```
[project]
name = "hw01"
version = "0.1.0"
description = "Linear regression with Gaussian basis functions. Learned paramete
rs through stochastic gradient descent"
readme = "README.md"
authors = [
  { name = "Josh Miao", email = "joshuamiao03@gmail.com" }
]
requires-python = ">=3.10,<4.0"
dependencies = [
  "structlog",
  "rich",
  "numpy",
  "jax",
  "flax",
  "optax",
  "matplotlib>=3.10.6,<4.0",
  "nnx>=0.0.8",
  "pydantic>=2.11.7",
  "pydantic-settings>=2.10.1",
  "ghostscript>=0.8.1",
]

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

[build-system]
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"

[tool.setuptools.package-dir]
"" = "src"

[tool.setuptools.packages.find]
where = ["src"]
include = ["hw01*"]
```

```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 (sin wave)."""

    num_features: int = 1
    num_samples: int = 50
    sigma_noise: float = 0.1


class ModelSettings(BaseModel):
    """Model architecture/hyperparameters."""

    num_basis: int = 9  # settled on 9 through trial and errror
    init_basis_sigma: float = 0.05  # originally .1 but settled on .05


class TrainingSettings(BaseModel):
    """Settings for model training."""

    batch_size: int = 16
    num_iters: int = 1000
    learning_rate: float = 0.05


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

    figsize: Tuple[int, int] = (6, 4)
    dpi: int = 200
    output_dir: Path = Path("output_graphs")


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

    debug: bool = False
    random_seed: int = 472  # seed = 472 for ECE472 lol
    data: DataSettings = DataSettings()
    model: ModelSettings = ModelSettings()
    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, ...]:
        """
```

```python
        Use TOML file as a primary source, then env, then dotenv.
        """
        return (
            init_settings,
            TomlConfigSettingsSource(settings_cls),
            env_settings,
            dotenv_settings,
            file_secret_settings,
        )


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

```python
from dataclasses import InitVar, dataclass, field
import numpy as np
import structlog
from typing import Tuple

log = structlog.get_logger()


@dataclass
class Data:
    """
  Generate noisy sine data:
    x ~ Uniform(0, 1)
    y = sin(2*pi*x) + Normal(0, sigma_noise)
  """

    rng: InitVar[np.random.Generator]
    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):
        log.info("Generating data", num_samples=self.num_samples, sigma=self.sigma)
        self.index = np.arange(self.num_samples)
        self.x = rng.uniform(0.0, 1.0, size=(self.num_samples, 1))
        clean_y = np.sin(2.0 * np.pi * self.x)
        self.y = rng.normal(loc=clean_y, scale=self.sigma).flatten()
        log.debug(
            "Sample of generated data",
            x_sample=self.x[:5].tolist(),
            y_sample=self.y[:5].tolist(),
        )

    def get_batch(
        self, rng: np.random.Generator, batch_size: int
    ) -> Tuple[np.ndarray, np.ndarray]:
        """
    Select a random batch (with replacement if batch_size > num_samples).
    Returns (x_batch, y_batch) where x_batch shape = (batch_size, 1), y_batch = (batch_size,)
    """
        if batch_size <= 0:
            raise ValueError("batch_size must be positive")
            log.critical("batch size was negative somehow", batch_size=batch_size)
        replace = batch_size > self.num_samples
        """
    get batch_size samples from self and replace once batch_size exceeds num_samples
    """
        choices = rng.choice(self.index, size=batch_size, replace=replace)
        x_batch = self.x[choices]
        y_batch = self.y[choices]
        log.debug("Selected training batch", batch_size=batch_size, replace=replace)
        return x_batch, y_batch
```

```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.RichTracebackForm
atter()
            ),
        ],
        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
import numpy as np
import structlog
from flax import nnx
from dataclasses import dataclass
from typing import Dict

log = structlog.get_logger()


@dataclass
class Parameters:
    """Container for logging / inspection of learned parameters."""

    mus: np.ndarray
    sigmas: np.ndarray
    weights: np.ndarray
    bias: float


class Linear(nnx.Module):
    """
    maps M basis outputs to scalaras
    """

    def __init__(self, *, rngs: nnx.Rngs, num_basis: int):
        log.info("Initializing Linear module", num_basis=num_basis)
        key = rngs.params()
        # small initialization scale
        self.w = nnx.Param(0.1 * jax.random.normal(key, (num_basis,)))
        """
        for the longest time i thought jnp.zeros() was sufficient to create a scalar.
        i didnt know i had to do jnp.zeros(()) ðM–^_M–^X-ðM–^_M–^X-ðM–^_M–^X-
        and yes i copy pasted this emoji just to emphasize my struggle
        """
        self.b = nnx.Param(jnp.zeros(()))
        log.debug(
            "Linear params initialized",
            w_shape=self.w.value.shape,
            b_shape=self.b.value.shape,
        )

    def __call__(self, phi: jax.Array) -> jax.Array:
        """
        phi shape: (batch, M) -> returns (batch,)
        """
        return jnp.dot(phi, self.w.value) + self.b.value

    # learned what decorators and warppers are today are you proud professor?
    @property
    def model(self) -> Dict[str, jnp.ndarray]:
        return {"w": self.w.value, "b": self.b.value}


class BasisExpansion(nnx.Module):
    """
    Gaussian basis functions with learnable centers mus and sigmas.
    phi_j(x | mu_j, sigma_j) = exp(-(x - mu_j)^2 / sigma_j^2)
    """

    def __init__(self, *, rngs: nnx.Rngs, num_basis: int, init_sigma: float = 0.
1):
        log.info(
            "Initializing BasisExpansion", num_basis=num_basis, init_sigma=init_sigma
        )
        key = rngs.params()
        # split to get reproducible mu and sigma inits
        k1, k2 = jax.random.split(key, 2)
```

```
        # initialize mu evenly across domain [0,1] with small noise
        mu_init = jnp.linspace(0.0, 1.0, num_basis)
        mu_init = mu_init + 0.01 * jax.random.normal(
            k1, (num_basis,)
        )  # small perturbation  to vary basis functions initially
        self.mu = nnx.Param(mu_init)

        # lets convert sigma to log in case it gets too small or negative
        log_sigma_init = jnp.log(jnp.ones(num_basis) * 0.05)  # smaller spread
        self.log_sigma = nnx.Param(log_sigma_init)

        # idk how this debug statement got this long. it was revolutionary when
i realized that i could put if statements in here
        log.debug(
            "Basis params initialized",
            mu_init=mu_init.tolist()
            if isinstance(mu_init, jnp.ndarray)
            else str(mu_init),
            log_sigma_init=log_sigma_init.tolist()
            if isinstance(log_sigma_init, jnp.ndarray)
            else str(log_sigma_init),
        )

    def __call__(self, x: jax.Array) -> jax.Array:
        """
        Compute phi for inputs x.

        x: shape (batch, 1) or (batch,)
        returns phi: (batch, M)
        """
        x = jnp.squeeze(x)  # -> (batch,)
        log.debug("x squeezed:", shape=x.shape)
        x = x[:, None]  # -> (batch, 1)
        log.info("x shape:", shape=x.shape)
        mu = self.mu.value[None, :]  # -> (1, M)
        log.info("mu shape:", shape=mu.shape)
        # i think this is the right place to exponentiate and get rid of the log
 for sigma
        sigma = jnp.exp(self.log_sigma.value)
        sigma = sigma[None, :]
        log.info("sigma shape:", shape=sigma.shape)

        diff = x - mu  # (batch, M) i think
        phi = jnp.exp(-(diff**2) / (sigma**2))
        return phi

    @property
    def model(self) -> Dict[str, jnp.ndarray]:
        return {"mu": self.mu.value, "sigma": jnp.exp(self.log_sigma.value)}


class GaussianRegressor(nnx.Module):
    """
    build the regression model:
      y_hat(x) = sum_j w_j * phi_j(x | mu_j, sigma_j) + b
    """

    def __init__(self, *, rngs: nnx.Rngs, num_basis: int, init_sigma: float = 0.
1):
        log.info(
            "Initializing GaussianRegressor", num_basis=num_basis, init_sigma=init_sigma
        )
        key = rngs.params()
        # split RNG for submodules
        k_basis, k_linear = jax.random.split(key, 2)
        self.basis = BasisExpansion(
            rngs=nnx.Rngs(params=k_basis), num_basis=num_basis, init_sigma=init_
sigma
```

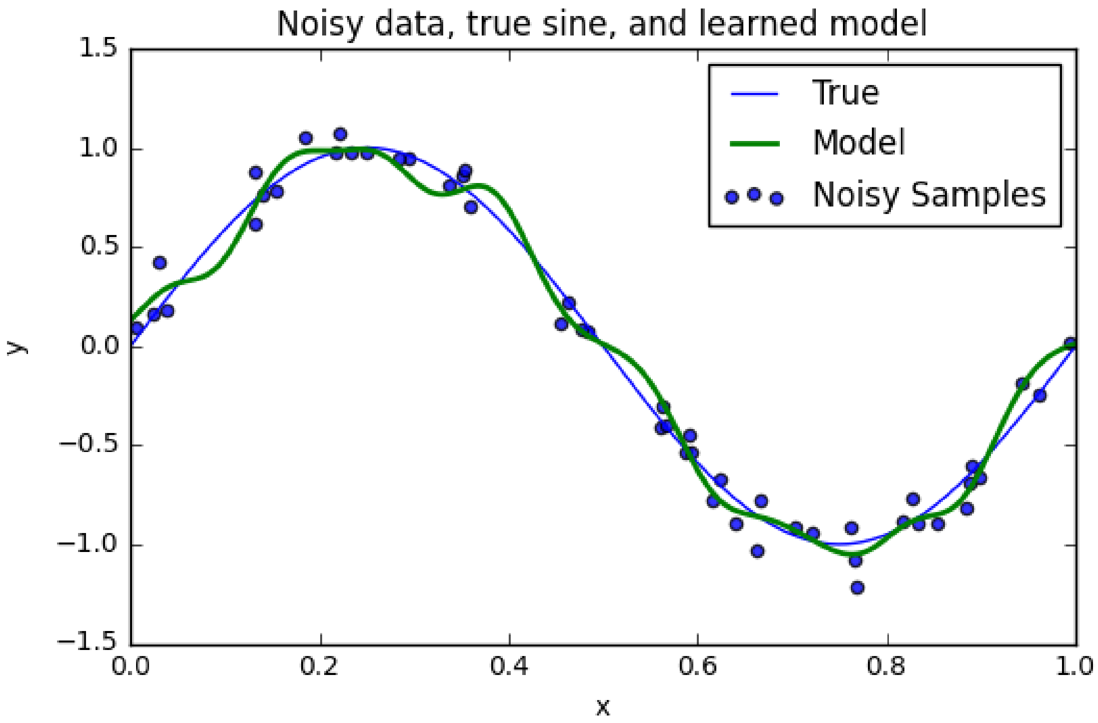
```
        )
        self.linear = Linear(rngs=nnx.Rngs(params=k_linear), num_basis=num_basis
)
        log.info("GaussianRegressor initialized")

    def __call__(self, x: jax.Array) -> jax.Array:
        """
    x: (batch, 1) -> returns (batch,)
    """
        phi = self.basis(x)
        return self.linear(phi)

    @property
    def model(self) -> Parameters:
        mus = np.array(self.basis.mu.value)
        sigmas = np.array(jnp.exp(self.basis.log_sigma.value))
        weights = np.array(self.linear.w.value)
        bias = float(np.array(self.linear.b.value).squeeze())
        return Parameters(mus=mus, sigmas=sigmas, weights=weights, bias=bias)
```

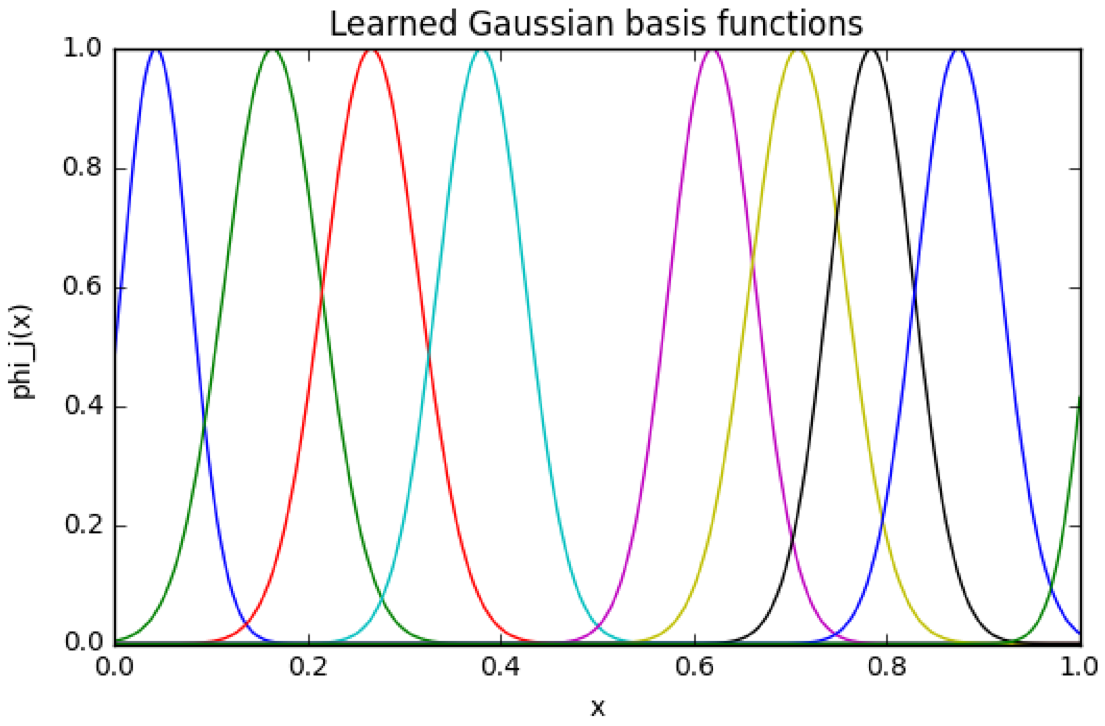
```python
import matplotlib
import matplotlib.pyplot as plt
import jax.numpy as jnp
import numpy as np
import structlog
from pathlib import Path

log = structlog.get_logger()
font = {"size": 10}
matplotlib.style.use("classic")
matplotlib.rc("font", **font)


def plot_fit(model: any, data: any, settings: any) -> None:
    """
    Plot the data points, the true noiseless sine, and the model
    """
    log.info("Plotting fit")
    xs = np.linspace(0.0, 1.0, 500)
    xs_jnp = jnp.asarray(xs)[:, None]
    y_pred = np.array(model(xs_jnp))
    y_true = np.sin(2.0 * np.pi * xs)

    fig, ax = plt.subplots(1, 1, figsize=settings.figsize, dpi=settings.dpi)
    ax.set_title("Noisy data, true sine, and learned model")
    ax.set_xlabel("x")
    ax.set_xlim(0.0, 1.0)
    ax.set_ylabel("y", labelpad=10)
    ax.plot(xs, y_true, "-", label="True")
    ax.plot(xs, y_pred, "-", linewidth=2, label="Model")
    ax.scatter(
        data.x.flatten(), data.y, marker="o", alpha=0.8, label="Noisy Samples"
    )
    ax.legend()
    plt.tight_layout()

    settings.output_dir.mkdir(parents=True, exist_ok=True)
    output_path = Path(settings.output_dir) / "fit.png"
    plt.savefig(output_path)
    plt.close(fig)
    log.info("Saved fit plot", path=str(output_path))


def plot_basis_functions(model: any, data: any, settings: any) -> None:
    """
    Plot each learned basis function across the input domain.
    """
    log.info("Plotting basis functions")
    xs = np.linspace(0.0, 1.0, 500)
    xs_jnp = jnp.asarray(xs)[:, None]
    # compute raw basis outputs (batch, M)
    phi = np.array(model.basis(xs_jnp))

    fig, ax = plt.subplots(1, 1, figsize=settings.figsize, dpi=settings.dpi)
    M = phi.shape[1]
    for j in range(M):
        ax.plot(xs, phi[:, j], label=f"phi_{j}")
    ax.set_title("Learned Gaussian basis functions")
    ax.set_xlabel("x")
    ax.set_ylabel("phi_j(x)", labelpad=10)
    # i dont think this plot needs a legend so im not gonna make one
    plt.tight_layout()

    settings.output_dir.mkdir(parents=True, exist_ok=True)
    output_path = Path(settings.output_dir) / "basis_functions.png"
    plt.savefig(output_path)
    plt.close(fig)
    log.info("Saved basis functions plot", path=str(output_path))
```

```
import jax.numpy as jnp
import numpy as np
import structlog
from flax import nnx
from tqdm import trange
from typing import Any

from .data import Data

log = structlog.get_logger()

def train_step(model: nnx.Module, optimizer: nnx.Optimizer, x: jnp.ndarray, y: j
np.ndarray) -> jnp.ndarray:
    """
    Performs a single training step using autodiff from nnx.
    Returns the scalar loss.
    """
    log.debug("Entering train_step", x_shape=x.shape, y_shape=y.shape)

    def loss_fn(model: nnx.Module):
        y_hat = model(x)
        return 0.5 * jnp.mean((y_hat - y) ** 2)
        """
        My basis functions are a little close together indicating redundancy and makes me afraid im overfitting.
        I want to introduce some sort of repulsive factor but I don’t know how and I kind of give up on trying.
        """

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


def train(
    model: nnx.Module,
    optimizer: nnx.Optimizer,
    data: Data,
    settings: Any,
    np_rng: np.random.Generator,
) -> None:
    """
    Train the provided model in−place using stochastic gradient descent.
    """
    log.info("Starting training", iters=settings.num_iters, batch_size=settings.batch_
size, lr=settings.learning_rate)
    bar = trange(settings.num_iters)
    for i in bar:
        x_np, y_np = data.get_batch(np_rng, settings.batch_size)
        x = jnp.asarray(x_np)
        y = jnp.asarray(y_np)
        loss = train_step(model, optimizer, x, y)
        bar.set_description(f"Loss @ {i} => {float(loss):.6f}")
        bar.refresh()
        if (i + 1) % max(1, settings.num_iters // 5) == 0:
            log.info("Intermediate training status", iter=i + 1, loss=float(loss))
    log.info("Training finished")
```

```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 GaussianRegressor
from .plotting import plot_fit, plot_basis_functions
from .training import train

log = structlog.get_logger()


def main() -> None:
    """CLI entry point for hw01 Gaussian−basis regression."""
    settings = load_settings()
    configure_logging()
    log.info("Settings loaded", settings=settings.model_dump())

    # JAX PRNG
    key = jax.random.PRNGKey(472)  # seed is 472 for ECE472 hehe
    data_key, model_key = jax.random.split(key)
    np_rng = np.random.default_rng(int(np.array(data_key)[0]))

    log.info(
        "Creating dataset",
        num_samples=settings.data.num_samples,
        sigma=settings.data.sigma_noise,
    )
    data = Data(
        rng=np_rng,
        num_samples=settings.data.num_samples,
        sigma=settings.data.sigma_noise,
    )

    # Build model
    log.info(
        "Creating model",
        num_basis=settings.model.num_basis,
        init_sigma=settings.model.init_basis_sigma,
    )
    model = GaussianRegressor(
        rngs=nnx.Rngs(params=model_key),
        num_basis=settings.model.num_basis,
        init_sigma=settings.model.init_basis_sigma,
    )
    log.debug("Initial model params:", model_params=model.model)

    # time to define how to optimize
    optimizer = nnx.Optimizer(
        model, optax.sgd(settings.training.learning_rate), wrt=nnx.Param
    )
    log.info("Optimizer created", optimizer="sgd", lr=settings.training.learning_rate
)

    # time to train
    train(model, optimizer, data, settings.training, np_rng)
    log.debug("Trained model params", model_params=model.model)

    # time to plot
    log.info("Generating plots")
    plot_fit(model, data, settings.plotting)
    plot_basis_functions(model, data, settings.plotting)
    log.info("plots generated in folder titled outputs")
    log.info("hw01 run complete!!!!!!")
```

```python

if __name__ == "__main__":
    main()
```


Noisy data, true sine, and learned model
True
Model
Noisy Samples
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


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