

Forward Boltz2 (boltz/src/boltz/model/models /boltz2.py)

```
1  def forward(
2      self,
3      feats: dict[str, Tensor],
4      recycling_steps: int = 0,
5      num_sampling_steps: Optional[int] = None,
6      multiplicity_diffusion_train: int = 1,
7      diffusion_samples: int = 1,
8      max_parallel_samples: Optional[int] = None,
9      run_confidence_sequentially: bool = False,
10 ) -> dict[str, Tensor]:
11     with torch.set_grad_enabled(
12         self.training and self.structure_prediction_training
13     ):
14         s_inputs = self.input_embedder(feats)
15
16         # Initialize the sequence embeddings
17         s_init = self.s_init(s_inputs)
18
19         # Initialize pairwise embeddings
20         z_init = (
21             self.z_init_1(s_inputs)[: , :, None]
22             + self.z_init_2(s_inputs)[: , None, :]
23         )
24         relative_position_encoding = self.rel_pos(feats)
25         z_init = z_init + relative_position_encoding
26         z_init = z_init + self.token_bonds(feats["token_bonds"].float())
27         if self.bond_type_feature:
28             z_init = z_init +
29                 ↪ self.token_bonds_type(feats["type_bonds"].long())
30         z_init = z_init + self.contact_conditioning(feats)
31
32         # Perform rounds of the pairwise stack
33         s = torch.zeros_like(s_init)
34         z = torch.zeros_like(z_init)
35
36         # Compute pairwise mask
37         mask = feats["token_pad_mask"].float()
38         pair_mask = mask[: , :, None] * mask[: , None, :]
39         if self.run_trunk_and_structure:
40             for i in range(recycling_steps + 1):
41                 with torch.set_grad_enabled(
42                     self.training
43                     and self.structure_prediction_training
44                     and (i == recycling_steps)
45                 ):
46                     # Issue with unused parameters in autocast
47                     if (
48                         self.training
```

```

48         and (i == recycling_steps)
49         and torch.is_autocast_enabled()
50     ):
51         torch.clear_autocast_cache()
52
53     # Apply recycling
54     s = s_init + self.s_recycle(self.s_norm(s))
55     z = z_init + self.z_recycle(self.z_norm(z))
56
57     # Compute pairwise stack
58     if self.use_templates:
59         if self.is_template_compiled and not
        ↪ self.training:
60             template_module =
        ↪ self.template_module._orig_mod # noqa:
        ↪ SLF001
61         else:
62             template_module = self.template_module
63
64     z = z + template_module(
65         z, feats, pair_mask,
66         ↪ use_kernels=self.use_kernels
67     )
68
69     if self.is_msa_compiled and not self.training:
70         msa_module = self.msa_module._orig_mod # noqa:
71         ↪ SLF001
72     else:
73         msa_module = self.msa_module
74
75     z = z + msa_module(
76         z, s_inputs, feats, use_kernels=self.use_kernels
77     )
78
79     # Revert to uncompiled version for validation
80     if self.is_pairformer_compiled and not self.training:
81         pairformer_module =
82         ↪ self.pairformer_module._orig_mod # noqa:
83         ↪ SLF001
84     else:
85         pairformer_module = self.pairformer_module
86
87     s, z = pairformer_module(
88         s,
89         z,
90         mask=mask,
91         pair_mask=pair_mask,
92         use_kernels=self.use_kernels,
93     )

```

```

91     pdistogram = self.distogram_module(z)
92     dict_out = {
93         "pdistogram": pdistogram,
94         "s": s,
95         "z": z,
96     }
97
98     if (
99         self.run_trunk_and_structure
100         and ((not self.training) or self.confidence_prediction)
101         and (not self.skip_run_structure)
102     ):
103         if self.checkpoint_diffusion_conditioning and self.training:
104             # TODO decide whether this should be with bf16 or not
105             q, c, to_keys, atom_enc_bias, atom_dec_bias,
106             ↪ token_trans_bias = (
107                 torch.utils.checkpoint.checkpoint(
108                     self.diffusion_conditioning,
109                     s,
110                     z,
111                     relative_position_encoding,
112                     feats,
113                 )
114             )
115         else:
116             q, c, to_keys, atom_enc_bias, atom_dec_bias,
117             ↪ token_trans_bias = (
118                 self.diffusion_conditioning(
119                     s_trunk=s,
120                     z_trunk=z,
121
122                     ↪ relative_position_encoding=relative_position_encoding,
123                     feats=feats,
124                 )
125             )
126         diffusion_conditioning = {
127             "q": q,
128             "c": c,
129             "to_keys": to_keys,
130             "atom_enc_bias": atom_enc_bias,
131             "atom_dec_bias": atom_dec_bias,
132             "token_trans_bias": token_trans_bias,
133         }
134
135         with torch.autocast("cuda", enabled=False):
136             struct_out = self.structure_module.sample(
137                 s_trunk=s.float(),
138                 s_inputs=s_inputs.float(),
139                 feats=feats,
140                 num_sampling_steps=num_sampling_steps,

```

```

138         atom_mask=feats["atom_pad_mask"].float(),
139         multiplicity=diffusion_samples,
140         max_parallel_samples=max_parallel_samples,
141         steering_args=self.steering_args,
142         diffusion_conditioning=diffusion_conditioning,
143     )
144     dict_out.update(struct_out)
145
146     if self.predict_bfactor:
147         pbfactor = self.bfactor_module(s)
148         dict_out["pbfactor"] = pbfactor
149
150     if self.training and self.confidence_prediction:
151         assert len(feats["coords"].shape) == 4
152         assert feats["coords"].shape[1] == 1, (
153             "Only one conformation is supported for confidence"
154         )
155
156     # Compute structure module
157     if self.training and self.structure_prediction_training:
158         atom_coords = feats["coords"]
159         B, K, L = atom_coords.shape[0:3]
160         assert K in (
161             multiplicity_diffusion_train,
162             1,
163         ) # TODO make check somewhere else, expand to m % N == 0, m
164         ↪ > N
165         atom_coords = atom_coords.reshape(B * K, L, 3)
166         atom_coords = atom_coords.repeat_interleave(
167             multiplicity_diffusion_train // K, 0
168         )
169         feats["coords"] = atom_coords # (multiplicity, L, 3)
170         assert len(feats["coords"].shape) == 3
171
172         with torch.autocast("cuda", enabled=False):
173             struct_out = self.structure_module(
174                 s_trunk=s.float(),
175                 s_inputs=s_inputs.float(),
176                 feats=feats,
177                 multiplicity=multiplicity_diffusion_train,
178                 diffusion_conditioning=diffusion_conditioning,
179             )
180             dict_out.update(struct_out)
181
182     elif self.training:
183         feats["coords"] = feats["coords"].squeeze(1)
184         assert len(feats["coords"].shape) == 3
185
186     if self.confidence_prediction:
187         dict_out.update(

```

```

187         self.confidence_module(
188             s_inputs=s_inputs.detach(),
189             s=s.detach(),
190             z=z.detach(),
191             x_pred=(
192                 dict_out["sample_atom_coords"].detach()
193                 if not self.skip_run_structure
194                 else
195                 ↪ feats["coords"].repeat_interleave(diffusion_samples,
196                 ↪ 0)
197             ),
198             feats=feats,
199             pred_distogram_logits=(
200                 dict_out["pdistogram"] [
201                     :, :, :, 0
202                 ].detach() # TODO only implemented for 1 distogram
203             ),
204             multiplicity=diffusion_samples,
205             run_sequentially=run_confidence_sequentially,
206             use_kernels=self.use_kernels,
207         )
208     )
209
210     if self.affinity_prediction:
211         pad_token_mask = feats["token_pad_mask"][0]
212         rec_mask = feats["mol_type"][0] == 0
213         rec_mask = rec_mask * pad_token_mask
214         lig_mask = feats["affinity_token_mask"][0].to(torch.bool)
215         lig_mask = lig_mask * pad_token_mask
216         cross_pair_mask = (
217             lig_mask[:, None] * rec_mask[None, :]
218             + rec_mask[:, None] * lig_mask[None, :]
219             + lig_mask[:, None] * lig_mask[None, :]
220         )
221         z_affinity = z * cross_pair_mask[None, :, :, None]
222
223         argsort = torch.argsort(dict_out["iptm"], descending=True)
224         best_idx = argsort[0].item()
225         coords_affinity =
226         ↪ dict_out["sample_atom_coords"].detach()[best_idx] [
227             None, None
228         ]
229         s_inputs = self.input_embedder(feats, affinity=True)
230
231         with torch.autocast("cuda", enabled=False):
232             if self.affinity_ensemble:
233                 dict_out_affinity1 = self.affinity_module1(
234                     s_inputs=s_inputs.detach(),
235                     z=z_affinity.detach(),
236                     x_pred=coords_affinity,

```

```

234         feats=feats,
235         multiplicity=1,
236         use_kernels=self.use_kernels,
237     )
238
239     dict_out_affinity1["affinity_probability_binary"] = (
240         torch.nn.functional.sigmoid(
241             dict_out_affinity1["affinity_logits_binary"]
242         )
243     )
244     dict_out_affinity2 = self.affinity_module2(
245         s_inputs=s_inputs.detach(),
246         z=z_affinity.detach(),
247         x_pred=coords_affinity,
248         feats=feats,
249         multiplicity=1,
250         use_kernels=self.use_kernels,
251     )
252     dict_out_affinity2["affinity_probability_binary"] = (
253         torch.nn.functional.sigmoid(
254             dict_out_affinity2["affinity_logits_binary"]
255         )
256     )
257
258     dict_out_affinity_ensemble = {
259         "affinity_pred_value": (
260             dict_out_affinity1["affinity_pred_value"]
261             + dict_out_affinity2["affinity_pred_value"]
262         )
263         / 2,
264         "affinity_probability_binary": (
265             dict_out_affinity1["affinity_probability_binary"]
266             +
267             ↵ dict_out_affinity2["affinity_probability_binary"]
268         )
269         / 2,
270     }
271
272     dict_out_affinity1 = {
273         "affinity_pred_value1": dict_out_affinity1[
274             "affinity_pred_value"
275         ],
276         "affinity_probability_binary1": dict_out_affinity1[
277             "affinity_probability_binary"
278         ],
279     }
280     dict_out_affinity2 = {
281         "affinity_pred_value2": dict_out_affinity2[
282             "affinity_pred_value"
283         ],

```

```

283         "affinity_probability_binary2": dict_out_affinity2[
284             "affinity_probability_binary"
285         ],
286     }
287     if self.affinity_mw_correction:
288         model_coef = 1.03525938
289         mw_coef = -0.59992683
290         bias = 2.83288489
291         mw = feats["affinity_mw"][0] ** 0.3
292         dict_out_affinity_ensemble["affinity_pred_value"] = (
293             model_coef
294             *
295             ↪ dict_out_affinity_ensemble["affinity_pred_value"]
296             + mw_coef * mw
297             + bias
298         )
299         dict_out.update(dict_out_affinity_ensemble)
300         dict_out.update(dict_out_affinity1)
301         dict_out.update(dict_out_affinity2)
302     else:
303         dict_out_affinity = self.affinity_module(
304             s_inputs=s_inputs.detach(),
305             z=z_affinity.detach(),
306             x_pred=coords_affinity,
307             feats=feats,
308             multiplicity=1,
309             use_kernels=self.use_kernels,
310         )
311         dict_out.update(
312             {
313                 "affinity_pred_value": dict_out_affinity[
314                     "affinity_pred_value"
315                 ],
316                 "affinity_probability_binary":
317                 ↪ torch.nn.functional.sigmoid(
318                     dict_out_affinity["affinity_logits_binary"]
319                 ),
320             }
321         )
322     return dict_out

```