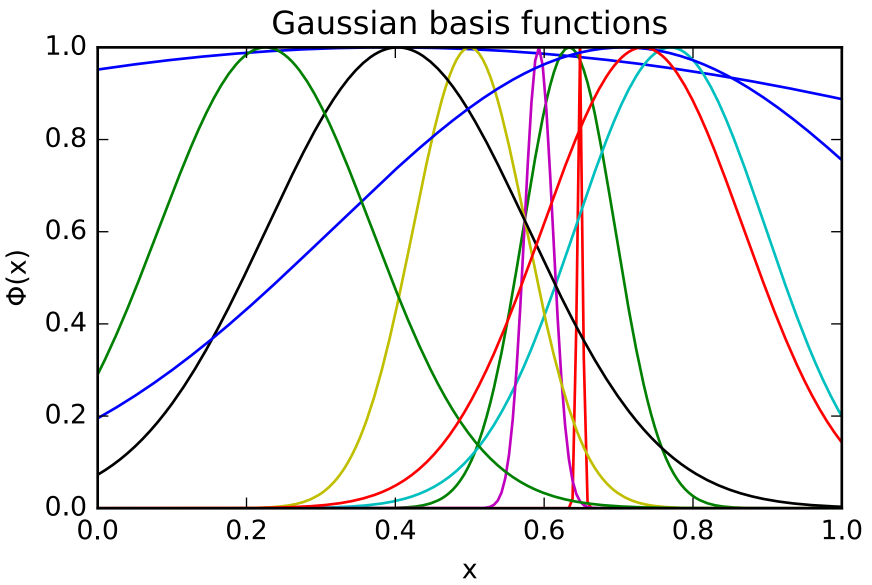
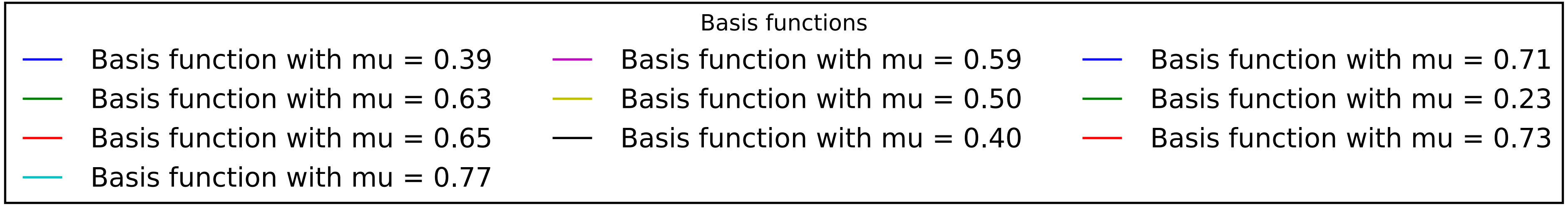
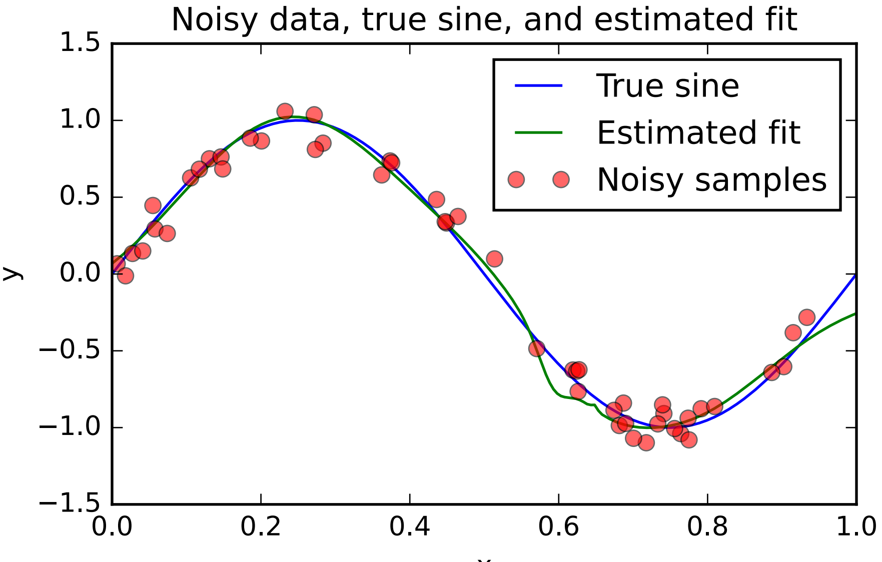

Gaussian basis functions
Φ(x)
x
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


Basis functions
Basis function with mu = 0.39
Basis function with mu = 0.63
Basis function with mu = 0.65
Basis function with mu = 0.77
Basis function with mu = 0.59
Basis function with mu = 0.50
Basis function with mu = 0.40
Basis function with mu = 0.71
Basis function with mu = 0.23
Basis function with mu = 0.73


Noisy data, true sine, and estimated fit
True sine
Estimated fit
Noisy samples
y
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

```
# pyproject.toml.jinja

[project]
name = "hw01"
version = "0.1.0"
description = "Linear regression"
readme = "README.md"
authors = [
    { name = "wongee (freddy) hong", email = "wongee.hong@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"
```

```
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 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 ModelSettings(BaseModel):
    """Settings for the basis expansion model."""

    num_basis: int = 10

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

    num_features: int = 1
    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
    model: ModelSettings = ModelSettings()
    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),
            env_settings,
            dotenv_settings,
            file_secret_settings,
        )


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

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

from .config import PlottingSettings
from .data import Data
from .model import BasisExpansionModel, TrueSineModel, NNXBasisExpansionModel

log = structlog.get_logger()

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

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


def compare_models(true_model: TrueSineModel, est_model: NNXBasisExpansionModel,
 data: Data, settings: PlottingSettings):
    """Plots true sine function vs estimated regression model."""
    xs = np.linspace(0, 1, 200)

    # true sine
    ys_true = true_model(xs)

    # estimated model
    ys_est = est_model(jnp.asarray(xs[:, None]))

    fig, ax = plt.subplots(1, 1, figsize=settings.figsize, dpi=settings.dpi)
    ax.set_title("Noisy data, true sine, and estimated fit")
    ax.plot(xs, ys_true, label="True sine")
    ax.plot(xs, ys_est, label="Estimated fit")
    ax.plot(data.x.squeeze(), data.y, "o", label="Noisy samples", alpha=0.6)
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.legend()

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

def plot_basis_functions(model: NNXBasisExpansionModel, settings: PlottingSettin
gs):
    xs = np.linspace(0, 1, 200)
    phi_matrix = []
    fig, ax = plt.subplots(figsize=settings.figsize, dpi=settings.dpi)
    mu_sig = jax.nn.sigmoid(model.mu.value)
    for mu, sigma in zip(mu_sig, model.sigma.value):
        mu = float(mu.item())
        phi = np.exp(-((xs - mu)**2) / (sigma**2))
        ax.plot(xs, phi, label=f"Basis function with mu = {mu:.2f}")
        phi_matrix.append(phi)

    ax.set_title("Gaussian basis functions")
    ax.set_xlabel("x")
    ax.set_ylabel("Î¦(x)")
    ax.legend(
    title="Basis functions",
    loc="upper center",
    bbox_to_anchor=(0.5, -0.15),
    ncol=3,
)
```

```python
    settings.output_dir.mkdir(parents=True, exist_ok=True)
    plt.savefig(settings.output_dir / "basis_functions.pdf", bbox_inches="tight")
```

```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 TrueSineModel, NNXBasisExpansionModel
from .plotting import compare_models, plot_basis_functions
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())

    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_generating_model = TrueSineModel(sigma=settings.data.sigma_noise)

    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_samples=settings.data.num_samples,
        sigma=settings.data.sigma_noise,
    )

    model = NNXBasisExpansionModel(
        rngs=nnx.Rngs(params=model_key), num_basis=settings.model.num_basis
    )
    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_models(data_generating_model, model, data, settings.plotting)

    if settings.data.num_features == 1:
        plot_basis_functions(model, settings.plotting)
    else:
        log.info("Skipping plotting for multi−feature models. Check feature number setting from config file"
)
```

```python
from dataclasses import dataclass

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

@dataclass
class BasisExpansionModel:
    """Represents a simple basis expansion model."""

    weights: np.ndarray
    sigma: np.ndarray
    mu: np.ndarray
    bias: float

@dataclass
class TrueSineModel:
    sigma: float

    def __call__(self, x: np.ndarray) -> np.ndarray:
        return np.sin(2 * np.pi * x)


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

    def __init__(self, *, rngs:nnx.Rngs, num_basis: int):
        self.num_basis = num_basis
        key = rngs.params()
        mu_key, w_key = jax.random.split(key) # splitting key to assign differen
t key value for mu and w
        self.mu = nnx.Param(jax.random.uniform(mu_key, (num_basis, 1)))
        self.sigma = nnx.Param(jnp.full((num_basis, 1), 0.1))
        self.w = nnx.Param(jax.random.normal(w_key, (num_basis, 1)))
        self.b = nnx.Param(jnp.zeros((1,1)))

    def __call__(self, x: jax.Array) -> jax.Array:
        mu = jax.nn.sigmoid(self.mu.value) # setting mu value within [0,1] range
        diff = x - mu.T
        phi = jnp.exp(-(diff**2) / (self.sigma.value.T**2))
        y_hat = phi @ self.w.value + self.b.value
        return jnp.squeeze(y_hat)

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

```
debug = false
random_seed = 31415

[model]
num_basis = 10

[data]
num_features = 1
num_samples = 50
sigma_noise = 0.1

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

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

```
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 NNXBasisExpansionModel

log = structlog.get_logger()

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

    def loss_fn(model: NNXBasisExpansionModel):
        y_hat = model(x)
        return 0.5 * jnp.mean((y - y_hat) ** 2) # Returning mean of the differen
ce to handle loss in batch of samples

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


def train(
    model: NNXBasisExpansionModel,
    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
from dataclasses import InitVar, dataclass, field

import numpy as np

from .model import BasisExpansionModel


@dataclass
class Data:
    """Handles generation of synthetic data for basis expansion regression on sine data."""

    model: BasisExpansionModel
    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 the model."""
        self.index = np.arange(self.num_samples)
        self.x = rng.uniform(0.0, 1.0, size=(self.num_samples, self.num_features
))
        epsilon = rng.normal(0.0, self.sigma, size=(self.num_samples, self.num_f
eatures))
        self.y = np.sin(2*np.pi*self.x) + epsilon

    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()
```