```
==> Running homework 'hw05'...
^[[2m2025-10-16T01:57:26.220760Z^[[0m [^[[32m^[[1minfo     ^[[0m] ^[[1mSettings
loaded              ^[[0m [^[[0m^[[1m^[[34mhw05^[[0m]^[[0m ^[[36msettings^[[0m=
^[[35m{'debug': False, 'random_seed': 427, 'data': {'max_length': 128}, 'trainin
g': {'batch_size': 64, 'num_epochs': 5, 'learning_rate': 0.0005, 'weight_decay':
 1e-05}, 'model': {'latent_dim': 512, 'hidden_dims': (512,), 'embed_dim': 256, '
dropout_rate': 0.1}}^[[0m
^[[2m2025-10-16T01:57:43.142282Z^[[0m [^[[32m^[[1minfo     ^[[0m] ^[[1mDatasets
tokenized           ^[[0m [^[[0m^[[1m^[[34mhw05.data^[[0m]^[[0m ^[[36mtest_size
^[[0m=^[[35m7600^[[0m ^[[36mtrain_size^[[0m=^[[35m120000^[[0m
^[[2m2025-10-16T01:57:43.159135Z^[[0m [^[[32m^[[1minfo     ^[[0m] ^[[1mTrain/Val
/Test splits created ^[[0m [^[[0m^[[1m^[[34mhw05.data^[[0m]^[[0m ^[[36mtest_size
^[[0m=^[[35m7600^[[0m ^[[36mtrain_size^[[0m=^[[35m108000^[[0m ^[[36mval_size^[[0
m=^[[35m12000^[[0m
^[[2m2025-10-16T01:57:43.159274Z^[[0m [^[[32m^[[1minfo     ^[[0m] ^[[1mToken ite
rator created       ^[[0m [^[[0m^[[1m^[[34mhw05.data^[[0m]^[[0m ^[[36mbatch_siz
e^[[0m=^[[35m64^[[0m
^[[2m2025-10-16T01:57:43.159639Z^[[0m [^[[32m^[[1minfo     ^[[0m] ^[[1mmodel ins
tantiated           ^[[0m [^[[0m^[[1m^[[34mhw05^[[0m]^[[0m ^[[36mvocab_size^[[0
m=^[[35m30522^[[0m
Epoch 1/5

Epoch 1:   0%|          | 0/1688 [00:00<?, ?it/s]
Epoch 1:   0%|          | 0/1688 [00:02<?, ?it/s, loss=1.39, acc=0.328]
Epoch 1:  50%|âM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^I     | 838/1688 [00:2
8<00:24, 34.71it/s, loss=0.279, acc=0.906]
Epoch 1: 100%|âM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^Hâ
M-^VM-^HâM-^VM-^HâM-^VM-^H| 1688/1688 [00:57<00:00, 29.56it/s, loss=0.341, acc=0
.875]
Val loss: 0.2386, Val acc: 0.9164
Epoch 2/5

Epoch 2:   0%|          | 0/1688 [00:00<?, ?it/s]
Epoch 2:  50%|âM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^I     | 836/1688 [00:2
6<00:23, 35.73it/s, loss=0.131, acc=0.953]
Epoch 2: 100%|âM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^Hâ
M-^VM-^HâM-^VM-^HâM-^VM-^H| 1688/1688 [00:54<00:00, 31.04it/s, loss=0.181, acc=0
.922]
Val loss: 0.2376, Val acc: 0.9209
Epoch 3/5

Epoch 3:   0%|          | 0/1688 [00:00<?, ?it/s]
Epoch 3:  50%|âM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^I     | 836/1688 [00:2
6<00:24, 35.22it/s, loss=0.157, acc=0.953]
Epoch 3: 100%|âM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^Hâ
M-^VM-^HâM-^VM-^HâM-^VM-^H| 1688/1688 [00:51<00:00, 32.86it/s, loss=0.161, acc=0
.953]
Val loss: 0.2823, Val acc: 0.9170
Epoch 4/5

Epoch 4:   0%|          | 0/1688 [00:00<?, ?it/s]
Epoch 4:  50%|âM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^I     | 836/1688 [00:3
1<00:26, 31.67it/s, loss=0.0471, acc=0.984]
Epoch 4: 100%|âM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^Hâ
M-^VM-^HâM-^VM-^HâM-^VM-^H| 1688/1688 [00:56<00:00, 29.83it/s, loss=0.102, acc=0
.969]
Val loss: 0.3477, Val acc: 0.9165
Epoch 5/5

Epoch 5:   0%|          | 0/1688 [00:00<?, ?it/s]
Epoch 5:  50%|âM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^I     | 843/1688 [00:2
7<00:24, 34.59it/s, loss=0.085, acc=0.953]
Epoch 5: 100%|âM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^Hâ
M-^VM-^HâM-^VM-^HâM-^VM-^H| 1688/1688 [00:55<00:00, 30.60it/s, loss=0.109, acc=0
.984]
Val loss: 0.4297, Val acc: 0.9097
Early stopping after 5 epochs (no improvement for 3 epochs)
Test loss: 0.2555, Test acc: 0.9169
Training complete
```

```
# pyproject.toml.jinja

[project]
name = "hw05"
version = "0.1.0"
description = "AG new classifier using MLP"
readme = "README.md"
authors = [
    { name = "Josh Miao", email = "josh.miao@cooper.edu" }
]
requires-python = ">=3.13"
dependencies = [
    "structlog",
    "rich",
    "numpy",
    "jax[cuda-12,cuda12]",
    "flax",
    "optax>=0.2.6",
    "typing>=3.10.0.0",
    "tqdm>=4.67.1",
    "pydantic>=2.12.2",
    "pydantic-settings>=2.11.0",
    "datasets>=4.2.0",
    "transformers>=4.57.1",
]

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

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

```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):
    max_length: int = 128

class ModelSettings(BaseModel):
    latent_dim: int = 512 #many extra dimensions to later support sparsity
    hidden_dims: tuple = (512,)
    embed_dim: int = 256
    dropout_rate: float = 0.1

class TrainingSettings(BaseModel):
    """Settings for model training."""
    batch_size: int = 64
    num_epochs: int = 5
    learning_rate: float = 5e-4
    weight_decay: float = 1e-5

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

    debug: bool = False
    random_seed: int = 427 #for ece 427 ofc
    data: DataSettings = DataSettings()
    training: TrainingSettings = TrainingSettings()
    model: ModelSettings = ModelSettings()

    model_config = SettingsConfigDict(
        toml_file=files("hw05").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
from datasets import load_dataset
from transformers import DistilBertTokenizerFast
import numpy as np
import structlog

log = structlog.get_logger()

def prepare_datasets(settings):
    """
  load and tokenize ag news using hugging face and distilbert
  create train val test splits
  Load and tokenize the AG News dataset and return batching helpers.
  """
    ds = load_dataset("ag_news")
    tokenizer = DistilBertTokenizerFast.from_pretrained("distilbert-base-uncased")

    max_len = settings.data.max_length

    def tokenize_fn(examples):
        return tokenizer(examples['text'], padding='max_length', truncation=True, m
ax_length=max_len)

    tokenized = ds.map(tokenize_fn, batched=True)
    tokenized.set_format(type='numpy', columns=['input_ids', 'attention_mask', 'label'])
    log.info("Datasets tokenized", train_size=len(tokenized["train"]), test_size=len(to
kenized["test"]))

    # Create a small validation split from the training set (10%)
    split = tokenized["train"].train_test_split(test_size=0.1, seed=472)
    train = split["train"]
    val = split["test"]
    test = tokenized["test"]
    log.info("Train/Val/Test splits created", train_size=len(train), val_size=len(val), te
st_size=len(test))

    def token_iterator(split, batch_size, shuffle=True):
        """
    convert hugging face dataset into batches of numpy dicts that the training function expects
    i have input ID f, attention makss(so paddubg tokens remain indpendent to real tokens), and labels
    methodology is build index array for the split, shuffle, select examples, convert to array, and yield the batch dictio
nary until the split is exhausted
    EZ PZ
    """
        n = len(split)
        indices = np.arange(n)
        if shuffle:
            np.random.shuffle(indices)
        for start in range(0, n, batch_size):
            batch_idx = indices[start:start+batch_size]
            batch = split.select(batch_idx)
            yield {
                'input_ids': np.array(batch['input_ids'], dtype=np.int32),
                'attention_mask': np.array(batch['attention_mask'], dtype=np.int32),
                'labels': np.array(batch['label'], dtype=np.int32)
            }
    log.info("Token iterator created", batch_size=settings.training.batch_size)
    return train, val, test, token_iterator, tokenizer

    log.info("data preprocessed ")
```

```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
import jax.numpy as jnp
from flax import linen as nnx
import structlog

log = structlog.get_logger()

class TextEmbeder(nnx.Module):
    vocab_size: int
    embed_dim: int

    @nnx.compact
    def __call__(self, input_ids):
        """
    train model embeddings intead of using pretrained embeddings so i have more control for ass. 7
    nnx.Embed was useful for creating a trainable embedding layer.
    """
        embed = nnx.Embed(num_embeddings=self.vocab_size, features=self.embed_di
m, name="token_embed")
        x = embed(input_ids) # (batch_size, seq_length, embed_dim)
        return x

class MLPEncoder(nnx.Module):
    latent_dim: int
    hidden_dims: tuple = (512,)
    activation: callable = jax.nn.relu
    dropout_rate: float = 0.0

    @nnx.compact
    def __call__(self,x, deterministic: bool = True):
        """
    Encode pooled features to a latent vector.
    The paper in ass. 7 talks about latent factors so im going to compress the token embeddings
    into a lower dimensional space that i'll call z for now that a decoder can use later.
    I just applied a sequence of dense layers with ReLU activation and dropout the nprojected to
    the latent dimension

    """
        for h_dim in self.hidden_dims:
            x = nnx.Dense(features=h_dim)(x)
            x = self.activation(x)
            x = nnx.Dropout(rate=self.dropout_rate)(x, deterministic=determinist
ic)

        # Final layer to latent space which will be the target of sparsity for a
ss. 7
        z = nnx.Dense(features=self.latent_dim)(x)
        return z

class ClassifierHead(nnx.Module):
    num_classes: int
    dropout_rate: float = 0.0

    @nnx.compact
    def __call__(self, z, deterministic: bool = True):
        """
    Project latent z to logits
    """
        x = nnx.Dropout(rate=self.dropout_rate)(z, deterministic=deterministic)
        logits = nnx.Dense(features=self.num_classes)(x)
        return logits

class TextMLPModel(nnx.Module):
    vocab_size: int
    embed_dim: int
    latent_dim: int
    num_classes: int
    hidden_dims: tuple = (512,)
    dropout_rate: float = 0.0
```



```python
    def setup(self):
        self.embedder = TextEmbeder(vocab_size=self.vocab_size, embed_dim=self.e
mbed_dim)
        self.encoder = MLPEncoder(latent_dim=self.latent_dim, hidden_dims=self.h
idden_dims, dropout_rate=self.dropout_rate)
        self.classifier = ClassifierHead(num_classes=self.num_classes, dropout_r
ate=self.dropout_rate)
    def __call__(self, input_ids, attention_mask=None, deterministic: bool = Tru
e):
        """
    Forward pass for the text MLP classifier. (batch, seq_len) input, logits is (batch, num_classes) and z is (batch, late
nt_dim)
    Embed tokens, mean pooling(i used max pooling for hw04 and that was ass(like pooling was the issue lol)), encod
e latent vector, create logit
    return a TWOple. get it?? because i have 2 items in my tuple?
    """
        x = self.embedder(input_ids)  # (batch, seq_len, embed_dim)

        # Pool token embeddings into a fixed-size vector per example.
        if attention_mask is None:
            # simple mean-pooling across sequence length
            pooled = jnp.mean(x, axis=1)
        else:
            # use attention mask for masked mean pooling
            mask = jnp.expand_dims(jnp.asarray(attention_mask, dtype=x.dtype), a
xis=-1)
            x_masked = x * mask
            summed = jnp.sum(x_masked, axis=1)
            denom = jnp.sum(mask, axis=1)
            pooled = summed / jnp.clip(denom, a_min=1e-9)

        # encoder to latent space
        z = self.encoder(pooled, deterministic=deterministic)
        # classifier to logits
        logits = self.classifier(z)
        return logits, z
```

```python
import structlog

import jax
import jax.numpy as jnp
import numpy as np

import optax

from tqdm import trange
from flax import nnx

from flax.training import train_state

log = structlog.get_logger()

class TrainState(train_state.TrainState):
    """
  mini checkpointing helper to hold my model parameters, optimizer transformer/state
  """
    pass

def create_train_state(rng, model, learning_rate, weight_decay):
    # Initialize model parameters with dummy input and mask to infer input shape
s
    dummy_input = jnp.ones((1, 128), dtype=jnp.int32)
    dummy_mask = jnp.ones((1, 128), dtype=jnp.int32)
    params = model.init(rng, dummy_input, dummy_mask)
    tx = optax.adamw(learning_rate=learning_rate, weight_decay=weight_decay)
    state = TrainState.create(apply_fn=model.apply, params=params, tx=tx)
    return state

@jax.jit
def train_step(state, batch, dropout_rng):

    def loss_fn(params):
        #cross entropy loss function
        logits, z = state.apply_fn(params, batch["input_ids"], batch.get("attention_ma
sk", None), deterministic=False, rngs={"dropout": dropout_rng})
        ce = optax.softmax_cross_entropy_with_integer_labels(logits, batch["labels
"]).mean()
        return ce, (logits, z)

    (loss, (logits, z)), grads = jax.value_and_grad(loss_fn, has_aux=True)(state
.params)
    state = state.apply_gradients(grads=grads)
    acc = jnp.mean(jnp.argmax(logits, -1) == batch["labels"])
    return state, loss, acc, logits, z

def train_loop(state, train_iter, num_steps=None, print_every=100):

    bar = trange(num_steps or 0) if num_steps is not None else None
    step = 0
    for batch in train_iter:
        batch_jax = {k: jnp.array(v) for k, v in batch.items()}
        state, loss, acc, _, _ = train_step(state, batch_jax, jax.random.PRNGKey
(472))
        if step % print_every == 0:
            print(f"Step {step} | Loss: {float(loss):.4f} | Acc: {float(acc):.4f}")
        step += 1
        if num_steps is not None and step >= num_steps:
            break
        if bar is not None:
            bar.update(1)
    return state


def train_and_evaluate(state, train_ds, val_ds, test_ds, token_iterator, setting
s, rng, patience: int = 3):
    """
```



```
  5 epochs over training split and evalues on validation split after each epoch.
  I decided to be ambition and utilize early stopping if val accuracy doesn't improve for 3 epochs
  """
    step = 0
    best_params = state.params
    best_val = -1.0
    wait = 0
    for epoch in range(settings.training.num_epochs):
        print(f"Epoch {epoch+1}/{settings.training.num_epochs}")
        train_iter = token_iterator(train_ds, settings.training.batch_size, shuf
fle=True)
        # Use tqdm for per-epoch progress
        n_batches = (len(train_ds) + settings.training.batch_size - 1) // settin
gs.training.batch_size
        with trange(n_batches, desc=f"Epoch {epoch+1}") as pbar:
            for i, batch in enumerate(train_iter):
                batch_jax = {k: jnp.array(v) for k, v in batch.items()}
                # split RNG for this training step and use it for dropout
                rng, step_rng = jax.random.split(rng)
                state, loss, acc, _, _ = train_step(state, batch_jax, step_rng)
                if step % 100 == 0:
                    pbar.set_postfix({'loss': float(loss), 'acc': float(acc)})
                step += 1
                pbar.update(1)
        # Eval on validation set at epoch end
        val_iter = token_iterator(val_ds, settings.training.batch_size, shuffle=
False)
        losses = []
        accs = []
        for batch in val_iter:
            batch_jax = {k: jnp.array(v) for k, v in batch.items()}
            loss, acc, _, _ = eval_step(state, batch_jax)
            losses.append(float(loss))
            accs.append(float(acc))
        val_loss = np.mean(losses)
        val_acc = np.mean(accs)
        print(f"Val loss: {val_loss:.4f}, Val acc: {val_acc:.4f}")

        # Early stopping on validation accuracy
        if val_acc > best_val:
            best_val = val_acc
            best_params = state.params
            wait = 0
        else:
            wait += 1
            if wait >= patience:
                print(f"Early stopping after {epoch+1} epochs (no improvement for {patience} epochs)")
                break

    # Restore best params
    state = state.replace(params=best_params)

    # Final evaluation on test set
    test_iter = token_iterator(test_ds, settings.training.batch_size, shuffle=Fa
lse)
    losses = []
    accs = []
    for batch in test_iter:
        batch_jax = {k: jnp.array(v) for k, v in batch.items()}
        loss, acc, _, _ = eval_step(state, batch_jax)
        losses.append(float(loss))
        accs.append(float(acc))
    print(f"Test loss: {np.mean(losses):.4f}, Test acc: {np.mean(accs):.4f}")

    return state

@jax.jit
def eval_step(state, batch):
    """
```

```
    Run a single evaluation step. dropout disabled here
    """
    logits, z = state.apply_fn(state.params, batch["input_ids"], batch.get("attention_
mask", None), deterministic=True)
    loss = optax.softmax_cross_entropy_with_integer_labels(logits, batch["labels"]
).mean()
    acc = jnp.mean(jnp.argmax(logits, -1) == batch["labels"])
    return loss, acc, logits, z
```

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

from .logging import configure_logging
from .config import load_settings
from .data import prepare_datasets
from .model import TextMLPModel
from .training import create_train_state, train_step, eval_step, train_and_evalu
ate

log = structlog.get_logger()

def main() -> None:
    """CLI entry point."""
    settings = load_settings()
    configure_logging()
    log = structlog.get_logger()
    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))
    # Prepare data and tokenizer (now returns train, val, test)
    train_ds, val_ds, test_ds, token_iterator, tokenizer = prepare_datasets(sett
ings)
    num_classes = 4

    model = TextMLPModel(
        vocab_size=tokenizer.vocab_size,
        embed_dim=settings.model.embed_dim,
        latent_dim=settings.model.latent_dim,
        num_classes=num_classes,
        hidden_dims=settings.model.hidden_dims,
    )
    log.info("model instantiated", vocab_size=tokenizer.vocab_size)

    #i moved tthe optimizer that would be here to training.py so __init__ bc i w
anna try using state to hold everything
    # Create training state (parameters + optimizer)
    rng = jax.random.PRNGKey(settings.random_seed)
    state = create_train_state(rng, model, settings.training.learning_rate, geta
ttr(settings.training, 'weight_decay', 0.0))

    # Prepare RNG for dropout and other stochastic components
    rng, dropout_rng = jax.random.split(rng)

    # Run the high-level training + evaluation loop (with progress bars)
    state = train_and_evaluate(state, train_ds, val_ds, test_ds, token_iterator,
 settings, dropout_rng)
    print("Training complete")
```