data.num_samples,
        sigma=settings.data.sigma_noise,
    )

    model = BasisModel(
        rngs=nnx.Rngs(params=model_key), num_basis_ftn=settings.data.num_basis_f
tn
    )
    log.debug("Initial model", model=model.model)

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

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

    log.debug("Trained model", model=model.model)

    compare_linear_models(data.model, model.model)

    if settings.data.num_features == 1:
        plot_fit(model, data, settings.plotting)
        base_fit(
            model, data, settings.plotting, num_basis_ftn=settings.data.num_basi
s_ftn
        )

    else:
        log.info("Skipping plotting for multi-feature models.")
```

```toml
debug = false
random_seed = 0xc61dd8fc

[data]
num_features = 1
num_basis_ftn = 6       #M
num_samples = 50        #N
sigma_noise = 0.1

[training]
batch_size = 16
num_iters = 300
learning_rate = 0.1

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

```
from dataclasses import dataclass

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


@dataclass
class Basis_Model_param:
    """Parameters for gaussian basis function."""

    mu: np.ndarray
    sigma: np.ndarray
    w: np.ndarray
    b: float


class BasisModel(nnx.Module):
    """A Flax NNX module for a basis model."""

    def __init__(self, *, rngs: nnx.Rngs, num_basis_ftn: int):
        self.num_basis_ftn = num_basis_ftn
        key = rngs.params()
        self.mu = nnx.Param(jax.random.uniform(key, (self.num_basis_ftn, 1)))
        self.sigma = nnx.Param(jax.random.normal(key, (self.num_basis_ftn, 1)))
        self.w = nnx.Param(
            jax.random.normal(key, (self.num_basis_ftn, 1)) * 0.2
        )  # initial weight is less using *0.2
        self.b = nnx.Param(jnp.zeros((1, 1)))

    def __call__(self, x: jax.Array) -> jax.Array:
        """Predicts the output for a given input."""
        # compute gaussian basis ftn
        gauss_basis = jnp.exp(-((x - self.mu.value.T) ** 2) / self.sigma.value.T
**2)

        return jnp.squeeze(gauss_basis @ self.w.value + self.b.value)

    @property
    def model(self) -> Basis_Model_param:
        """Returns the model parameters."""
        return Basis_Model_param(
            mu=np.array(self.mu.value).reshape([self.num_basis_ftn]),
            sigma=np.array(self.sigma.value).reshape([self.num_basis_ftn]),
            w=np.array(self.w.value).reshape([self.num_basis_ftn]),
            b=np.array(self.b.value).squeeze(),
        )

    @property
    def model_params(self) -> Basis_Model_param:
        """Return the model parameters."""
        return Basis_Model_param(
            mu=np.array(self.mu.value).reshape([self.num_basis_ftn]),
            sigma=np.array(self.sigma.value).reshape([self.num_basis_ftn]),
            w=np.array(self.w.value).reshape([self.num_basis_ftn]),
            b=np.array(self.b.value).squeeze(),
        )
```

```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 BasisModel

log = structlog.get_logger()


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

    def loss_fn(model: BasisModel):
        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: BasisModel,
    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 jax.numpy as jnp
import numpy as np
import structlog

from .config import PlottingSettings
from .data import Data
from .model import Basis_Model_param, BasisModel

log = structlog.get_logger()

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

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


def compare_linear_models(a: Basis_Model_param, b: Basis_Model_param):
    """Prints a comparison of two linear models."""
    log.info("Comparing models", true=a, estimated=b)
    print("w,  w_hat")
    for w_a, w_b in zip(a.w, b.w):
        print(f"{w_a:0.2f}, {w_b:0.2f}")

    print(f"{a.b:0.2f}, {b.b:0.2f}")


def plot_fit(
    model: BasisModel,
    data: Data,
    settings: PlottingSettings,
):
    """Plots the linear fit and saves it to a file."""
    log.info("Plotting fit")
    fig, ax = plt.subplots(1, 1, figsize=settings.figsize, dpi=settings.dpi)

    ax.set_title("Linear fit")
    ax.set_xlabel("x")
    ax.set_ylim(-np.amax(data.y) * 1.5, np.amax(data.y) * 1.5)
    h = ax.set_ylabel("y", labelpad=10)
    h.set_rotation(0)

    xs = np.linspace(0, 1, 100)
    xs = xs[:, np.newaxis]
    ax.plot(xs, np.squeeze(model(jnp.asarray(xs))), '-', label='Estimated function')
    ax.plot(np.squeeze(data.x), data.y, 'o')
    ax.plot(xs, np.sin(2 * np.pi * xs), '--', label='Sine wave')

    plt.tight_layout()

    settings.output_dir.mkdir(parents=True, exist_ok=True)
    output_path = settings.output_dir / "hw01_fit.pdf"
    plt.savefig(output_path)
    log.info("Saved plot", path=str(output_path))


def base_fit(
    model: BasisModel,
    data: Data,
    settings: PlottingSettings,
    num_basis_ftn: int,
):
    """Plots the basis ftn and saves it to a file."""
    log.info("Plotting fit")
    fig2, ax2 = plt.subplots(1, 1, figsize=settings.figsize, dpi=settings.dpi)
```

```python
    ax2.set_title("Base for fit")
    ax2.set_xlabel("x")
    ax2.set_ylim(0, np.amax(data.y))
    h = ax2.set_ylabel("y", labelpad=10)
    h.set_rotation(0)

    x = np.linspace(-1, 1, 100)

    params = model.model_params
    for i in range(num_basis_ftn):
        ax2.plot(x, jnp.exp(-((x - params.mu[i]) ** 2) / (params.sigma[i] ** 2))
)

    plt.tight_layout()
    settings.output_dir.mkdir(parents=True, exist_ok=True)
    output_path = settings.output_dir / "hw01_basis.pdf"
    plt.savefig(output_path)
    log.info("Saved plot", path=str(output_path))
```

```
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("hw01").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,
    )
```

```
from dataclasses import InitVar, dataclass, field

import numpy as np

from .model import Basis_Model_param


@dataclass
class Data:
    """Handles generation of synthetic data for linear regression."""

    model: Basis_Model_param
    rng: InitVar[np.random.Generator]
    num_features: int
    num_basis_ftn: int
    num_samples: int
    sigma: 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 based on the model."""
        self.index = np.arange(self.num_samples)
        self.x = rng.uniform(0, 1, size=(self.num_samples, self.num_features))
        clean_y = np.sin(2 * np.pi * self.x) + self.model.b  # bias=sigma gauss
noise
        self.y = rng.normal(loc=clean_y, scale=self.sigma)

    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 = "hw01"
version = "0.1.0"
description = " linear regression of a noisy sine wave using a set of Gaussian b
asis functions."
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",
]

[project.scripts]
hw01 = "hw01:main"

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

[tool.setuptools.package-data]
"hw01" = ["config.toml"]
```