```
# pyproject.toml.jinja

[project]
name = "hw04"
version = "0.1.0"
description = "CIFAR10"
readme = "README.md"
authors = [
    { name = "Josh Miao", email = "josh.miao@cooper.edu" }
]
requires-python = ">=3.13"
dependencies = [
    "structlog",
    "rich",
    "numpy",
    "jax[cuda12]",
    "flax",
    "typing>=3.10.0.0",
    "tqdm>=4.67.1",
    "pydantic>=2.11.9",
    "pydantic-settings>=2.11.0",
    "optax>=0.2.6",
    "dataclasses>=0.8",
    "tensorflow>=2.20.0",
    "nnx>=0.0.8",
    "diffusers==0.11.1",
    "orbax>=0.1.9",
    "orbax-checkpoint>=0.11.25",
    "pathlib>=1.0.1",
    "path>=17.1.1",
]

[project.scripts]
hw04 = "hw04:main"
test10 = "hw04:test10"
test100 = "hw04:test100"

[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, List

from pydantic import BaseModel
from pydantic_settings import (
    BaseSettings,
    PydanticBaseSettingsSource,
    SettingsConfigDict,
    TomlConfigSettingsSource,
)

class DataSettings(BaseModel):
    """Settings for data generation."""
    train_ratio: float = 0.8

#in data.py, change back to cifar10 and change num_classes here in model seeting
s back to 10
class ModelSettings(BaseModel):
    input_depth: int = 3
    num_classes: int = 10
    layer_depth: List[int] = [64, 128, 256]
    layer_kernel_sizes: List[List[int]] = [[3, 3], [3,3], [3,3]]
    strides: List[int] = [1, 1, 1]
     #dropout_rate: float = 0.2
    shape: List[int] = [32, 32]
    noise_std: int = 0.05

class TrainingSettings(BaseModel):
    """Settings for model training."""
    batch_size: int = 256
    num_iters: int = 150000
    learning_rate: float = 1.5e-4
    L2_weight: float = 1.5e-4
    #tried to implement cosine decay with linear warmup but that was low key che
eks
    #warmup_steps: int = 800
    #initial_rate: float = 1e-5
    #end_value: 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("hw04").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 (
```



```python
            init_settings,
            TomlConfigSettingsSource(settings_cls),
            env_settings,
            dotenv_settings,
            file_secret_settings,
        )

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

```
from dataclasses import dataclass, InitVar, field
import numpy as np
import structlog
import tensorflow as tf

log = structlog.get_logger()

@dataclass
class cifar10:
    """retreive data. also i know that my class name is misleading when im training cifar100
  kinda too lazy to change it."""
    rng: InitVar[np.random.Generator]
    train_ratio: float = 0.8

    # Data arrays (initialized post-construction)
    x_train: np.ndarray = field(init=False)
    x_val: np.ndarray = field(init=False)
    x_test: np.ndarray = field(init=False)
    y_train: np.ndarray = field(init=False)
    y_val: np.ndarray = field(init=False)
    y_test: np.ndarray = field(init=False)

    index: np.ndarray = field(init=False)

    def __post_init__(self, rng:np.random.Generator):
        # Load and normalize cifar10 data
        (x_train_raw, y_train_raw), (x_test_raw, y_test_raw) = tf.keras.datasets
.cifar10.load_data()

        # convert to float32 in [0,1]
        x_train_raw = x_train_raw.astype(np.float32) / 255.0
        x_test_raw = x_test_raw.astype(np.float32) / 255.0

        # labels come as shape (N,1) âM-^@M-^T flatten to (N,)
        y_train_raw = y_train_raw.reshape(-1)
        y_test_raw = y_test_raw.reshape(-1)

        # shuffle and split training into train/val
        train_idx, val_idx = self.split_indices(len(y_train_raw), rng)

        self.x_train = x_train_raw[train_idx]
        self.y_train = y_train_raw[train_idx]
        self.x_val = x_train_raw[val_idx]
        self.y_val = y_train_raw[val_idx]
        self.x_test = x_test_raw
        self.y_test = y_test_raw
        self.index = np.arange(len(self.y_train))

    def split_indices(self, size: int, rng: np.random.Generator):
        indices = np.arange(size)
        rng.shuffle(indices)
        split = int(self.train_ratio * size)
        return indices[:split], indices[split:]

    def get_batch(self, rng: np.random.Generator, batch_size: int):
        choices = rng.choice(self.index, size=batch_size)
        return self.x_train[choices], self.y_train[choices]

    def get_validation(self):
        return self.x_val, self.y_val

    def get_test(self):
        return self.x_test, self.y_test
```

```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("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.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 nnx
from typing import Union, Tuple, List
import structlog
import math

from .config import ModelSettings

settings = ModelSettings()

log = structlog.get_logger()

#Pooling
class Pooling(nnx.Module):
    def __init__(self, *, window=(2,2), strides=(2,2), padding="SAME"):
        self.window = window
        self.strides = strides
        self.padding = padding

    def __call__(self, x, train=False):
        return jax.lax.reduce_window(
            x,
            -jnp.inf,
            jax.lax.max,
            window_dimensions=(1, self.window[0], self.window[1], 1),
            window_strides=(1, self.strides[0], self.strides[1], 1),
            padding=self.padding,
        )

# GroupNorm
class GroupNorm(nnx.Module):
    """
  Group normalization with learnable affine parameters per channel.
  """
    def __init__(self, *, num_channels: int, num_groups: int = 8, epsilon: float
 = 1e-5):
        # store simple config
        self.num_channels = int(num_channels)
        self.num_groups = int(num_groups)
        self.epsilon = float(epsilon)

        # choose group count that divides channels (fallback by decrementing)
        G = min(self.num_groups, self.num_channels)
        while G > 1 and (self.num_channels % G) != 0:
            G -= 1
        self._G = max(1, G)

        # learnable affine parameters (per-channel)
        # nnx.Param wraps a leaf parameter; we initialize to gamma=1, beta=0
        self.scale = nnx.Param(jnp.ones((self.num_channels,), dtype=jnp.float32)
)
        self.bias = nnx.Param(jnp.zeros((self.num_channels,), dtype=jnp.float32)
)

    def __call__(self, x: jnp.ndarray, train: bool = False) -> jnp.ndarray:
        # x: (N, H, W, C)
        N, H, W, C = x.shape
        assert C == self.num_channels, "GroupNorm: channel mismatch"

        # reshape into groups: (N, H, W, G, C_per_group
        Cg = C // self._G
        xg = x.reshape((N, H, W, self._G, Cg))

        # compute mean/var over (H, W, Cperg)
        mean = jnp.mean(xg, axis=(1, 2, 4), keepdims=True)
        var = jnp.var(xg, axis=(1, 2, 4), keepdims=True)

        xg = (xg - mean) / jnp.sqrt(var + self.epsilon)
```



```python
        x_norm = xg.reshape((N, H, W, C))

        # apply affine per-channel (broadcast over N,H,W)
        gamma = self.scale.value if hasattr(self.scale, "value") else self.scale
        beta = self.bias.value if hasattr(self.bias, "value") else self.bias
        gamma = gamma.reshape((1, 1, 1, C))
        beta = beta.reshape((1, 1, 1, C))

        return x_norm * gamma + beta

#Convolution
class Convolution(nnx.Module):
    """
  Single convolution with L2
  """
    def __init__(
        self,
        *,
        keys,
        L2_weight: float,
        in_features: int,
        out_features: int,
        kernel_size: Union[int, Tuple[int, int]],
        strides: Union[int, Tuple[int, int]],
        padding: str = "SAME",
    ):
        # store config
        self.L2_weight = float(L2_weight)
        self.padding = padding

        # normalize kernel and strides
        if isinstance(kernel_size, int):
            kh = kw = int(kernel_size)
        else:
            kh, kw = map(int, kernel_size)

        if isinstance(strides, int):
            sh = sw = int(strides)
        else:
            sh, sw = strides

        self.kernel_size = (kh, kw)
        self.strides = (sh, sw)

        # compute fan_in and He (Kaiming) std for ReLU
        fan_in = kh * kw * int(in_features)
        he_std = math.sqrt(2.0 / float(fan_in))

        # generate kernel
        key = keys if keys is not None else jax.random.PRNGKey(0)
        kernel_shape = (kh, kw, int(in_features), int(out_features))
        init_kernel = jax.random.normal(key, kernel_shape, dtype=jnp.float32) *
jnp.array(he_std, dtype=jnp.float32)
        self.kernel = nnx.Param(init_kernel)

    def __call__(self, x: jnp.ndarray, train: bool = False) -> jnp.ndarray:
        """
    x expected shape: (N, H, W, C) (NHWC).
    Returns NHWC.
    """
        kernel_arr = self.kernel.value if hasattr(self.kernel, "value") else self
.kernel

        out = jax.lax.conv_general_dilated(
            lhs=x,
            rhs=kernel_arr,
            window_strides=self.strides,
            padding=self.padding,
            dimension_numbers=("NHWC", "HWIO", "NHWC"),
```

```
        )
        return out

    def L2_loss(self) -> jnp.ndarray:
        k = self.kernel.value if hasattr(self.kernel, "value") else self.kernel
        return jnp.sum(jnp.square(k)) * self.L2_weight


class GaussianNoise(nnx.Module):
    def __init__(self, *, std: float, key):
        self.std = std
        self.rngs = nnx.Rngs(noise=key)

    def __call__(self, x, train: bool = False):
        if not train or self.std <= 0:
            return x
        noise = jax.random.normal(self.rngs.noise(), x.shape) * self.std
        return x + noise

# ResidualBlock
class ResidualBlock(nnx.Module):
    """
  Basic residual block: Conv -> GroupNorm -> ReLU -> Conv -> GroupNorm -> skip -> ReLU
  """
    def __init__(
        self,
        *,
        keys,# tuple of three keys: (conv1_key, conv2_key, skip_proj_key)
        L2_weight: float,
        in_features: int,
        out_features: int,
        kernel_size: Tuple[int, int] = (3, 3),
        strides: Tuple[int, int] = (1, 1),
    ):
        conv1_key, conv2_key, proj_key = keys

        # conv1: in_features -> out_features
        self.conv1 = Convolution(
            keys=conv1_key,
            L2_weight=L2_weight,
            in_features=in_features,
            out_features=out_features,
            kernel_size=kernel_size,
            strides=strides,
        )
        self.norm1 = GroupNorm(num_channels=out_features)

        # conv2: out_features -> out_features
        self.conv2 = Convolution(
            keys=conv2_key,
            L2_weight=L2_weight,
            in_features=out_features,
            out_features=out_features,
            kernel_size=kernel_size,
            strides=(1, 1),
        )
        self.norm2 = GroupNorm(num_channels=out_features)

        # projection on skip if needed
        if in_features != out_features or strides != (1, 1):
            self.proj = nnx.Conv(
                in_features=in_features,
                out_features=out_features,
                kernel_size=(1, 1),
                strides=strides,
                padding="SAME",
                rngs=nnx.Rngs(params=proj_key),
            )
        else:
```



```
            self.proj = None

        self.noise = GaussianNoise(std=settings.noise_std, key = proj_key)

    def __call__(self, x: jnp.ndarray, train: bool = False) -> jnp.ndarray:
        identity = x

        out = self.conv1(x, train)
        out = self.norm1(out, train)
        out = jax.nn.relu(out)

        if train:
            out = self.noise(out,train)

        out = self.conv2(out, train)
        out = self.norm2(out, train)

        if self.proj is not None:
            identity = self.proj(identity)

        out = out + identity
        return jax.nn.relu(out)

    def L2_loss(self) -> jnp.ndarray:
        loss = self.conv1.L2_loss() + self.conv2.L2_loss()
        if self.proj is not None:
            loss += jnp.sum(jnp.square(self.proj.kernel.value))
        return loss

# classify
class Classify(nnx.Module):
    """
  CNN classifier built from ResidualBlocks.
  """
    def __init__(
        self,
        *,
        input_depth: int,
        layer_depths: List[int],
        layer_kernel_sizes: List[Tuple[int, int]],
        strides: List[Union[int, Tuple[int,int]]],
        num_classes: int,
        L2_weight: float,
        rngs: nnx.Rngs,
        shape: List[int],
    ):
        keys = rngs.params()              # root key (jax.Array)

        input_features = [input_depth] + layer_depths[:-1]
        layers = []

        # We'll need 3 keys per residual block plus one key for final linear:
        total_keys = len(layer_depths) * 3 + 1
        all_keys = jax.random.split(keys, total_keys)
        block_keys = all_keys[:-1]
        final_key = all_keys[-1]

        for i, out_ch in enumerate(layer_depths):
            k1, k2, k3 = block_keys[3*i:3*i+3]
            stride_val = (strides[i], strides[i]) if isinstance(strides[i], int)
 else strides[i]
            block = ResidualBlock(
                keys=(k1, k2, k3),
                L2_weight=L2_weight,
                in_features=input_features[i],
                out_features=out_ch,
                kernel_size=tuple(layer_kernel_sizes[i]),
                strides=stride_val,
            )
```

```
            layers.append(block)

            # pooling between blocks (except after last)
            if i < len(layer_depths) - 1:
                layers.append(Pooling(window=(2,2), strides=(2,2)))

        self.layers = nnx.List(layers)

        # compute flatten size after downsampling
        h, w = shape[0], shape[1]
        for s in strides:
            sh, sw = (s, s) if isinstance(s, int) else s
            h = math.ceil(h / sh)
            w = math.ceil(w / sw)
        # account for pooling between blocks
        h = h // (2 ** (len(layer_depths) - 1))
        w = w // (2 ** (len(layer_depths) - 1))
        flatten_size = h * w * layer_depths[-1]

        # final linear
        self.final_layer = nnx.Linear(
            in_features=flatten_size,
            out_features=num_classes,
            rngs=nnx.Rngs(params=final_key),
        )

    def __call__(self, x: jnp.ndarray, train: bool = False) -> jnp.ndarray:
        out = x
        for layer in self.layers:
            # pooling and blocks all accept (x, train) signature
            out = layer(out, train)
        out = out.reshape((out.shape[0], -1))
        return self.final_layer(out)

    def L2_loss(self) -> jnp.ndarray:
        return sum(layer.L2_loss() for layer in self.layers if hasattr(layer, "
L2_loss"))
```

```
==> Running homework 'hw04'...
/mnt/c/Users/joshu/OneDrive/Cooper Union/Deep Learning/hw04/.venv/lib/python3.13
/site-packages/pydantic/main.py:463: UserWarning: Pydantic serializer warnings:
  PydanticSerializationUnexpectedValue(Expected `int` - serialized value may not
 be as expected [input_value=0.05, input_type=float])
  return self.__pydantic_serializer__.to_python(
INFO:hw04:^[[2m2025-10-09T06:40:23.690409Z^[[0m [^[[32m^[[1minfo     ^[[0m] ^[[1
mSettings loaded               ^[[0m [^[[0m^[[1m^[[34mhw04^[[0m]^[[0m ^[[36msett
ings^[[0m=^[[35m{'debug': False, 'random_seed': 427, 'data': {'train_ratio': 0.8
}, 'training': {'batch_size': 256, 'num_iters': 150000, 'learning_rate': 0.00015
, 'L2_weight': 0.00015}, 'model': {'input_depth': 3, 'num_classes': 10, 'layer_d
epth': [64, 128, 256], 'layer_kernel_sizes': [[3, 3], [3, 3], [3, 3]], 'strides'
: [1, 1, 1], 'shape': [32, 32], 'noise_std': 0.05}}^[[0m
INFO:hw04:^[[2m2025-10-09T06:40:28.617145Z^[[0m [^[[32m^[[1minfo     ^[[0m] ^[[1
mcifar10 data retrieved        ^[[0m [^[[0m^[[1m^[[34mhw04^[[0m]^[[0m ^[[36mshap
e^[[0m=^[[35m(40000, 32, 32, 3)^[[0m
INFO:hw04.training:^[[2m2025-10-09T06:40:52.363057Z^[[0m [^[[32m^[[1minfo     ^[
[0m] ^[[1mstarted training             ^[[0m [^[[0m^[[1m^[[34mhw04.training^[[0
m]^[[0m

  0%|          | 0/150000 [00:00<?, ?it/s]
loss at 0 => 463.223602:   0%|          | 0/150000 [00:48<?, ?it/s]
loss at 50000 => 0.610804:  33%|âM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^N      | 5000
1/150000 [1:37:45<2:51:23,  9.72it/s]
loss at 100000 => 0.401484:  67%|âM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^Hâ
M-^VM-^HâM-^VM-^K   | 100001/150000 [3:11:07<1:23:43,  9.95it/s]
loss at 149999 => 0.378595: 100%|âM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^Hâ
M-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^H| 150000/150000 [4:45:29<00:00,  8.7
6it/s]
INFO:hw04.training:^[[2m2025-10-09T11:26:22.316114Z^[[0m [^[[32m^[[1minfo     ^[
[0m] ^[[1mdone training                ^[[0m [^[[0m^[[1m^[[34mhw04.training^[[0
m]^[[0m
INFO:hw04:^[[2m2025-10-09T11:26:22.317243Z^[[0m [^[[32m^[[1minfo     ^[[0m] ^[[1
mtraining concluded            ^[[0m [^[[0m^[[1m^[[34mhw04^[[0m]^[[0m
INFO:hw04:^[[2m2025-10-09T11:26:22.605988Z^[[0m [^[[32m^[[1minfo     ^[[0m] ^[[1
msaved most recent checkpoint  ^[[0m [^[[0m^[[1m^[[34mhw04^[[0m]^[[0m
INFO:hw04.test:^[[2m2025-10-09T11:27:09.263316Z^[[0m [^[[32m^[[1minfo     ^[[0m]
 ^[[1mEvaluated validation set: 10000 samples, 8740 correct, Accuracy=87.40%^[[0
m [^[[0m^[[1m^[[34mhw04.test^[[0m]^[[0m
INFO:hw04:^[[2m2025-10-09T11:27:09.263809Z^[[0m [^[[32m^[[1minfo     ^[[0m] ^[[1
mvalidation accuracy:          ^[[0m [^[[0m^[[1m^[[34mhw04^[[0m]^[[0m ^[[36mval_
accuracy^[[0m=^[[35m0.874^[[0m
INFO:hw04.test:^[[2m2025-10-09T11:27:11.177033Z^[[0m [^[[32m^[[1minfo     ^[[0m]
 ^[[1mEvaluated test set: 10000 samples, 8703 correct, Accuracy=87.03%^[[0m [^[[
0m^[[1m^[[34mhw04.test^[[0m]^[[0m
INFO:hw04:^[[2m2025-10-09T11:27:11.177436Z^[[0m [^[[32m^[[1minfo     ^[[0m] ^[[1
mreal test accuracy:           ^[[0m [^[[0m^[[1m^[[34mhw04^[[0m]^[[0m ^[[36maccu
racy^[[0m=^[[35m0.8703^[[0m
INFO:hw04:^[[2m2025-10-09T11:27:11.177541Z^[[0m [^[[32m^[[1minfo     ^[[0m] ^[[1
myippeeeee, close enough to state of the art i give up^[[0m [^[[0m^[[1m^[[34mhw0
4^[[0m]^[[0m
```

```
==> Running homework 'hw04'...
/mnt/c/Users/joshu/OneDrive/Cooper Union/Deep Learning/hw04/.venv/lib/python3.13
/site-packages/pydantic/main.py:463: UserWarning: Pydantic serializer warnings:
  PydanticSerializationUnexpectedValue(Expected `int` - serialized value may not
 be as expected [input_value=0.05, input_type=float])
  return self.__pydantic_serializer__.to_python(
INFO:hw04:^[[2m2025-10-09T00:53:22.982046Z^[[0m [^[[32m^[[1minfo     ^[[0m] ^[[1
mSettings loaded               ^[[0m [^[[0m^[[1m^[[34mhw04^[[0m]^[[0m ^[[36msett
ings^[[0m=^[[35m{'debug': False, 'random_seed': 427, 'data': {'train_ratio': 0.8
}, 'training': {'batch_size': 256, 'num_iters': 175000, 'learning_rate': 0.00015
, 'L2_weight': 0.00015}, 'model': {'input_depth': 3, 'num_classes': 100, 'layer_
depth': [64, 128, 256], 'layer_kernel_sizes': [[3, 3], [3, 3], [3, 3]], 'strides
': [1, 1, 1], 'shape': [32, 32], 'noise_std': 0.05}}^[[0m
INFO:hw04:^[[2m2025-10-09T00:53:28.617587Z^[[0m [^[[32m^[[1minfo     ^[[0m] ^[[1
mcifar100 data retrieved       ^[[0m [^[[0m^[[1m^[[34mhw04^[[0m]^[[0m ^[[36mshap
e^[[0m=^[[35m(40000, 32, 32, 3)^[[0m
INFO:hw04.training:^[[2m2025-10-09T00:53:48.260701Z^[[0m [^[[32m^[[1minfo     ^[
[0m] ^[[1mstarted training             ^[[0m [^[[0m^[[1m^[[34mhw04.training^[[0
m]^[[0m

  0%|          | 0/175000 [00:00<?, ?it/s]
loss at 0 => 465.868805:   0%|          | 0/175000 [00:44<?, ?it/s]
loss at 50000 => 1.318601:  29%|âM-^VM-^HâM-^VM-^HâM-^VM-^J       | 50001/175000
 [1:36:43<3:54:45,  8.87it/s]
loss at 100000 => 0.769379:  57%|âM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^Hâ
M-^VM-^K    | 100001/175000 [3:12:34<2:10:13,  9.60it/s]
loss at 150000 => 0.637122:  86%|âM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^Hâ
M-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^L | 150001/175000 [4:47:41<41:22, 10.07it/s]
loss at 174964 => 0.592589: 100%|âM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^Hâ
M-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^I| 174965/175000 [5:35:21<00:03,  9.7
6it/s]
loss at 174999 => 0.812800: 100%|âM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^Hâ
M-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^HâM-^VM-^H| 175000/175000 [5:35:28<00:00,  8.6
9it/s]
INFO:hw04.training:^[[2m2025-10-09T06:29:16.912630Z^[[0m [^[[32m^[[1minfo     ^[
[0m] ^[[1mdone training                ^[[0m [^[[0m^[[1m^[[34mhw04.training^[[0
m]^[[0m
INFO:hw04:^[[2m2025-10-09T06:29:16.921884Z^[[0m [^[[32m^[[1minfo     ^[[0m] ^[[1
mtraining concluded            ^[[0m [^[[0m^[[1m^[[34mhw04^[[0m]^[[0m
INFO:hw04:^[[2m2025-10-09T06:29:17.291625Z^[[0m [^[[32m^[[1minfo     ^[[0m] ^[[1
msaved most recent checkpoint  ^[[0m [^[[0m^[[1m^[[34mhw04^[[0m]^[[0m
INFO:hw04.test:^[[2m2025-10-09T06:30:04.425090Z^[[0m [^[[32m^[[1minfo     ^[[0m]
 ^[[1mEvaluated validation set: 10000 samples, 8167 correct (top-5), Accuracy=81
.67%^[[0m [^[[0m^[[1m^[[34mhw04.test^[[0m]^[[0m
INFO:hw04:^[[2m2025-10-09T06:30:04.425528Z^[[0m [^[[32m^[[1minfo     ^[[0m] ^[[1
mvalidation top5 accuracy:     ^[[0m [^[[0m^[[1m^[[34mhw04^[[0m]^[[0m ^[[36mval_
accuracy^[[0m=^[[35m0.8167^[[0m
INFO:hw04.test:^[[2m2025-10-09T06:30:06.457077Z^[[0m [^[[32m^[[1minfo     ^[[0m]
 ^[[1mEvaluated test set: 10000 samples, 8134 correct (top-5), Accuracy=81.34%^[
[0m [^[[0m^[[1m^[[34mhw04.test^[[0m]^[[0m
INFO:hw04:^[[2m2025-10-09T06:30:06.457469Z^[[0m [^[[32m^[[1minfo     ^[[0m] ^[[1
mreal test top5 accuracy:      ^[[0m [^[[0m^[[1m^[[34mhw04^[[0m]^[[0m ^[[36maccu
racy^[[0m=^[[35m0.8134^[[0m
INFO:hw04:^[[2m2025-10-09T06:30:06.457569Z^[[0m [^[[32m^[[1minfo     ^[[0m] ^[[1
mthis is not state of the art :(^[[0m [^[[0m^[[1m^[[34mhw04^[[0m]^[[0m
```

I started by literally training on cifar10 without changing anythin meaning
i didnt use data augmentaation, add group normalization, or residual blocks.
this was purely a CNN w 2 convolutional layers followedby fully connected layers
and i got like 65% which isny bad for relying almost solely on the inductive
biases of a convolution. It trained quickly but obviously didnt generalize.

I then added data augmentation with crop, color jitter, flips(vertical and horiz
ontal),
and 90 degree rotations which yielded a cifar10 accuracy of rouhgly 80%. Adding
group normalization made it less intensive for my GPU to train larger batch size
s faster.
Additionally, i used the residual blocks from the nnx module to implement skip
connections. i used kaiming he initialization bc the PReLU paper said it was goo
d lol.
All of this got me to an accuracy of like 92% on the validation set which was my
 best
score and better than what i currently have lol. I then added awgn to activation
s during training
because i thought that it would help regualarize and be able to simmulate data v
ariability but
it low key made no difference. I actually thought i was so smart for adding awgn
 in the modul
but i guess im just not that guy. After hitting a plateu, i tried adjusting my d
ata augmentations
by getting rid of rotation bc it was just too annoying to consider how the orien
tation of certain
objects should be included also jax has no rotation function so unless its by 90
 degrees, idk how
to rotate easily. I then added awgn to the pictures to try and make the models m
ore robust as well
as greyscale, as suggest by Vaibhav. He also said to try inverting the colors an
d Professor,
you are so right because he said that you said it makes no fucking sense bc why
would it? His
rational was that a green frog inverted to a black frog is still a frog???? firs
t of all inverting
green is not black, its magenta. and second of all, most animals come only in ce
rtain colors??? Anyway,
sorry for the side tangent. It must have been Vaibav's bad luck because my model
 declined in performance
after including awgn and greyscale to the data augmentation to 88% on the valida
tion set so that's where
I'm at now. It's not state of the art. Every paper boasts 99% on CIFAR10 but i f
ound some random guy
boasting his 93% which i will interpret as state of the art bc he is all but one
 man and not the
DL avengers assembled to conquer CIFAR10, CIFAR100, and every possible dataset.

It's worth noting that for the trial ruuns, i trained until the loss started to
converge to a number
around 1 which took about 15k its and 20 minutes. I really thought increasing th
at number to like 200k
would help because the loss would drop to like .2 in some cases but i only got a
n increased performance
of a couple percent meaning that i probably reached a point where i was overfitt
ing.

Sorry that i couldnt even get 90 percent on CIFAR100 top5 accuacy btw despite ev
en my extension. I tried;
I really did. Thanks for the extension though.

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

from .model import Classify
from .data import cifar10

log = structlog.get_logger()

def test_accuracy(
    model: Classify,
    data: cifar10,
    batch_size: int,
    validation: bool = True
) -> float:

    if validation == True:
        x_np, y_np = data.get_validation()
    else:
        x_np, y_np = data.get_test()

    num_samples = x_np.shape[0]

    # ensure labels are 1-D integer arrays
    y_np = np.asarray(y_np).reshape(-1)

    #batch predictions
    preds = []
    for start in range(0, num_samples, batch_size):
        end = start + batch_size
        xb = jnp.asarray(x_np[start:end])
        logits = model(xb, train=False)
        batch_pred = jnp.argmax(logits, axis=-1)
        preds.append(batch_pred)

    #compute accuracy
    all_preds = jnp.concatenate(preds, axis=0)
    all_labels = jnp.asarray(y_np)
    correct = jnp.sum(all_preds == all_labels)
    accuracy = correct / num_samples

    log.info(
        "Evaluated %s set: %d samples, %d correct, Accuracy=%.2f%%",
        "validation" if validation else "test",
        int(num_samples),
        int(correct),
        float(accuracy * 100),
    )

    return float(accuracy)

def test_top5(
    model: Classify,
    data: cifar10,
    batch_size: int,
    validation: bool = True
) -> float:
    """Compute top-5 accuracy on the validation or test set."""

    if validation:
        x_np, y_np = data.get_validation()
    else:
        x_np, y_np = data.get_test()

    num_samples = x_np.shape[0]

    # ensure labels are 1-D integer arrays
    y_np = np.asarray(y_np).reshape(-1)
```



```python
    correct_top5 = 0

    for start in range(0, num_samples, batch_size):
        end = start + batch_size
        xb = jnp.asarray(x_np[start:end])
        yb = jnp.asarray(y_np[start:end])

        logits = model(xb, train=False)

        # get indices of top 5 predictions per sample
        top5 = jnp.argsort(logits, axis=-1)[:, -5:]  # shape (batch_size, 5)

        # check if true label is in top 5 predictions
        correct_batch = jnp.any(top5 == yb[:, None], axis=-1)
        correct_top5 += jnp.sum(correct_batch)

    top5_accuracy = correct_top5 / num_samples

    log.info(
        "Evaluated %s set: %d samples, %d correct (top-5), Accuracy=%.2f%%",
        "validation" if validation else "test",
        int(num_samples),
        int(correct_top5),
        float(top5_accuracy * 100),
    )

    return float(top5_accuracy)
```

```python
import jax.numpy as jnp
import jax
import numpy as np
import optax
import structlog
import orbax.checkpoint as ocp
from typing import Optional
from pathlib import Path

from flax import nnx
from tqdm import trange
from .config import TrainingSettings
from .data import cifar10
from .model import Classify

log = structlog.get_logger()

@nnx.jit
def train_step_cifar10(
    model: Classify,
    optimizer: nnx.Optimizer,
    x: jnp.ndarray,
    y: jnp.ndarray,
 ):

    """perform on training step"""

    # x shape: (batch_size, 32, 32, 3); y shape: (batch_size,)
    def loss_function(model: Classify):
        logits = model(x, True)
        ce_loss = jnp.mean(optax.softmax_cross_entropy_with_integer_labels(logit
s, y))
        L2_loss = model.L2_loss()

        return ce_loss + L2_loss

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

def augment(
    key: jnp.array = jax.random.PRNGKey(472),
    images: jnp.array = jnp.ndarray,
    *,
    p_hflip: float = 0.35, #probabilities for performing these augmentations
    pad: int = 4,
    brightness: float = 0.2,#changes in brightness, conttrast, and saturation
    contrast: float = 0.2,
    saturation: float = 0.2,
    p_cutout: float = 0.35,
    # i define this in the cutout function bc i break vmap otherwise and im too
lazy to figure out why: max_cutout: int = 8,
    p_greyscale: float = 0.3,
    p_noise: float = 0.85,
    noise_std: float = 0.15,    # stddev of additive Gaussian noise
) -> jnp.ndarray:
    """
  Batch augmentation
  images: float32 in [0,1], shape (B,H,W,C).
  Returns augmented images, same shape and dtype.
  """
    B, H, W, C = images.shape

    # Split top-level key into independent subkeys.
    # We need separate keys for flips, crop, color, cutout, greyscale, noise_mas
k, noise_rng.
    key, k_flip_h, k_crop, k_color, k_cutout, k_grey, k_noise_mask, k_noise = (
        jax.random.split(key, 8)
    )
```



```python
    """i got rid of rotation because many classes in the cifar datasets dont make sense to flip and rotating
  it by a small amount instead didnt really help much. also it probably didnt help that i couldnt find a roation funciton i
n the
  jax documentation lol"""

    # Horizontal flip
    hmask = jax.random.bernoulli(k_flip_h, p_hflip, (B,))
    images = jnp.where(hmask[:, None, None, None], images[:, :, ::-1, :], images
)

    """i also axed vertical flip because its genunely just not worth it given that many of the classes in both cifar data
sets
  #wouldnt realistically have many vertically flipped images"""

    # Crop (random crop from reflected padding)
    padded = jnp.pad(images, ((0, 0), (pad, pad), (pad, pad), (0, 0)), mode="refl
ect")
    max_x = padded.shape[1] - H
    max_y = padded.shape[2] - W
    k_crop_x, k_crop_y = jax.random.split(k_crop, 2)
    xs = jax.random.randint(k_crop_x, (B,), 0, max_x + 1)
    ys = jax.random.randint(k_crop_y, (B,), 0, max_y + 1)

    def crop_one(img, x, y):
        return jax.lax.dynamic_slice(img, (x, y, 0), (H, W, C))

    images = jax.vmap(crop_one)(padded, xs, ys)

    # Color jitter (brightness / contrast / saturation)
    k_b, k_c, k_s = jax.random.split(k_color, 3)
    bright_f = jax.random.uniform(k_b, (B,), minval=-brightness, maxval=brightne
ss)
    contrast_f = jax.random.uniform(k_c, (B,), minval=1.0 - contrast, maxval=1.0
 + contrast)
    sat_f = jax.random.uniform(k_s, (B,), minval=-saturation, maxval=saturation)

    def apply_color(bf, cf, sf, img):
        # brightness
        img = img + bf
        # contrast
        mean = jnp.mean(img, axis=(0, 1), keepdims=True)
        img = (img - mean) * cf[..., None, None] + mean
        # saturation (convert to luminance and lerp)
        lum = img[..., 0] * 0.2989 + img[..., 1] * 0.5870 + img[..., 2] * 0.1140
        lum = lum[..., None]
        img = img * (1.0 - sf)[..., None, None] + lum * sf[..., None, None]
        return jnp.clip(img, 0.0, 1.0)

    images = jax.vmap(apply_color)(bright_f, contrast_f, sat_f, images)

    # Cutout
    cut_mask = jax.random.bernoulli(k_cutout, p_cutout, (B,))  # which images ge
t cutout
    # create 4 keys per image (one for cut_h, cut_w, ky, kx)
    per_cut_keys = jax.random.split(k_cutout, B * 4).reshape((B, 4, 2))

    def do_cutout(karr, do, img, max_cutout = 8):
        Hc, Wc, Cc = img.shape

        def true_fn(args):
            karr, img = args
            k0, k1, k2, k3 = karr

            # pick random cutout size
            cut_h = jax.random.randint(k0, (), 1, max_cutout + 1)
            cut_w = jax.random.randint(k1, (), 1, max_cutout + 1)

            # pick random top-left corner
```

```
            ky = jax.random.randint(k2, (), 0, Hc - cut_h + 1)
            kx = jax.random.randint(k3, (), 0, Wc - cut_w + 1)

            # create a mask of ones (static shape)
            mask = jnp.ones((Hc, Wc, Cc), dtype=img.dtype)

            # create indices for cutout
            rows = jnp.arange(Hc)
            cols = jnp.arange(Wc)
            row_mask = (rows < ky) | (rows >= ky + cut_h)
            col_mask = (cols < kx) | (cols >= kx + cut_w)

            # broadcast row/col masks
            cut_mask_local = jnp.outer(row_mask, col_mask).astype(img.dtype)
            cut_mask_local = cut_mask_local[..., None]  # expand to (H, W, 1)
            mask = mask * cut_mask_local           # apply mask

            return img * mask

        def false_fn(args):
            return args[1]

        return jax.lax.cond(do, true_fn, false_fn, (karr, img))

    images = jax.vmap(do_cutout)(per_cut_keys, cut_mask, images)

    #greyscale
    if p_greyscale > 0.0:
        grey_mask = jax.random.bernoulli(k_grey, p_greyscale, (B,))  # which ima
ges -> greyscale
        # compute luminance and stack to 3 channels for those images
        lum = images[..., 0] * 0.2989 + images[..., 1] * 0.5870 + images[..., 2]
 * 0.1140  # (B,H,W)
        lum = lum[..., None]  # (B,H,W,1)
        grey_images = jnp.concatenate([lum, lum, lum], axis=-1)  # (B,H,W,3)
        images = jnp.where(grey_mask[:, None, None, None], grey_images, images)

    #awgn
    if p_noise > 0.0 and noise_std > 0.0:
        noise_mask = jax.random.bernoulli(k_noise_mask, p_noise, (B,))
        noise = jax.random.normal(k_noise, (B, H, W, C)) * noise_std
        noisy = images + noise
        noisy = jnp.clip(noisy, 0.0, 1.0)
        images = jnp.where(noise_mask[:, None, None, None], noisy, images)

    return jnp.clip(images, 0.0, 1.0)

def cifar10_train(
    model: Classify,
    optimizer: nnx.Optimizer,
    data: cifar10,
    settings: TrainingSettings,
    np_rng: np.random.Generator,
) -> None:
    """train with grad descend"""
    log.info("started training")

    # create a persistent jnp PRNGKey seeded from the numpy RNG you already have
    jnp_key = jax.random.PRNGKey(int(np_rng.integers(0, 2 ** 31 - 1)))

    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, dtype=jnp.float32)
        y = jnp.asarray(y_np, dtype=jnp.int32).reshape(-1)

        # get subkey and augment the batch (augmentation is outside the jitted t
rain step)
        jnp_key, subkey = jax.random.split(jnp_key)
```



```
        x_aug = augment(subkey, x)

        loss = train_step_cifar10(model, optimizer, x_aug, y)

        bar.set_description(f"loss at {i} => {loss:.6f}")
        bar.refresh()

    log.info("done training")
```

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

from .config import load_settings
from .data import cifar10
from .model import Classify
from .logging import configure_logging
from .training import cifar10_train
from .test import test_accuracy, test_top5

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(472) #472 for ECE 472
    data_key, model_key = jax.random.split(key)
    np_rng = np.random.default_rng(np.array(data_key))

    data = cifar10(
        rng = np_rng,
        train_ratio = settings.data.train_ratio,
    )
    if settings.model.num_classes == 10:
        log.info("cifar10 data retrieved", shape = data.x_train.shape)
    else:
        log.info("cifar100 data retrieved", shape = data.x_train.shape)

    model = Classify(
        rngs = nnx.Rngs(params=model_key),
        num_classes = settings.model.num_classes,
        shape = settings.model.shape,
        input_depth = settings.model.input_depth,
        layer_depths = settings.model.layer_depth,
        layer_kernel_sizes = settings.model.layer_kernel_sizes,
        strides = settings.model.strides,
        L2_weight = settings.training.L2_weight,
    )

    optimizer_schedule = optax.cosine_decay_schedule(settings.training.learning_
rate, settings.training.num_iters,) #same schedule as you
    optimizer = nnx.Optimizer(
        model, optax.adam(optimizer_schedule), wrt=nnx.Param
    )

    cifar10_train(model, optimizer, data, settings.training, np_rng)
    log.info("training concluded")

    ckpt_dir = ocp.test_utils.erase_and_create_empty("/saves")
    graphdef, state = nnx.split(model)
    checkpointer = ocp.StandardCheckpointer()
    checkpointer.save(ckpt_dir/ "cifar10", state)
    log.info("saved most recent checkpoint")

    #validation and test set and stuff
    if settings.model.num_classes == 10:
        validation_accuracy = test_accuracy(
            data = data,
            batch_size = settings.training.batch_size,
            model = model,
            validation = True,
        )
        log.info("validation accuracy:", val_accuracy = validation_accuracy)
        real_test_accuracy = test_accuracy(
            data = data,
            batch_size = settings.training.batch_size,
            model = model,
            validation = False,
        )
        log.info("real test accuracy:", accuracy = real_test_accuracy)

    else:
        validation_accuracy = test_top5(
            data = data,
            batch_size = settings.training.batch_size,
            model = model,
            validation = True,
        )
        log.info("validation top5 accuracy:", val_accuracy = validation_accuracy)
        real_test_accuracy = test_top5(
            data = data,
            batch_size = settings.training.batch_size,
            model = model,
            validation = False,
        )
        log.info("real test top5 accuracy:", accuracy = real_test_accuracy)

    if real_test_accuracy >= 0.85:
        log.info("yippeeeee, close enough to state of the art i give up")
    else:
        log.info("this is not state of the art :(")

def test10():
    settings = load_settings()
    configure_logging()
    log = structlog.get_logger()
    log.info("Settings loaded", settings=settings.model_dump())

    # JAX PRNG
    key = jax.random.PRNGKey(472) #472 for ECE 472
    data_key, model_key = jax.random.split(key)
    np_rng = np.random.default_rng(np.array(data_key))

    log.info("evalutating on test set for CIFAR10!!")

    model = Classify(
        rngs = nnx.Rngs(params=model_key),
        num_classes = settings.model.num_classes,
        shape = settings.model.shape,
        input_depth = settings.model.input_depth,
        layer_depths = settings.model.layer_depth,
        layer_kernel_sizes = settings.model.layer_kernel_sizes,
        strides = settings.model.strides,
        L2_weight = settings.training.L2_weight,
    )

    data = cifar10(
        rng = np_rng,
        train_ratio = settings.data.train_ratio,
    )

    ckpt_dir = Path("/saves").resolve()
    checkpointer = ocp.StandardCheckpointer()
    graphdef, state = nnx.split(model)
    state = jax.tree_util.tree_map(ocp.utils.to_shape_dtype_struct, state)
    restored = checkpointer.restore(ckpt_dir/ 'most_recent', state)
    model = nnx.merge(graphdef, restored)
    log.info("restored model from checkpoint", model = model)

    real_test_accuracy = test_accuracy(
```

```
            data = data,
            batch_size = settings.training.batch_size,
            model = model,
            validation = False,
        )
    log.info("real test accuracy:", accuracy = real_test_accuracy)

def test100():
    settings = load_settings()
    configure_logging()
    log = structlog.get_logger()
    log.info("Settings loaded", settings=settings.model_dump())

    # JAX PRNG
    key = jax.random.PRNGKey(472) #472 for ECE 472
    data_key, model_key = jax.random.split(key)
    np_rng = np.random.default_rng(np.array(data_key))

    log.info("evalutating on test set for CIFAR100!!")

    model = Classify(
        rngs = nnx.Rngs(params=model_key),
        num_classes = settings.model.num_classes,
        shape = settings.model.shape,
        input_depth = settings.model.input_depth,
        layer_depths = settings.model.layer_depth,
        layer_kernel_sizes = settings.model.layer_kernel_sizes,
        strides = settings.model.strides,
        L2_weight = settings.training.L2_weight,
    )

    data = cifar10( #should be cifar100 but im too lazy to change
        rng = np_rng,
        train_ratio = settings.data.train_ratio,
    )
    ckpt_dir = Path("/saves").resolve()
    checkpointer = ocp.StandardCheckpointer()
    graphdef, state = nnx.split(model)
    state = jax.tree_util.tree_map(ocp.utils.to_shape_dtype_struct, state)
    restored = checkpointer.restore(ckpt_dir/ 'most_recent', state)
    model = nnx.merge(graphdef, restored)
    log.info("restored model from checkpoint", model = model)

    real_test_accuracy = test_accuracy(
            data = data,
            batch_size = settings.training.batch_size,
            model = model,
            validation = False,
        )
    log.info("real test accuracy:", accuracy = real_test_accuracy)
```