from typing import Any

import polars as pl import torch from pytorch_lightning import LightningModule from torch import Tensor, nn

from integrator.model.distributions import BaseDistribution from integrator.model.encoders import (IntensityEncoder, MLPMetadataEncoder, ShoeboxEncoder,) from integrator.model.integrators import BaseIntegrator from integrator.model.loss import BaseLoss

def get_outputs(vars: dict, data_dim: str,) -> dict: # default network outputs out = { “rates”: vars[“rate”], “counts”: vars[“counts”], “masks”: vars[“masks”], “qbg”: vars[“qbg”], “qp”: vars[“qp”], “qp_mean”: vars[“qp”].mean, “qi”: vars[“qi”], “intensity_mean”: vars[“qi”].mean, “intensity_var”: vars[“qi”].variance, “profile”: vars[“qp”].mean, “zp”: vars[“zp”], }

```
if vars["reference"] is not None:
    reference = vars["reference"]

    if data_dim == "3d":
        ref_3d = {
            "dials_I_sum_value": reference[:, 6],
            "dials_I_sum_var": reference[:, 7],
            "dials_I_prf_value": reference[:, 8],
            "dials_I_prf_var": reference[:, 9],
            "refl_ids": reference[:, -1].int().tolist(),
            "x_c": reference[:, 0],
            "y_c": reference[:, 1],
            "z_c": reference[:, 2],
            "x_c_mm": reference[:, 3],
            "y_c_mm": reference[:, 4],
            "z_c_mm": reference[:, 5],
            "dials_bg_mean": reference[:, 10],
            "dials_bg_sum_value": reference[:, 11],
            "dials_bg_sum_var": reference[:, 12],
            "d": reference[:, 13],
        }
        for k, v in ref_3d.items():
            out[k] = v

    elif data_dim == "2d":
        ref_2d = {
            "dials_I_sum_value": reference[:, 3],
            "dials_I_sum_var": reference[:, 4],
            "dials_I_prf_value": reference[:, 3],
            "dials_I_prf_var": reference[:, 4],
            "refl_ids": reference[:, -1].tolist(),
            "x_c": reference[:, 9],
            "y_c": reference[:, 10],
            "z_c": reference[:, 11],
            "dials_bg_mean": reference[:, 0],
            "dials_bg_sum_value": reference[:, 0],
```

```
            "dials_bg_sum_var": reference[:, 1],
        }

        for k, v in ref_2d.items():
            out[k] = v

elif vars["reference"] is None:
    return out

else:
    print("Invalid output data")

return out
```

-

class IntegratorA(BaseIntegrator): “““IntegratorA uses no additional experimental metadata.”“”

```
encoder1: ShoeboxEncoder | IntensityEncoder
"""Encoder to get profile distribution"""
encoder2: ShoeboxEncoder | IntensityEncoder
"""Encoder to get intensity & background distributions"""

def __init__(
    self,
    encoder1: ShoeboxEncoder | IntensityEncoder,
    encoder2: ShoeboxEncoder | IntensityEncoder,
    qbg: BaseDistribution,
    qp: BaseDistribution,
    qi: BaseDistribution,
    loss: BaseLoss,
    data_dim: str = "3d",
    d: int = 3,
    h: int = 21,
    w: int = 21,
    *,
    lr: float = 1e-3,
    weight_decay: float = 0.0,
    mc_samples: int = 100,
    max_iterations: int = 4,
    renyi_scale: float = 0.00,
    encoder_out: int,
):
    super().__init__(
        qbg=qbg,
        qi=qi,
        qp=qp,
        loss=loss,
```

```
            data_dim=data_dim,
            d=d,
            h=h,
            w=w,
            lr=lr,
            weight_decay=weight_decay,
            mc_samples=mc_samples,
            max_iterations=max_iterations,
            renyi_scale=renyi_scale,
            encoder_out=encoder_out,
        )

        self.data_dim: str = data_dim
        self.encoder1 = encoder1
        self.encoder2 = encoder2

    # def forward(self, counts, shoebox, metadata, masks, reference):
    def forward(
        self,
        counts: Tensor,
        shoebox: Tensor,
        masks: Tensor,
        reference: Tensor | None = None,
    ) -> dict[str, Any]:
        """
        Forward model architecture:
        ```mermaid
        flowchart LR

            counts --> encoder1
            counts --> encoder2

            encoder1 --> qp
            encoder2 --> qi
            encoder2 --> qbg

        ```
        """

        # Unpack batch
        counts = torch.clamp(counts, min=0) * masks

        profile_rep = self.encoder1(
            shoebox.reshape(shoebox.shape[0], 1, *(self.shoebox_shape))
        )
        intensity_rep = self.encoder2(
            shoebox.reshape(shoebox.shape[0], 1, *(self.shoebox_shape))
        )

        qbg = self.qbg(intensity_rep)
        qp = self.qp(profile_rep)
```

```
    qi = self.qi(intensity_rep)

    zbg = qbg.rsample([self.mc_samples]).unsqueeze(-1).permute(1, 0, 2)
    zp = qp.rsample([self.mc_samples]).permute(1, 0, 2)
    zI = qi.rsample([self.mc_samples]).unsqueeze(-1).permute(1, 0, 2)

    rate = zI * zp + zbg

    # calculate profile renyi entropy
    avg_reynyi_entropy = (-(zp.pow(2).sum(-1).log())).mean(-1)
    out = get_outputs(locals(), self.data_dim)
    return out
```

-

class tmepIntegratorB(BaseIntegrator): “““IntegratorB uses enables the use of metadata”“”

```
encoder1: ShoeboxEncoder | IntensityEncoder
"""Encoder to get profile distribution"""
encoder2: ShoeboxEncoder | IntensityEncoder
"""Encoder to get intensity & background distributions"""
encoder3: MLPMetadataEncoder | None
"""Encoder for experimental metadata"""

def __init__(
    self,
    encoder1: ShoeboxEncoder | IntensityEncoder,
    encoder2: ShoeboxEncoder | IntensityEncoder,
    encoder3: MLPMetadataEncoder | None,
    qbg: BaseDistribution,
    qp: BaseDistribution,
    qi: BaseDistribution,
    loss: BaseLoss,
    data_dim: str,
    d: int = 3,
    h: int = 21,
    w: int = 21,
    *,
    lr: float = 1e-3,
    weight_decay: float = 0.0,
    mc_samples: int = 100,
    max_iterations: int = 4,
    renyi_scale: float = 0.00,
    encoder_out: int,
):
    super().__init__(
        qbg=qbg,
        qi=qi,
```

```
            qp=qp,
            loss=loss,
            data_dim=data_dim,
            d=d,
            h=h,
            w=w,
            lr=lr,
            weight_decay=weight_decay,
            mc_samples=mc_samples,
            max_iterations=max_iterations,
            renyi_scale=renyi_scale,
            encoder_out=encoder_out,
        )

        self.encoder1 = encoder1
        self.encoder2 = encoder2
        self.encoder3 = encoder3

        self.linear = nn.Linear(self.encoder_out * 2, self.encoder_out)

    def forward(
        self,
        counts: Tensor,
        shoebox: Tensor,
        masks: Tensor,
        reference: Tensor | None = None,
    ) -> dict[str, Any]:
        """
        Forward model architecture:
        ```mermaid
        flowchart LR

            counts --> encoder1
            counts --> encoder2
            metadata --> encoder3

            encoder1 --> qp
            encoder2 --> torch.concat
            encoder3 --> torch.concat
            torch.concat --> qi
            torch.concat --> qbg

        ```

        Args:
            counts: Raw photon count Tensor
            shoebox: Standardized photon count Tensor
            masks: Dead-pixel mask
            reference: Optional metadata Tensor

        Returns:
```

```
        """
        # Unpack batch
        counts = torch.clamp(counts, min=0) * masks

        x_profile = self.encoder1(
            shoebox.reshape(shoebox.shape[0], 1, *(self.shoebox_shape))
        )
        x_intensity = self.encoder2(
            shoebox.reshape(shoebox.shape[0], 1, *(self.shoebox_shape))
        )

        if self.encoder3 is not None and reference is None:
            assert ValueError(
                "A metadata encoder (encoder 3) was provided, but no reference data was found. Please provide a `reference.pt` dataset"
            )

        metadata = torch.nn.Identity()

        if self.encoder3 is not None and reference is not None:
            if self.data_dim == "2d" and reference is not None:
                # TODO: Change the datatypes in the DataLoader
                metadata = (reference[:, [6, 7, 8, 9, 10, 11]]).float()
                print("metadata type:", type(metadata))

            elif self.data_dim == "3d" and reference is not None:
                metadata = reference[:, [6, 7, 8, 9, 10, 11]]

            print(metadata)

            x_metadata = self.encoder3(metadata)

            x_intensity = torch.concat([x_intensity, x_metadata], dim=-1)
            x_intensity = self.linear(x_intensity)

        qbg = self.qbg(x_intensity)
        qi = self.qi(x_intensity)
        qp = self.qp(x_profile)

        zbg = qbg.rsample([self.mc_samples]).unsqueeze(-1).permute(1, 0, 2)
        zp = qp.rsample([self.mc_samples]).permute(1, 0, 2)
        zI = qi.rsample([self.mc_samples]).unsqueeze(-1).permute(1, 0, 2)

        rate = zI * zp + zbg

        # calculate profile renyi entropy
        avg_reynyi_entropy = (-(zp.pow(2).sum(-1).log())).mean(-1)
        out = get_outputs(locals(), self.data_dim)
        return out
```

-

class IntegratorB(LightningModule): encoder1: ShoeboxEncoder | IntensityEncoder “““Encoder to get profile distribution”“” encoder2: ShoeboxEncoder | IntensityEncoder “““Encoder to get intensity & background distributions”“” encoder3: MLPMetadataEncoder | None “““Encoder for experimental metadata”“” qbg: BaseDistribution “““Surrogate posterior shoebox Background”“” qp: BaseDistribution “““Surrogate posterior of spot Profile”“” qi: BaseDistribution “““Surrogate posterior of the spot Intensity”“” data_dim: str “““Dimensionality of diffraction data (2d or 3d)”“” loss: BaseLoss “““Loss function to optimize.”“” d: int “““Depth of input shoebox.”“” h: int “““Height on input shoebox.”“” w: int “““Width of input shoebox.”“” lr: float weight_decay: float “““Weight decay value for Adam optimizer.”“” mc_samples: int “““Number of samples to use for Monte Carlo approximations”“” max_iterations: int renyi_scale: float encoder_out: int

```
def __init__(
    self,
    encoder1: ShoeboxEncoder | IntensityEncoder,
    encoder2: ShoeboxEncoder | IntensityEncoder,
    encoder3: MLPMetadataEncoder | None,
    qbg: BaseDistribution,
    qp: BaseDistribution,
    qi: BaseDistribution,
    loss: BaseLoss,
    data_dim: str = "3d",  # defaults to rotation data
    d: int = 3,
    h: int = 21,
    w: int = 21,
    *,
    lr: float = 1e-3,
    weight_decay: float = 0.0,
    mc_samples: int = 100,
    max_iterations: int = 4,
    renyi_scale: float = 0.00,
    encoder_out: int,
    predict_keys: tuple[str, ...] = (
        "intensity_mean",
        "intensity_var",
        "refl_ids",
        "dials_I_sum_value",
        "dials_I_sum_var",
        "dials_I_prf_value",
        "dials_I_prf_var",
        "dials_bg_mean",
        "qbg_mean",
        "qbg_scale",
        "x_c",
        "y_c",
        "z_c",
```

```
        ),
    ):
        super().__init__()
        self.qbg = qbg
        self.qp = qp
        self.qi = qi
        self.d = d
        self.h = h
        self.w = w
        self.loss = loss
        self.renyi_scale = renyi_scale
        self.data_dim = data_dim
        self.encoder_out = encoder_out

        # encoders
        self.encoder1 = encoder1
        self.encoder2 = encoder2
        self.encoder3 = encoder3

        if self.encoder3 is not None:
            self.linear = nn.Linear(self.encoder_out * 2, self.encoder_out)

        # lists to track avg traning metrics
        self.train_loss = []
        self.train_kl = []
        self.train_nll = []

        # lists to track avg validation metrics
        self.val_loss = []
        self.val_kl = []
        self.val_nll = []
        self.lr = lr
        self.automatic_optimization = True
        self.weight_decay = weight_decay
        self.mc_samples = mc_samples
        self.max_iterations = max_iterations
        self.predict_keys = predict_keys

        #
        if self.data_dim == "3d":
            self.shoebox_shape = (self.d, self.h, self.w)
        elif self.data_dim == "2d":
            self.shoebox_shape = (self.h, self.w)

        # dataframes to keep track of val/train epoch metrics
        self.schema = [
            ("epoch", int),
            ("avg_loss", float),
            ("avg_kl", float),
            ("avg_nll", float),
        ]
```

```python
        self.train_df = pl.DataFrame(schema=self.schema)
        self.val_df = pl.DataFrame(schema=self.schema)

    def calculate_intensities(self, counts, qbg, qp, masks):
        with torch.no_grad():
            counts = counts * masks  # [B,P]
            zbg = qbg.rsample([self.mc_samples]).unsqueeze(-1).permute(1, 0, 2)
            zp = qp.mean.unsqueeze(1)

            vi = zbg + 1e-6
            intensity = torch.tensor([0.0])

            # kabsch sum
            for _ in range(self.max_iterations):
                num = (
                    (counts.unsqueeze(1) - zbg) * zp * masks.unsqueeze(1) / vi
                )
                denom = zp.pow(2) / vi
                intensity = num.sum(-1) / denom.sum(
                    -1
                )  # [batch_size, mc_samples]
                vi = (intensity.unsqueeze(-1) * zp) + zbg
                vi = vi.mean(-1, keepdim=True)
            kabsch_sum_mean = intensity.mean(-1)
            kabsch_sum_var = intensity.var(-1)

            # profile masking
            zp = zp * masks.unsqueeze(1)  # profiles
            thresholds = torch.quantile(
                zp,
                0.99,
                dim=-1,
                keepdim=True,
            )  # threshold values
            profile_mask = zp > thresholds

            masked_counts = counts.unsqueeze(1) * profile_mask

            profile_masking_I = (masked_counts - zbg * profile_mask).sum(-1)

            profile_masking_mean = profile_masking_I.mean(-1)

            profile_masking_var = profile_masking_I.var(-1)

            intensities = {
                "profile_masking_mean": profile_masking_mean,
                "profile_masking_var": profile_masking_var,
                "kabsch_sum_mean": kabsch_sum_mean,
```

```
            "kabsch_sum_var": kabsch_sum_var,
        }

        return intensities

def forward(
    self,
    counts: Tensor,
    shoebox: Tensor,
    masks: Tensor,
    reference: Tensor | None = None,
) -> dict[str, Any]:
    """
    Forward model architecture:
    ```mermaid
    flowchart LR

        counts --> encoder1
        counts --> encoder2
        metadata --> encoder3

        encoder1 --> qp
        encoder2 --> torch.concat
        encoder3 --> torch.concat
        torch.concat --> qi
        torch.concat --> qbg

    ```

    Args:
        counts: Raw photon count Tensor
        shoebox: Standardized photon count Tensor
        masks: Dead-pixel mask
        reference: Optional metadata Tensor

    Returns:

    """
    # Unpack batch
    counts = torch.clamp(counts, min=0) * masks

    x_profile = self.encoder1(
        shoebox.reshape(shoebox.shape[0], 1, *(self.shoebox_shape))
)
    x_intensity = self.encoder2(
        shoebox.reshape(shoebox.shape[0], 1, *(self.shoebox_shape))
)

    if self.encoder3 is not None and reference is None:
        assert ValueError(
            "A metadata encoder (encoder 3) was provided, but no
```

```
                    reference data was found. Please provide a `reference.pt
                    ` dataset"
            )

        metadata = torch.nn.Identity()

        if self.encoder3 is not None and reference is not None:
            if self.data_dim == "2d" and reference is not None:
                # TODO: Change the datatypes in the DataLoader
                metadata = (reference[:, [6, 7, 8, 9, 10, 11]]).float()
                print("metadata type:", type(metadata))

            elif self.data_dim == "3d" and reference is not None:
                metadata = reference[:, [0, 1, 2, 3, 4, 5, 13]]

            print(metadata)

            x_metadata = self.encoder3(metadata)

            x_intensity = torch.concat([x_intensity, x_metadata], dim=-1)
            x_intensity = self.linear(x_intensity)

        qbg = self.qbg(x_intensity)
        qi = self.qi(x_intensity)
        qp = self.qp(x_profile)

        zbg = qbg.rsample([self.mc_samples]).unsqueeze(-1).permute(1, 0, 2)
        zp = qp.rsample([self.mc_samples]).permute(1, 0, 2)
        zI = qi.rsample([self.mc_samples]).unsqueeze(-1).permute(1, 0, 2)

        rate = zI * zp + zbg

        # calculate profile renyi entropy
        avg_reynyi_entropy = (-(zp.pow(2).sum(-1).log())).mean(-1)
        out = get_outputs(locals(), self.data_dim)
        return out

    def on_train_epoch_end(self):
        # calculate epoch averages
        avg_train_loss = sum(self.train_loss) / len(self.train_loss)
        avg_kl = sum(self.train_kl) / len(self.train_kl)
        avg_nll = sum(self.train_nll) / len(self.train_nll)

        # log averages to weights & biases
        self.log("train_loss", avg_train_loss)
        self.log("avg_kl", avg_kl)
        self.log("avg_nll", avg_nll)

        # create epoch dataframe
        epoch_df = pl.DataFrame(
            {
```

```
                "epoch": self.current_epoch,
                "avg_loss": avg_train_loss,
                "avg_kl": avg_kl,
                "avg_nll": avg_nll,
            }
        )

        # udpate training dataframe
        self.train_df = pl.concat([self.train_df, epoch_df])

        # clear all lists
        self.train_loss = []
        self.train_kl = []
        self.train_nll = []

    def on_validation_epoch_end(self):
        """Validation step processing"""
        avg_val_loss = sum(self.val_loss) / len(self.val_loss)
        avg_kl = sum(self.val_kl) / len(self.val_kl)
        avg_nll = sum(self.val_nll) / len(self.val_nll)

        self.log("validation_loss", avg_val_loss)
        self.log("validation_avg_kl", avg_kl)
        self.log("validation_avg_nll", avg_nll)

        epoch_df = pl.DataFrame(
            {
                "epoch": self.current_epoch,
                "avg_loss": avg_val_loss,
                "avg_kl": avg_kl,
                "avg_nll": avg_nll,
            }
        )
        self.val_df = pl.concat([self.val_df, epoch_df])

        self.val_loss = []
        self.avg_kl = []
        self.val_nll = []

    def training_step(self, batch, _batch_idx):
        counts, shoebox, masks, reference = batch
        outputs = self(counts, shoebox, masks, reference)

        # Calculate loss
        loss_dict = self.loss(
            rate=outputs["rates"],
            counts=outputs["counts"],
            q_p=outputs["qp"],
            q_i=outputs["qi"],
            q_bg=outputs["qbg"],
            masks=outputs["masks"],
```

```
    )

    renyi_loss = (
        (
            -torch.log(
                outputs["qp"]
                .rsample([self.mc_samples])
                .permute(1, 0, 2)
                .pow(2)
                .sum(-1)
            )
        )
        .mean(1)
        .sum()
    ) * self.renyi_scale
    self.log("renyi_loss", renyi_loss)

    for k, v in loss_dict.items():
        key = f"train_{k}"
        value = v.mean()
        self.log(key, value)

    self.log("Mean(qi.mean)", outputs["qi"].mean.mean())
    self.log("Min(qi.mean)", outputs["qi"].mean.min())
    self.log("Max(qi.mean)", outputs["qi"].mean.max())
    self.log("Mean(qbg.mean)", outputs["qbg"].mean.mean())
    self.log("Min(qbg.mean)", outputs["qbg"].mean.min())
    self.log("Max(qbg.mean)", outputs["qbg"].mean.max())
    self.log("Mean(qbg.variance)", outputs["qbg"].variance.mean())

    self.train_loss.append(loss_dict["total_loss"].mean())
    self.train_kl.append(loss_dict["kl_mean"].mean())
    self.train_nll.append(loss_dict["neg_ll_mean"].mean())

    return loss_dict["total_loss"].mean() + renyi_loss.sum()

def configure_optimizers(self):
    return torch.optim.Adam(
        self.parameters(), lr=self.lr, weight_decay=self.weight_decay
    )

def validation_step(self, batch, _batch_idx):
    """

    Args:
        batch ():
        _batch_idx ():

    Returns:

    """
```

```
    # Unpack batch
    counts, shoebox, masks, reference = batch
    outputs = self(counts, shoebox, masks, reference)

    loss_dict = self.loss(
        rate=outputs["rates"],
        counts=outputs["counts"],
        q_p=outputs["qp"],
        q_i=outputs["qi"],
        q_bg=outputs["qbg"],
        masks=outputs["masks"],
    )

    for k, v in loss_dict.items():
        key = f"val_{k}"
        value = v.mean()
        self.log(key, value)

    self.val_loss.append(loss_dict["total_loss"].mean())
    self.val_kl.append(loss_dict["kl_mean"].mean())
    self.val_nll.append(loss_dict["neg_ll_mean"].mean())

    return outputs

def predict_step(self, batch, _batch_idx):
    """Prediction step

    Args:
        batch: Inpute Tensor data

    Returns:


    """
    counts, shoebox, masks, reference = batch
    outputs = self(counts, shoebox, masks, reference)

    return {k: v for k, v in outputs.items() if k in self.predict_keys}
```

-

if **name** == “**main**”: import torch

```
from integrator.utils import (
    create_data_loader,
    create_integrator,
    load_config,
)
```

```
from utils import CONFIGS

torch.set_default_dtype(torch.float32)

# Model A 3D
config = load_config(CONFIGS["integratorA_3D"])
integrator = create_integrator(config.dict())
data_loader = create_data_loader(config.dict())
counts, shoebox, masks, reference = next(
    iter(data_loader.train_dataloader())
)
out_2d = integrator(counts, shoebox, masks, reference)

# Model A 2D
config = load_config(CONFIGS["integratorA_2D"])

integrator = create_integrator(config.dict())
data_loader = create_data_loader(config.dict())
counts, shoebox, masks, reference = next(
    iter(data_loader.train_dataloader())
)
out_2d = integrator(counts, shoebox, masks, reference)

# Model B 3D
CONFIGS["integratorB_3D"].as_posix()
config = load_config(CONFIGS["integratorB_3D"])
config.model_dump()["integrator"]
integrator = create_integrator(config.dict())
data_loader = create_data_loader(config.dict())
counts, shoebox, masks, reference = next(
    iter(data_loader.train_dataloader())
)
out_3d = integrator(counts, shoebox, masks, reference)

# Model B 2D
CONFIGS["integratorB_2D"].as_posix()

config = load_config(CONFIGS["integratorB_2D"])

config.model_dump()["integrator"]

integrator = create_integrator(config.dict())

data_loader = create_data_loader(config.dict())

counts, shoebox, masks, reference = next(
    iter(data_loader.train_dataloader())
)
out_3d = integrator(counts, shoebox, masks, reference)

# 2D - no metadata
```

```
config = load_config(CONFIGS["integrator_2D_2encoders"])
```