```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 NNXGaussianModel

log = structlog.get_logger()


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

    def loss_fn(model: NNXGaussianModel):
        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: NNXGaussianModel,
    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)
        # log.debug("Y value", y=y)

        loss = train_step(model, optimizer, x, y)
        # log.debug("Training step", step=i, loss=loss)
        bar.set_description(f"Loss @ {i} => {loss:.6f}")
        bar.refresh()
    log.info("Training finished")
```

```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 NNXGaussianModel
from .training import train
from .plotting import plot_fit


def main() -> None:
    """CLI entry point."""
    settings = load_settings()
    configure_logging()
    log = structlog.get_logger()
    log.info("Hello from hw01!")
    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))

    data = Data(
        rng=np_rng,
        num_features=settings.data.num_features,
        num_samples=settings.data.num_samples,
        sigma=settings.data.sigma_noise,
    )
    # log.debug("Data generated", x=data.x, y=data.y)

    model = NNXGaussianModel(
        rngs=nnx.Rngs(params=model_key), num_features=settings.data.num_features
    )

    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)

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

```python
from dataclasses import dataclass

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

log = structlog.get_logger()


@dataclass
class GaussianModel:
    """Represents a simple gaussian model."""

    weights: np.ndarray
    mean: np.ndarray
    sd: np.ndarray
    bias: float


class NNXGaussianModel(nnx.Module):
    """A Flax NNX module for a gaussian regression model."""

    def __init__(self, *, rngs: nnx.Rngs, num_features: int):
        self.num_features = num_features
        key = rngs.params()
        self.w = nnx.Param(jax.random.normal(key, (self.num_features, 1)))  # We
ights
        self.mu = nnx.Param(jnp.linspace(0, 1, self.num_features))  # Mean
        # log.debug("Initialized mu", mu=self.mu.value)
        self.sigma = nnx.Param(jnp.array([0.1] * self.num_features))  # SD
        self.b = nnx.Param(jnp.zeros((1, 1)))  # Bias

    def __call__(self, x: jax.Array) -> jax.Array:
        """Predicts the output array y_hat for given input array x."""
        phi = jnp.exp(
            -((x - self.mu.value) ** 2) / (self.sigma.value**2)
        )  # exp((-(x - mu)^2) / sigma^2)
        y_hat = phi @ self.w.value + self.b.value
        return jnp.squeeze(y_hat)

    @property
    def model(self) -> GaussianModel:
        """Returns the underlying simple gaussian model."""
        return GaussianModel(
            weights=np.array(self.w.value).reshape([self.num_features]),
            mean=np.array(self.mu.value).reshape([self.num_features]),
            sd=np.array(self.sigma.value).reshape([self.num_features]),
            bias=np.array(self.b.value).squeeze(),
        )
```

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

import numpy as np


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

    rng: InitVar[np.random.Generator]
    num_features: 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 y = sin(2*pi*x)."""
        self.index = np.arange(self.num_samples)
        self.x = rng.uniform(0.1, 0.9, size=(self.num_samples, 1))
        clean_y = np.sin(self.x * 2 * np.pi)
        noise = rng.normal(0, self.sigma, size=clean_y.shape)
        noisy_y = clean_y + noise
        self.y = noisy_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].flatten()
```

```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("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.RichTracebackFormatter()
            ),
        ],
        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 typing import Tuple

from pydantic import BaseModel
from pydantic_settings import BaseSettings


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

    num_features: 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 = 0.05


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

    figsize: Tuple[int, int] = (10, 3)
    dpi: int = 200
    output_dir: Path = Path("hw01/artifacts")


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

    debug: bool = False
    random_seed: int = 31415
    data: DataSettings = DataSettings()
    training: TrainingSettings = TrainingSettings()
    plotting: PlottingSettings = PlottingSettings()


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

```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 NNXGaussianModel

log = structlog.get_logger()

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

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


def plot_fit(
    model: NNXGaussianModel,
    data: Data,
    settings: PlottingSettings,
):
    """Plots the fit and saves it to a file."""
    log.info("Plotting fit")
    fig, ax = plt.subplots(1, 2, figsize=settings.figsize, dpi=settings.dpi)

    ax[0].set_title("Fit")
    ax[0].set_xlabel("x")
    ax[0].set_ylim(np.amin(data.y) * 1.5, np.amax(data.y) * 1.5)
    h = ax[0].set_ylabel("y", labelpad=10)
    h.set_rotation(0)

    ax[1].set_title("Bases for Fit")
    ax[1].set_xlabel("x")
    ax[1].set_ylim(0, np.amax(data.y))
    h = ax[1].set_ylabel("y", labelpad=10)
    h.set_rotation(0)

    xs = np.linspace(0, 1, 1000)
    xs = xs[:, np.newaxis]
    ax[0].plot(
        xs,
        np.squeeze(model(jnp.asarray(xs))),
        "--",
        xs,
        np.sin(xs * 2 * np.pi),
        "-",
        np.squeeze(data.x),
        data.y,
        "o",
    )
    # log.info("y", y=np.squeeze(data.y), y_size=data.y.shape)

    for i in range(model.num_features):
        phi = np.exp(-((xs - model.mu.value[i]) ** 2) / (model.sigma.value[i] **
 2))
        # log.debug("Basis function", i=i, mu=model.mu.value[i], sigma=model.sig
ma.value[i])
        ax[1].plot(xs, phi, label=f"Basis {i + 1}")
    ax[1].legend(loc="best", fontsize=8)

    plt.tight_layout()

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



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

```
# pyproject.toml.jinja

[project]
name = "hw01"
version = "0.1.0"
description = "Linear regression of a noisy sine wave using a set of Gaussian ba
sis function with learned location and scale parameters."
readme = "README.md"
authors = [
    { name = "Donghyun Park", email = "donghyun.park@cooper.edu" }
]
requires-python = ">=3.13"
dependencies = [
    "structlog",
    "rich",
    "numpy",
    "jax",
    "flax",
    "tqdm>=4.67.1",
    "matplotlib>=3.10.6",
    "pydantic>=2.11.7",
    "pydantic-settings>=2.10.1",
]

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

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

```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 NNXGaussianModel

log = structlog.get_logger()


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

    def loss_fn(model: NNXGaussianModel):
        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: NNXGaussianModel,
    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)
        # log.debug("Y value", y=y)

        loss = train_step(model, optimizer, x, y)
        # log.debug("Training step", step=i, loss=loss)
        bar.set_description(f"Loss @ {i} => {loss:.6f}")
        bar.refresh()
    log.info("Training finished")
```

```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 NNXGaussianModel
from .training import train
from .plotting import plot_fit


def main() -> None:
    """CLI entry point."""
    settings = load_settings()
    configure_logging()
    log = structlog.get_logger()
    log.info("Hello from hw01!")
    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))

    data = Data(
        rng=np_rng,
        num_features=settings.data.num_features,
        num_samples=settings.data.num_samples,
        sigma=settings.data.sigma_noise,
    )
    # log.debug("Data generated", x=data.x, y=data.y)

    model = NNXGaussianModel(
        rngs=nnx.Rngs(params=model_key), num_features=settings.data.num_features
    )

    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)

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

```python
from dataclasses import dataclass

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

log = structlog.get_logger()


@dataclass
class GaussianModel:
    """Represents a simple gaussian model."""

    weights: np.ndarray
    mean: np.ndarray
    sd: np.ndarray
    bias: float


class NNXGaussianModel(nnx.Module):
    """A Flax NNX module for a gaussian regression model."""

    def __init__(self, *, rngs: nnx.Rngs, num_features: int):
        self.num_features = num_features
        key = rngs.params()
        self.w = nnx.Param(jax.random.normal(key, (self.num_features, 1)))  # We
ights
        self.mu = nnx.Param(jnp.linspace(0, 1, self.num_features))  # Mean
        # log.debug("Initialized mu", mu=self.mu.value)
        self.sigma = nnx.Param(jnp.array([0.1] * self.num_features))  # SD
        self.b = nnx.Param(jnp.zeros((1, 1)))  # Bias

    def __call__(self, x: jax.Array) -> jax.Array:
        """Predicts the output array y_hat for given input array x."""
        phi = jnp.exp(
            -((x - self.mu.value) ** 2) / (self.sigma.value**2)
        )  # exp((-(x - mu)^2) / sigma^2)
        y_hat = phi @ self.w.value + self.b.value
        return jnp.squeeze(y_hat)

    @property
    def model(self) -> GaussianModel:
        """Returns the underlying simple gaussian model."""
        return GaussianModel(
            weights=np.array(self.w.value).reshape([self.num_features]),
            mean=np.array(self.mu.value).reshape([self.num_features]),
            sd=np.array(self.sigma.value).reshape([self.num_features]),
            bias=np.array(self.b.value).squeeze(),
        )
```

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

import numpy as np


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

    rng: InitVar[np.random.Generator]
    num_features: 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 y = sin(2*pi*x)."""
        self.index = np.arange(self.num_samples)
        self.x = rng.uniform(0.1, 0.9, size=(self.num_samples, 1))
        clean_y = np.sin(self.x * 2 * np.pi)
        noise = rng.normal(0, self.sigma, size=clean_y.shape)
        noisy_y = clean_y + noise
        self.y = noisy_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].flatten()
```

```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("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.RichTracebackFormatter()
            ),
        ],
        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 typing import Tuple

from pydantic import BaseModel
from pydantic_settings import BaseSettings


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

    num_features: 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 = 0.05


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

    figsize: Tuple[int, int] = (10, 3)
    dpi: int = 200
    output_dir: Path = Path("hw01/artifacts")


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

    debug: bool = False
    random_seed: int = 31415
    data: DataSettings = DataSettings()
    training: TrainingSettings = TrainingSettings()
    plotting: PlottingSettings = PlottingSettings()


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

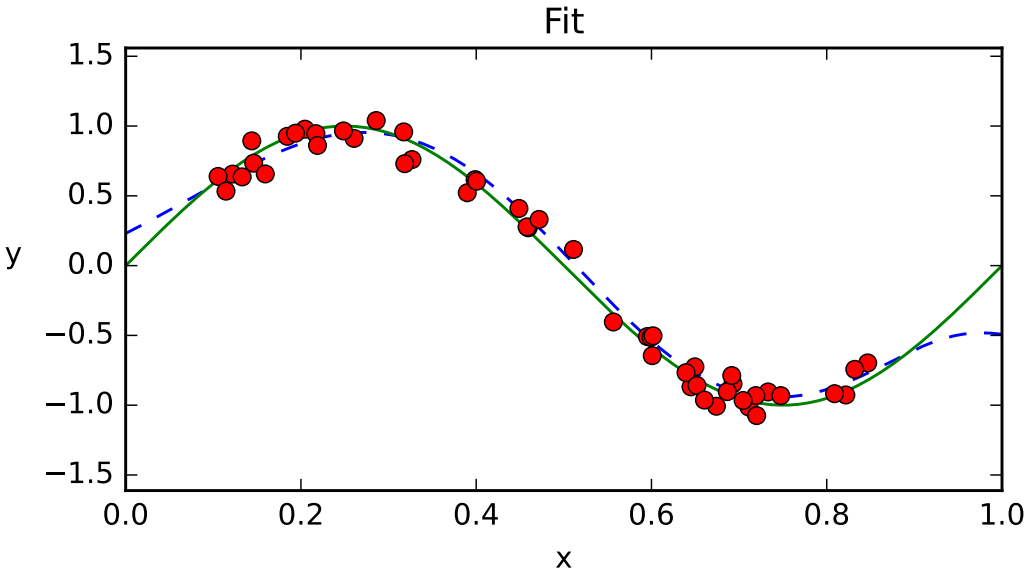
```python
import matplotlib
import matplotlib.pyplot as plt
import jax.numpy as jnp
import numpy as np
import structlog

from .config import PlottingSettings
from .data import Data
from .model import NNXGaussianModel

log = structlog.get_logger()

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

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


def plot_fit(
    model: NNXGaussianModel,
    data: Data,
    settings: PlottingSettings,
):
    """Plots the fit and saves it to a file."""
    log.info("Plotting fit")
    fig, ax = plt.subplots(1, 2, figsize=settings.figsize, dpi=settings.dpi)

    ax[0].set_title("Fit")
    ax[0].set_xlabel("x")
    ax[0].set_ylim(np.amin(data.y) * 1.5, np.amax(data.y) * 1.5)
    h = ax[0].set_ylabel("y", labelpad=10)
    h.set_rotation(0)

    ax[1].set_title("Bases for Fit")
    ax[1].set_xlabel("x")
    ax[1].set_ylim(0, np.amax(data.y))
    h = ax[1].set_ylabel("y", labelpad=10)
    h.set_rotation(0)

    xs = np.linspace(0, 1, 1000)
    xs = xs[:, np.newaxis]
    ax[0].plot(
        xs,
        np.squeeze(model(jnp.asarray(xs))),
        "--",
        xs,
        np.sin(xs * 2 * np.pi),
        "-",
        np.squeeze(data.x),
        data.y,
        "o",
    )
    # log.info("y", y=np.squeeze(data.y), y_size=data.y.shape)

    for i in range(model.num_features):
        phi = np.exp(-((xs - model.mu.value[i]) ** 2) / (model.sigma.value[i] **
 2))
        # log.debug("Basis function", i=i, mu=model.mu.value[i], sigma=model.sig
ma.value[i])
        ax[1].plot(xs, phi, label=f"Basis {i + 1}")
    ax[1].legend(loc="best", fontsize=8)

    plt.tight_layout()

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

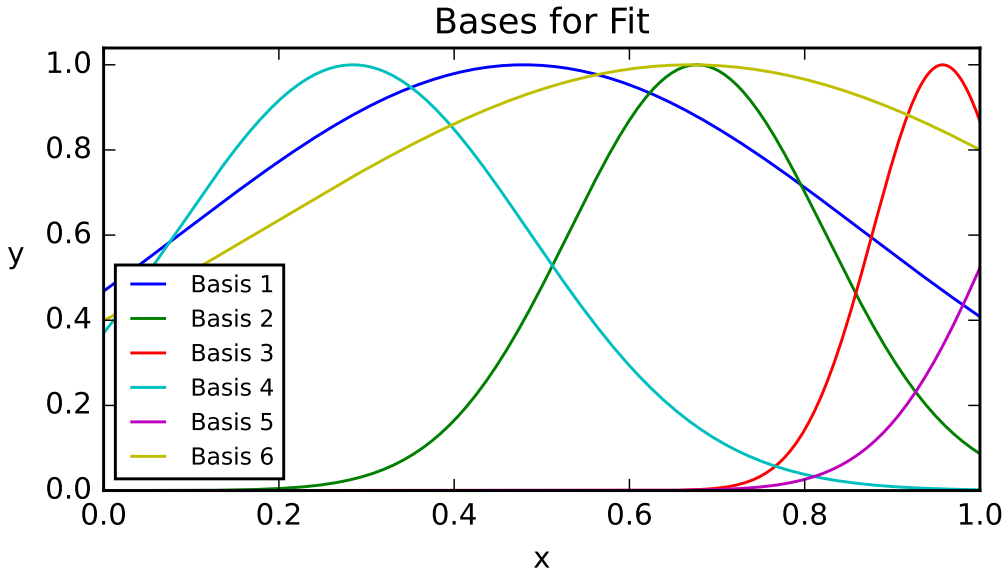


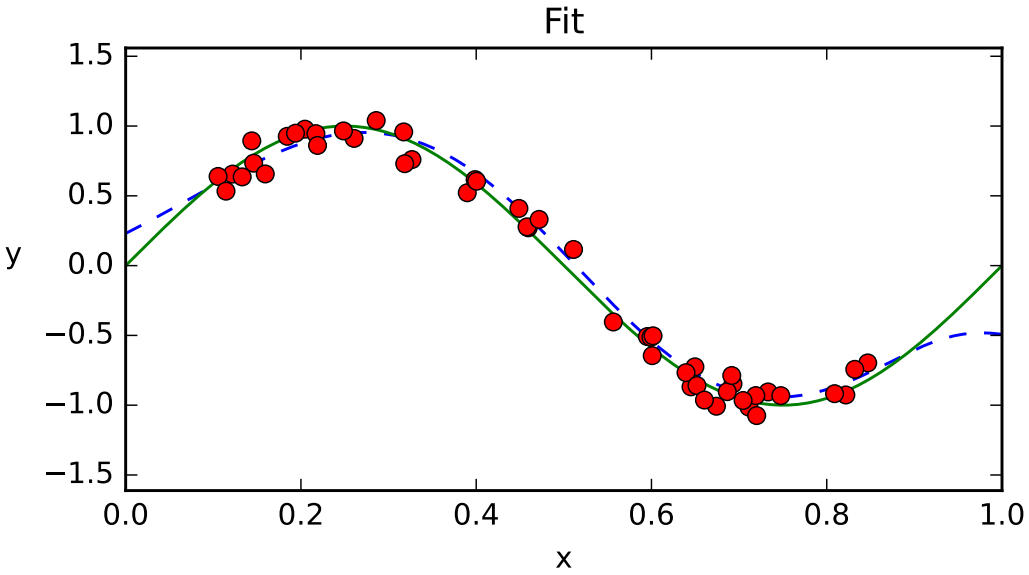
```python
    plt.savefig(output_path)
    log.info("Saved plot", path=str(output_path))
```

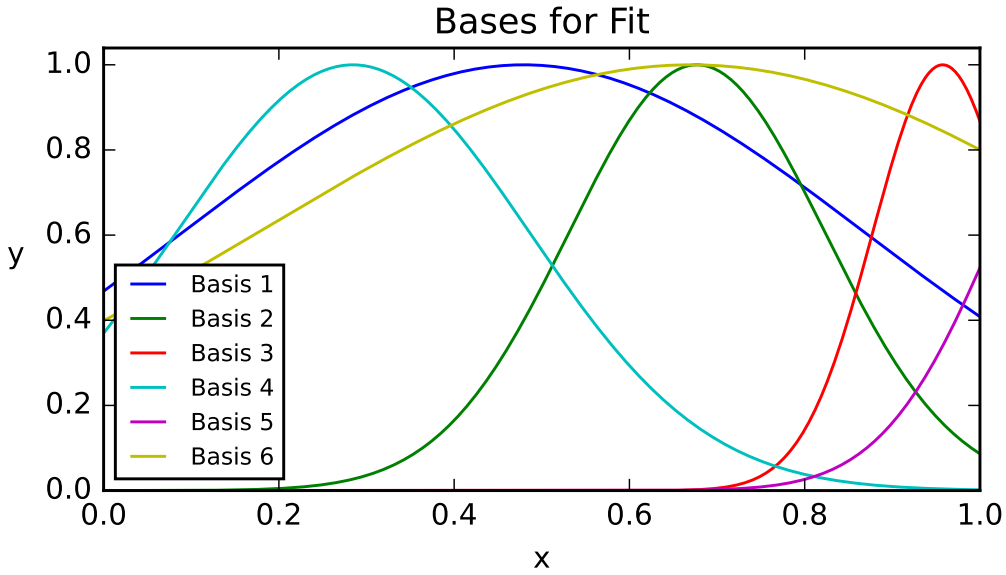
```
debug = false
random_seed = 0xc61dd8fc

[data]
num_features = 1
num_samples = 50
sigma_noise = 0.5

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

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

```
# pyproject.toml.jinja

[project]
name = "hw01"
version = "0.1.0"
description = "Linear regression of a noisy sine wave using a set of Gaussian ba
sis function with learned location and scale parameters."
readme = "README.md"
authors = [
    { name = "Donghyun Park", email = "donghyun.park@cooper.edu" }
]
requires-python = ">=3.13"
dependencies = [
    "structlog",
    "rich",
    "numpy",
    "jax",
    "flax",
    "tqdm>=4.67.1",
    "matplotlib>=3.10.6",
    "pydantic>=2.11.7",
    "pydantic-settings>=2.10.1",
]

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

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


Fit
y
x
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0.0
0.2
0.4
0.6
0.8
1.0


Bases for Fit
y
x
Basis 1
Basis 2
Basis 3
Basis 4
Basis 5
Basis 6
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


Fit
y
x
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0.0
0.2
0.4
0.6
0.8
1.0


Bases for Fit
y
x
Basis 1
Basis 2
Basis 3
Basis 4
Basis 5
Basis 6
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0