

Nov 03, 2025 0:07

pyproject.toml

Page 1/1

```
# pyproject.toml.jinja

[project]
name = "hw06"
version = "0.1.0"
description = "Multi-head attention"
readme = "README.md"
authors = [
    { name = "wongee(freddy) hong", email = "wongee.hong@cooper.edu" }
]
requires-python = ">=3.11"
dependencies = [
    "structlog",
    "numpy",
    "tensorflow",
    "tensorflow_datasets",
    "pydantic-settings",
    "matplotlib",
    "tqdm",
    "jax",
    "jaxlib",
    "flax",
    "optax",
    "scikit-learn",
    "transformers",
    "datasets",
]
[project.scripts]
hw06 = "hw06:main"

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

Nov 05, 2025 17:50

logging.py

Page 1/2

```

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():
        if hasattr(value, "numpy"):
            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

import logging
import sys
from pathlib import Path
import structlog

# keep your existing helpers if you have them:
# - FormattedFloat
# - custom_serializer_processor

def configure_logging(
    log_dir: Path = Path("hw06/artifacts"),
    log_name_json: str = "log.json",
    log_name_txt: str = "log.txt",
):
    log_dir.mkdir(parents=True, exist_ok=True)
    json_path = log_dir / log_name_json
    txt_path = log_dir / log_name_txt

    shared_processors = [
        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.stdlib.ProcessorFormatter.wrap_for_formatter,
    ]

    structlog.configure(
        processors=shared_processors,
        logger_factory=structlog.stdlib.LoggerFactory(),
        wrapper_class=structlog.stdlib.BoundLogger,
        cache_logger_on_first_use=True,
    )

    console_renderer = structlog.dev.ConsoleRenderer(
        colors=sys.stdout.isatty(),

```

Nov 05, 2025 17:50

logging.py

Page 2/2

```

        exception_formatter=structlog.dev.RichTracebackFormatter(),
    )
    json_renderer = structlog.processors.JSONRenderer()
    text_renderer = structlog.dev.ConsoleRenderer(colors=False)

    console_handler = logging.StreamHandler(sys.stdout)
    console_handler.setFormatter(
        structlog.stdlib.ProcessorFormatter(
            processors=[
                structlog.stdlib.ProcessorFormatter.remove_processors_meta,
                console_renderer,
            ]
        )
    )

    json_file_handler = logging.FileHandler(json_path, mode="w", encoding="utf-8")
    json_file_handler.setFormatter(
        structlog.stdlib.ProcessorFormatter(
            processors=[
                structlog.stdlib.ProcessorFormatter.remove_processors_meta,
                json_renderer,
            ]
        )
    )

    text_file_handler = logging.FileHandler(txt_path, mode="w", encoding="utf-8")
    text_file_handler.setFormatter(
        structlog.stdlib.ProcessorFormatter(
            processors=[
                structlog.stdlib.ProcessorFormatter.remove_processors_meta,
                text_renderer,
            ]
        )
    )

    root = logging.getLogger()
    root.handlers.clear()
    root.addHandler(console_handler)
    root.addHandler(json_file_handler)
    root.addHandler(text_file_handler)

    log_level = logging.getLevelName((sys.argv and "INFO") or "INFO")
    root.setLevel(log_level)

    logging.getLogger("matplotlib").setLevel(logging.WARNING)
    logging.getLogger("PIL").setLevel(logging.WARNING)

    return {"json": json_path, "txt": txt_path}

```

Nov 05, 2025 17:54

init_.py

Page 1/1

```
# __init__.py
import structlog
from flax import nnx
import jax
import jax.numpy as jnp

from .logging import configure_logging
from .test import grad_future_leak_self_attn, check_perm_equivariance_mha
from .model import EncoderBlock, MultiHeadAttention

def main() -> None:
    """CLI entry point."""
    configure_logging()
    log = structlog.get_logger()

    # Creating data and MultiHeadAttention (MHA)
    B, T, D, H = 2, 6, 64, 8
    key = jax.random.key(0)
    key_model, key_x = jax.random.split(key)

    x = jax.random.normal(key_x, (B, T, D))
    perm = jnp.array([2, 0, 4, 1, 5, 3])
    mha = MultiHeadAttention(model_dim=D, head_size=H, rngs=nnx.Rngs(params=key_model))

    # Testing Permutation equivariance
    log.info("perm_check.mha.start", B=B, T=T, D=D, H=H)
    passed_mha = check_perm_equivariance_mha(mha, x, perm)
    log.info("perm_check.mha", passed=bool(passed_mha))

    # Testing gradient for masked MHA
    key = jax.random.key(1)

    # testing gradient at t_mid
    t_mid = T // 2

    future_masked = grad_future_leak_self_attn(mha, T, D, t_mid, use_mask=True)
    future_unmasked = grad_future_leak_self_attn(mha, T, D, t_mid, use_mask=False)

    max_future_masked = float(jnp.max(jnp.abs(future_masked)))      # ~ 0 with mask
    max_future_unmasked = float(jnp.max(jnp.abs(future_unmasked)))  # > 0 without mask

    log.info("future grad matrices for masked MHA", masked=str(future_masked.tolist()))
    log.info("future grad matrices for unmasked MHA", unmasked=str(future_unmasked.tolist()))

    # Log the summary scalars
    log.info("max value of future grad matrices", max_masked=max_future_masked, max_unmasked=max_future_unmasked)

    # Masked case: expect = 0
    log.info("future_grad_check", case="masked", passed=(max_future_masked == 0), max_future=max_future_masked)

    # Unmasked case: expect > 0
    log.info("future_grad_check", case="unmasked", passed=(max_future_unmasked > 0), max_future=max_future_unmasked)
```

Nov 05, 2025 17:30

test.py

Page 1/1

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

def check_perm_equivariance_mha(mha, x, perm):
    # No mask, no PE
    y = mha(x)                      # (B, T, D)
    y_perm_in = mha(x[:, perm, :])    # permute input
    y_perm_out = y[:, perm, :]        # permute outputs
    return jnp.allclose(y_perm_in, y_perm_out, atol=1e-5)

def causal_mask(T: int) -> jnp.ndarray:
    return jnp.tril(jnp.ones((T, T), dtype=bool))[None, None, :, :]

def grad_future_leak_self_attn(mha, Ty: int, D: int, t: int, use_mask: bool):
    key = jax.random.key(0)
    y_in = jax.random.normal(key, (1, Ty, D))
    mask = causal_mask(Ty) if use_mask else None

    def f(y_):
        y_out = mha(y_, mask=mask)      # self-attn: Q, K, V from y_
        return y_out[:, t, :].sum()

    g = jax.grad(lambda y: f(y).sum())(y_in)  # (1, Ty, D)
    future = g[:, (t+1):, :]                  # grads wrt future positions
    return future
```

Nov 05, 2025 17:38

model.py

Page 1/3

```

from flax import nnx
import jax
import jax.numpy as jnp

class MultiHeadAttention(nn.Module):
    """MultiHeadAttention class for self-attention, cross-attention and masked/unmasked"""
    def __init__(self,
                 model_dim: int,
                 head_size: int,
                 rngs: nn.Rngs,
                 ):
        assert model_dim % head_size == 0, "model_dim should be divisible by head size"
        self.model_dim = model_dim
        self.head_size = head_size
        self.d_k = model_dim // head_size
        self.Wq = nnx.Linear(model_dim, model_dim, rngs=rngs)
        self.Wk = nnx.Linear(model_dim, model_dim, rngs=rngs)
        self.Wv = nnx.Linear(model_dim, model_dim, rngs=rngs)
        self.Wo = nnx.Linear(model_dim, model_dim, rngs=rngs)

    def attention(self, Q, K, V, mask: jnp.ndarray | None):
        attention = (Q @ jnp.swapaxes(K, -1, -2)) / jnp.sqrt(self.d_k)

        # handling case for masked attention
        if mask is not None:
            attention = jnp.where(mask, attention, attention - 1e9)
        attention = jax.nn.softmax(attention, axis=-1)
        output = attention @ V
        return output

    def split_head(self, x: jnp.ndarray):
        # splitting head for Multihead attention
        B, T, D = x.shape
        H, d_k = self.head_size, self.d_k
        x = x.reshape(B, T, H, d_k) # (B, T, H, d_k)
        x = jnp.swapaxes(x, 1, 2) # (B, H, T, d_k)
        return x

    def combine_head(self, x: jnp.ndarray):
        # combining heads after computing attentions
        B, H, T, d_k = x.shape
        x = jnp.swapaxes(x, 1, 2) # (B, T, H, d_k)
        return x.reshape(B, T, H * d_k)

    def __call__(self, x: jnp.ndarray, *, cross: jnp.ndarray | None = None, mask: jnp.ndarray | None = None):
        Q = self.Wq(x)

        # Handling case for cross attention
        y = x if cross is None else cross
        K = self.Wk(y)
        V = self.Wv(y)

        Q = self.split_head(Q)
        K = self.split_head(K)
        V = self.split_head(V)
        attention_value = self.attention(Q, K, V, mask)

        output = self.Wo(self.combine_head(attention_value))
        return output

class FeedForward(nn.Module):
    def __init__(self,
                 n_embd: int,
                 rngs: nn.Rngs
                 ):
        # From paper, input and output is d_model = 512 and d_ff = 4*d_model = 2048
        self,
        n_embd: int,
        rngs: nn.Rngs
    
```

Nov 05, 2025 17:38

model.py

Page 2/3

```

048     # Implemented the same way with paper
        self.linear1 = nnx.Linear(n_embd, 4 * n_embd, rngs=rngs)
        self.linear2 = nnx.Linear(4 * n_embd, n_embd, rngs=rngs)

    def __call__(self, x):
        x = self.linear1(x)
        x = jax.nn.relu(x)
        x = self.linear2(x)
        return x

class EncoderBlock(nn.Module):
    def __init__(self,
                 model_dim: int,
                 head_size: int,
                 *,
                 rngs: nn.Rngs
                 ):
        self.head_size = head_size
        self.model_dim = model_dim
        self.attention = MultiHeadAttention(model_dim, head_size, rngs)
        self.ff = FeedForward(model_dim, rngs)
        self.norm1 = nnx.LayerNorm(model_dim, rngs=rngs)
        self.norm2 = nnx.LayerNorm(model_dim, rngs=rngs)

    def __call__(self, x):
        # For encoder block, self attention and unmasked
        y = x + self.attention(x, cross=None, mask=None)
        x1 = self.norm1(y)
        z = x1 + self.ff(x1)
        x2 = self.norm2(z)
        return x2

class DecoderBlock(nn.Module):
    def __init__(self,
                 model_dim: int,
                 head_size: int,
                 *,
                 rngs: nn.Rngs
                 ):
        self.head_size = head_size
        self.model_dim = model_dim
        self.attention = MultiHeadAttention(model_dim, head_size, rngs)
        self.ff = FeedForward(model_dim, rngs)
        self.norm1 = nnx.LayerNorm(model_dim, rngs=rngs)
        self.norm2 = nnx.LayerNorm(model_dim, rngs=rngs)
        self.norm3 = nnx.LayerNorm(model_dim, rngs=rngs)

    def __call__(self, x, x_encoder, mask):
        # For decoder block, first attention is masked self-attention
        y1 = x + self.attention(x, cross=None, mask=mask)
        x1 = self.norm1(y1)
        # second attention is unmasked cross attention
        y2 = x1 + self.attention(x1, cross=x_encoder, mask=None)
        x2 = self.norm2(y2)
        y3 = x2 + self.ff(x2)
        output = self.norm3(y3)
        return output

class Transformer(nn.Module):
    def __init__(self,
                 model_dim: int,
                 head_size: int,
                 vocab_size: int,
                 num_layers: int,
                 *,
                 rngs: nn.Rngs,
                 )
        self,
        model_dim: int,
        head_size: int,
        vocab_size: int,
        num_layers: int,
        rngs: nn.Rngs,
    
```

Nov 05, 2025 17:38

model.py

Page 3/3

```

):
    self.model_dim = model_dim
    self.head_size = head_size
    self.vocab_size = vocab_size
    self.num_layers = num_layers

    self.x_embed = nnx.Embed(num_embeddings=vocab_size, features=model_dim,
rngs=rngs)
    self.y_embed = nnx.Embed(num_embeddings=vocab_size, features=model_dim,
rngs=rngs)

    self.EncoderBlocks = [EncoderBlock(model_dim, head_size, rngs) for _ in
range(num_layers)]
    self.DecoderBlocks = [DecoderBlock(model_dim, head_size, rngs) for _ in
range(num_layers)]
    self.Linear = nnx.Linear(model_dim, vocab_size, rngs=rngs)

    def PositionalEncoding(self, seq_length:int , model_dim: int):
        positions = jnp.arange(seq_length)[:, None]
        dims = jnp.arange(model_dim)[None,:] # if model_dim = 8, [0,1,2 ... ,8]

        # since embedding is sin for even and cos for odd, we need i = dims//2 to
o create
        # [0,0,1,1,2,2...]
        pos_enc = positions / (10000 ** ((2 * dims//2) / model_dim))
        pos_enc = jnp.where(dims % 2 == 0, jnp.sin(pos_enc), jnp.cos(pos_enc))
        return pos_enc

    def __call__ (self, x: jnp.ndarray , y: jnp.ndarray):
        x = self.x_embed(x) * jnp.sqrt(self.model_dim)
        Bx, Tx, Dx = x.shape

        x_pos_enc = self.PositionalEncoding(Tx, self.model_dim)
        x = x + x_pos_enc[None, :, :]

        for block in self.EncoderBlocks:
            x = self.block(x)
        encoder_out = x

        y = self.y_embed(y) * jnp.sqrt(self.model_dim)
        By, Ty, Dy = y.shape

        y_pos_enc = self.PositionalEncoding(Ty, self.model_dim)
        y = y + y_pos_enc[None, :, :]
        self_mask = jnp.tril(jnp.ones((Ty, Ty), dtype=bool))[None, None, :, :]

        for block in self.DecoderBlocks:
            y = self.block(y, encoder_out, self_mask)
        decoder_out = y
        logits = self.Linear(decoder_out)
        return logits

```