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

```python
def forward(
        self,
        feats: dict[str, Tensor],
        recycling_steps: int = 0,
        num_sampling_steps: Optional[int] = None,
        multiplicity_diffusion_train: int = 1,
        diffusion_samples: int = 1,
        max_parallel_samples: Optional[int] = None,
        run_confidence_sequentially: bool = False,
    ) -> dict[str, Tensor]:
        with torch.set_grad_enabled(
            self.training and self.structure_prediction_training
        ):
            s_inputs = self.input_embedder(feats)

            # Initialize the sequence embeddings
            s_init = self.s_init(s_inputs)

            # Initialize pairwise embeddings
            z_init = (
                self.z_init_1(s_inputs)[:, :, None]
                + self.z_init_2(s_inputs)[:, None, :]
            )
            relative_position_encoding = self.rel_pos(feats)
            z_init = z_init + relative_position_encoding
            z_init = z_init + self.token_bonds(feats["token_bonds"].float())
            if self.bond_type_feature:
                z_init = z_init +
                ↪  self.token_bonds_type(feats["type_bonds"].long())
            z_init = z_init + self.contact_conditioning(feats)

            # Perform rounds of the pairwise stack
            s = torch.zeros_like(s_init)
            z = torch.zeros_like(z_init)

            # Compute pairwise mask
            mask = feats["token_pad_mask"].float()
            pair_mask = mask[:, :, None] * mask[:, None, :]
            if self.run_trunk_and_structure:
                for i in range(recycling_steps + 1):
                    with torch.set_grad_enabled(
                        self.training
                        and self.structure_prediction_training
                        and (i == recycling_steps)
                    ):
                        # Issue with unused parameters in autocast
                        if (
                            self.training
```

```
                    and (i == recycling_steps)
                    and torch.is_autocast_enabled()
                ):
                    torch.clear_autocast_cache()

                # Apply recycling
                s = s_init + self.s_recycle(self.s_norm(s))
                z = z_init + self.z_recycle(self.z_norm(z))

                # Compute pairwise stack
                if self.use_templates:
                    if self.is_template_compiled and not self.training:
                        template_module = self.template_module._orig_mod  # noqa: SLF001
                    else:
                        template_module = self.template_module

                    z = z + template_module(
                        z, feats, pair_mask, use_kernels=self.use_kernels
                    )

                if self.is_msa_compiled and not self.training:
                    msa_module = self.msa_module._orig_mod  # noqa: SLF001
                else:
                    msa_module = self.msa_module

                z = z + msa_module(
                    z, s_inputs, feats, use_kernels=self.use_kernels
                )

                # Revert to uncompiled version for validation
                if self.is_pairformer_compiled and not self.training:
                    pairformer_module = self.pairformer_module._orig_mod  # noqa: SLF001
                else:
                    pairformer_module = self.pairformer_module

                s, z = pairformer_module(
                    s,
                    z,
                    mask=mask,
                    pair_mask=pair_mask,
                    use_kernels=self.use_kernels,
                )

```

```
                pdistogram = self.distogram_module(z)
                dict_out = {
                    "pdistogram": pdistogram,
                    "s": s,
                    "z": z,
                }

                if (
                    self.run_trunk_and_structure
                    and ((not self.training) or self.confidence_prediction)
                    and (not self.skip_run_structure)
                ):
                    if self.checkpoint_diffusion_conditioning and self.training:
                        # TODO decide whether this should be with bf16 or not
                        q, c, to_keys, atom_enc_bias, atom_dec_bias,
                        ↪  token_trans_bias = (
                            torch.utils.checkpoint.checkpoint(
                                self.diffusion_conditioning,
                                s,
                                z,
                                relative_position_encoding,
                                feats,
                            )
                        )
                    else:
                        q, c, to_keys, atom_enc_bias, atom_dec_bias,
                        ↪  token_trans_bias = (
                            self.diffusion_conditioning(
                                s_trunk=s,
                                z_trunk=z,

                                ↪  relative_position_encoding=relative_position_encoding,
                                feats=feats,
                            )
                        )
                    diffusion_conditioning = {
                        "q": q,
                        "c": c,
                        "to_keys": to_keys,
                        "atom_enc_bias": atom_enc_bias,
                        "atom_dec_bias": atom_dec_bias,
                        "token_trans_bias": token_trans_bias,
                    }

                    with torch.autocast("cuda", enabled=False):
                        struct_out = self.structure_module.sample(
                            s_trunk=s.float(),
                            s_inputs=s_inputs.float(),
                            feats=feats,
                            num_sampling_steps=num_sampling_steps,
```

```
                        atom_mask=feats["atom_pad_mask"].float(),
                        multiplicity=diffusion_samples,
                        max_parallel_samples=max_parallel_samples,
                        steering_args=self.steering_args,
                        diffusion_conditioning=diffusion_conditioning,
                    )
                    dict_out.update(struct_out)

                if self.predict_bfactor:
                    pbfactor = self.bfactor_module(s)
                    dict_out["pbfactor"] = pbfactor

            if self.training and self.confidence_prediction:
                assert len(feats["coords"].shape) == 4
                assert feats["coords"].shape[1] == 1, (
                    "Only one conformation is supported for confidence"
                )

            # Compute structure module
            if self.training and self.structure_prediction_training:
                atom_coords = feats["coords"]
                B, K, L = atom_coords.shape[0:3]
                assert K in (
                    multiplicity_diffusion_train,
                    1,
                )  # TODO make check somewhere else, expand to m % N == 0, m
                ↪  > N
                atom_coords = atom_coords.reshape(B * K, L, 3)
                atom_coords = atom_coords.repeat_interleave(
                    multiplicity_diffusion_train // K, 0
                )
                feats["coords"] = atom_coords  # (multiplicity, L, 3)
                assert len(feats["coords"].shape) == 3

                with torch.autocast("cuda", enabled=False):
                    struct_out = self.structure_module(
                        s_trunk=s.float(),
                        s_inputs=s_inputs.float(),
                        feats=feats,
                        multiplicity=multiplicity_diffusion_train,
                        diffusion_conditioning=diffusion_conditioning,
                    )
                    dict_out.update(struct_out)

            elif self.training:
                feats["coords"] = feats["coords"].squeeze(1)
                assert len(feats["coords"].shape) == 3

        if self.confidence_prediction:
            dict_out.update(
```

```
                self.confidence_module(
                    s_inputs=s_inputs.detach(),
                    s=s.detach(),
                    z=z.detach(),
                    x_pred=(
                        dict_out["sample_atom_coords"].detach()
                        if not self.skip_run_structure
                        else
                        ↪  feats["coords"].repeat_interleave(diffusion_samples,
                        ↪  0)
                    ),
                    feats=feats,
                    pred_distogram_logits=(
                        dict_out["pdistogram"][
                            :, :, :, 0
                        ].detach()  # TODO only implemented for 1 distogram
                    ),
                    multiplicity=diffusion_samples,
                    run_sequentially=run_confidence_sequentially,
                    use_kernels=self.use_kernels,
                )
            )

        if self.affinity_prediction:
            pad_token_mask = feats["token_pad_mask"][0]
            rec_mask = feats["mol_type"][0] == 0
            rec_mask = rec_mask * pad_token_mask
            lig_mask = feats["affinity_token_mask"][0].to(torch.bool)
            lig_mask = lig_mask * pad_token_mask
            cross_pair_mask = (
                lig_mask[:, None] * rec_mask[None, :]
                + rec_mask[:, None] * lig_mask[None, :]
                + lig_mask[:, None] * lig_mask[None, :]
            )
            z_affinity = z * cross_pair_mask[None, :, :, None]

            argsort = torch.argsort(dict_out["iptm"], descending=True)
            best_idx = argsort[0].item()
            coords_affinity =
            ↪  dict_out["sample_atom_coords"].detach()[best_idx][
                None, None
            ]
            s_inputs = self.input_embedder(feats, affinity=True)

            with torch.autocast("cuda", enabled=False):
                if self.affinity_ensemble:
                    dict_out_affinity1 = self.affinity_module1(
                        s_inputs=s_inputs.detach(),
                        z=z_affinity.detach(),
                        x_pred=coords_affinity,
```

```
                    feats=feats,
                    multiplicity=1,
                    use_kernels=self.use_kernels,
                )

                dict_out_affinity1["affinity_probability_binary"] = (
                    torch.nn.functional.sigmoid(
                        dict_out_affinity1["affinity_logits_binary"]
                    )
                )
                dict_out_affinity2 = self.affinity_module2(
                    s_inputs=s_inputs.detach(),
                    z=z_affinity.detach(),
                    x_pred=coords_affinity,
                    feats=feats,
                    multiplicity=1,
                    use_kernels=self.use_kernels,
                )
                dict_out_affinity2["affinity_probability_binary"] = (
                    torch.nn.functional.sigmoid(
                        dict_out_affinity2["affinity_logits_binary"]
                    )
                )

                dict_out_affinity_ensemble = {
                    "affinity_pred_value": (
                        dict_out_affinity1["affinity_pred_value"]
                        + dict_out_affinity2["affinity_pred_value"]
                    )
                    / 2,
                    "affinity_probability_binary": (
                        dict_out_affinity1["affinity_probability_binary"]
                        +
                         ↪  dict_out_affinity2["affinity_probability_binary"]
                    )
                    / 2,
                }

                dict_out_affinity1 = {
                    "affinity_pred_value1": dict_out_affinity1[
                        "affinity_pred_value"
                    ],
                    "affinity_probability_binary1": dict_out_affinity1[
                        "affinity_probability_binary"
                    ],
                }
                dict_out_affinity2 = {
                    "affinity_pred_value2": dict_out_affinity2[
                        "affinity_pred_value"
                    ],
```

```
                        "affinity_probability_binary2": dict_out_affinity2[
                            "affinity_probability_binary"
                        ],
                    }
                    if self.affinity_mw_correction:
                        model_coef = 1.03525938
                        mw_coef = -0.59992683
                        bias = 2.83288489
                        mw = feats["affinity_mw"][0] ** 0.3
                        dict_out_affinity_ensemble["affinity_pred_value"] = (
                            model_coef
                            *
                             ↪  dict_out_affinity_ensemble["affinity_pred_value"]
                            + mw_coef * mw
                            + bias
                        )

                    dict_out.update(dict_out_affinity_ensemble)
                    dict_out.update(dict_out_affinity1)
                    dict_out.update(dict_out_affinity2)
                else:
                    dict_out_affinity = self.affinity_module(
                        s_inputs=s_inputs.detach(),
                        z=z_affinity.detach(),
                        x_pred=coords_affinity,
                        feats=feats,
                        multiplicity=1,
                        use_kernels=self.use_kernels,
                    )
                    dict_out.update(
                        {
                            "affinity_pred_value": dict_out_affinity[
                                "affinity_pred_value"
                            ],
                            "affinity_probability_binary":
                             ↪  torch.nn.functional.sigmoid(
                                dict_out_affinity["affinity_logits_binary"]
                            ),
                        }
                    )

        return dict_out
```