
```
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"], }
```

```
1  if vars["reference"] is not None:
2      reference = vars["reference"]
3
4      if data_dim == "3d":
5          ref_3d = {
6              "dials_I_sum_value": reference[:, 6],
7              "dials_I_sum_var": reference[:, 7],
8              "dials_I_prf_value": reference[:, 8],
9              "dials_I_prf_var": reference[:, 9],
10             "refl_ids": reference[:, -1].int().tolist(),
11             "x_c": reference[:, 0],
12             "y_c": reference[:, 1],
13             "z_c": reference[:, 2],
14             "x_c_mm": reference[:, 3],
15             "y_c_mm": reference[:, 4],
16             "z_c_mm": reference[:, 5],
17             "dials_bg_mean": reference[:, 10],
18             "dials_bg_sum_value": reference[:, 11],
19             "dials_bg_sum_var": reference[:, 12],
20             "d": reference[:, 13],
21         }
22         for k, v in ref_3d.items():
23             out[k] = v
24
25     elif data_dim == "2d":
26         ref_2d = {
27             "dials_I_sum_value": reference[:, 3],
28             "dials_I_sum_var": reference[:, 4],
29             "dials_I_prf_value": reference[:, 3],
30             "dials_I_prf_var": reference[:, 4],
31             "refl_ids": reference[:, -1].tolist(),
32             "x_c": reference[:, 9],
33             "y_c": reference[:, 10],
34             "z_c": reference[:, 11],
35             "dials_bg_mean": reference[:, 0],
36             "dials_bg_sum_value": reference[:, 0],
```

```

37         "dials_bg_sum_var": reference[:, 1],
38     }
39
40     for k, v in ref_2d.items():
41         out[k] = v
42
43 elif vars["reference"] is None:
44     return out
45
46 else:
47     print("Invalid output data")
48
49 return out

```

-

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

```

1  encoder1: ShoeboxEncoder | IntensityEncoder
2  """Encoder to get profile distribution"""
3  encoder2: ShoeboxEncoder | IntensityEncoder
4  """Encoder to get intensity & background distributions"""
5
6  def __init__(
7      self,
8      encoder1: ShoeboxEncoder | IntensityEncoder,
9      encoder2: ShoeboxEncoder | IntensityEncoder,
10     qbg: BaseDistribution,
11     qp: BaseDistribution,
12     qi: BaseDistribution,
13     loss: BaseLoss,
14     data_dim: str = "3d",
15     d: int = 3,
16     h: int = 21,
17     w: int = 21,
18     *,
19     lr: float = 1e-3,
20     weight_decay: float = 0.0,
21     mc_samples: int = 100,
22     max_iterations: int = 4,
23     renyi_scale: float = 0.00,
24     encoder_out: int,
25 ):
26     super().__init__(
27         qbg=qbg,
28         qi=qi,
29         qp=qp,
30         loss=loss,

```

```

31         data_dim=data_dim,
32         d=d,
33         h=h,
34         w=w,
35         lr=lr,
36         weight_decay=weight_decay,
37         mc_samples=mc_samples,
38         max_iterations=max_iterations,
39         renyi_scale=renyi_scale,
40         encoder_out=encoder_out,
41     )
42
43     self.data_dim: str = data_dim
44     self.encoder1 = encoder1
45     self.encoder2 = encoder2
46
47 # def forward(self, counts, shoebox, metadata, masks, reference):
48 def forward(
49     self,
50     counts: Tensor,
51     shoebox: Tensor,
52     masks: Tensor,
53     reference: Tensor | None = None,
54 ) -> dict[str, Any]:
55     """
56     Forward model architecture:
57     ```mermaid
58     flowchart LR
59
60         counts --> encoder1
61         counts --> encoder2
62
63         encoder1 --> qp
64         encoder2 --> qi
65         encoder2 --> qbg
66
67     ```
68     """
69
70     # Unpack batch
71     counts = torch.clamp(counts, min=0) * masks
72
73     profile_rep = self.encoder1(
74         shoebox.reshape(shoebox.shape[0], 1, *(self.shoebox_shape))
75     )
76     intensity_rep = self.encoder2(
77         shoebox.reshape(shoebox.shape[0], 1, *(self.shoebox_shape))
78     )
79
80     qbg = self.qbg(intensity_rep)
81     qp = self.qp(profile_rep)

```

```

82     qi = self.qi(intensity_rep)
83
84     zbg = qbg.rsamples([self.mc_samples]).unsqueeze(-1).permute(1, 0, 2)
85     zp = qp.rsamples([self.mc_samples]).permute(1, 0, 2)
86     zI = qi.rsamples([self.mc_samples]).unsqueeze(-1).permute(1, 0, 2)
87
88     rate = zI * zp + zbg
89
90     # calculate profile renyi entropy
91     avg_renyi_entropy = (-(zp.pow(2).sum(-1).log()))).mean(-1)
92     out = get_outputs(locals(), self.data_dim)
93     return out

```

-

class tmeplIntegratorB(BaseIntegrator): """IntegratorB uses enables the use of metadata"""

```

1  encoder1: ShoeboxEncoder | IntensityEncoder
2  """Encoder to get profile distribution"""
3  encoder2: ShoeboxEncoder | IntensityEncoder
4  """Encoder to get intensity & background distributions"""
5  encoder3: MLPMetadataEncoder | None
6  """Encoder for experimental metadata"""
7
8  def __init__(
9      self,
10     encoder1: ShoeboxEncoder | IntensityEncoder,
11     encoder2: ShoeboxEncoder | IntensityEncoder,
12     encoder3: MLPMetadataEncoder | None,
13     qbg: BaseDistribution,
14     qp: BaseDistribution,
15     qi: BaseDistribution,
16     loss: BaseLoss,
17     data_dim: str,
18     d: int = 3,
19     h: int = 21,
20     w: int = 21,
21     *,
22     lr: float = 1e-3,
23     weight_decay: float = 0.0,
24     mc_samples: int = 100,
25     max_iterations: int = 4,
26     renyi_scale: float = 0.00,
27     encoder_out: int,
28 ):
29     super().__init__(
30         qbg=qbg,
31         qi=qi,

```

```

32         qp=qp,
33         loss=loss,
34         data_dim=data_dim,
35         d=d,
36         h=h,
37         w=w,
38         lr=lr,
39         weight_decay=weight_decay,
40         mc_samples=mc_samples,
41         max_iterations=max_iterations,
42         renyi_scale=renyi_scale,
43         encoder_out=encoder_out,
44     )
45
46     self.encoder1 = encoder1
47     self.encoder2 = encoder2
48     self.encoder3 = encoder3
49
50     self.linear = nn.Linear(self.encoder_out * 2, self.encoder_out)
51
52     def forward(
53         self,
54         counts: Tensor,
55         shoebox: Tensor,
56         masks: Tensor,
57         reference: Tensor | None = None,
58     ) -> dict[str, Any]:
59         """
60         Forward model architecture:
61         ```mermaid
62         flowchart LR
63
64             counts --> encoder1
65             counts --> encoder2
66             metadata --> encoder3
67
68             encoder1 --> qp
69             encoder2 --> torch.concat
70             encoder3 --> torch.concat
71             torch.concat --> qi
72             torch.concat --> qbg
73
74             ...
75
76         Args:
77             counts: Raw photon count Tensor
78             shoebox: Standardized photon count Tensor
79             masks: Dead-pixel mask
80             reference: Optional metadata Tensor
81
82         Returns:

```

```

83
84     """
85     # Unpack batch
86     counts = torch.clamp(counts, min=0) * masks
87
88     x_profile = self.encoder1(
89         shoebox.reshape(shoebox.shape[0], 1, *(self.shoebox_shape))
90     )
91     x_intensity = self.encoder2(
92         shoebox.reshape(shoebox.shape[0], 1, *(self.shoebox_shape))
93     )
94
95     if self.encoder3 is not None and reference is None:
96         assert ValueError(
97             "A metadata encoder (encoder 3) was provided, but no
98             reference data was found. Please provide a `reference.pt
99             ` dataset"
100         )
101
102     metadata = torch.nn.Identity()
103
104     if self.encoder3 is not None and reference is not None:
105         if self.data_dim == "2d" and reference is not None:
106             # TODO: Change the datatypes in the DataLoader
107             metadata = (reference[:, [6, 7, 8, 9, 10, 11]]).float()
108             print("metadata type:", type(metadata))
109
110             elif self.data_dim == "3d" and reference is not None:
111                 metadata = reference[:, [6, 7, 8, 9, 10, 11]]
112
113                 print(metadata)
114
115                 x_metadata = self.encoder3(metadata)
116
117                 x_intensity = torch.concat([x_intensity, x_metadata], dim=-1)
118                 x_intensity = self.linear(x_intensity)
119
120         qbg = self.qbg(x_intensity)
121         qi = self.qi(x_intensity)
122         qp = self.qp(x_profile)
123
124         zbg = qbg.rsamples([self.mc_samples]).unsqueeze(-1).permute(1, 0, 2)
125         zp = qp.rsamples([self.mc_samples]).permute(1, 0, 2)
126         zI = qi.rsamples([self.mc_samples]).unsqueeze(-1).permute(1, 0, 2)
127
128         rate = zI * zp + zbg
129
130         # calculate profile renyi entropy
131         avg_renyi_entropy = (-(zp.pow(2).sum(-1).log())).mean(-1)
132         out = get_outputs(locals(), self.data_dim)
133         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

```

```

1  def __init__(
2      self,
3      encoder1: ShoeboxEncoder | IntensityEncoder,
4      encoder2: ShoeboxEncoder | IntensityEncoder,
5      encoder3: MLPMetadataEncoder | None,
6      qbg: BaseDistribution,
7      qp: BaseDistribution,
8      qi: BaseDistribution,
9      loss: BaseLoss,
10     data_dim: str = "3d", # defaults to rotation data
11     d: int = 3,
12     h: int = 21,
13     w: int = 21,
14     *,
15     lr: float = 1e-3,
16     weight_decay: float = 0.0,
17     mc_samples: int = 100,
18     max_iterations: int = 4,
19     renyi_scale: float = 0.00,
20     encoder_out: int,
21     predict_keys: tuple[str, ...] = (
22         "intensity_mean",
23         "intensity_var",
24         "refl_ids",
25         "dials_I_sum_value",
26         "dials_I_sum_var",
27         "dials_I_prf_value",
28         "dials_I_prf_var",
29         "dials_bg_mean",
30         "qbg_mean",
31         "qbg_scale",
32         "x_c",
33         "y_c",
34         "z_c",

```

```

35     ),
36 ):
37     super().__init__()
38     self.qbg = qbg
39     self.qp = qp
40     self.qi = qi
41     self.d = d
42     self.h = h
43     self.w = w
44     self.loss = loss
45     self.renyi_scale = renyi_scale
46     self.data_dim = data_dim
47     self.encoder_out = encoder_out
48
49     # encoders
50     self.encoder1 = encoder1
51     self.encoder2 = encoder2
52     self.encoder3 = encoder3
53
54     if self.encoder3 is not None:
55         self.linear = nn.Linear(self.encoder_out * 2, self.encoder_out)
56
57     # lists to track avg training metrics
58     self.train_loss = []
59     self.train_kl = []
60     self.train_nll = []
61
62     # lists to track avg validation metrics
63     self.val_loss = []
64     self.val_kl = []
65     self.val_nll = []
66     self.lr = lr
67     self.automatic_optimization = True
68     self.weight_decay = weight_decay
69     self.mc_samples = mc_samples
70     self.max_iterations = max_iterations
71     self.predict_keys = predict_keys
72
73     #
74     if self.data_dim == "3d":
75         self.shoebox_shape = (self.d, self.h, self.w)
76     elif self.data_dim == "2d":
77         self.shoebox_shape = (self.h, self.w)
78
79     # dataframes to keep track of val/train epoch metrics
80     self.schema = [
81         ("epoch", int),
82         ("avg_loss", float),
83         ("avg_kl", float),
84         ("avg_nll", float),
85     ]

```

```

86     self.train_df = pl.DataFrame(schema=self.schema)
87     self.val_df = pl.DataFrame(schema=self.schema)
88
89     def calculate_intensities(self, counts, qbg, qp, masks):
90         with torch.no_grad():
91             counts = counts * masks # [B,P]
92             zbg = qbg.rsample([self.mc_samples]).unsqueeze(-1).permute(1,
93                 0, 2)
94             zp = qp.mean.unsqueeze(1)
95
96             vi = zbg + 1e-6
97             intensity = torch.tensor([0.0])
98
99             # kabsch sum
100            for _ in range(self.max_iterations):
101                num = (
102                    (counts.unsqueeze(1) - zbg) * zp * masks.unsqueeze(1) /
103                    vi
104                )
105                denom = zp.pow(2) / vi
106                intensity = num.sum(-1) / denom.sum(
107                    -1
108                ) # [batch_size, mc_samples]
109                vi = (intensity.unsqueeze(-1) * zp) + zbg
110                vi = vi.mean(-1, keepdim=True)
111                kabsch_sum_mean = intensity.mean(-1)
112                kabsch_sum_var = intensity.var(-1)
113
114            # profile masking
115            zp = zp * masks.unsqueeze(1) # profiles
116            thresholds = torch.quantile(
117                zp,
118                0.99,
119                dim=-1,
120                keepdim=True,
121            ) # threshold values
122            profile_mask = zp > thresholds
123
124            masked_counts = counts.unsqueeze(1) * profile_mask
125
126            profile_masking_I = (masked_counts - zbg * profile_mask).sum(
127                -1)
128
129            profile_masking_mean = profile_masking_I.mean(-1)
130            profile_masking_var = profile_masking_I.var(-1)
131
132            intensities = {
133                "profile_masking_mean": profile_masking_mean,
134                "profile_masking_var": profile_masking_var,
135                "kabsch_sum_mean": kabsch_sum_mean,

```

```

134         "kabsch_sum_var": kabsch_sum_var,
135     }
136
137     return intensities
138
139 def forward(
140     self,
141     counts: Tensor,
142     shoebox: Tensor,
143     masks: Tensor,
144     reference: Tensor | None = None,
145 ) -> dict[str, Any]:
146     """
147     Forward model architecture:
148     ```mermaid
149     flowchart LR
150
151         counts --> encoder1
152         counts --> encoder2
153         metadata --> encoder3
154
155         encoder1 --> qp
156         encoder2 --> torch.concat
157         encoder3 --> torch.concat
158         torch.concat --> qi
159         torch.concat --> qbg
160
161     ```
162
163     Args:
164         counts: Raw photon count Tensor
165         shoebox: Standardized photon count Tensor
166         masks: Dead-pixel mask
167         reference: Optional metadata Tensor
168
169     Returns:
170
171     """
172     # Unpack batch
173     counts = torch.clamp(counts, min=0) * masks
174
175     x_profile = self.encoder1(
176         shoebox.reshape(shoebox.shape[0], 1, *(self.shoebox_shape))
177     )
178     x_intensity = self.encoder2(
179         shoebox.reshape(shoebox.shape[0], 1, *(self.shoebox_shape))
180     )
181
182     if self.encoder3 is not None and reference is None:
183         assert ValueError(
184             "A metadata encoder (encoder 3) was provided, but no

```

```

        reference data was found. Please provide a `reference.pt
        ` dataset"
185     )
186
187     metadata = torch.nn.Identity()
188
189     if self.encoder3 is not None and reference is not None:
190         if self.data_dim == "2d" and reference is not None:
191             # TODO: Change the datatypes in the DataLoader
192             metadata = (reference[:, [6, 7, 8, 9, 10, 11]]).float()
193             print("metadata type:", type(metadata))
194
195         elif self.data_dim == "3d" and reference is not None:
196             metadata = reference[:, [0, 1, 2, 3, 4, 5, 13]]
197
198         print(metadata)
199
200         x_metadata = self.encoder3(metadata)
201
202         x_intensity = torch.concat([x_intensity, x_metadata], dim=-1)
203         x_intensity = self.linear(x_intensity)
204
205         qbg = self.qbg(x_intensity)
206         qi = self.qi(x_intensity)
207         qp = self.qp(x_profile)
208
209         zbg = qbg.rsamle([self.mc_samples]).unsqueeze(-1).permute(1, 0, 2)
210         zp = qp.rsamle([self.mc_samples]).permute(1, 0, 2)
211         zI = qi.rsamle([self.mc_samples]).unsqueeze(-1).permute(1, 0, 2)
212
213         rate = zI * zp + zbg
214
215         # calculate profile renyi entropy
216         avg_renyi_entropy = (- (zp.pow(2).sum(-1).log())) .mean(-1)
217         out = get_outputs(locals(), self.data_dim)
218         return out
219
220     def on_train_epoch_end(self):
221         # calculate epoch averages
222         avg_train_loss = sum(self.train_loss) / len(self.train_loss)
223         avg_kl = sum(self.train_kl) / len(self.train_kl)
224         avg_nll = sum(self.train_nll) / len(self.train_nll)
225
226         # log averages to weights & biases
227         self.log("train_loss", avg_train_loss)
228         self.log("avg_kl", avg_kl)
229         self.log("avg_nll", avg_nll)
230
231         # create epoch dataframe
232         epoch_df = pl.DataFrame(
233             {

```

```

234         "epoch": self.current_epoch,
235         "avg_loss": avg_train_loss,
236         "avg_kl": avg_kl,
237         "avg_nll": avg_nll,
238     }
239 )
240
241 # update training dataframe
242 self.train_df = pl.concat([self.train_df, epoch_df])
243
244 # clear all lists
245 self.train_loss = []
246 self.train_kl = []
247 self.train_nll = []
248
249 def on_validation_epoch_end(self):
250     """Validation step processing"""
251     avg_val_loss = sum(self.val_loss) / len(self.val_loss)
252     avg_kl = sum(self.val_kl) / len(self.val_kl)
253     avg_nll = sum(self.val_nll) / len(self.val_nll)
254
255     self.log("validation_loss", avg_val_loss)
256     self.log("validation_avg_kl", avg_kl)
257     self.log("validation_avg_nll", avg_nll)
258
259     epoch_df = pl.DataFrame(
260         {
261             "epoch": self.current_epoch,
262             "avg_loss": avg_val_loss,
263             "avg_kl": avg_kl,
264             "avg_nll": avg_nll,
265         }
266     )
267     self.val_df = pl.concat([self.val_df, epoch_df])
268
269     self.val_loss = []
270     self.avg_kl = []
271     self.val_nll = []
272
273 def training_step(self, batch, _batch_idx):
274     counts, shoebox, masks, reference = batch
275     outputs = self(counts, shoebox, masks, reference)
276
277     # Calculate loss
278     loss_dict = self.loss(
279         rate=outputs["rates"],
280         counts=outputs["counts"],
281         q_p=outputs["qp"],
282         q_i=outputs["qi"],
283         q_bg=outputs["qbg"],
284         masks=outputs["masks"],

```

```

285     )
286
287     renyi_loss = (
288         (
289             -torch.log(
290                 outputs["qp"]
291                 .rsample([self.mc_samples])
292                 .permute(1, 0, 2)
293                 .pow(2)
294                 .sum(-1)
295             )
296         )
297         .mean(1)
298         .sum()
299     ) * self.renyi_scale
300     self.log("renyi_loss", renyi_loss)
301
302     for k, v in loss_dict.items():
303         key = f"train_{k}"
304         value = v.mean()
305         self.log(key, value)
306
307     self.log("Mean(qi.mean)", outputs["qi"].mean.mean())
308     self.log("Min(qi.mean)", outputs["qi"].mean.min())
309     self.log("Max(qi.mean)", outputs["qi"].mean.max())
310     self.log("Mean(qbg.mean)", outputs["qbg"].mean.mean())
311     self.log("Min(qbg.mean)", outputs["qbg"].mean.min())
312     self.log("Max(qbg.mean)", outputs["qbg"].mean.max())
313     self.log("Mean(qbg.variance)", outputs["qbg"].variance.mean())
314
315     self.train_loss.append(loss_dict["total_loss"].mean())
316     self.train_kl.append(loss_dict["kl_mean"].mean())
317     self.train_nll.append(loss_dict["neg_ll_mean"].mean())
318
319     return loss_dict["total_loss"].mean() + renyi_loss.sum()
320
321 def configure_optimizers(self):
322     return torch.optim.Adam(
323         self.parameters(), lr=self.lr, weight_decay=self.weight_decay
324     )
325
326 def validation_step(self, batch, _batch_idx):
327     """
328
329     Args:
330         batch ():
331         _batch_idx ():
332
333     Returns:
334
335     """

```

```

336     # Unpack batch
337     counts, shoebox, masks, reference = batch
338     outputs = self(counts, shoebox, masks, reference)
339
340     loss_dict = self.loss(
341         rate=outputs["rates"],
342         counts=outputs["counts"],
343         q_p=outputs["qp"],
344         q_i=outputs["qi"],
345         q_bg=outputs["qbg"],
346         masks=outputs["masks"],
347     )
348
349     for k, v in loss_dict.items():
350         key = f"val_{k}"
351         value = v.mean()
352         self.log(key, value)
353
354     self.val_loss.append(loss_dict["total_loss"].mean())
355     self.val_kl.append(loss_dict["kl_mean"].mean())
356     self.val_nll.append(loss_dict["neg_ll_mean"].mean())
357
358     return outputs
359
360 def predict_step(self, batch, _batch_idx):
361     """Prediction step
362
363     Args:
364         batch: Inpute Tensor data
365
366     Returns:
367
368     """
369
370     counts, shoebox, masks, reference = batch
371     outputs = self(counts, shoebox, masks, reference)
372
373     return {k: v for k, v in outputs.items() if k in self.predict_keys}

```

-

if name == "main": import torch

```

1  from integrator.utils import (
2      create_data_loader,
3      create_integrator,
4      load_config,
5  )

```

```

6 from utils import CONFIGS
7
8 torch.set_default_dtype(torch.float32)
9
10 # Model A 3D
11 config = load_config(CONFIGS["integratorA_3D"])
12 integrator = create_integrator(config.dict())
13 data_loader = create_data_loader(config.dict())
14 counts, shoebox, masks, reference = next(
15     iter(data_loader.train_dataloader())
16 )
17 out_2d = integrator(counts, shoebox, masks, reference)
18
19 # Model A 2D
20 config = load_config(CONFIGS["integratorA_2D"])
21
22 integrator = create_integrator(config.dict())
23 data_loader = create_data_loader(config.dict())
24 counts, shoebox, masks, reference = next(
25     iter(data_loader.train_dataloader())
26 )
27 out_2d = integrator(counts, shoebox, masks, reference)
28
29 # Model B 3D
30 CONFIGS["integratorB_3D"].as_posix()
31 config = load_config(CONFIGS["integratorB_3D"])
32 config.model_dump()["integrator"]
33 integrator = create_integrator(config.dict())
34 data_loader = create_data_loader(config.dict())
35 counts, shoebox, masks, reference = next(
36     iter(data_loader.train_dataloader())
37 )
38 out_3d = integrator(counts, shoebox, masks, reference)
39
40 # Model B 2D
41 CONFIGS["integratorB_2D"].as_posix()
42
43 config = load_config(CONFIGS["integratorB_2D"])
44
45 config.model_dump()["integrator"]
46
47 integrator = create_integrator(config.dict())
48
49 data_loader = create_data_loader(config.dict())
50
51 counts, shoebox, masks, reference = next(
52     iter(data_loader.train_dataloader())
53 )
54 out_3d = integrator(counts, shoebox, masks, reference)
55
56 # 2D - no metadata

```

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