```
# pyproject.toml.jinja

[project]
name = "hw04"
version = "0.1.0"
description = "Classify CIFAR10 and CIFAR100"
readme = "README.md"
authors = [
    { name = "Donghyun Park", email = "donghyun.park@cooper.edu" }
]
requires-python = ">=3.13"
dependencies = [
    "structlog",
    "rich",
    "numpy",
    "jax",
    "flax",
    "tensorflow-datasets>=4.9.9",
    "pydantic>=2.11.9",
    "pydantic-settings>=2.11.0",
    "orbax>=0.1.9",
    "tensorflow>=2.20.0",
]

[project.scripts]
hw04 = "hw04:main"
test = "hw04:run_test"

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

```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."""

    dataset_name: str = "cifar10"
    percent_train: int = 90


class ModelSettings(BaseModel):
    """Settings for model architecture."""

    input_height: int = 32
    input_width: int = 32
    input_depth: int = 3
    layer_depths: list[int] = [
        32,
        32,
        32,
        64,
        64,
        64,
        128,
        128,
        128,
    ]  # resnet 18 layers
    strides: list[int] = [1, 1, 1, 2, 1, 1, 2, 1, 1]
    num_groups: list[int] = [1, 1, 1, 2, 2, 2, 4, 4, 4]
    layer_kernel_sizes: list[tuple[int, int]] = [
        (3, 3),
        (3, 3),
        (3, 3),
        (3, 3),
        (3, 3),
        (3, 3),
        (3, 3),
        (3, 3),
        (3, 3),
    ]
    num_classes: int = 10  # 10 for cifar10, 100 for cifar100


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

    batch_size: int = 128
    epochs: int = 12500
    learning_rate: float = 0.001
    l2_reg: float = 0.0001
    momentum: float = 0.9


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

    figsize: Tuple[int, int] = (10, 10)
    dpi: int = 200
    output_dir: Path = Path("hw04/artifacts")


class LoggingSettings(BaseModel):
    """Settings for logging."""

    log_level: str = "INFO"
```



```python
    log_format: str = "plain"  # "json" or "plain"
    log_output: str = "stdout"  # "stdout" or "file"
    output_dir: Path = Path("hw04/artifacts")


class SavingSettings(BaseModel):
    """Settings for model saving."""

    output_dir_10: Path = Path("hw04/saves/cifar-10")
    output_dir_100: Path = Path("hw04/saves/cifar-100")


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

    debug: bool = False
    random_seed: int = 31415
    data: DataSettings = DataSettings()
    model: ModelSettings = ModelSettings()
    training: TrainingSettings = TrainingSettings()
    plotting: PlottingSettings = PlottingSettings()
    logging: LoggingSettings = LoggingSettings()
    saving: SavingSettings = SavingSettings()


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

```python
import structlog
import jax
import numpy as np
import optax
import orbax.checkpoint as ocp
from flax import nnx
from pathlib import Path

from .logging import configure_logging
from .config import load_settings
from .data import Data
from .model import Classifier
from .training_testing import train, test

"""
Discussion:
According to Wikipedia, state of the art for CIFAR-10 and CIFAR-100 performance can be seen here:
https://en.wikipedia.org/wiki/CIFAR-10#Research_papers_claiming_state-of-the-art_results_on_CIFAR-10
From the linked papers based on the paper "Wide Residual Networks",
as it seems to implement the ResNet architecture without too many modifications or even other approaches like transfor
mers

ResNet-18 architecture was implemented, so ideally we would get better performance that 16 layer ResNet show on pa
ge 8,
where the accuracies were 95.44% and 78.41% for CIFAR-10, and CIFAR-100 respectively.
We will be benchmarking on these values for our system.

Different approaches like data augmentation, hyperparameter adjustments (learning rate, decay, iterations, layer depths
etc) were attempted
but constrained by computational power+time was the biggest factor.
Certain hyperparamters would cause the loss value to blow up exponentially, causing the system to perform badly
In the end, numbers hyperparameters used to run training was settled on, achieving about
72% top1-accuracy on CIFAR-10
50% top5-accuracy on CIFAR-100

"""


def main() -> None:
    """CLI entry point."""
    configure_logging()
    log = structlog.get_logger()
    log.info("Hello from hw04!")

    settings = load_settings()
    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,
        dataset_name=settings.data.dataset_name,
        percent_train=settings.data.percent_train,
    )

    log.info("Loaded dataset", dataset=settings.data.dataset_name)

    num_classes = settings.model.num_classes if settings.data.dataset_name == "c
ifar10" else 100
    model = Classifier(
        key=model_key,
        input_depth=settings.model.input_depth,
        layer_depths=settings.model.layer_depths,
        layer_kernel_sizes=settings.model.layer_kernel_sizes,
        num_classes=num_classes,
        num_groups=settings.model.num_groups,
```



```python
        strides=settings.model.strides,
    )

    learning_rate_schedule = optax.cosine_decay_schedule(
        init_value=settings.training.learning_rate,
        decay_steps=settings.training.epochs,
        alpha=0.001,
    )

    optimizer_chain = optax.chain(
        optax.sgd(
            learning_rate=learning_rate_schedule,
            momentum=settings.training.momentum,
        ),
        optax.add_decayed_weights(settings.training.l2_reg),
    )

    optimizer = nnx.Optimizer(
        model,
        optimizer_chain,
        wrt=nnx.Param,
    )

    top1_accuracy, top5_accuracy = train(
        model, optimizer, data, settings.training, np_rng
    )
    log.info("Finished Training model")

    save_dir = (
        settings.saving.output_dir_10
        if settings.data.dataset_name == "cifar10"
        else settings.saving.output_dir_100
    )
    log.info("savedir", save_dir=save_dir)
    save_dir = Path.cwd() / save_dir
    ckpt_dir = ocp.test_utils.erase_and_create_empty(save_dir)
    _, state = nnx.split(model)
    checkpointer = ocp.StandardCheckpointer()
    checkpointer.save(ckpt_dir / "state", state)
    checkpointer.wait_until_finished()
    log.info("Saved model")

    if (
        settings.data.dataset_name == "cifar10"
        and (top1_accuracy > 0.9544 or top5_accuracy > 0.9)
    ) or (
        settings.data.dataset_name == "cifar100"
        and (top1_accuracy > 0.7841 or top5_accuracy > 0.9)
    ):
        log.info(
            "Achieved target accuracy on validation set",
            accuracy=top1_accuracy,
            top5_accuracy=top5_accuracy,
        )
    else:
        log.info(
            "Did not achieve target accuracy on validation set",
            accuracy=top1_accuracy,
            top5_accuracy=top5_accuracy,
        )


def run_test() -> None:
    """CLI entry point."""
    configure_logging()
    log = structlog.get_logger()
    log.info("Running test!")

    settings = load_settings()
```

```
    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,
        dataset_name=settings.data.dataset_name,
        percent_train=settings.data.percent_train,
    )

    log.info("Loaded dataset", dataset=settings.data.dataset_name)

    num_classes = settings.model.num_classes if settings.data.dataset_name == "c
ifar10" else 100
    model = Classifier(
        key=model_key,
        input_depth=settings.model.input_depth,
        layer_depths=settings.model.layer_depths,
        layer_kernel_sizes=settings.model.layer_kernel_sizes,
        num_classes=num_classes,
        num_groups=settings.model.num_groups,
        strides=settings.model.strides,
    )

    # recreate model
    save_dir = (
        settings.saving.output_dir_10
        if settings.data.dataset_name == "cifar10"
        else settings.saving.output_dir_100
    )
    ckpt_dir = Path.cwd() / save_dir
    checkpointer = ocp.StandardCheckpointer()
    graphdef, state = nnx.split(model)
    state_restored = checkpointer.restore(ckpt_dir / "state", state)
    model = nnx.merge(graphdef, state_restored)

    log.info("Loaded model")

    test(model, data)
```

```python
import jax
import jax.numpy as jnp
import numpy as np
import structlog
import optax
from flax import nnx
from tqdm import trange

from .config import TrainingSettings
from .data import Data
from .model import Classifier

log = structlog.get_logger()


def calc_values(x, y):
    loss = optax.softmax_cross_entropy_with_integer_labels(x, y).mean()
    accuracy = jnp.mean(jnp.argmax(x, axis=-1) == y)
    _, top_5_indices = jax.lax.top_k(x, k=5)
    y = y.reshape(-1, 1)
    top5_accuracy = jnp.mean(jnp.any(top_5_indices == y, axis=-1))
    return [loss, accuracy, top5_accuracy]


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

    def loss_fn(model: Classifier):
        y_hat = model(x)
        loss, accuracy, _ = calc_values(y_hat, y)
        return loss, accuracy

    (loss, accuracy), grads = nnx.value_and_grad(loss_fn, has_aux=True)(model)
    optimizer.update(model, grads)  # In-place update of model parameters
    return loss, accuracy


def train(
    model: Classifier,
    optimizer: nnx.Optimizer,
    data: Data,
    settings: TrainingSettings,
    np_rng: np.random.Generator,
) -> list[float]:
    """Train the model using SGD."""
    log.info("Starting training", **settings.model_dump())
    bar = trange(settings.epochs)
    for i in bar:
        train_image_batch, train_label_batch = data.get_batch(
            np_rng, settings.batch_size
        )

        loss, accuracy = train_step(
            model, optimizer, train_image_batch, train_label_batch
        )

        bar.set_description(f"Loss @ {i} => {loss:.6f}, Acc @ {accuracy:.6f}")
        bar.refresh()
        if i % 2500 == 0:
            log.info("Training step", step=i, loss=loss, accuracy=accuracy)

    log.info("Training step", step=settings.epochs, loss=loss, accuracy=accuracy)
    log.info("Training finished")

    # test on validation set
    results = calc_values(model(data.test_image_set), data.test_label_set)
```



```python
    top1_accuracy = results[1]
    top5_accuracy = results[2]
    log.info("Top 1 accuracy", accuracy=top1_accuracy)
    log.info("Top 5 accuracy", accuracy=top5_accuracy)

    return [top1_accuracy, top5_accuracy]


def test(
    model: Classifier,
    data: Data,
) -> None:
    """Test the model using test dataset."""
    results = calc_values(model(data.test_image_set), data.test_label_set)
    top1_accuracy = results[1]
    top5_accuracy = results[2]
    log.info("Top 1 accuracy", accuracy=top1_accuracy)
    log.info("Top 5 accuracy", accuracy=top5_accuracy)
```

```python
import jax
import structlog
from flax import nnx
from jax import numpy as jnp

log = structlog.get_logger()


class GroupNorm(nnx.Module):
    """A Flax NNX module for Group Normalization."""

    def __init__(self, *, rngs: nnx.Rngs, num_groups: int, num_features: int):
        self.norm = nnx.GroupNorm(
            num_groups=num_groups, num_features=num_features, rngs=rngs
        )

    def __call__(self, x: jax.Array) -> jax.Array:
        return self.norm(x)


class Conv2d(nnx.Module):
    """A Flax NNX module for a 2D convolutional layer."""

    def __init__(
        self,
        *,
        rngs: nnx.Rngs,
        in_features: int,
        out_features: int,
        kernel_size: tuple[int, int],
        strides: int,
    ):
        self.conv = nnx.Conv(
            in_features=in_features,
            out_features=out_features,
            kernel_size=kernel_size,
            strides=strides,
            padding="SAME",
            rngs=rngs,
        )

    def __call__(self, x: jax.Array) -> jax.Array:
        return self.conv(x)


class ResidualBlock(nnx.Module):
    """A Flax NNX module for a residual block."""

    def __init__(
        self,
        *,
        key: jax.random.PRNGKey,
        in_features: int,
        out_features: int,
        kernel_size: tuple[int, int],
        num_groups: int,
        strides: int,
    ):
        res_keys = jax.random.split(key, 5)

        self.conv1 = Conv2d(
            in_features=in_features,
            out_features=out_features,
            kernel_size=kernel_size,
            strides=strides,
            rngs=nnx.Rngs(params=res_keys[0]),
        )
        self.gn1 = GroupNorm(
            num_groups=num_groups,
            num_features=in_features,
            rngs=nnx.Rngs(params=res_keys[1]),
        )
        self.conv2 = Conv2d(
            in_features=out_features,
            out_features=out_features,
            kernel_size=kernel_size,
            strides=1,
            rngs=nnx.Rngs(params=res_keys[2]),
        )
        self.gn2 = GroupNorm(
            num_groups=num_groups,
            num_features=out_features,
            rngs=nnx.Rngs(params=res_keys[3]),
        )

        if (in_features != out_features) or (strides != 1):
            self.shortcut = Conv2d(
                in_features=in_features,
                out_features=out_features,
                kernel_size=(1, 1),
                strides=strides,
                rngs=nnx.Rngs(params=res_keys[4]),
            )
        else:
            self.shortcut = lambda x: x  # Identity

        self.activation = jax.nn.relu

    def __call__(self, x: jax.Array) -> jax.Array:
        residual = self.shortcut(x)
        fx = self.gn1(x)
        fx = self.activation(fx)
        fx = self.conv1(fx)
        fx = self.gn2(fx)
        fx = self.activation(fx)
        fx = self.conv2(fx)

        return residual + fx


class Classifier(nnx.Module):
    """A Flax NNX module for an CNN model."""

    def __init__(
        self,
        *,
        key: jax.random.PRNGKey,
        input_depth: int,
        layer_depths: list[int],
        layer_kernel_sizes: list[tuple[int, int]],
        num_classes: int,
        num_groups: int,
        strides: list[int],
    ):
        keys = jax.random.split(key, len(layer_depths) + 2)
        # layer_depths and layer_kernel_sizes must have the same length
        assert len(layer_depths) == len(layer_kernel_sizes)

        self.start_conv = Conv2d(
            rngs=nnx.Rngs(params=keys[0]),
            in_features=input_depth,
            out_features=layer_depths[0],
            kernel_size=layer_kernel_sizes[0],
            strides=strides[0],
        )

        self.res_layers = nnx.List([])
```

```
        for i in range(1, len(layer_depths)):
            self.res_layers.append(
                ResidualBlock(
                    key=keys[i],
                    in_features=layer_depths[i - 1],
                    out_features=layer_depths[i],
                    kernel_size=layer_kernel_sizes[i],
                    num_groups=num_groups[i],
                    strides=strides[i],
                )
            )

        self.dense = nnx.Linear(
            in_features=layer_depths[-1],
            out_features=num_classes,
            rngs=nnx.Rngs(params=keys[-1]),
        )

    def __call__(self, x: jax.Array) -> jax.Array:
        # """Iterates through the CNN layers."""
        # x = jax.nn.relu(self.start_gn(self.start_conv(x)))
        x = jax.nn.relu(self.start_conv(x))
        for layer in self.res_layers:
            x = layer(x)

        x = jnp.mean(x, axis=(1, 2))
        x = self.dense(x)

        return x
```

```python
import logging
import os
from pathlib import Path

import jax
import numpy as np
import structlog

from .config import LoggingSettings


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."""
    log_settings = LoggingSettings()
    log_settings.output_dir.mkdir(parents=True, exist_ok=True)
    log_path = log_settings.output_dir / "hw04.log"
    logging.basicConfig(
        format="%(message)s",
        level=logging.INFO,
        filename=log_path,
        filemode="w",
    )

    # Set the level for the application's logger
    log_level = os.environ.get("LOG_LEVEL", "INFO").upper()
    logging.getLogger("hw04").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.processors.JSONRenderer(),
        ],
        context_class=dict,
        logger_factory=structlog.stdlib.LoggerFactory(),
        wrapper_class=structlog.stdlib.BoundLogger,
        cache_logger_on_first_use=True,
    )
```

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

import numpy as np
import structlog
import tensorflow_datasets as tfds
import tensorflow as tf

log = structlog.get_logger()


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

    rng: InitVar[np.random.Generator]
    dataset_name: str
    percent_train: int
    train_image_set: np.ndarray = field(init=False)
    train_label_set: np.ndarray = field(init=False)
    val_image_set: np.ndarray = field(init=False)
    val_label_set: np.ndarray = field(init=False)
    test_image_set: np.ndarray = field(init=False)
    test_label_set: np.ndarray = field(init=False)
    train_index: np.ndarray = field(init=False)

    def __post_init__(self, rng: np.random.Generator) -> None:
        """Generate training and validating data."""

        def normalize(image, label):
            image = tf.cast(image, tf.float32) / 255.0
            return image, label

        # load data from TF
        (cifar_train, cifar_val) = tfds.load(
            self.dataset_name,
            split=[
                "train[:{percent}%]".format(percent=self.percent_train),
                "train[{percent}%:]".format(percent=self.percent_train),
            ],
            shuffle_files=True,
            as_supervised=True,
        )

        cifar_train = cifar_train.map(normalize, num_parallel_calls=tf.data.AUTO
TUNE)
        cifar_train = tfds.as_numpy(cifar_train)
        cifar_val = cifar_val.map(normalize, num_parallel_calls=tf.data.AUTOTUNE
)
        cifar_val = tfds.as_numpy(cifar_val)

        cifar_test = tfds.load(
            self.dataset_name, split="test", shuffle_files=True, as_supervised=Tr
ue
        )
        cifar_test = tfds.as_numpy(cifar_test)

        self.train_image_set = np.stack([x for x, _ in cifar_train])
        self.train_label_set = np.stack([y for _, y in cifar_train])
        self.val_image_set = np.stack([x for x, _ in cifar_val])
        self.val_label_set = np.stack([y for _, y in cifar_val])
        self.test_image_set = np.stack([x for x, _ in cifar_test])
        self.test_label_set = np.stack([y for _, y in cifar_test])
        self.train_index = np.arange(self.train_image_set.shape[0])

        log.debug(
            "Data initialized",
            train_images=self.train_image_set.shape,
            train_labels=self.train_label_set.shape,
            val_images=self.val_image_set.shape,
```



```python
            val_labels=self.val_label_set.shape,
        )

    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.train_index, size=batch_size)

        images = self.train_image_set[choices]
        labels = self.train_label_set[choices].flatten()

        images = self.augment(rng, images)

        return images, labels

    def augment(self, rng: np.random.Generator, images: np.ndarray):
        batch_size, height, width, _ = images.shape
        augmented_images = []

        for image in images:
            if rng.random() > 0.5:
                image = np.fliplr(image)

            padding = 4
            padded_image = np.pad(
                image, ((padding, padding), (padding, padding), (0, 0)), mode="r
eflect"
            )

            h_start = rng.integers(0, padding * 2)
            w_start = rng.integers(0, padding * 2)

            image = padded_image[
                h_start : h_start + height, w_start : w_start + width, :
            ]
            augmented_images.append(image)

        return np.stack(augmented_images)
```