```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 sklearn.model_selection import KFold

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

"""
Discussion:
Implemented a MLP that intakes a word emebedding 384−width vector of the ag news article pass through HuggingFac
e's SentenceTransformer
MLP hyperparameters: 64 hidden layer nodes, 2 layers, 128 embedding dimensions, batch size 128
Model performed very poorly with accuracy of 46% and loss of 1.2, likely due to using SentenceTransformer
Implementing custom word embedding matrix peformed far better, likely due to the limited word domain rather than th
e large pre−training done on SentenceTransformer
5 fold cross validation showed 91.37% mean accuracy on validation set
Test set accuracy of 90.26%
"""


def main() -> None:
    """CLI entry point."""
    configure_logging()
    log = structlog.get_logger()
    log.info("Hello from hw05!")

    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,
        vocab_size=settings.data.vocab_size,
        max_seq_length=settings.data.max_seq_length,
    )

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

    kfold = KFold(n_splits=settings.training.k_folds, shuffle=True, random_state
=settings.random_seed)
    fold_accuracies = []

    for fold, (train_idx, val_idx) in enumerate(kfold.split(data.train_text_set)
):
        log.info(f"Starting Fold {fold + 1}/{settings.training.k_folds}")

        train_tokens = data.train_text_set[train_idx]
        train_labels = data.train_label_set[train_idx]
        val_tokens = data.train_text_set[val_idx]
        val_labels = data.train_label_set[val_idx]

        log.info("Train/Val split", train_shape=train_tokens.shape, val_shape=val_tok
ens.shape)

        model = MLP(
            rngs=nnx.Rngs(params=model_key),
```



```python
            vocab_size=settings.model.vocab_size,
            embedding_dim=settings.model.embedding_dim,
            hidden_layer_depth=settings.model.layer_depths,
            num_hidden_layers=settings.model.num_hidden_layers,
            num_classes=settings.model.num_classes,
        )

        params = nnx.state(model, nnx.Param)
        total_params = sum(np.prod(x.shape) for x in jax.tree_util.tree_leaves(p
arams))
        log.info(f"Total parameters: {total_params:,}")

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

        optimizer = nnx.Optimizer(
            model,
            optax.adamw(
                learning_rate=learning_rate_schedule,
                weight_decay=settings.training.l2_reg,
            ),
            wrt=nnx.Param,
        )

        val_accuracy = train(
            model,
            optimizer,
            [train_tokens, train_labels, val_tokens, val_labels],
            settings.training,
            fold,
            np_rng,
        )
        log.info(f"Fold {fold + 1} Validation Accuracy: {val_accuracy:.4f}")
        fold_accuracies.append(val_accuracy)

    mean_accuracy = np.mean(fold_accuracies)
    std_accuracy = np.std(fold_accuracies)
    log.info(
        "Cross−Validation Finished",
        mean_accuracy=f"{mean_accuracy:.4f}",
        std_deviation=f"{std_accuracy:.4f}",
    )

    save_dir = settings.saving.output_dir
    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")


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,
        vocab_size=settings.data.vocab_size,
        max_seq_length=settings.data.max_seq_length,
    )

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

    model = MLP(
        rngs=nnx.Rngs(params=model_key),
        vocab_size=settings.model.vocab_size,
        embedding_dim=settings.model.embedding_dim,
        hidden_layer_depth=settings.model.layer_depths,
        num_hidden_layers=settings.model.num_hidden_layers,
        num_classes=settings.model.num_classes,
    )

    # recreate model
    ckpt_dir = Path.cwd() / settings.saving.output_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 structlog
from flax import nnx
from jax import numpy as jnp

log = structlog.get_logger()


class GLU(nnx.Module):
    def __init__(
        self,
        *,
        input_layer_depth: int,
        output_layer_depth: int,
        rngs: nnx.Rngs,
    ):
        self.glu = nnx.Linear(input_layer_depth, output_layer_depth * 2, rngs=rn
gs)

    def __call__(self, x: jax.Array) -> jax.Array:
        gate, activation = jnp.split(self.glu(x), 2, axis=-1)
        return gate * nnx.sigmoid(activation)


class HiddenLayer(nnx.Module):
    def __init__(
        self,
        *,
        layer_depth: int,
        rngs: nnx.Rngs,
    ):
        self.norm = nnx.GroupNorm(num_features=layer_depth, rngs=rngs)
        self.hidden_layer = GLU(
            input_layer_depth=layer_depth, output_layer_depth=layer_depth, rngs=
rngs
        )

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


class HiddenLayers(nnx.Module):
    def __init__(
        self,
        *,
        num_hidden_layers: int,
        layer_depth: int,
        rngs: nnx.Rngs,
    ):
        @nnx.split_rngs(splits=num_hidden_layers)
        @nnx.vmap(in_axes=(0,), out_axes=0)
        def create_layers(rngs: nnx.Rngs):
            return HiddenLayer(layer_depth=layer_depth, rngs=rngs)

        self.layers = create_layers(rngs)
        self.num_hidden_layers = num_hidden_layers

    def __call__(self, x: jax.Array) -> jax.Array:
        if self.num_hidden_layers > 0:

            @nnx.scan(in_axes=(nnx.Carry, 0), out_axes=nnx.Carry)
            def forward_body(carry, layer):
                # skip connection
                return nnx.relu(carry + layer(carry))

            x = forward_body(x, self.layers)
        return x
```



```python
class MLP(nnx.Module):
    """A Flax NNX module for an MLP model."""

    def __init__(
        self,
        *,
        rngs: nnx.Rngs,
        vocab_size: int,
        embedding_dim: int,
        hidden_layer_depth: int,
        num_hidden_layers: int,
        num_classes: int,
    ):
        self.embedding = nnx.Embed(
            num_embeddings=vocab_size,
            features=embedding_dim,
            rngs=rngs,
        )

        self.input_layer = GLU(
            input_layer_depth=embedding_dim,
            output_layer_depth=hidden_layer_depth,
            rngs=rngs,
        )

        self.hidden_layers = HiddenLayers(
            num_hidden_layers=num_hidden_layers,
            layer_depth=hidden_layer_depth,
            rngs=rngs,
        )

        self.output_layer = nnx.Linear(hidden_layer_depth, num_classes, rngs=rng
s)

    def __call__(self, x: jax.Array) -> jax.Array:
        """Iterates through the MLP layers."""
        x = self.embedding(x)
        x = jnp.mean(x, axis=1)

        x = self.input_layer(x)
        x = self.hidden_layers(x)
        x = self.output_layer(x)
        return x
```

```python
from dataclasses import InitVar, dataclass, field
from collections import Counter
import numpy as np
import structlog
import tensorflow_datasets as tfds

log = structlog.get_logger()


@dataclass
class Data:
    """Handles generation of tokenized data for MLP with trainable embeddings."""

    rng: InitVar[np.random.Generator]
    dataset_name: str
    vocab_size: int = 10000  # Size of vocabulary
    max_seq_length: int = 50  # Maximum sequence length
    train_text_set: np.ndarray = field(init=False)
    train_label_set: np.ndarray = field(init=False)
    test_text_set: np.ndarray = field(init=False)
    test_label_set: np.ndarray = field(init=False)
    train_index: np.ndarray = field(init=False)
    word_to_idx: dict = field(init=False)
    idx_to_word: dict = field(init=False)

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

        # load data from TF
        (ag_train, ag_test) = tfds.load(
            self.dataset_name,
            split=["train", "test"],
            shuffle_files=True,
        )

        def combine_text(x):
            text = x["title"] + " " + x["description"]
            return text, x["label"]

        ag_train = ag_train.map(combine_text)
        ag_test = ag_test.map(combine_text)

        # Convert to numpy arrays
        train_texts = [x.numpy().decode('utf-8') for x, _ in ag_train]
        self.train_label_set = np.array([y for _, y in ag_train])
        test_texts = [x.numpy().decode('utf-8') for x, _ in ag_test]
        self.test_label_set = np.array([y for _, y in ag_test])

        log.debug(
            "Data initialized",
            train_samples=len(train_texts),
            test_samples=len(test_texts),
            train_labels=self.train_label_set.shape,
            test_labels=self.test_label_set.shape,
        )

        log.info("Building vocabulary")
        self.build_vocabulary(train_texts)

        # Tokenize and pad sequences
        log.info("Tokenizing text")
        self.train_text_set = self.tokenize_texts(train_texts)
        self.test_text_set = self.tokenize_texts(test_texts)
        self.train_index = np.arange(self.train_text_set.shape[0])

        log.debug(
            "Data processed",
            train_text_shape=self.train_text_set.shape,
            test_text_shape=self.test_text_set.shape,
```



```python
            vocab_size=len(self.word_to_idx),
            max_seq_length=self.max_seq_length,
        )

    def build_vocabulary(self, texts: list) -> None:
        """Build vocabulary from training texts."""
        word_counts = Counter()

        for text in texts:
            words = text.lower().split()
            word_counts.update(words)

        most_common = word_counts.most_common(self.vocab_size - 2)

        # Create word to index mapping
        self.word_to_idx = {
            '<PAD>': 0,
            '<UNK>': 1,
        }

        for idx, (word, _) in enumerate(most_common, start=2):
            self.word_to_idx[word] = idx

        self.idx_to_word = {idx: word for word, idx in self.word_to_idx.items()}

    def tokenize_texts(self, texts: list) -> np.ndarray:
        """Convert texts to padded token sequences."""
        tokenized = []

        for text in texts:
            words = text.lower().split()
            # Convert words to indices
            indices = [self.word_to_idx.get(word, 1) for word in words]  # 1 is
<UNK>

            # Truncate or pad to max_seq_length
            if len(indices) > self.max_seq_length:
                indices = indices[:self.max_seq_length]
            else:
                indices.extend([0] * (self.max_seq_length - len(indices)))  # 0
is <PAD>

            tokenized.append(indices)

        return np.array(tokenized, dtype=np.int32)

    def get_batch(
        self, rng: np.random.Generator, batch_size: int
    ) -> tuple[np.ndarray, np.ndarray]:
        """Select random subset of examples for training batch. Not used in this assignment."""
        choices = rng.choice(self.train_index, size=batch_size)

        texts = self.train_text_set[choices]
        labels = self.train_label_set[choices].flatten()

        return texts, labels
```

```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("hw05").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
import jax.numpy as jnp
import numpy as np
import structlog
import optax
import matplotlib.pyplot as plt
from pathlib import Path
from flax import nnx
from tqdm import trange

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

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)
    return loss, accuracy


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

    def loss_fn(model: MLP):
        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: MLP,
    optimizer: nnx.Optimizer,
    data: list,
    settings: TrainingSettings,
    fold: int,
    np_rng: np.random.Generator,
) -> float:
    """Train the model using SGD."""
    log.info("Starting training", **settings.model_dump())
    bar = trange(settings.num_iters)

    train_tokens = data[0]
    train_labels = data[1]
    train_index = np.arange(train_tokens.shape[0])

    losses = []
    accuracies = []
    for i in bar:
        choices = np_rng.choice(train_index, size=settings.batch_size)
        train_token_batch = train_tokens[choices]
        train_label_batch = train_labels[choices]

        loss, accuracy = train_step(
            model, optimizer, train_token_batch, train_label_batch
        )
        losses.append(loss)
        accuracies.append(accuracy)

        bar.set_description(f"Loss @ {i} => {loss:.6f}, Acc @ {accuracy:.6f}")
        bar.refresh()
```



```python
        if i % 1000 == 0:
            log.info("Training step", step=i, loss=loss, accuracy=accuracy)
            log.debug("Predicted labels", predicted=jnp.argmax(model(train_token_batc
h), axis=-1), true=train_label_batch)

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

    # test on validation set
    _, accuracy = calc_values(model(data[2]), data[3])
    log.info("Validation set accuracy", accuracy=accuracy, fold=fold)
    return accuracy


def test(
    model: MLP,
    data: Data,
) -> None:
    """Test the model using test dataset."""
    _, accuracy = calc_values(model(data.test_text_set), data.test_label_set)
    log.info("Test set accuracy", accuracy=accuracy)
```

```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 = "ag_news_subset"
    percent_train: int = 90
    vocab_size: int = 10000
    max_seq_length: int = 50


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

    vocab_size: int = 10000
    embedding_dim: int = 128
    max_seq_length: int = 50
    layer_depths: int = 64
    num_hidden_layers: int = 2
    num_classes: int = 4  # World, Sports, Business, Sci/Tech


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

    k_folds: int = 5
    batch_size: int = 128
    num_iters: int = 10000
    learning_rate: float = 0.001
    l2_reg: float = 0.0001


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

    figsize: Tuple[int, int] = (10, 10)
    dpi: int = 200
    output_dir: Path = Path("hw05/artifacts")


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

    log_level: str = "INFO"
    log_format: str = "plain"  # "json" or "plain"
    log_output: str = "stdout"  # "stdout" or "file"
    output_dir: Path = Path("hw05/artifacts")


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

    output_dir: Path = Path("hw05/saves/ag-news")


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()
```



```python
    logging: LoggingSettings = LoggingSettings()
    saving: SavingSettings = SavingSettings()


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

```
# pyproject.toml.jinja

[project]
name = "hw05"
version = "0.1.0"
description = "Classify the AG News dataset."
readme = "README.md"
authors = [
    { name = "Donghyun Park", email = "donghyun.park@cooper.edu" }
]
requires-python = ">=3.13"
dependencies = [
    "structlog",
    "rich",
    "numpy",
    "jax",
    "flax",
    "tensorflow>=2.20.0",
    "tensorflow-datasets>=4.9.9",
    "pydantic>=2.12.1",
    "pydantic-settings>=2.11.0",
    "orbax>=0.1.9",
    "sentence-transformers>=5.1.1",
    "tf-keras>=2.20.1",
    "matplotlib>=3.10.7",
]

[project.scripts]
hw05 = "hw05:main"
test = "hw05:run_test"

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