```
root@LegionT5:/mnt/c/Users/joshu/OneDrive/Cooper Union/Deep Learning/hw06# just
run hw06
==> Running homework 'hw06'...
2025-11-06T04:46:43.263310Z [info     ] Loaded settings                [hw06] se
ttings={'d_model': 64, 'num_heads': 4, 'ff_dim': 256, 'dropout_rate': 0.0, 'use_
bias': True}
2025-11-06T04:46:44.433905Z [info     ] building TransformerBlock      [hw06] d_
model=64 dropout=0 ff_dim=256 num_heads=4
2025-11-06T04:46:44.434109Z [info     ] Running tests. im so nervous   [hw06]
2025-11-06T04:46:58.162895Z [info     ] test_result                    [hw06.tes
ts] message=passed passed=True test=mha_shape_and_attn_sums
2025-11-06T04:47:08.710704Z [info     ] test_result                    [hw06.tes
ts] message=passed passed=True test=mha_deterministic
2025-11-06T04:47:21.698067Z [info     ] test_result                    [hw06.tes
ts] message=passed passed=True test=mha_permutation_equivariance
2025-11-06T04:48:05.501712Z [info     ] test_result                    [hw06.tes
ts] message=passed passed=True test=transformer_grad_flow
2025-11-06T04:48:24.393277Z [info     ] test_result                    [hw06.tes
ts] message=passed passed=True test=head_collapse_single_vs_manual
2025-11-06T04:48:33.442183Z [info     ] test_result                    [hw06.tes
ts] message=passed passed=True test=softmax_stability_scaling
2025-11-06T04:48:42.185660Z [info     ] test_result                    [hw06.tes
ts] message=passed passed=True test=head_additivity_and_reconstruction
2025-11-06T04:48:45.954266Z [info     ] test_result                    [hw06.tes
ts] message=passed passed=True test=zero_out_head_and_contribution
2025-11-06T04:48:46.901266Z [info     ] test_result                    [hw06.tes
ts] message='passed (AssertionError)' passed=True test=divisibility_check_error
2025-11-06T04:48:53.731616Z [info     ] test_result                    [hw06.tes
ts] message=passed passed=True test=single_token_T1
2025-11-06T04:49:17.870213Z [info     ] test_result                    [hw06.tes
ts] message=passed passed=True test=backprop_through_attention
2025-11-06T04:49:17.870432Z [info     ] Test summary                   [hw06] pa
ssed=True results=[('mha_shape_and_attn_sums', True, 'passed'), ('mha_determinis
tic', True, 'passed'), ('mha_permutation_equivariance', True, 'passed'), ('trans
former_grad_flow', True, 'passed'), ('head_collapse_single_vs_manual', True, 'pa
ssed'), ('softmax_stability_scaling', True, 'passed'), ('head_additivity_and_rec
onstruction', True, 'passed'), ('zero_out_head_and_contribution', True, 'passed'
), ('divisibility_check_error', True, 'passed (AssertionError)'), ('single_token
_T1', True, 'passed'), ('backprop_through_attention', True, 'passed')]
2025-11-06T04:49:17.870643Z [info     ] All tests passed yippeeeeeeeeeeee [hw06]
```

```
# pyproject.toml.jinja

[project]
name = "hw06"
version = "0.1.0"
description = "Transformer Block with Multi Head Attention"
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>=2.12.3",
    "pydantic-settings>=2.11.0",
    "optax>=0.2.6",
    "pathlib>=1.0.1",
    "importlib>=1.0.4",
]

[project.scripts]
hw06 = "hw06: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, List

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

class ModelSettings(BaseModel):
    d_model: int = 64
    num_heads: int = 4
    ff_dim: int = 256 #feedforward dimension
    dropout_rate: float = 0.0
    use_bias: bool = True

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

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

    model_config = SettingsConfigDict(
        toml_file=files("hw06").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
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("hw06").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 structlog
import jax
import jax.numpy as jnp
import flax.linen as nnx
from typing import Optional, Tuple
from .config import ModelSettings

log = structlog.get_logger()

Array = jnp.ndarray

class MultiHeadAttention(nnx.Module):
    """Multi-head self-attention (1-D sequence)
  Returns (out, attn_weights) when return_attn=True to aid testing.
  """
    d_model: int
    num_heads: int
    dropout_rate: float = 0.0
    use_bias: bool = True

    @nnx.compact
    def __call__(self,
                 x: Array,
                 deterministic: bool = True,
                 return_attn: bool = False
                 ) -> Tuple[Array, Optional[Array]]:
        """
    x: (batch, seq_len, d_model)
    """
        B, T, D = x.shape
        assert D == self.d_model, "input embedding dim must match d_model"
        assert D % self.num_heads == 0, "d_model must be divisible by num_heads"
        head_dim = D // self.num_heads

        log.debug("MHA.call", batch=B, seq_len=T, d_model=D, num_heads=self.num_h
eads, head_dim=head_dim)

        # Combined linear for qkv: (B, T, 3*D)
        qkv = nnx.Dense(3 * D, use_bias=self.use_bias, name="qkv_proj")(x)
        # split and reshape => (B, num_heads, T, head_dim)
        qkv = qkv.reshape(B, T, 3, self.num_heads, head_dim)
        qkv = jnp.transpose(qkv, (2, 0, 3, 1, 4))  # (3, B, H, T, head_dim)
        q, k, v = qkv[0], qkv[1], qkv[2]

        # compute attention logits: (B, H, T, T)
        logits = jnp.einsum("bhqd,bhkd->bhqk", q, k)
        logits = logits * (1.0 / jnp.sqrt(head_dim))

        attn = jax.nn.softmax(logits, axis=-1)
        attn = nnx.Dropout(rate=self.dropout_rate)(attn, deterministic=determini
stic)

        out_per_head = jnp.einsum("bhqk,bhkd->bhqd", attn, v)
        out_per_head = jnp.transpose(out_per_head, (0, 2, 1, 3))  # (B, T, H, he
ad_dim)
        out = out_per_head.reshape(B, T, D)

        out = nnx.Dense(D, use_bias=self.use_bias, name="out_proj")(out)
        out = nnx.Dropout(rate=self.dropout_rate)(out, deterministic=determinist
ic)

        log.debug("MHA.output", out_shape=out.shape)

        if return_attn: # attn shape: (B, H, T, T)
            return out, attn
        return out, None


class TransformerBlock(nnx.Module):
```



```python
    """Single Transformer block with pre-LN, MHA, feed-forward, residuals.
  Designed for 1-D sequences. LayerNorm before sublayer
  """
    d_model: int
    num_heads: int
    ff_dim: int
    dropout_rate: float = 0.0
    pre_norm: bool = True

    @nnx.compact
    def __call__(self,
                 x: Array,
                 deterministic: bool = True,
                 return_attn: bool = False
                 ) -> Tuple[Array, Optional[Array]]:
        """
    x: (B, T, d_model)
    returns (out, attn_weights_if_requested)
    """
        log.debug("TransformerBlock.call", x_shape=x.shape, pre_norm=self.pre_norm)

        #MHA sublayer
        if self.pre_norm:
            y = nnx.LayerNorm(name="ln_1")(x)
        else:
            y = x

        mha = MultiHeadAttention(self.d_model, self.num_heads, dropout_rate=self
.dropout_rate, name="mha")
        attn_out, attn_weights = mha(y, deterministic=deterministic, return_attn
=return_attn)
        x = x + attn_out
        log.debug("TransformerBlock.after_mha", x_shape=x.shape)

        #Feed-forward sublayer
        if self.pre_norm:
            y = nnx.LayerNorm(name="ln_2")(x)
        else:
            y = x

        y = nnx.Dense(self.ff_dim, name="ff_1")(y)
        y = nnx.relu(y)
        y = nnx.Dropout(rate=self.dropout_rate)(y, deterministic=deterministic)
        y = nnx.Dense(self.d_model, name="ff_2")(y)
        y = nnx.Dropout(rate=self.dropout_rate)(y, deterministic=deterministic)

        out = x + y
        log.debug("TransformerBlock.output", out_shape=out.shape)
        return out, (attn_weights if return_attn else None)
```

```python
# tests.py
import structlog
import jax
import jax.numpy as jnp
import numpy as np
from jax import random
from typing import List, Tuple
from .model import MultiHeadAttention, TransformerBlock

log = structlog.get_logger()
KEY = random.PRNGKey(427)


def _init_module(mod, x, rng=KEY):
    return mod.init(rng, x, deterministic=True)


def test_mha_shape_and_attn_sums() -> Tuple[bool, str]:
    """Shape checks and attention rows sum to 1 after softmax."""
    B, T, D = 2, 8, 32
    H = 4
    x = random.normal(KEY, (B, T, D))

    mha = MultiHeadAttention(d_model=D, num_heads=H, dropout_rate=0.0)
    params = _init_module(mha, x)

    out, attn = mha.apply(params, x, deterministic=True, return_attn=True)

    assert out.shape == (B, T, D)
    assert attn.shape == (B, H, T, T)
    # each query distribution sums to 1
    row_sums = jnp.sum(attn, axis=-1)
    assert jnp.allclose(row_sums, jnp.ones_like(row_sums), atol=1e-5)
    return True, "passed"

def test_mha_deterministic_behavior() -> Tuple[bool, str]:
    """Fixed PRNG/ deterministic=True yields identical outputs across calls."""
    B, T, D = 2, 6, 16
    H = 2
    x = random.normal(KEY, (B, T, D))
    mha = MultiHeadAttention(d_model=D, num_heads=H, dropout_rate=0.0)
    params = _init_module(mha, x)

    out1, _ = mha.apply(params, x, deterministic=True, return_attn=False)
    out2, _ = mha.apply(params, x, deterministic=True, return_attn=False)

    assert jnp.allclose(out1, out2), "outputs differ despite deterministic=True"
    return True, "passed"

def test_permutation_equivariance_mha() -> Tuple[bool, str]:
    """Without positional encodings, MHA should be permutation equivariant:
    permuting input positions permutes outputs the same way.
  """
    B, T, D = 2, 7, 24
    H = 3
    x = random.normal(KEY, (B, T, D))
    perm = jax.random.permutation(KEY, T)

    mha = MultiHeadAttention(d_model=D, num_heads=H, dropout_rate=0.0)
    params = _init_module(mha, x)

    out, _ = mha.apply(params, x, deterministic=True, return_attn=False)
    out_perm, _ = mha.apply(params, x[:, perm, :], deterministic=True, return_at
tn=False)

    # out_perm should equal out with positions permuted
    assert jnp.allclose(out[:, perm, :], out_perm, atol=1e-6)
    return True, "passed"
```



```python
def test_transformerblock_grad_flow_basic() -> Tuple[bool, str]:
    """Quick gradient sanity check: gradient of sum(out) wrt inputs is nonzero."""

    B, T, D = 1, 5, 16
    x = random.normal(KEY, (B, T, D))

    block = TransformerBlock(d_model=D, num_heads=4, ff_dim=4 * D, dropout_rate=
0.0)
    params = block.init(KEY, x, deterministic=True)

    def loss_fn(inp):
        out, _ = block.apply(params, inp, deterministic=True, return_attn=False)
        return jnp.sum(out)

    g = jax.grad(loss_fn)(x)
    # gradient should have same shape and not be all zeros
    assert g.shape == x.shape
    assert jnp.any(jnp.abs(g) > 0.0)
    return True, "passed"

def _manual_single_head_forward_from_params(params, x):
    """Given params from MultiHeadAttention with num_heads=1, compute manual forward."""
    log.debug("manual_single_head_forward start", x_shape=x.shape)
    p = params['params']
    qkv_k = p['qkv_proj']['kernel']
    qkv_b = p['qkv_proj'].get('bias', None)
    out_k = p['out_proj']['kernel']
    out_b = p['out_proj'].get('bias', None)

    B, T, D = x.shape
    H = 1
    head_dim = D // H

    qkv = jnp.einsum("btd,df->btf", x, qkv_k) #big thanks to mr. eric eng for showi
ng me einstein summations
    if qkv_b is not None:
        qkv += qkv_b
    qkv = qkv.reshape(B, T, 3, H, head_dim)
    qkv = jnp.transpose(qkv, (2, 0, 3, 1, 4))
    q, k, v = qkv[0], qkv[1], qkv[2]

    logits = jnp.einsum("bhqd,bhkd->bhqk", q, k) / jnp.sqrt(head_dim)
    attn = jax.nn.softmax(logits, axis=-1)
    out_per_head = jnp.einsum("bhqk,bhkd->bhqd", attn, v)
    out_per_head = jnp.transpose(out_per_head, (0, 2, 1, 3))
    out_pre = out_per_head.reshape(B, T, D)

    out = jnp.einsum("btd,df->btf", out_pre, out_k)
    if out_b is not None:
        out += out_b

    log.debug("manual_single_head_forward.end", out_shape=out.shape)
    return out, attn, out_pre


def test_head_collapse_single_vs_manual() -> Tuple[bool, str]:
    """When num_heads=1, module should equal manual single-head computation (outputs and grads)."""
    #not liskov sub exactly but i think this achieves the same test
    log.debug("test_head_collapse start")
    B, T, D = 2, 6, 16
    x = random.normal(KEY, (B, T, D))

    mha = MultiHeadAttention(d_model=D, num_heads=1, dropout_rate=0.0)
    params = _init_module(mha, x)

    out_mod, _ = mha.apply(params, x, deterministic=True, return_attn=True)
    out_manual, _, _ = _manual_single_head_forward_from_params(params, x)

    if not jnp.allclose(out_mod, out_manual, atol=1e-6):
```

```
        return False, "head-collapse forward mismatch"

    def loss_mod(inp): return jnp.sum(mha.apply(params, inp, deterministic=True)
[0])
    def loss_manual(inp): return jnp.sum(_manual_single_head_forward_from_params
(params, inp)[0])

    g_mod, g_manual = jax.grad(loss_mod)(x), jax.grad(loss_manual)(x)

    if not jnp.allclose(g_mod, g_manual, atol=1e-5):
        return False, "head-collapse input gradients mismatch"

    log.debug("test_head_collapse passed")
    return True, "passed"


def test_softmax_stability_scaling() -> Tuple[bool, str]:
    """Ensure softmax doesn't blow up with very large logits (scaling in place)."""
    log.debug("test_softmax_stability start")
    B, T, D, H = 1, 4, 32, 4
    x = random.normal(KEY, (B, T, D)) * 1e3
    mha = MultiHeadAttention(d_model=D, num_heads=H, dropout_rate=0.0)
    params = _init_module(mha, x)

    out, attn = mha.apply(params, x, deterministic=True, return_attn=True)
    if not (jnp.all(jnp.isfinite(out)) and jnp.all(jnp.isfinite(attn))):
        return False, "non-finite values in output or attention"

    log.debug("test_softmax_stability passed")
    return True, "passed"


def test_head_additivity_and_reconstruction() -> Tuple[bool, str]:
    """Reconstruct final output from per-head outputs + out_proj and compare to module output."""
    log.debug("test_head_additivity started")
    B, T, D, H = 2, 5, 32, 4
    x = random.normal(KEY, (B, T, D))
    mha = MultiHeadAttention(d_model=D, num_heads=H, dropout_rate=0.0)
    params = _init_module(mha, x)
    out_mod, _ = mha.apply(params, x, deterministic=True, return_attn=True)

    p = params['params']
    qkv_k, qkv_b = p['qkv_proj']['kernel'], p['qkv_proj'].get('bias', None)
    out_k, out_b = p['out_proj']['kernel'], p['out_proj'].get('bias', None)

    qkv = jnp.einsum("btd,df->btf", x, qkv_k)
    if qkv_b is not None:
        qkv += qkv_b
    head_dim = D // H
    qkv = qkv.reshape(B, T, 3, H, head_dim).transpose(2, 0, 3, 1, 4)
    q, k, v = qkv[0], qkv[1], qkv[2]

    logits = jnp.einsum("bhqd,bhkd->bhqk", q, k) / jnp.sqrt(head_dim)
    attn = jax.nn.softmax(logits, axis=-1)
    out_per_head = jnp.einsum("bhqk,bhkd->bhqd", attn, v)
    pre_concat = out_per_head.transpose(0, 2, 1, 3).reshape(B, T, D)

    recon = jnp.einsum("btd,df->btf", pre_concat, out_k)
    if out_b is not None:
        recon += out_b

    if not jnp.allclose(recon, out_mod, atol=1e-6):
        return False, "reconstructed output != module output"

    log.debug("test_head_additivity passed")
    return True, "passed"


def test_zero_out_head_and_contribution() -> Tuple[bool, str]:
```



```
    """Zero one head's qkv params; diff = contribution of that head."""
    log.debug("test_zero_out_head start")
    from jax import tree_util

    B, T, D, H, head_to_zero = 2, 5, 32, 4, 1
    x = random.normal(KEY, (B, T, D))
    mha = MultiHeadAttention(d_model=D, num_heads=H, dropout_rate=0.0)
    params = _init_module(mha, x)
    out_orig, _ = mha.apply(params, x, deterministic=True, return_attn=True)

    p = params['params']
    qkv_k, qkv_b = p['qkv_proj']['kernel'], p['qkv_proj'].get('bias', None)
    out_k, out_b = p['out_proj']['kernel'], p['out_proj'].get('bias', None)

    head_dim = D // H
    qkv = jnp.einsum("btd,df->btf", x, qkv_k)
    if qkv_b is not None:
        qkv += qkv_b
    qkv = qkv.reshape(B, T, 3, H, head_dim).transpose(2, 0, 3, 1, 4)
    q, k, v = qkv[0], qkv[1], qkv[2]

    logits = jnp.einsum("bhqd,bhkd->bhqk", q, k) / jnp.sqrt(head_dim)
    attn = jax.nn.softmax(logits, axis=-1)
    out_per_head = jnp.einsum("bhqk,bhkd->bhqd", attn, v)
    out_per_head_t = out_per_head.transpose(0, 2, 1, 3)

    # compute contribution of that head
    mask = jnp.arange(H) == head_to_zero
    pre_concat_head = (out_per_head_t * mask[None, None, :, None]).reshape(B, T,
 D)
    head_contrib = jnp.einsum("btd,df->btf", pre_concat_head, out_k)

    # zero head params
    params_zeroed = tree_util.tree_map(lambda x: x.copy() if hasattr(x, 'copy')
else x, params)
    def zero_head_qkv(kern, h):
        kern = kern.at[:, h*head_dim:(h+1)*head_dim].set(0.0)
        kern = kern.at[:, D + h*head_dim:D + (h+1)*head_dim].set(0.0)
        kern = kern.at[:, 2*D + h*head_dim:2*D + (h+1)*head_dim].set(0.0)
        return kern
    qkv_kz = zero_head_qkv(qkv_k, head_to_zero)
    params_zeroed['params']['qkv_proj']['kernel'] = qkv_kz
    if qkv_b is not None:
        b = qkv_b.at[head_to_zero*head_dim:(head_to_zero+1)*head_dim].set(0.0)
        b = b.at[D + head_to_zero*head_dim:D + (head_to_zero+1)*head_dim].set(0.
0)
        b = b.at[2*D + head_to_zero*head_dim:2*D + (head_to_zero+1)*head_dim].se
t(0.0)
        params_zeroed['params']['qkv_proj']['bias'] = b

    out_zeroed, _ = mha.apply(params_zeroed, x, deterministic=True, return_attn=
True)
    diff = out_orig - out_zeroed

    if not jnp.allclose(diff, head_contrib, atol=1e-5):
        return False, "zero-out-head diff != head contribution"

    log.debug("test_zero_out_head passed")
    return True, "passed"


def test_divisibility_check_error() -> Tuple[bool, str]:
    """d_model % num_heads != 0 should assert / error during init or call."""
    log.debug("test_divisibility_check start")
    B, T, D, H = 1, 4, 30, 4
    x = random.normal(KEY, (B, T, D))
    try:
        mha = MultiHeadAttention(d_model=D, num_heads=H, dropout_rate=0.0)
        _ = _init_module(mha, x)
```

```
    except (AssertionError, ValueError) as e:
        log.debug("test_divisibility_check passed", err_type=type(e).__name__)
        return True, f"passed ({type(e).__name__})"
    return False, "expected error for non-divisible d_model but succeeded"


def test_single_token_sequence_T1() -> Tuple[bool, str]:
    """Ensure T=1 works and shapes are preserved."""
    log.debug("test_single_token_T1 start")
    B, T, D, H = 2, 1, 16, 4
    x = random.normal(KEY, (B, T, D))
    mha = MultiHeadAttention(d_model=D, num_heads=H, dropout_rate=0.0)
    params = _init_module(mha, x)
    out, attn = mha.apply(params, x, deterministic=True, return_attn=True)
    assert out.shape == (B, T, D)
    assert attn.shape == (B, H, T, T)
    assert jnp.all(jnp.isfinite(out))
    log.debug("test_single_token_T1 passed")
    return True, "passed"


def test_backprop_through_attention_weights() -> Tuple[bool, str]:
    """Gradient through attention: d loss / d x should be non-zero."""
    log.debug("test_backprop_through_attention start")
    B, T, D, H = 1, 6, 24, 4
    x = random.normal(KEY, (B, T, D))
    mha = MultiHeadAttention(d_model=D, num_heads=H, dropout_rate=0.0)
    params = _init_module(mha, x)

    def loss_fn(inp):
        out, _ = mha.apply(params, inp, deterministic=True, return_attn=True)
        return jnp.sum(out[:, 0, :])

    g = jax.grad(loss_fn)(x)
    if not jnp.any(jnp.abs(g) > 0.0):
        return False, "gradient through attention is zero"
    log.debug("test_backprop_through_attention passed")
    return True, "passed"


def run_tests() -> List[Tuple[str, bool, str]]:
    tests = [
        ("mha_shape_and_attn_sums", test_mha_shape_and_attn_sums),
        ("mha_deterministic", test_mha_deterministic_behavior),
        ("mha_permutation_equivariance", test_permutation_equivariance_mha),
        ("transformer_grad_flow", test_transformerblock_grad_flow_basic),
        ("head_collapse_single_vs_manual", test_head_collapse_single_vs_manual),
        ("softmax_stability_scaling", test_softmax_stability_scaling),
        ("head_additivity_and_reconstruction", test_head_additivity_and_reconstruction),
        ("zero_out_head_and_contribution", test_zero_out_head_and_contribution),
        ("divisibility_check_error", test_divisibility_check_error),
        ("single_token_T1", test_single_token_sequence_T1),
        ("backprop_through_attention", test_backprop_through_attention_weights),
    ]
    results = []
    for name, fn in tests:
        try:
            ok, msg = fn()
        except Exception as e:
            ok, msg = False, f"exception: {e}"
        log.info("test_result", test=name, passed=ok, message=msg)
        results.append((name, ok, msg))
    return results
```

```python
import structlog
import jax
import numpy as np

from .logging import configure_logging
from .config import load_settings
from .model import TransformerBlock
from .tests import run_tests

log = structlog.get_logger()

def main() -> None:
    """CLI entry point."""
    configure_logging()
    settings = load_settings()
    log.info("Loaded settings", settings=settings.model.dict() if hasattr(settings,
"model") else str(settings))

    # PRNG
    seed = int(getattr(settings, "random_seed", 427))
    key = jax.random.PRNGKey(seed)
    data_key, model_key = jax.random.split(key)

    # instantiate model
    cfg = settings.model
    d_model = int(cfg.d_model)
    num_heads = int(cfg.num_heads)
    dropout = float(getattr(cfg, "dropout_rate", 0.0))
    ff_dim = max(4 * d_model, getattr(cfg, "ff_dim", 4 * d_model))  # fallback

    log.info("building TransformerBlock", d_model=d_model, num_heads=num_heads, ff_dim
=ff_dim, dropout=dropout)
    block = TransformerBlock(d_model=d_model, num_heads=num_heads, ff_dim=ff_dim
, dropout_rate=dropout)

    #tets time
    log.info("Running tests. im so nervous")
    results = run_tests()

    passed = all(r[1] for r in results)
    log.info("Test summary", passed=passed, results=results)

    if not passed:
        # print helpful failure summary
        failures = [(name, msg) for name, ok, msg in results if not ok]
        for name, msg in failures:
            log.error("Test failed bruh", test=name, reason=msg)
        raise SystemExit(1)
    else:
        log.info("All tests passed yippeeeeeeeeeee")
```