■ Training Code Export (Complete Code - No Truncation)

- [PY] config.py
- [PY] dataset.py
- [PY] evaluation\metrics.py
- [PY] inference\engine.py
- [PY] models\dvx.py
- [PY] models\encoder.py
- [PY] models\extrusion.py
- [PY] models\heads.py
- [PY] models\model.py
- [PY] train.py
- [PY] training\losses.py
- [PY] training\trainer.py
- [PY] utils\visualization.py

■ File: config.py

```
1: """
 2: Configuration settings for the Neural-Geometric 3D Model Generator
 3: Enhanced with dynamic curriculum and adaptive training strategies
 5: from dataclasses import dataclass
 6: from typing import Tuple, Dict, Any, Optional, List
 7: import torch
 8:
 9:
10: @dataclass
11: class DataConfig:
12:
       """Data-related configuration"""
13:
       data dir: str = "./data/floorplans"
14:
      image_size: Tuple[int, int] = (256, 256)
                                                # keep full resolution for accuracy
15:
       voxel_size: int = 64
16:
       batch_size: int = 4
                                                  # balance speed & memory
17:
       num workers: int = 8
                                                  # faster dataloader (tune per CPU)
18:
       augment: bool = True
19:
20:
21: @dataclass
22: class ModelConfig:
        """Model architecture configuration optimized for high accuracy"""
23:
24:
       input_channels: int = 3
25:
       num_classes: int = 5
       feature_dim: int = 512
26:
                                  # reduced from 768 ? faster while keeping strong accuracy
27:
      num_attributes: int = 6
28:
      voxel_size: int = 64
29:
       max polygons: int = 20
                                  # enough for complex layouts
30:
       max_points: int = 50
                                  # good detail without huge cost
31:
       dropout: float = 0.05
       use_attention: bool = True
32:
       use_deep_supervision: bool = True
33:
        # Auxiliary heads for novel training strategies
35:
36:
       use_latent_consistency: bool = True
37:
       use_graph_constraints: bool = True
       latent_embedding_dim: int = 256
38:
39:
40:
41: @dataclass
42: class CurriculumConfig:
43:
        """Dynamic curriculum learning configuration"""
44:
        # Adaptive stage transitioning
45:
       use_dynamic_curriculum: bool = True
46:
       stage_switch_patience: int = 5
       min_improvement_threshold: float = 0.001
47:
48:
       plateau_detection_window: int = 3
49:
50:
        # GradNorm / gradient tracking
51:
        gradient_norm_window: int = 100
52:
53:
        # Objectives for multi-objective optimization
        objectives: Optional[List[str]] = None
54:
55:
56:
        # Topology-aware scheduling
57:
        topology_schedule: str = "progressive" # "progressive", "linear_ramp", "exponential"
58:
        topology_start_weight: float = 0.1
59:
        topology_end_weight: float = 1.0
60:
        topology_ramp_epochs: int = 20
61:
62:
        # config.py (snippet ? add into the existing config class/dict)
        # Mixed precision and training conveniences
63:
64:
        use_mixed_precision = True
                                            # enable AMP
65:
       cache_in_memory = False
                                             # set True if host RAM can hold dataset
66:
       accumulation_steps = 1
                                            # effective batch size multiplier
67:
       dvx_step_freq = 1
                                             # run DVX refinement every N steps (1 = every step)
68:
                                            # DataLoader persistent workers
       persistent workers = True
69:
                                             # DataLoader prefetch
       prefetch factor = 4
70:
       num_workers = 8
                                             # default num workers for DataLoader (tune by CPU)
71:
        # Progressive resolution settings (example)
```

```
72:
         voxel_size_stage = { "stage1": 32, "stage2": 32, "stage3": 64 } # voxel sizes per stage
 73:
         image_size_stage = { "stage1": (128,128), "stage2": (192,192), "stage3": (256,256)}
 74:
 75:
 76:
         # Loss component scheduling
         loss_schedule: Dict[str, str] = None
 77:
 78:
 79:
         # Multi-objective optimization (GradNorm)
 :08
         use_gradnorm: bool = True
         gradnorm_alpha: float = 0.12
 81:
         gradnorm_update_freq: int = 5
 82:
 83:
 84:
         # Graph constraint scheduling
         graph_weight_schedule: str = "delayed_ramp"
 85:
        graph_start_epoch: int = 15
 86:
 87:
        graph_end_weight: float = 0.5
 88:
 89:
         def __post_init__(self):
             # Provide default loss schedule if not set
 90:
 91:
             if self.loss_schedule is None:
 92:
                 self.loss_schedule = {
                      "segmentation": "static",
 93:
 94:
                     "dice": "static",
                     "sdf": "early_decay",
 95:
                     "attributes": "static",
 96:
                     "polygon": "staged_ramp",
 97:
 98:
                     "voxel": "late_ramp",
                     "topology": "progressive",
 99:
100:
                     "latent_consistency": "mid_ramp",
101:
                     "graph": "delayed_ramp",
102:
                 }
103:
104:
            # Default objectives used by GradNorm / trainer monitoring
            if self.objectives is None:
105:
106:
                 self.objectives = [
107:
                     "segmentation",
                     "dice",
108:
109:
                     "sdf",
110:
                     "attributes",
111:
                     "polygon",
112:
                     "voxel",
                     "topology",
113:
114:
                     "latent_consistency",
115:
                     "graph",
116:
                 ]
117:
118:
119: @dataclass
120: class TrainingConfig:
121:
         """Training configuration with adaptive strategies"""
122:
        device: str = "cuda" if torch.cuda.is_available() else "cpu"
124:
        # Dynamic epoch limits (maxima; curriculum may switch earlier)
125:
        max_stagel_epochs: int = 40
126:
         max_stage2_epochs: int = 25
127:
        max_stage3_epochs: int = 60
128:
129:
         # Minimum epochs per stage (avoid switching too early)
130:
        min_stagel_epochs: int = 8
        min_stage2_epochs: int = 5
131:
132:
        min_stage3_epochs: int = 12
133:
134:
         # Learning rates (per stage)
135:
         stage1_lr: float = 1e-5 # was 3e-4
136:
        stage1_weight_decay: float = 1e-5
137:
138:
        stage2_lr: float = 5e-6 # was 1e-4
139:
        stage2_weight_decay: float = 1e-5
140:
141:
        stage3_lr: float = 1e-6 # was 5e-5
142:
         stage3_weight_decay: float = 1e-5
143:
144:
         # Advanced training techniques
```

```
145:
        use_mixed_precision: bool = True
        use_cosine_restarts: bool = True
146:
147:
        warmup_epochs: int = 5
148:
        grad_clip_norm: float = 0.5
149:
150:
         # Gradient monitoring for dynamic weighting
151:
         track_gradient_norms: bool = True
152:
        gradient_norm_window: int = 10  # rolling window for gradient tracking
153:
154:
         # Checkpointing
155:
         checkpoint_freq: int = 1
156:
157:
         # Curriculum configuration
158:
        curriculum: CurriculumConfig = None
159:
160:
        def __post_init__(self):
161:
            if self.curriculum is None:
162:
                 self.curriculum = CurriculumConfig()
163:
164:
165: @dataclass
166: class LossConfig:
167:
        """Loss function weights (will be dynamically adjusted during training)"""
168:
         # Base weights (starting values)
169:
         seg_weight: float = 1.0
170:
        dice_weight: float = 1.0
171:
        sdf_weight: float = 0.5
172:
        attr_weight: float = 1.0
173:
        polygon_weight: float = 1.0
174:
        voxel_weight: float = 1.0
175:
        topology_weight: float = 0.1 # start low, ramp up
176:
177:
         # New loss components
        latent_consistency_weight: float = 0.5
178:
        graph_constraint_weight: float = 0.3
179:
180:
181:
         # Dynamic weighting parameters
182:
        enable_dynamic_weighting: bool = True
183:
        weight_update_freq: int = 10
        weight_momentum: float = 0.9
184:
185:
186:
187: @dataclass
188: class InferenceConfig:
189:
        """Inference configuration"""
190:
        model_path: str = "final_model.pth"
        test_images_dir: str = "./data/test_images"
191:
192:
       output_dir: str = "./outputs"
193:
        export_intermediate: bool = True
194:
        polygon_threshold: float = 0.5
195:
197: # Curriculum stage transition logic
198: class StageTransitionCriteria:
199:
         """Defines criteria for automatic stage transitions"""
200:
201:
         @staticmethod
202:
         def should_transition_from_stagel(train_losses, val_losses, config: CurriculumConfig) -> bool:
203:
             """Check if should transition from Stage 1 to Stage 2""
204:
             if len(val_losses) < config.plateau_detection_window:</pre>
205:
                 return False
206:
207:
             # Check for plateau in segmentation + dice losses
208:
             recent_losses = val_losses[-config.plateau_detection_window:]
             if len(recent_losses) < 2:</pre>
209:
210:
                 return False
211:
212:
             # Calculate improvement rate
213:
             old_avg = sum(recent_losses[:len(recent_losses)//2]) / (len(recent_losses)//2)
214:
             new_avg = sum(recent_losses[len(recent_losses)//2:]) / (len(recent_losses) -
                     len(recent_losses)//2)
215:
216:
             improvement_rate = (old_avg - new_avg) / (old_avg + 1e-8)
```

```
217:
             return improvement_rate < config.min_improvement_threshold</pre>
218:
219:
         @staticmethod
220:
        def should_transition_from_stage2(polygon_losses, config: CurriculumConfig) -> bool:
221:
             """Check if should transition from Stage 2 to Stage 3"""
222:
             if len(polygon_losses) < config.plateau_detection_window:</pre>
223:
                 return False
224:
225:
             # Check polygon loss plateau
226:
             recent_losses = polygon_losses[-config.plateau_detection_window:]
             if len(recent_losses) < 2:</pre>
227:
                return False
228:
229:
             old_avg = sum(recent_losses[:len(recent_losses)//2]) / (len(recent_losses)//2)
230:
231:
            new_avg = sum(recent_losses[len(recent_losses)//2:]) / (len(recent_losses) -
                      len(recent_losses)//2)
232:
233:
             improvement_rate = (old_avg - new_avg) / (old_avg + 1e-8)
234:
             return improvement_rate < config.min_improvement_threshold</pre>
236:
237: # Default configurations (import these in your trainer)
238: DEFAULT_DATA_CONFIG = DataConfig()
239: DEFAULT_MODEL_CONFIG = ModelConfig()
240: DEFAULT_TRAINING_CONFIG = TrainingConfig()
241: DEFAULT_LOSS_CONFIG = LossConfig()
242: DEFAULT_INFERENCE_CONFIG = InferenceConfig()
```

■ File: dataset.py

```
______
 2: Dataset classes for the Neural-Geometric 3D Model Generator
 3: Enhanced with in-memory caching for faster training
 4: """
 5:
 6: import cv2
 7: import json
 8: import numpy as np
 9: import torch
10: from torch.utils.data import Dataset
11: from pathlib import Path
12: from typing import Dict, List, Tuple, Optional, Union
13: import time
14:
15: from config import DEFAULT_DATA_CONFIG
16:
17:
 18: class AdvancedFloorPlanDataset(Dataset):
19:
      Research-grade dataset with complete ground truth:
 21:
        - Floorplan image + segmentation mask
 22:
        - Attribute dictionary (geometric parameters)
 23:
        - Ground-truth mesh + voxelized occupancy
 24:
        - Polygon outlines for vectorization supervision
 25:
 26:
        Enhanced with optional in-memory caching for performance
 27:
 28:
 29:
       def __init__(
 30:
           self,
 31:
            data_dir: str = None,
 32:
            split: str = "train",
           image_size: Tuple[int, int] = None,
33:
 34:
            voxel_size: int = None,
35:
           augment: bool = None,
 36:
           config=None,
37:
            # Use config if provided, otherwise defaults from config.py
38:
 39:
           if config is None:
               config = DEFAULT_DATA_CONFIG
 40:
 41:
```

```
self.data_dir = Path(data_dir or config.data_dir)
 42:
 43:
             self.split = split
 44:
             self.image_size = image_size or config.image_size
 45:
             self.voxel_size = voxel_size or config.voxel_size
 46:
             self.augment = (
 47:
                 augment if augment is not None else config.augment
 48:
             ) and split == "train"
 49:
 50:
             # Collect all samples that contain every required file
             self.samples = self._find_complete_samples()
 51:
             print(f"Found {len(self.samples)} complete samples for {split}")
 52:
 53:
 54:
             # NEW: In-memory caching for performance
 55:
             self.cache_in_memory = getattr(config, "cache_in_memory", False)
 56:
             self. cache = None
 57:
             if self.cache_in_memory and len(self.samples) > 0:
 58:
 59:
                 print(f"[DATA] Preloading {len(self.samples)} samples into RAM (cache_in_memory=True).")
 60:
                 print("[DATA]] This may take significant memory but will speed up training...")
 61:
 62:
                 # Estimate memory usage
 63:
                 estimated_mb = self._estimate_memory_usage()
 64:
                 print(f"[DATA] Estimated memory usage: {estimated_mb:.1f} MB")
 65:
                 start_time = time.time()
 66:
 67:
                 self._preload_cache()
 68:
                 load_time = time.time() - start_time
 69:
                 print(f"[DATA] Cache preloading completed in {load_time:.2f}s")
 70:
 71:
         def _estimate_memory_usage(self):
 72:
              ""Estimate memory usage for caching"""
 73:
             if not self.samples:
 74:
                 return 0.0
 75:
 76:
             H, W = self.image_size
 77:
             n_samples = len(self.samples)
 78:
 79:
             # Rough estimates in bytes
 :08
             image_bytes = H * W * 3 # RGB uint8
             mask_bytes = H * W # grayscale uint8
 81:
             voxel_bytes = self.voxel_size ** 3 * 4 # float32
 82:
 83:
             json_bytes = 1024 # rough estimate for params + polygons
 84:
             total_per_sample = image_bytes + mask_bytes + voxel_bytes + json_bytes
 85:
 86:
             total_mb = (total_per_sample * n_samples) / (1024 * 1024)
 87:
 88:
             return total mb
 89:
 90:
         def _preload_cache(self):
 91:
              """Preload all samples into memory"""
 92:
             self._cache = []
 93:
 94:
             for i, sample in enumerate(self.samples):
 95:
                 if i % 100 == 0:
 96:
                     print(f"[DATA] Loading sample {i+1}/{len(self.samples)}")
 97:
 98:
 99:
                     # Load image
100:
                     img = cv2.imread(str(sample["image"]))
101:
                     if img is None:
102:
                         print(f"Warning: Could not load image {sample['image']}")
103:
                         continue
                     img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
104:
105:
                     img = cv2.resize(img, self.image_size) # (W, H) format for cv2.resize
106:
107:
                     # Load mask
                     mask = cv2.imread(str(sample["mask"]), cv2.IMREAD_GRAYSCALE)
108:
109:
                     if mask is None:
110:
                         print(f"Warning: Could not load mask {sample['mask']}")
111:
112:
                     mask = cv2.resize(mask, self.image_size, interpolation=cv2.INTER_NEAREST)
113:
114:
                     # Load voxel data
```

```
115:
                     t.rv:
116:
                        voxel_data = np.load(sample["voxel"])
117:
                        vox = voxel_data["voxels"] # Keep as numpy array
118:
                     except Exception as e:
                        print(f"Warning: Could not load voxel data {sample['voxel']}: {e}")
119:
120:
                         # Create dummy voxel data
121:
                        vox = np.zeros((self.voxel_size, self.voxel_size, self.voxel_size),
                                 dtype=np.float32)
122:
                     # Load parameters
123:
124:
                     try:
                        with open(sample["params"], "r") as f:
125:
126:
                            params = json.load(f)
127:
                     except Exception as e:
128:
                        print(f"Warning: Could not load params {sample['params']}: {e}")
129:
                        params = self._get_default_attributes()
130:
131:
                     # Load polygons
132:
                     try:
133:
                        with open(sample["polygon"], "r") as f:
134:
                            polygons = json.load(f)
135:
                     except Exception as e:
136:
                        print(f"Warning: Could not load polygons {sample['polygon']}: {e}")
137:
                        polygons = {"walls": []}
138:
139:
                     self._cache.append({
140:
                        "image": img,
                         "mask": mask,
141:
                        "vox": vox,
142:
143:
                        "params": params,
144:
                         "polygons": polygons,
145:
                         "sample_id": sample["image"].parent.name,
                    })
146:
147:
                except Exception as e:
148:
149:
                    print(f"Error loading sample {i}: {e}")
150:
                     continue
151:
152:
       def _get_default_attributes(self):
             """Return default attributes for missing param files"""
153:
154:
            return {
155:
                "wall_height": 2.6,
                "wall_thickness": 0.15,
157:
                "window_base_height": 0.7,
158:
                 "window_height": 0.95,
159:
                 "door_height": 2.6,
160:
                 "pixel_scale": 0.02,
161:
             }
162:
         # ------
163:
164:
        def _find_complete_samples(self):
165:
             """Locate samples that contain all the expected files."""
166:
            samples = []
167:
            split_dir = self.data_dir / self.split
168:
169:
             if not split_dir.exists():
170:
                print(f"Warning: Split directory {split_dir} does not exist")
171:
                return samples
172:
173:
             for sample_dir in split_dir.iterdir():
174:
                if not sample_dir.is_dir():
175:
                    continue
176:
177:
                required_files = {
                    "image": sample_dir / "image.png",
178:
179:
                     "mask": sample_dir / "mask.png",
                     "params": sample_dir / "params.json",
180:
181:
                     "mesh": sample_dir / "model.obj",
                     "voxel": sample_dir / "voxel_GT.npz",
182:
183:
                     "polygon": sample_dir / "polygon.json",
184:
185:
186:
                if all(f.exists() for f in required_files.values()):
```

```
187:
                    samples.append(required files)
188:
189:
            return samples
190:
         # -----
191:
192:
        def __len__(self):
193:
            return len(self._cache) if self._cache is not None else len(self.samples)
194:
195:
        def __getitem__(self, idx):
196:
             # Use cached data if available
197:
198:
            if self._cache is not None:
199:
                cached_sample = self._cache[idx]
200:
                image = cached_sample['image']
201:
                mask = cached sample['mask']
202:
                vox = cached_sample['vox']
203:
                attributes = cached_sample['params']
204:
                polygons_gt = cached_sample['polygons']
                sample_id = cached_sample['sample_id']
205:
206:
            else:
207:
                # Fallback: load from disk on-the-fly
208:
                sample = self.samples[idx]
209:
                # Load image and mask
210:
                image = cv2.imread(str(sample["image"]))
211:
212:
                image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
213:
                image = cv2.resize(image, self.image_size)
214:
215:
                mask = cv2.imread(str(sample["mask"]), cv2.IMREAD_GRAYSCALE)
216:
                mask = cv2.resize(mask, self.image_size, interpolation=cv2.INTER_NEAREST)
217:
218:
                # Load attributes
                with open(sample["params"], "r") as f:
219:
220:
                    attributes = json.load(f)
221:
222:
                # Load voxel ground truth
223:
                voxel_data = np.load(sample["voxel"])
224:
                vox = voxel_data["voxels"]
225:
226:
                # Load polygon ground truth
                with open(sample["polygon"], "r") as f:
227:
                    polygons_gt = json.load(f)
228:
229:
230:
                sample_id = sample["image"].parent.name
231:
232:
             # Normalize image to [0,1]
            image = image.astype(np.float32) / 255.0
233:
234:
235:
            # Clean mask (remove class 5 if present)
            mask[mask == 5] = 0
236:
237:
238:
            # Convert to tensors
239:
            image_tensor = torch.from_numpy(image).float().permute(2, 0, 1)
240:
            mask tensor = torch.from numpy(mask).long()
241:
            voxels_tensor = torch.from_numpy(vox.astype(np.float32))
242:
            attr_tensor = self._process_attributes(attributes)
243:
244:
            polygon_tensor = self._process_polygons(polygons_gt)
245:
246:
             # Apply augmentation if enabled
247:
            if self.augment:
248:
                image_tensor, mask_tensor = self._augment(image_tensor, mask_tensor)
249:
250:
             # Add validation before returning
251:
            self._validate_sample_data(idx, image_tensor, mask_tensor, attr_tensor, voxels_tensor,
                     polygon_tensor)
252:
253:
            return {
                "image": image_tensor,
254:
255:
                "mask": mask_tensor,
256:
                "attributes": attr_tensor,
                 "voxels_gt": voxels_tensor,
257:
258:
                 "polygons_gt": polygon_tensor,
```

```
259:
                "sample_id": sample_id,
260:
            }
261:
        # -----
262:
        def _validate_sample_data(self, idx, image, mask, attributes, voxels, polygons):
263:
264:
            """Validate sample data for NaN/Inf values"""
265:
            tensors_to_check = [
266:
               ("image", image),
                ("mask", mask),
267:
268:
                ("attributes", attributes),
269:
                ("voxels", voxels),
270:
                ("polygons", polygons["polygons"])
271:
272:
273:
            corrupted_data = False
274:
275:
            for name, tensor in tensors_to_check:
276:
                if torch.isnan(tensor).any():
277:
                    print(f"ERROR: {name} contains NaN values at sample {idx}")
278:
                    corrupted_data = True
279:
                if torch.isinf(tensor).any():
280:
                    print(f"ERROR: {name} contains Inf values at sample {idx}")
281:
                    corrupted_data = True
282:
283:
            if corrupted data:
284:
                print(f"WARNING: Corrupted data detected in sample {idx}, replacing with safe fallback
                         values")
285:
286:
                # Replace corrupted tensors with safe fallback values
287:
                for name, tensor in tensors_to_check:
288:
                    if torch.isnan(tensor).any() or torch.isinf(tensor).any():
289:
                        if name == "image":
290:
                            # Replace with zeros (black image)
291:
                            image.data = torch.zeros_like(image)
                        elif name == "mask":
292:
293:
                            # Replace with zeros (background class)
294:
                            mask.data = torch.zeros_like(mask).long()
295:
                        elif name == "attributes":
296:
                            # Replace with reasonable default values (0.5 normalized)
297:
                            attributes.data = torch.ones_like(attributes) * 0.5
298:
                        elif name == "voxels":
299:
                            # Replace with empty voxel grid
300:
                            voxels.data = torch.zeros_like(voxels)
                        elif name == "polygons":
301:
302:
                            # Replace polygons with zeros
303:
                            polygons["polygons"].data = torch.zeros_like(polygons["polygons"])
304:
305:
        # -----
306:
        def _process_attributes(self, attributes):
307:
             ""Convert attribute dictionary to a normalized tensor."""
            \# Normalize common architectural parameters into [0,1]
308:
309:
            attr_list = [
310:
                attributes.get("wall_height", 2.6) / 5.0,
311:
                attributes.get("wall_thickness", 0.15) / 0.5,
312:
                attributes.get("window_base_height", 0.7) / 3.0,
                attributes.get("window_height", 0.95) / 2.0,
313:
                attributes.get("door_height", 2.6) / 5.0,
314:
315:
                attributes.get("pixel_scale", 0.01) / 0.02,
316:
            ]
317:
318:
            # Ensure no NaN/Inf values in attribute processing
319:
            safe_attr_list = []
320:
            for val in attr_list:
321:
                if np.isnan(val) or np.isinf(val):
                    safe_attr_list.append(0.5) # Default normalized value
322:
323:
                    safe\_attr\_list.append(max(0.0, \, min(1.0, \, val))) \quad \# \ Clamp \ to \ [0,1]
324:
325:
326:
            return torch.tensor(safe_attr_list, dtype=torch.float32)
327:
328:
329:
        def _process_polygons(self, polygons_gt):
330:
             ""Convert polygon ground truth into a fixed tensor representation.
```

```
331:
             Handles both formats:
332:
             1. Nested dict: { "walls": [...], "doors": [...], ... }
333:
             2. Flat list: [ {"type": "wall", "points": [...]}, ... ]
334:
             max_polygons = 30  # number of polygons per sample
335:
336:
             max_points = 100
                                  # max points per polygon
337:
338:
             processed = torch.zeros(max_polygons, max_points, 2)
339:
             valid_mask = torch.zeros(max_polygons, dtype=torch.bool)
340:
341:
             poly_idx = 0
342:
343:
             try:
344:
                 # --- Case 1: dict format ---
345:
                 if isinstance(polygons_gt, dict):
346:
                     for class_name, polygon_list in polygons_gt.items():
347:
                         if not isinstance(polygon_list, list):
348:
                              continue
349:
                         for polygon in polygon_list:
350:
                             if poly_idx >= max_polygons:
351:
                                 break
352:
                              if "points" not in polygon:
353:
                                  continue
354:
355:
                             points = np.array(polygon["points"])
356:
                              if len(points) > max_points:
357:
                                  # Subsample evenly if too many points
358:
                                  indices = np.linspace(0, len(points) - 1, max_points, dtype=int)
359:
                                  points = points[indices]
360:
361:
                              # Check for NaN/Inf in points
362:
                              if np.any(np.isnan(points)) or np.any(np.isinf(points)):
363:
                                  print(f"Warning: Invalid polygon points detected, skipping polygon")
364:
                                  continue
365:
366:
                              # Normalize to [0,1] relative to image size
367:
                             points = points / np.array(self.image_size)
368:
                              # Clamp to valid range
369:
                             points = np.clip(points, 0.0, 1.0)
370:
371:
                              processed[poly_idx, : len(points)] = torch.from_numpy(points).float()
372:
                             valid_mask[poly_idx] = True
373:
                             poly_idx += 1
374:
375:
                 # --- Case 2: list format ---
376:
                 elif isinstance(polygons_gt, list):
                     for polygon in polygons_gt:
377:
378:
                         if poly_idx >= max_polygons:
379:
                             break
380:
                         if "points" not in polygon:
381:
                             continue
382:
383:
                         points = np.array(polygon["points"])
384:
                         if len(points) > max_points:
385:
                              indices = np.linspace(0, len(points) - 1, max_points, dtype=int)
                             points = points[indices]
386:
387:
388:
                         # Check for NaN/Inf in points
389:
                         if np.any(np.isnan(points)) or np.any(np.isinf(points)):
                             \verb|print(f"Warning: Invalid polygon points detected, skipping polygon")|\\
390:
391:
                             continue
392:
393:
                         points = points / np.array(self.image_size)
394:
                         points = np.clip(points, 0.0, 1.0)
395:
396:
                         processed[poly_idx, : len(points)] = torch.from_numpy(points).float()
397:
                         valid_mask[poly_idx] = True
398:
                         poly_idx += 1
399:
400:
             except Exception as e:
401:
                 print(f"Warning: Error processing polygons: {e}")
402:
                 # Return safe empty polygon data
403:
                 processed = torch.zeros(max_polygons, max_points, 2)
```

```
404:
               valid_mask = torch.zeros(max_polygons, dtype=torch.bool)
405:
406:
            return {"polygons": processed, "valid_mask": valid_mask}
407:
        # -----
408:
409:
        def _augment(self, image, mask):
410:
            """Enhanced data augmentation with rotations, flips, and intensity changes."""
411:
            # Random rotation (multiples of 90° only for architectural data)
412:
            if torch.rand(1) < 0.5:
               k = torch.randint(1, 4, (1,)).item()
413:
414:
               image = torch.rot90(image, k, dims=[1, 2])
415:
               mask = torch.rot90(mask, k, dims=[0, 1])
416:
417:
            # Random horizontal flip
418:
            if torch.rand(1) < 0.5:
419:
               image = torch.flip(image, dims=[2])
420:
               mask = torch.flip(mask, dims=[1])
421:
            # Random vertical flip
422:
423:
            if torch.rand(1) < 0.5:</pre>
424:
               image = torch.flip(image, dims=[1])
425:
               mask = torch.flip(mask, dims=[0])
426:
427:
            # Slight brightness/contrast adjustment with safety checks
            if torch.rand(1) < 0.3:
428:
               brightness = torch.rand(1) * 0.2 - 0.1 # ±0.1
429:
430:
               contrast = torch.rand(1) * 0.2 + 0.9
                                                    # 0.9-1.1
               image = torch.clamp(image * contrast + brightness, 0, 1)
431:
432:
433:
               # Additional safety check for augmented image
434:
               if torch.isnan(image).any() or torch.isinf(image).any():
435:
                   print("Warning: Augmentation produced invalid values, reverting to original")
436:
                   # Revert to safe values
437:
                   image = torch.clamp(image, 0, 1)
                   image = torch.where(torch.isnan(image) | torch.isinf(image),
438:
439:
                                   torch.zeros_like(image), image)
440:
441:
           return image, mask
442:
443:
        # ------
444:
        def get_cache_info(self):
            """Return information about caching status"""
445:
446:
            return {
               "cache_enabled": self.cache_in_memory,
447:
448:
               "cache_loaded": self._cache is not None,
449:
               "cached_samples": len(self._cache) if self._cache else 0,
450:
               "total_samples": len(self.samples),
451:
               "estimated_memory_mb": self._estimate_memory_usage() if self.cache_in_memory else 0
452:
            }
453:
454:
        def disable cache(self):
455:
            """Disable caching and free memory"""
456:
            if self._cache is not None:
457:
               print(f"[DATA] Disabling cache and freeing memory for {len(self._cache)} samples")
458:
               self._cache = None
459:
               self.cache_in_memory = False
460:
461:
        def enable_cache(self):
462:
            """Enable caching if not already enabled"""
463:
            if not self.cache_in_memory and self.samples:
464:
               self.cache_in_memory = True
465:
               print("[DATA] Enabling cache...")
466:
               self._preload_cache()
467:
468:
470: # Synthetic sample generator for testing without dataset
471: # -----
472: def create_synthetic_data_sample():
473:
       """Generate a synthetic floorplan with attributes, voxels, and polygons."""
474:
       image = np.ones((256, 256, 3), dtype=np.uint8) * 255
475:
       mask = np.zeros((256, 256), dtype=np.uint8)
476:
```

```
477:
         # Simple square room
        room_points = np.array([[50, 50], [200, 50], [200, 200], [50, 200]])
478:
479:
         cv2.fillPoly(mask, [room_points], 1) # Room = class 1
480:
        cv2.polylines(image, [room_points], True, (0, 0, 0), 3)
481:
482:
        # Add door
483:
        cv2.rectangle(mask, (90, 50), (110, 70), 2) # Door = class 2
484:
        cv2.rectangle(image, (90, 50), (110, 70), (255, 0, 0), -1)
485:
         # Attributes
486:
487:
        attributes = {
             "wall_height": 2.6,
488:
489:
             "wall_thickness": 0.15,
490:
            "window_base_height": 0.7,
491:
            "window height": 0.95,
492:
            "door_height": 2.6,
            "pixel_scale": 0.02,
493:
494:
        }
495:
496:
        # Simple voxel GT
497:
        voxels = np.zeros((64, 64, 64), dtype=bool)
498:
        voxels[:20, 10:50, 10:50] = True
499:
500:
        # Polygon GT
        polygons = {"walls": [{"points": room_points.tolist()}]}
501:
502:
503:
        return image, mask, attributes, voxels, polygons
504:
506: class SyntheticFloorPlanDataset(Dataset):
507:
508:
         Synthetic dataset for testing and development when real data is not available
509:
510:
511:
        def __init__(self, num_samples=1000, image_size=(256, 256), voxel_size=64):
512:
             self.num_samples = num_samples
513:
             self.image_size = image_size
514:
            self.voxel_size = voxel_size
515:
516:
        def __len__(self):
517:
             return self.num_samples
518:
519:
       def __getitem__(self, idx):
520:
             \# Generate deterministic synthetic data based on index
521:
            np.random.seed(idx)
522:
             torch.manual_seed(idx)
523:
524:
            image, mask, attributes, voxels, polygons_gt = create_synthetic_data_sample()
525:
526:
             # Convert to tensors
527:
            image_tensor = torch.from_numpy(image.astype(np.float32) / 255.0).permute(2, 0, 1)
528:
            mask_tensor = torch.from_numpy(mask).long()
529:
            voxels_tensor = torch.from_numpy(voxels.astype(np.float32))
530:
531:
             \ensuremath{\sharp} Process attributes and polygons using same methods as main dataset
532:
             dataset = AdvancedFloorPlanDataset.__new__(AdvancedFloorPlanDataset)
             dataset.image_size = self.image_size
533:
534:
535:
            attr_tensor = dataset._process_attributes(attributes)
536:
            polygon_tensor = dataset._process_polygons(polygons_gt)
537:
538:
             return {
539:
                 "image": image_tensor,
540:
                 "mask": mask_tensor,
                 "attributes": attr_tensor,
541:
542:
                 "voxels_gt": voxels_tensor,
                 "polygons_gt": polygon_tensor,
543:
544:
                 "sample_id": f"synthetic_{idx:06d}",
             }
545:
```

■ File: evaluation\metrics.py

```
-----
 1: """
 2: Evaluation metrics and utilities for the Neural-Geometric 3D Model Generator
 3: """
 4:
 5: import torch
 6: import numpy as np
 7: from torch.utils.data import DataLoader
 9: from models.model import NeuralGeometric3DGenerator
10: from dataset import AdvancedFloorPlanDataset
11:
12:
13: def compute_iou(pred, target):
14:
        """Compute IoU for segmentation"""
15:
       intersection = (pred & target).float().sum()
16:
        union = (pred | target).float().sum()
17:
       return (intersection / (union + 1e-6)).item()
18:
19:
20: def compute_3d_iou(pred, target):
21:
        """Compute 3D IoU for voxel grids"""
        pred_bool = pred.bool()
22:
       target_bool = target.bool()
23:
24:
25:
        intersection = (pred_bool & target_bool).float().sum()
26:
        union = (pred_bool | target_bool).float().sum()
27:
28:
       return (intersection / (union + 1e-6)).item()
29:
30:
31: def compute_polygon_metrics(pred_polygons, gt_polygons, validity_pred, validity_gt):
        """Compute metrics for polygon prediction"""
        # Chamfer distance between polygon sets
33:
34:
        valid_pred = pred_polygons[validity_pred > 0.5]
35:
        valid_gt = gt_polygons[validity_gt]
36:
37:
        if len(valid_pred) == 0 or len(valid_gt) == 0:
            return {"chamfer_distance": float('inf'), "validity_accuracy": 0.0}
38:
39:
40:
        # Simplified chamfer distance computation
41:
        chamfer_dist = 0.0
42:
        for pred_poly in valid_pred:
43:
            min_dist = float('inf')
44:
            for gt_poly in valid_gt:
45:
                dist = torch.norm(pred_poly - gt_poly, dim=-1).min().item()
46:
                min_dist = min(min_dist, dist)
            chamfer_dist += min_dist
47:
48:
49:
       chamfer_dist /= len(valid_pred)
50:
51:
        # Validity accuracy
52:
       validity_acc = ((validity_pred > 0.5) == validity_gt).float().mean().item()
53:
54:
       return {
            "chamfer_distance": chamfer_dist,
55:
56:
            "validity_accuracy": validity_acc
57:
        }
58:
59:
60: class ModelEvaluator:
61:
        """Comprehensive model evaluation"""
62:
        def __init__(self, model_path, device="cuda"):
63:
64:
           self.device = device
65:
           self.model = NeuralGeometric3DGenerator()
66:
67:
            # Load model
68:
           checkpoint = torch.load(model_path, map_location=device)
69:
           self.model.load_state_dict(checkpoint["model_state_dict"])
70:
           self.model.to(device)
71:
            self.model.eval()
```

```
72:
 73:
             print(f"Loaded model from {model_path}")
 74:
 75:
         def evaluate_dataset(self, test_dataset):
             """Comprehensive evaluation on test dataset"""
 76:
 77:
             test_loader = DataLoader(test_dataset, batch_size=1, shuffle=False)
 78:
 79:
             # Metrics storage
 :08
             metrics = {
                 "segmentation": {"ious": [], "class_ious": []},
 81:
                 "attributes": {"maes": [], "mses": []},
 82:
                 "voxels": {"ious": [], "dice_scores": []},
 83:
 84:
                 "polygons": {"chamfer_distances": [], "validity_accs": []},
             }
 85:
 86:
 87:
             with torch.no_grad():
                 for batch_idx, batch in enumerate(test_loader):
 88:
 89:
                     batch = {k: v.to(self.device) if torch.is_tensor(v) else v
 90:
                             for k, v in batch.items()}
 91:
 92:
                     predictions = self.model(batch["image"])
 93:
 94:
                     # Evaluate segmentation
 95:
                     seg_metrics = self._evaluate_segmentation(
                         predictions["segmentation"], batch["mask"]
 96:
 97:
 98:
                     metrics["segmentation"]["ious"].append(seg_metrics["iou"])
 99:
                     metrics["segmentation"]["class_ious"].append(seg_metrics["class_ious"])
100:
101:
                     # Evaluate attributes
102:
                     attr_metrics = self._evaluate_attributes(
103:
                         predictions["attributes"], batch["attributes"]
104:
                     metrics["attributes"]["maes"].append(attr_metrics["mae"])
105:
106:
                     metrics["attributes"]["mses"].append(attr_metrics["mse"])
107:
108:
                     # Evaluate voxels
109:
                     voxel_metrics = self._evaluate_voxels(
110:
                         predictions["voxels_pred"], batch["voxels_gt"]
111:
112:
                     metrics["voxels"]["ious"].append(voxel_metrics["iou"])
113:
                     metrics["voxels"]["dice_scores"].append(voxel_metrics["dice"])
114:
115:
                     # Evaluate polygons
116:
                     poly_metrics = self._evaluate_polygons(
117:
                         predictions["polygons"],
                         predictions["polygon_validity"],
118:
119:
                         batch["polygons_gt"]
120:
121:
                     metrics["polygons"]["chamfer_distances"].append(poly_metrics["chamfer_distance"])
                     metrics["polygons"]["validity_accs"].append(poly_metrics["validity_accuracy"])
122:
123:
124:
                     if (batch_idx + 1) % 10 == 0:
125:
                         print(f"Evaluated {batch_idx + 1}/{len(test_loader)} samples")
126:
             return self._compute_summary_metrics(metrics)
127:
128:
129:
         def _evaluate_segmentation(self, pred_seg, target_mask):
130:
              """Evaluate segmentation performance""
131:
             seg_pred = torch.argmax(pred_seg, dim=1)
132:
133:
             # Overall IoU
134:
             overall_iou = compute_iou(seg_pred, target_mask)
135:
             # Per-class IoU
136:
137:
             num_classes = pred_seg.shape[1]
138:
             class_ious = []
139:
140:
             for c in range(num_classes):
141:
                 pred_c = (seg_pred == c)
142:
                 target_c = (target_mask == c)
143:
144:
                 if target_c.sum() > 0: # Only compute if class exists in ground truth
```

```
iou_c = compute_iou(pred_c, target_c)
145:
146:
                     class_ious.append(iou_c)
147:
148:
            return {
                "iou": overall_iou,
149:
150:
                 "class_ious": class_ious
151:
             }
152:
        def _evaluate_attributes(self, pred_attrs, target_attrs):
153:
              """Evaluate attribute prediction"""
154:
155:
            mae = torch.mean(torch.abs(pred_attrs - target_attrs)).item()
156:
            mse = torch.mean((pred_attrs - target_attrs) ** 2).item()
157:
            return {"mae": mae, "mse": mse}
158:
159:
160:
        def _evaluate_voxels(self, pred_voxels, target_voxels):
161:
              """Evaluate 3D voxel prediction"""
162:
             pred_binary = (torch.sigmoid(pred_voxels) > 0.5).float()
             target_float = target_voxels.float()
163:
164:
165:
             # 3D IoU
166:
             iou_3d = compute_3d_iou(pred_binary, target_float)
167:
168:
             # 3D Dice score
             intersection = (pred_binary * target_float).sum()
169:
             dice = (2 * intersection) / (pred_binary.sum() + target_float.sum() + 1e-6)
170:
171:
172:
            return {
                 "iou": iou_3d,
173:
174:
                 "dice": dice.item()
175:
             }
176:
177:
        def _evaluate_polygons(self, pred_polygons, pred_validity, gt_polygons):
178:
             """Evaluate polygon prediction""'
179:
            return compute_polygon_metrics(
180:
                pred_polygons[0],
181:
                 gt_polygons["polygons"][0],
182:
                 pred validity[0].
183:
                 gt_polygons["valid_mask"][0]
184:
185:
186:
         def _compute_summary_metrics(self, metrics):
187:
             """Compute summary statistics"""
188:
             summary = {}
189:
190:
             # Segmentation
191:
            summary["segmentation_mIoU"] = np.mean(metrics["segmentation"]["ious"])
192:
             summary["segmentation_std"] = np.std(metrics["segmentation"]["ious"])
193:
194:
             # Attributes
             summary["attribute_MAE"] = np.mean(metrics["attributes"]["maes"])
195:
196:
             summary["attribute_MAE_std"] = np.std(metrics["attributes"]["maes"])
197:
198:
             # Voxels
199:
             summary["voxel_mIoU"] = np.mean(metrics["voxels"]["ious"])
             summary["voxel_mIoU_std"] = np.std(metrics["voxels"]["ious"])
200:
             summary["voxel_dice"] = np.mean(metrics["voxels"]["dice_scores"])
201:
202:
203:
             # Polygons
             valid_chamfer = [d for d in metrics["polygons"]["chamfer_distances"] if d != float('inf')]
204:
             if valid_chamfer:
205:
206:
                 summary["polygon_chamfer"] = np.mean(valid_chamfer)
207:
                 summary["polygon_chamfer_std"] = np.std(valid_chamfer)
208:
             else:
                 summary["polygon_chamfer"] = float('inf')
209:
210:
                 summary["polygon_chamfer_std"] = 0.0
211:
             summary["polygon_validity_acc"] = np.mean(metrics["polygons"]["validity_accs"])
212:
213:
214:
             return summary
215:
216:
         def print_evaluation_results(self, summary):
217:
             """Print formatted evaluation results"""
```

```
print("=" * 60)
218:
219:
            print("COMPREHENSIVE EVALUATION RESULTS")
            print("=" * 60)
220:
221:
            print(f"Segmentation mIoU: {summary['segmentation_mIoU']:.4f} ±
222:
                      {summary['segmentation_std']:.4f}")
            print(f"Attribute MAE: {summary['attribute_MAE']:.4f} ± {summary['attribute_MAE_std']:.4f}")
223:
224:
            print(f"Voxel 3D mIoU: {summary['voxel_mIoU']:.4f} ± {summary['voxel_mIoU_std']:.4f}")
225:
            print(f"Voxel Dice Score: {summary['voxel_dice']:.4f}")
226:
227:
            if summary['polygon_chamfer'] != float('inf'):
                print(f"Polygon Chamfer Distance: {summary['polygon_chamfer']:.4f} ±
228:
                          {summary['polygon_chamfer_std']:.4f}")
229:
            else:
230:
                print("Polygon Chamfer Distance: No valid polygons")
231:
232:
            print(f"Polygon Validity Accuracy: {summary['polygon_validity_acc']:.4f}")
233:
            print("=" * 60)
234:
235:
236: def evaluate_model(model_path, data_dir="./data/floorplans"):
237:
         """Standalone evaluation function""
238:
         # Load test dataset
239:
        test_dataset = AdvancedFloorPlanDataset(data_dir, split="test")
240:
241:
        if len(test dataset) == 0:
242:
            print("No test samples found!")
            return None
243:
244:
245:
        # Create evaluator
        evaluator = ModelEvaluator(model_path)
246:
247:
248:
        # Run evaluation
249:
        summary = evaluator.evaluate_dataset(test_dataset)
250:
251:
         # Print results
252:
         evaluator.print_evaluation_results(summary)
253:
254:
        return summary
```

■ File: inference\engine.py

```
______
 2: Research-grade inference engine for 2D to 3D floorplan generation
 3: """
 4:
 5: import torch
 6: import cv2
 7: import numpy as np
 8: import json
 9: import trimesh
10: from pathlib import Path
11:
12: from models.model import NeuralGeometric3DGenerator
13: from config import DEFAULT_INFERENCE_CONFIG
14:
15:
16: class ResearchInferenceEngine:
17:
18:
       Complete inference system that converts 2D floorplans to 3D models
19:
       following the deterministic export pipeline
 20:
 21:
 22:
       def __init__(self, model_path=None, device="cuda", config=None):
23:
           if config is None:
 24:
               config = DEFAULT_INFERENCE_CONFIG
 25:
           self.device = device
 26:
 27:
           self.config = config
           self.model = NeuralGeometric3DGenerator()
 28:
 29:
```

```
30:
             # Load trained model
 31:
             model_path = model_path or config.model_path
 32:
             checkpoint = torch.load(model_path, map_location=device)
             self.model.load_state_dict(checkpoint["model_state_dict"])
 33:
 34:
             self.model.to(device)
 35:
             self.model.eval()
 36:
 37:
             print(f"Loaded trained model from {model_path}")
 38:
 39:
         def generate_3d_model(
 40:
             self,
 41:
             image_path: str,
 42:
             output_path: str,
 43:
             export_intermediate: bool = None
 44:
 45:
 46:
             Complete pipeline: Image -> Segmentation -> Polygons -> 3D Model
 47:
 48:
             export_intermediate = export_intermediate or self.config.export_intermediate
 49:
 50:
             # Load and preprocess image
 51:
             image = self._load_image(image_path)
 52:
 53:
             with torch.no_grad():
 54:
                 # Neural network inference
 55:
                 predictions = self.model(image)
 56:
 57:
                 # Extract predictions
 58:
                 segmentation = predictions["segmentation"]
 59:
                 attributes = predictions["attributes"]
 60:
                 polygons = predictions["polygons"]
 61:
                 validity = predictions["polygon_validity"]
 62:
 63:
                 print("Neural network inference complete")
 64:
 65:
                 # Convert to deterministic representations
 66:
                 mask_np = self._extract_mask(segmentation)
 67:
                 attributes_dict = self._extract_attributes(attributes)
 68:
                 polygons_list = self._extract_polygons(polygons, validity)
 69:
 70:
                 print(f"Extracted: {len(polygons_list)} valid polygons")
 71:
 72:
                 # Export intermediate results if requested
 73:
                 if export_intermediate:
 74:
                     self._export_intermediates(
 75:
                         mask_np, attributes_dict, polygons_list, Path(output_path).parent
 76:
 77:
 78:
                 # Generate 3D model using deterministic pipeline
 79:
                 success = self._generate_deterministic_3d(
                     mask_np, attributes_dict, polygons_list, output_path
 80:
 81:
 82:
 83:
                 return success
 84:
         def _load_image(self, image_path):
 85:
              """Load and preprocess input image"""
 86:
 87:
             image = cv2.imread(image_path)
 88:
             if image is None:
                 raise ValueError(f"Could not load image from {image_path}")
 89:
 90:
 91:
             image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
 92:
             image = cv2.resize(image, (256, 256))
 93:
             image = torch.from_numpy(image / 255.0).float()
 94:
             image = image.permute(2, 0, 1).unsqueeze(0)
 95:
             return image.to(self.device)
 96:
         def _extract_mask(self, segmentation):
 97:
              """Convert soft segmentation to hard mask"""
 98:
 99:
             seg_pred = torch.argmax(segmentation, dim=1)
100:
             mask_np = seg_pred.squeeze().cpu().numpy().astype(np.uint8)
101:
             return mask np
102:
```

```
def _extract_attributes(self, attributes):
103:
             """Convert normalized attributes back to physical values"""
104:
105:
             attr_np = attributes.squeeze().cpu().numpy()
106:
107:
             # Denormalize (reverse of normalization in dataset)
108:
             attributes_dict = {
                 "wall_height": float(attr_np[0] * 5.0),
109:
110:
                 "wall_thickness": float(attr_np[1] * 0.5),
                 "window_base_height": float(attr_np[2] * 3.0),
111:
                 "window_height": float(attr_np[3] * 2.0),
112:
                 "door_height": float(attr_np[4] * 5.0),
113:
                 "pixel_scale": float(attr_np[5] * 0.02),
114:
115:
             }
116:
117:
             return attributes dict
118:
119:
         def _extract_polygons(self, polygons, validity, threshold=None):
120:
              """Extract valid polygons from network predictions"""
             threshold = threshold or self.config.polygon_threshold
121:
122:
             batch_size, num_polys, num_points, _ = polygons.shape
123:
124:
             polygons_list = []
125:
126:
             for poly_idx in range(num_polys):
                 if validity[0, poly_idx] > threshold: # Only valid polygons
127:
128:
                     poly_points = polygons[0, poly_idx].cpu().numpy()
129:
130:
                     # Remove zero-padded points
131:
                     valid_points = poly_points[poly_points.sum(axis=1) > 0]
132:
                     if len(valid_points) >= 3: # Minimum for a polygon
133:
134:
                         # Convert to image coordinates (assuming 256x256 input)
135:
                         valid_points = valid_points * 256
136:
                         polygons_list.append(
137:
                             {
138:
                                  "points": valid_points.tolist(),
139:
                                  "class": "wall", # Simplified - in practice classify polygon type
140:
                             }
141:
                         )
142:
143:
             return polygons_list
144:
145:
         def _export_intermediates(self, mask, attributes, polygons, output_dir):
146:
             """Export intermediate results for debugging/analysis"""
147:
             output_dir = Path(output_dir)
148:
             output_dir.mkdir(exist_ok=True)
149:
150:
             # Export mask
             cv2.imwrite(str(output_dir / "predicted_mask.png"), mask * 50)
151:
152:
153:
             # Export attributes
             with open(output_dir / "predicted_attributes.json", "w") as f:
154:
155:
                 json.dump(attributes, f, indent=2)
156:
157:
             # Export polygons
             with open(output_dir / "predicted_polygons.json", "w") as f:
158:
159:
                 json.dump(polygons, f, indent=2)
160:
161:
             # Visualize polygons on mask
             vis_img = np.zeros((256, 256, 3), dtype=np.uint8)
162:
163:
             vis_img[:, :, 0] = mask * 50 # Background
164:
165:
             for poly in polygons:
166:
                 points = np.array(poly["points"], dtype=np.int32)
                 cv2.polylines(vis\_img, [points], True, (0, 255, 0), 2)
167:
168:
169:
             cv2.imwrite(str(output_dir / "polygon_visualization.png"), vis_img)
170:
171:
             print(f"Intermediate results exported to {output_dir}")
172:
173:
         def _generate_deterministic_3d(self, mask, attributes, polygons, output_path):
174:
              ""Generate 3D model using deterministic geometric operations"""
175:
             trv:
```

```
176:
                 # Initialize mesh components
177:
                 vertices = []
178:
                 faces = []
179:
                 vertex_count = 0
180:
181:
                 # Extract geometric parameters
182:
                 wall_height = attributes.get("wall_height", 2.6)
183:
                 wall_thickness = attributes.get("wall_thickness", 0.15)
                 pixel_scale = attributes.get("pixel_scale", 0.01)
184:
185:
186:
                     f "Generating 3D model with wall_height={wall_height:.2f}m,
187:
                              thickness={wall_thickness:.2f}m"
188:
189:
190:
                 # Process each polygon (walls, rooms, etc.)
191:
                 for poly_idx, polygon in enumerate(polygons):
192:
                     poly_vertices, poly_faces = self._extrude_polygon_3d(
                         polygon["points"],
193:
194:
                         wall_height,
195:
                         wall_thickness,
196:
                         pixel scale,
197:
                         vertex_count,
198:
                     )
199:
200:
                     vertices.extend(poly_vertices)
201:
                     faces.extend(poly_faces)
202:
                     vertex_count += len(poly_vertices)
203:
204:
                 # Add floor and ceiling
205:
                 floor_verts, floor_faces = self._generate_floor_ceiling(
206:
                     mask, pixel_scale, wall_height, vertex_count
207:
                 vertices.extend(floor_verts)
208:
209:
                 faces.extend(floor_faces)
210:
211:
                 if len(vertices) == 0:
212:
                     print("No geometry generated")
213:
                     return False
214:
215:
                 # Create mesh
216:
                 mesh = trimesh.Trimesh(vertices=np.array(vertices), faces=np.array(faces))
217:
218:
                 # Clean up mesh
219:
                 mesh.remove_duplicate_faces()
220:
                 mesh.remove_unreferenced_vertices()
                mesh.fix_normals()
221:
222:
223:
                 # Export
224:
                 mesh.export(output_path)
225:
                 print(f"3D model exported to {output_path}")
226:
                print(
227:
                     f"Mesh statistics: {len(mesh.vertices)} vertices, {len(mesh.faces)} faces"
228:
229:
                 return True
230:
231:
232:
             except Exception as e:
233:
                 print(f"Error generating 3D model: {str(e)}")
234:
                 return False
235:
236:
         def _extrude_polygon_3d(self, points, height, thickness, scale, vertex_offset):
237:
              ""Extrude a 2D polygon to create 3D wall geometry""
238:
             vertices = []
239:
             faces = []
240:
             # Convert points to 3D coordinates
241:
242:
             points_3d = []
243:
             for point in points:
                x = (point[0] - 128) * scale # Center and scale
244:
245:
                 z = (128 - point[1]) * scale # Flip Y and scale
246:
                points_3d.append([x, 0, z])
247:
```

```
# Create bottom vertices (y=0)
248:
249:
             bottom_outer = points_3d
250:
             bottom_inner = self._inset_polygon(points_3d, thickness)
251:
252:
             # Create top vertices (y=height)
253:
             top_outer = [[p[0], height, p[2]] for p in bottom_outer]
254:
             top_inner = [[p[0], height, p[2]] for p in bottom_inner]
255:
256:
             # Combine all vertices
             all_vertices = bottom_outer + bottom_inner + top_outer + top_inner
257:
258:
             vertices.extend(all_vertices)
259:
            n_points = len(points_3d)
261:
262:
             # Generate faces for walls
             for i in range(n_points):
263:
264:
                next_i = (i + 1) % n_points
265:
266:
                 # Outer wall faces
267:
                 v1 = vertex_offset + i # bottom outer
268:
                 v2 = vertex_offset + next_i # bottom outer next
269:
                 v3 = vertex_offset + 2 * n_points + next_i # top outer next
                 v4 = vertex_offset + 2 * n_points + i # top outer
270:
271:
272:
                faces.extend([[v1, v2, v3], [v1, v3, v4]])
273:
274:
                 # Inner wall faces (reverse winding)
275:
                 v1 = vertex_offset + n_points + i # bottom inner
276:
                v2 = vertex_offset + n_points + next_i # bottom inner next
277:
                 v3 = vertex\_offset + 3 * n\_points + next\_i # top inner next
278:
                 v4 = vertex_offset + 3 * n_points + i # top inner
279:
280:
                faces.extend([[v1, v3, v2], [v1, v4, v3]])
281:
282:
             # Top cap (between outer and inner)
283:
             for i in range(n_points):
                next_i = (i + 1) % n_points
284:
285:
286:
                 v1 = vertex_offset + 2 * n_points + i # top outer
                 v2 = vertex\_offset + 2 * n\_points + next\_i # top outer next
287:
                 v3 = vertex_offset + 3 * n_points + next_i # top inner next
288:
                 v4 = vertex_offset + 3 * n_points + i # top inner
289:
290:
291:
                 faces.extend([[v1, v2, v3], [v1, v3, v4]])
292:
293:
             # Bottom cap (between outer and inner)
294:
             for i in range(n_points):
295:
                next_i = (i + 1) % n_points
296:
297:
                 v1 = vertex_offset + i # bottom outer
                v2 = vertex_offset + next_i # bottom outer next
298:
                 v3 = vertex_offset + n_points + next_i # bottom inner next
300:
                 v4 = vertex_offset + n_points + i # bottom inner
301:
302:
                 faces.extend([[v1, v3, v2], [v1, v4, v3]])
303:
304:
             return vertices, faces
305:
306:
         def _inset_polygon(self, points, inset_distance):
307:
              """Create inset polygon for wall thickness"""
308:
             if len(points) < 3:
309:
                return points
310:
311:
             # Simple inset by moving each point inward along angle bisector
             inset_points = []
312:
313:
             n = len(points)
314:
315:
             for i in range(n):
316:
                 prev_i = (i - 1) % n
317:
                next_i = (i + 1) % n
318:
319:
                 p_prev = np.array(points[prev_i])
320:
                 p_curr = np.array(points[i])
```

```
321:
                 p_next = np.array(points[next_i])
322:
323:
                 # Vectors to adjacent points
324:
                 v1 = p_curr - p_prev
325:
                 v2 = p_next - p_curr
326:
327:
                 # Normalize vectors (in XZ plane, ignore Y)
328:
                 v1_norm = np.array([v1[0], 0, v1[2]])
329:
                 v2\_norm = np.array([v2[0], 0, v2[2]])
330:
                v1_len = np.linalg.norm(v1_norm)
331:
332:
                v2_len = np.linalg.norm(v2_norm)
333:
334:
                if v1_len > 1e-6:
335:
                     v1 norm /= v1 len
336:
                if v2_len > 1e-6:
337:
                     v2_norm /= v2_len
338:
                 # Angle bisector
339:
340:
                 bisector = v1_norm + v2_norm
341:
                bisector_len = np.linalg.norm(bisector)
342:
343:
                 if bisector_len > 1e-6:
344:
                     bisector /= bisector_len
345:
346:
                     # Move point inward
347:
                     inset_point = p_curr - bisector * inset_distance
                     inset_points.append([inset_point[0], inset_point[1], inset_point[2]])
348:
349:
350:
                     inset_points.append(points[i])
351:
352:
             return inset_points
353:
         def _generate_floor_ceiling(self, mask, scale, wall_height, vertex_offset):
354:
355:
              """Generate floor and ceiling geometry from segmentation mask"""
356:
             vertices = []
357:
             faces = []
358:
359:
             # Find floor regions (assuming class 0 = floor/room)
360:
            floor_mask = (mask == 0).astype(np.uint8)
361:
362:
             # Find contours
363:
            contours, _ = cv2.findContours(
                floor_mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE
364:
365:
             )
366:
367:
             for contour in contours:
368:
                if cv2.contourArea(contour) < 100: # Skip small regions
369:
                     continue
370:
371:
                 # Simplify contour
                 epsilon = 0.02 * cv2.arcLength(contour, True)
372:
373:
                 approx = cv2.approxPolyDP(contour, epsilon, True)
374:
375:
                 if len(approx) < 3:
376:
                     continue
377:
378:
                 # Convert to 3D coordinates
379:
                 floor_points = []
380:
                 for point in approx.reshape(-1, 2):
381:
                     x = (point[0] - 128) * scale
382:
                     z = (128 - point[1]) * scale
383:
                     floor_points.append([x, 0, z]) # Floor at y=0
384:
                 ceiling_points = []
385:
386:
                 for point in approx.reshape(-1, 2):
387:
                     x = (point[0] - 128) * scale
388:
                     z = (128 - point[1]) * scale
                     ceiling_points.append([x, wall_height, z]) # Ceiling at y=wall_height
389:
390:
391:
                 # Add vertices
392:
                 n_points = len(floor_points)
393:
                 vertices.extend(floor_points)
```

```
394:
                 vertices.extend(ceiling_points)
395:
396:
                 # Triangulate floor
397:
                 if n_points >= 3:
                     for i in range(1, n_points - 1):
398:
399:
                         faces.append(
400:
                              [vertex_offset, vertex_offset + i + 1, vertex_offset + i]
401:
402:
403:
                     # Triangulate ceiling (reverse winding)
404:
                      for i in range(1, n_points - 1):
405:
                         faces.append(
406:
407:
                                  vertex_offset + n_points,
408:
                                  vertex_offset + n_points + i,
409:
                                  vertex_offset + n_points + i + 1,
410:
                             1
411:
412:
413:
                 vertex_offset += 2 * n_points
414:
415:
             return vertices, faces
416:
417:
         def process_batch(self, image_paths, output_dir):
418:
             """Process multiple images in batch"""
             output_dir = Path(output_dir)
419:
420:
             output_dir.mkdir(exist_ok=True)
421:
422:
             results = []
423:
424:
             for img_path in image_paths:
425:
                 img_path = Path(img_path)
                 print(f"Processing: {img_path.name}")
426:
427:
                 output_path = output_dir / f"{img_path.stem}_model.obj"
428:
429:
430:
                 try:
431:
                     success = self.generate_3d_model(
432:
                         str(img_path), str(output_path), export_intermediate=True
433:
434:
435:
                     results.append({
436:
                         "input": str(img_path),
437:
                         "output": str(output_path),
438:
                          "success": success
439:
                     })
440:
441:
                     if success:
442:
                         print(f"? Generated: {output_path}")
443:
                      else:
                         print(f"? Failed: {img_path.name}")
444:
445:
446:
                 except Exception as e:
447:
                     print(f"? Error processing {img_path.name}: {str(e)}")
448:
                     results.append({
                         "input": str(img_path),
449:
450:
                         "output": str(output_path),
451:
                         "success": False,
452:
                          "error": str(e)
                     })
453:
454:
455:
             return results
```

■ File: models\dvx.py

1: """
2: Robust Differentiable Vectorization (DVX) module.
3:
4: Improvements vs naive DVX:
5: - Projects backbone feature maps to `feature_dim` if channels don't match via 1x1 conv.
6: - Multi-step iterative refinement (improves final polygon accuracy).

```
7: - Safe guards for shapes, device handling, and grid-sampling.
8: - Returns init_polygons, final polygons, per-step displacements, and validity scores.
9:
10: Usage:
11: - features: dict of feature maps (e.g. "p2", "p4"), each tensor (B, C, H, W).
12: - segmentation: (B, 1, H_img, W_img) or similar ? only used for optional initialization logic.
13: """
14:
15: from typing import Dict, Any, Optional, Tuple
16: import torch
17: import torch.nn as nn
18: import torch.nn.functional as F
19:
20:
21: class DifferentiableVectorization(nn.Module):
22:
      def __init__(
23:
           self.
24:
           max_polygons: int = 20,
           max_points: int = 50,
25:
26:
           feature_dim: int = 256,
27:
           displacement_scale: float = 0.12,
           num_refinement_steps: int = 3,
28:
29:
           align_corners: bool = False,
           padding_mode: str = "border", # options for grid_sample
30:
31:
           use_proj_conv: bool = True,
       ):
32:
33:
34:
           Args:
35:
               max_polygons: maximum polygons to predict per image
36:
                max_points: number of control points per polygon
37:
                feature_dim: number of channels the DVX expects (will project backbone features to this)
38:
                displacement_scale: multiplier for predicted displacement (tanh output)
39:
               num_refinement_steps: how many iterative refinement steps to apply (>=1)
40:
                align_corners: align_corners for F.grid_sample
41:
                padding_mode: padding_mode for F.grid_sample
42:
                use_proj_conv: whether to use 1x1 conv to project backbone features to feature_dim
                         (recommended)
43:
44:
           super().__init__()
45:
           assert max_points > 2, "max_points must be > 2"
46:
           assert num_refinement_steps >= 1
47:
           self.max_polygons = int(max_polygons)
49:
           self.max_points = int(max_points)
50:
           self.feature_dim = int(feature_dim)
51:
            self.displacement_scale = float(displacement_scale)
52:
           self.num_refinement_steps = int(num_refinement_steps)
53:
           self.align_corners = bool(align_corners)
54:
           self.padding_mode = padding_mode
55:
           self.use_proj_conv = bool(use_proj_conv)
56:
57:
            # init_net: from pooled p4 -> flattened -> produce normalized coords in [0,1]
58:
            # AdaptiveAvgPool2d(8) -> (B, C, 8, 8) -> flatten -> Linear(C*8*8 -> hidden)
59:
           hidden = max(512, feature_dim * 2)
60:
            self.init_pool = nn.AdaptiveAvgPool2d(8)
61:
            # we'll create a projector conv for p4/p2 channels if necessary at runtime
62:
63:
            # but also create an MLP init_net that assumes feature_dim channels after pooling
64:
            self.init_mlp = nn.Sequential(
65:
                nn.Flatten(),
66:
                nn.Linear(self.feature_dim * 8 * 8, hidden),
67:
                nn.ReLU(inplace=True),
68:
                nn.Linear(hidden, 1024),
69:
                nn.ReLU(inplace=True),
70:
                nn.Linear(1024, self.max_polygons * self.max_points * 2),
71:
                nn.Sigmoid(),
72:
           )
73:
74:
            # refinement network: maps (feature_dim + 2) -> displacement in [-1,1]
75:
           self.refine net = nn.Sequential(
                nn.Linear(self.feature_dim + 2, 256),
76:
77:
                nn.ReLU(inplace=True),
78:
                nn.Linear(256, 128),
```

```
79:
                 nn.ReLU(inplace=True).
                 nn.Linear(128, 2),
 80:
 81:
                 nn.Tanh(),
 82:
             )
 83:
 84:
             # validity net (reads flattened coords only)
 85:
             self.validity_net = nn.Sequential(
                 nn.Linear(self.max_points * 2, 128),
 86:
 87:
                 nn.ReLU(inplace=True),
 88:
                 nn.Linear(128, 1),
 89:
                 nn.Sigmoid(),
 90:
             )
 91:
             # projector convs (create lazily when first seen a feature channel mismatch)
 92:
 93:
             # stored per-key: e.g., self._proj_convs['p2'] = nn.Conv2d(in_ch, feature_dim, 1)
 94:
             self._proj_convs = nn.ModuleDict()
 95:
             self._proj_created = set()
 96:
 97:
         def _ensure_projector(self, key: str, in_channels: int):
 98:
 99:
             Ensure a 1x1 conv exists that projects `in_channels` -> self.feature_dim for feature map
                       `key`.
100:
101:
             if not self.use_proj_conv:
102:
                 return None
             if key in self._proj_created:
103:
104:
                 return self._proj_convs[key]
105:
106:
             if in_channels != self.feature_dim:
107:
                 conv = nn.Conv2d(in_channels, self.feature_dim, kernel_size=1, stride=1, padding=0)
108:
                 # initialize conv: kaiming
                 nn.init.kaiming_normal_(conv.weight, a=0.2)
109:
110:
                 if conv.bias is not None:
111:
                     nn.init.zeros_(conv.bias)
                 self._proj_convs[key] = conv
112:
113:
             else:
114:
                 # identity mapping using 1x1 conv with weights = identity-like is tricky
115:
                 # Instead simply keep no conv; we'll pass feature as-is
116:
                 self._proj_convs[key] = nn.Identity()
117:
             self._proj_created.add(key)
118:
             return self._proj_convs[key]
119:
120:
         def _project_feature(self, key: str, feat: torch.Tensor) -> torch.Tensor:
121:
122:
             Project or verify feature map to have self.feature_dim channels.
123:
             If projector conv wasn't present and channels == feature_dim, returns feat unchanged.
124:
125:
             in_ch = feat.shape[1]
126:
             proj = self._ensure_projector(key, in_ch)
127:
             if proj is None:
128:
                 # projection not desired; assert channels match
129:
                 if in_ch != self.feature_dim:
130:
                     raise RuntimeError(
131:
                         f"Feature '{key}' channels ({in_ch}) != feature_dim ({self.feature_dim}) "
132:
                         "and projection disabled."
133:
                     )
134:
                 return feat
135:
             # if proj is Identity, apply it still (fast path)
136:
             return proj(feat)
137:
        def forward(
138:
139:
             self,
140:
             features: Dict[str, torch.Tensor],
141:
             segmentation: Optional[torch.Tensor] = None,
             return_all_steps: bool = False,
142:
143:
         ) -> Dict[str, Any]:
144:
145:
             features: dict with keys like "p2", "p4" containing tensors (B, C, H, W)
146:
             segmentation: optional (B, 1, H_img, W_img) or similar (not strictly required)
147:
             return_all_steps: if True returns per-step intermediate polygons & displacements
148:
149:
             # pick features for init and refinement
150:
             p4 = features.get("p4", None)
```

```
151:
            p2 = features.get("p2", None)
152:
153:
            if p4 is None and p2 is None:
                 raise ValueError("At least one of 'p4' or 'p2' must be present in features.")
154:
155:
156:
             # prefer p4 for init; fallback to p2 if not present
157:
             init_feat = p4 if p4 is not None else p2
158:
            refine_feat = p2 if p2 is not None else p4
159:
             B = init_feat.shape[0]
160:
161:
162:
             # Project features to feature_dim (if needed)
163:
             init_feat = self._project_feature("p4_init", init_feat)
164:
             refine_feat = self._project_feature("p2_refine", refine_feat)
165:
166:
             # -- Initialize polygons --
167:
             # Pool then MLP; ensure init_mlp expects feature_dim channels
168:
             pooled = self.init_pool(init_feat) # [B, C', 8, 8]
             if pooled.shape[1] != self.feature_dim:
169:
170:
                 # If the projector returned Identity but pooled channels mismatch, try to apply a
                         runtime projector
                 pooled = self._project_feature("p4_init_postpool", pooled)
171:
172:
173:
             init_logits = self.init_mlp(pooled) # [B, max_polygons * max_points * 2]
174:
             init_polygons = init_logits.view(B, self.max_polygons, self.max_points, 2) # normalized
                      [0.1]
175:
             # Iterative refinement
176:
177:
             polygons = init_polygons.clone()
178:
             per_step_displacements = []
179:
             for step in range(self.num_refinement_steps):
180:
                 \# sample features at the polygon control-point locations
181:
                 displ = self._single_refine_step(polygons, refine_feat)
182:
                 per_step_displacements.append(displ)
183:
                 polygons = torch.clamp(polygons + displ, 0.0, 1.0)
184:
185:
             # final validity
186:
             validity = self._predict_validity(polygons)
187:
188:
             out: Dict[str, Any] = {
189:
                 "polygons": polygons, # [B, P, N, 2]
                 "validity": validity, # [B, P]
190:
191:
                 "init_polygons": init_polygons,
192:
                 "refinement_displacements": per_step_displacements, # list of [B, P, N, 2]
193:
             }
194:
195:
             if return_all_steps:
196:
                 out["all_step_polygons"] = [
197:
                     torch.clamp(init_polygons + sum(per_step_displacements[:i + 1]), 0.0, 1.0)
198:
                     for i in range(len(per_step_displacements))
                 1
199:
200:
201:
             return out
202:
203:
         def _single_refine_step(self, polygons: torch.Tensor, feature_map: torch.Tensor) ->
                 torch.Tensor:
204:
205:
            One refinement step: sample features at polygon points, predict displacement (scaled),
                     return displacement.
206:
             polygons: [B, P, N, 2] in [0,1]
207:
            feature_map: [B, C, H, W] with C == feature_dim (or projected)
208:
             returns displacement: [B, P, N, 2] in [-displacement_scale, displacement_scale]
209:
210:
             B, P, N, _ = polygons.shape
211:
             # flatten pts to sample
212:
            coords = polygons.view(B, -1, 2) # [B, P*N, 2], coords in [0,1]
213:
            grid = coords * 2.0 - 1.0 # to [-1,1]
214:
            # grid_sample expects (B, H_out, W_out, 2); use W_out=1
215:
            grid_sample = grid.view(B, -1, 1, 2)
216:
            sampled = F.grid_sample(
217:
                feature map,
218:
                 grid sample,
219:
                 mode="bilinear",
```

```
padding_mode=self.padding_mode,
220:
221:
                align_corners=self.align_corners,
222:
            ) # [B, C, P*N, 1]
            sampled = sampled.squeeze(-1).permute(0, 2, 1).contiguous() # [B, P*N, C]
223:
224:
225:
            # combine sampled features and coords (coords in [0,1])
226:
            input_feats = torch.cat([sampled, coords], dim=-1) \# [B, P*N, C+2]
227:
            # predict displacements in [-1,1] via tanh on last layer
228:
            disp = self.refine_net(input_feats) # [B, P*N, 2], values ~[-1,1]
229:
            disp = disp.view(B, P, N, 2)
            disp = disp * self.displacement_scale # scale
230:
231:
            return disp
232:
233:
       def _predict_validity(self, polygons: torch.Tensor) -> torch.Tensor:
234:
            B, P, N, _ = polygons.shape
235:
            if N != self.max_points:
                # If someone truncated or padded points, adapt: flatten to last dim whatever it is
236:
237:
                poly_flat = polygons.view(B * P, -1)
238:
            else:
                poly_flat = polygons.view(B * P, -1)
240:
            validity = self.validity_net(poly_flat) # [B*P, 1]
241:
            validity = validity.view(B, P)
242:
            return validity
243:
244:
245: # ----- quick unit test / smoke test -----
246: def _smoke_test():
247:
        torch.manual_seed(0)
        B = 2
249:
       C1 = 384 # different from feature_dim to test projector conv
250:
        C2 = 128
251:
        H2, W2 = 64, 64
252:
        H4, W4 = 16, 16
253:
254:
        # create dummy backbone features with different channels
        p2 = torch.randn(B, C1, H2, W2)
256:
        p4 = torch.randn(B, C2, H4, W4)
        seg = torch.rand(B, 1, H2 * 4, W2 * 4) # just a placeholder
257:
258:
259:
       dvx = DifferentiableVectorization(
260:
            max_polygons=4,
261:
            max_points=16,
262:
           feature_dim=256,
           displacement_scale=0.08,
263:
           num_refinement_steps=3,
264:
265:
            align_corners=False,
           padding_mode="border",
266:
267:
            use_proj_conv=True,
268:
        )
269:
270:
        # ensure module moves projector convs to device when dvx.to(device) called
271:
        dvx = dvx.eval() # inference mode ok
272:
        # Forward pass
273:
        out = dvx({"p2": p2, "p4": p4}, seg, return_all_steps=True)
274:
        print("polygons shape:", out["polygons"].shape) # expected [B, P, N, 2]
        print("validity shape:", out["validity"].shape) # expected [B, P]
275:
        print("init shape:", out["init_polygons"].shape)
276:
277:
       print("refinement steps:", len(out["refinement_displacements"]))
278:
        # check ranges
        assert out["polygons"].min().item() >= 0.0 - 1e-6
279:
280:
        assert out["polygons"].max().item() <= 1.0 + 1e-6
281:
        print("smoke test passed")
282:
283:
284: if __name__ == "__main__":
        _smoke_test()
```

■ File: models\encoder.py

1: """

```
2: Encoder architecture for multi-scale feature extraction
```

```
3: """
4:
5: import torch
6: import torch.nn as nn
7: import torch.nn.functional as F
9:
10: class ResidualBlock(nn.Module):
        """Basic residual block for the encoder"""
11:
12:
       def __init__(self, in_channels, out_channels, stride=1):
14:
           super().__init__()
15:
           self.conv1 = nn.Conv2d(in_channels, out_channels, 3, stride, 1, bias=False)
16:
17:
           self.bn1 = nn.BatchNorm2d(out channels)
18:
           self.conv2 = nn.Conv2d(out_channels, out_channels, 3, 1, 1, bias=False)
           self.bn2 = nn.BatchNorm2d(out_channels)
19:
20:
           self.shortcut = nn.Sequential()
21:
           if stride != 1 or in_channels != out_channels:
23:
                self.shortcut = nn.Sequential(
24:
                    nn.Conv2d(in_channels, out_channels, 1, stride, bias=False),
25:
                    nn.BatchNorm2d(out_channels),
26:
                )
27:
       def forward(self, x):
28:
29:
           out = F.relu(self.bn1(self.conv1(x)))
30:
            out = self.bn2(self.conv2(out))
31:
           out += self.shortcut(x)
32:
           return F.relu(out)
33:
34:
35: class MultiScaleEncoder(nn.Module):
37:
       Advanced encoder with skip connections and multi-scale feature extraction
38:
       Based on ResNet architecture with Feature Pyramid Network (FPN)
39:
40:
41:
       def __init__(self, input_channels=3, feature_dim=512):
42:
           super().__init__()
43:
44:
            # Stem
            self.stem = nn.Sequential(
46:
               nn.Conv2d(input_channels, 64, 7, 2, 3, bias=False),
47:
                nn.BatchNorm2d(64),
48:
                nn.ReLU(inplace=True),
49:
               nn.MaxPool2d(3, 2, 1),
50:
           )
51:
52:
            # ResNet blocks
53:
           self.layer1 = self._make_layer(64, 64, 2, stride=1) # 64x64
           self.layer2 = self._make_layer(64, 128, 2, stride=2) # 32x32
54:
55:
           self.layer3 = self._make_layer(128, 256, 2, stride=2) # 16x16
56:
           self.layer4 = self._make_layer(256, 512, 2, stride=2) # 8x8
57:
            # FPN lateral connections
58:
           self.lateral4 = nn.Conv2d(512, feature_dim, 1)
59:
60:
           self.lateral3 = nn.Conv2d(256, feature_dim, 1)
61:
            self.lateral2 = nn.Conv2d(128, feature_dim, 1)
            self.lateral1 = nn.Conv2d(64, feature_dim, 1)
62:
63:
64:
            # FPN output layers
            self.smooth4 = nn.Conv2d(feature_dim, feature_dim, 3, 1, 1)
65:
66:
            self.smooth3 = nn.Conv2d(feature_dim, feature_dim, 3, 1, 1)
67:
           self.smooth2 = nn.Conv2d(feature_dim, feature_dim, 3, 1, 1)
68:
           self.smooth1 = nn.Conv2d(feature_dim, feature_dim, 3, 1, 1)
69:
70:
            # Global context
71:
           self.global_pool = nn.AdaptiveAvgPool2d(1)
           self.global_fc = nn.Sequential(
72:
73:
                nn.Linear(512, feature_dim),
74:
                nn.ReLU().
75:
                nn.Linear(feature_dim, feature_dim)
```

```
76:
 77:
 78:
         def _make_layer(self, in_channels, out_channels, blocks, stride=1):
 79:
             layers = []
             layers.append(ResidualBlock(in_channels, out_channels, stride))
 80:
 81:
             for in range(1, blocks):
 82:
                 layers.append(ResidualBlock(out_channels, out_channels))
 83:
             return nn.Sequential(*layers)
 84:
        def forward(self, x):
 85:
 86:
            # Bottom-up pathway
 87:
            x = self.stem(x) # 64x64
 88:
 89:
            c1 = self.layer1(x) # 64x64
            c2 = self.layer2(c1) # 32x32
 90:
 91:
            c3 = self.layer3(c2) # 16x16
            c4 = self.layer4(c3) # 8x8
 92:
 93:
            # Global context
 94:
 95:
            global_feat = self.global_pool(c4).flatten(1)
 96:
            global_feat = self.global_fc(global_feat)
 97:
 98:
            # Top-down pathway (FPN)
            p4 = self.lateral4(c4)
99:
100:
            p3 = self.lateral3(c3) + F.interpolate(p4, scale_factor=2)
101:
           p2 = self.lateral2(c2) + F.interpolate(p3, scale_factor=2)
102:
            p1 = self.lateral1(c1) + F.interpolate(p2, scale_factor=2)
103:
104:
            # Smooth
105:
           p4 = self.smooth4(p4)
           p3 = self.smooth3(p3)
106:
107:
            p2 = self.smooth2(p2)
108:
            p1 = self.smooth1(p1)
109:
110:
            return {
111:
                "p1": p1, # 64x64
                 "p2": p2, # 32x32
112:
                 "p3": p3, # 16x16
113:
114:
                 "p4": p4, # 8x8
                 "global": global_feat,
115:
116:
             }
```

■ File: models\extrusion.py

```
______
 2: Vectorized Differentiable 3D extrusion module for converting polygons to 3D occupancy
 3: Optimized version with GPU-accelerated vectorized operations
 4: """
 5:
 6: import torch
 7: import torch.nn as nn
 8: import torch.nn.functional as F
 9: import math
10: import logging
11:
12:
13: # -----
14: # Logging and sanitization helper
16: logger = logging.getLogger(__name__)
17: if not logger.handlers:
18:
       handler = logging.StreamHandler()
      handler.setFormatter(logging.Formatter("%(asctime)s | %(levelname)s | %(message)s"))
19:
20:
      logger.addHandler(handler)
21: logger.setLevel(logging.INFO)
22:
23:
24: def _sanitize_normalized_height(value, sample_id=None, default=0.6):
25:
26:
       Ensure normalized height value is finite and in [0,1].
27:
       Returns a float in [0,1].
```

```
28:
29:
30:
          value: torch scalar tensor or float
           sample_id: optional identifier for logging (string or int)
31:
32:
          default: fallback normalized height
33:
34:
35:
          if isinstance(value, torch.Tensor):
36:
              raw = float(value.item())
37:
           else:
              raw = float(value)
38:
39:
       except Exception:
40:
          raw = float("nan")
41:
42:
       # Build label for logging
       sid = f"[sample={sample_id}]" if sample_id is not None else ""
43:
44:
45:
       # Check finite
46:
       if not math.isfinite(raw):
47:
          logger.warning(f"{sid} Invalid wall height value (not finite): {raw}; using default
                   {default}")
48:
          raw = default
49:
       # Clamp to [0,1]
50:
       if raw < 0.0 or raw > 1.0:
51:
          logger.warning(f"{sid} Wall height normalized {raw} out of [0,1]; clamping.")
52:
53:
          raw = max(0.0, min(1.0, raw))
54:
55:
      return raw
56:
57:
58: def _sanitize_tensor(tensor, default_value=0.0, name="tensor"):
59:
      Sanitize an entire tensor by replacing NaN/Inf values with default.
60:
61:
62:
      Arqs:
63:
          tensor: Input tensor
64:
          default_value: Value to replace invalid entries with
65:
          name: Name for logging
66:
67:
      Returns:
68:
       Sanitized tensor
69:
       if tensor.numel() == 0:
70:
71:
          return tensor
72:
73:
       # Check for any invalid values
74:
       invalid_mask = ~torch.isfinite(tensor)
75:
       num_invalid = invalid_mask.sum().item()
76:
77:
       if num_invalid > 0:
         logger.warning(f"Found {num_invalid} invalid values in {name}, replacing with
78:
                   {default_value}")
79:
          tensor = tensor.clone()
80:
           tensor[invalid_mask] = default_value
81:
82:
       return tensor
83:
84:
85: # -----
86: # Main extrusion module
87: # -----
88: class DifferentiableExtrusion(nn.Module):
89:
90:
       Vectorized Differentiable 3D extrusion module
91:
       Converts polygons + attributes to soft 3D occupancy grids
92:
93:
94:
       def __init__(self, voxel_size: int = 64):
95:
          super().__init__()
96:
          self.voxel_size = int(voxel_size)
97:
          self.register_buffer("_coords", None)
98:
```

```
99:
         def _ensure_coords(self, device):
100:
             """Initialize or update coordinate grid if needed"""
101:
             if (self._coords is None or
102:
                 self._coords.device != device or
                 self._coords.shape[0] != (self.voxel_size * self.voxel_size)):
103:
104:
105:
                 H = W = self.voxel_size
                 y, x = torch.meshgrid(
106:
107:
                     torch.arange(H, device=device),
108:
                     torch.arange(W, device=device),
109:
                     indexing="ij"
110:
                 )
111:
                 coords = torch.stack([x.flatten().float(), y.flatten().float()], dim=1) # [H*W, 2]
112:
                 coords = coords / float(self.voxel_size - 1)
113:
                 self.register_buffer("_coords", coords)
114:
115:
         def polygon_sdf(self, polygon_xy):
116:
117:
             Compute signed distance field for a polygon using vectorized operations.
118:
119:
             device = polygon_xy.device
120:
             self._ensure_coords(device)
121:
             pts = self._coords # [M, 2]
122:
            P = polygon_xy.shape[0]
123:
            if P < 2:
124:
125:
                 return torch.full((pts.shape[0],), 1.0, device=device)
126:
127:
             # Sanitize polygon coordinates
128:
             polygon_xy = _sanitize_tensor(polygon_xy, default_value=0.0, name="polygon_xy")
129:
130:
             v0 = polygon_xy.unsqueeze(1)
            v1 = torch.roll(polygon_xy, shifts=-1, dims=0).unsqueeze(1)
131:
132:
            pts_exp = pts.unsqueeze(0)
133:
134:
             e = v1 - v0
135:
             v = pts_exp - v0
             e_norm_sq = (e^{**2}).sum(dim=2, keepdim=True) + 1e-8
136:
137:
             t = (v * e).sum(dim=2, keepdim=True) / e_norm_sq
138:
            t_{clamped} = t.clamp(0.0, 1.0)
139:
140:
             proj = v0 + t_clamped * e
141:
             diff = pts_exp - proj
142:
             dists = torch.norm(diff, dim=2)
143:
144:
             # Sanitize distances before min operation
145:
             dists = _sanitize_tensor(dists, default_value=1.0, name="distances")
146:
             min_dist_per_point, _ = dists.min(dim=0)
147:
148:
            x_pts = pts[:, 0].unsqueeze(0)
149:
             y_pts = pts[:, 1].unsqueeze(0)
             x0, y0 = v0[..., 0], v0[..., 1]
150:
151:
             x1, y1 = v1[..., 0], v1[..., 1]
152:
153:
             y\_crosses = ((y0 \le y\_pts) & (y1 > y\_pts)) | ((y1 \le y\_pts) & (y0 > y\_pts))
             inter_x = x0 + (x1 - x0) * ((y_pts - y0) / (y1 - y0 + 1e-8))
154:
             crossings = (inter_x > x_pts) & y_crosses
155:
156:
             crossing_count = crossings.sum(dim=0)
157:
             inside = (crossing_count % 2 == 1)
158:
159:
             sdf = min_dist_per_point.clone()
160:
             sdf[inside] = -sdf[inside]
161:
162:
             # Final sanitization of SDF output
             sdf = _sanitize_tensor(sdf, default_value=1.0, name="sdf")
163:
164:
             return sdf
165:
         def forward(self, polygons, attributes, validity_scores, sample_ids=None):
166:
167:
             Convert polygons to 3D voxel occupancy.
168:
169:
             sample_ids: optional list/array of identifiers (e.g., filenames or dataset indices)
170:
171:
             device = polygons.device
```

```
B, N, P, _ = polygons.shape
172:
           D = H = W = self.voxel_size
173:
174:
175:
            # Sanitize input tensors
            polygons = _sanitize_tensor(polygons, default_value=0.0, name="input_polygons")
176:
177:
            attributes = _sanitize_tensor(attributes, default_value=0.6, name="input_attributes")
178:
            validity_scores = _sanitize_tensor(validity_scores, default_value=0.0,
                    name="input_validity_scores")
179:
180:
            voxels = torch.zeros((B, D, H, W), device=device)
181:
182:
            for b in range(B):
183:
               # pick identifier if available
184:
               sid = sample_ids[b] if sample_ids is not None else b
185:
186:
               # Sanitize height with logging
187:
               wall_height_normalized = attributes[b, 0]
188:
               sanitized_norm = _sanitize_normalized_height(
                   wall_height_normalized, sample_id=sid, default=0.6
189:
190:
191:
192:
               wall_height_m = sanitized_norm * 5.0
193:
               height_frac = wall_height_m / 5.0
194:
               height_voxels = int(round(height_frac * D))
195:
               height_voxels = max(1, min(D, height_voxels))
196:
197:
               # Process each polygon for this batch
198:
               validity_mask = validity_scores[b] >= 0.5
199:
               if not validity_mask.any():
200:
                   continue
201:
202:
               combined_mask = torch.zeros((H, W), device=device)
203:
               sharpness = 100.0
204:
205:
               for n in range(N):
206:
                   if not validity_mask[n]:
207:
                       continue
208:
209:
                   polygon = polygons[b, n] # [P, 2]
210:
211:
                   # Filter out zero-padded vertices
212:
                   vertex_mask = (polygon.sum(dim=1) != 0.0)
213:
                   if vertex_mask.sum().item() < 3:</pre>
214:
                       continue
215:
216:
                   valid_polygon = polygon[vertex_mask]
217:
218:
                   # Compute SDF for this polygon
219:
                   sdf = self.polygon_sdf(valid_polygon)
220:
                   mask = torch.sigmoid(-sdf * sharpness)
                   mask_2d = mask.view(H, W)
221:
222:
223:
                   # Sanitize mask before combining
224:
                   225:
                   combined_mask = torch.maximum(combined_mask, mask_2d)
226:
               # Create 3D mask by extruding to the computed height
227:
228:
               mask_3d = combined_mask.unsqueeze(0).expand(height_voxels, -1, -1)
229:
230:
               # Sanitize final mask before assignment
               mask_3d = _sanitize_tensor(mask_3d, default_value=0.0, name=f"final_mask_3d_b{b}")
231:
232:
               voxels[b, :height_voxels] = mask_3d
233:
234:
            # Final sanitization of output
           voxels = _sanitize_tensor(voxels, default_value=0.0, name="output_voxels")
235:
           return voxels
237:
238:
239: # -----
240: # Fast extrusion module
241: # -----
242: class DifferentiableExtrusionFast(nn.Module):
243:
```

```
244:
         Optimized version that batches polygon processing.
245:
246:
247:
         def __init__(self, voxel_size: int = 64):
248:
             super().__init__()
249:
             self.voxel_size = int(voxel_size)
250:
             self.register_buffer("_coords", None)
251:
252:
         def _ensure_coords(self, device):
             if (self._coords is None or
253:
254:
                 self._coords.device != device or
255:
                 self._coords.shape[0] != (self.voxel_size * self.voxel_size)):
257:
                 H = W = self.voxel_size
                 y, x = torch.meshgrid(
258:
259:
                     torch.arange(H, device=device),
260:
                     torch.arange(W, device=device),
261:
                     indexing="ij"
262:
                 )
263:
                 coords = torch.stack([x.flatten().float(), y.flatten().float()], dim=1)
264:
                 coords = coords / float(self.voxel_size - 1)
265:
                 self.register_buffer("_coords", coords)
266:
267:
        def batch_polygon_sdf(self, polygons_batch, validity_mask):
268:
             device = polygons_batch.device
             self._ensure_coords(device)
269:
270:
271:
            N, P, _ = polygons_batch.shape
272:
            M = self._coords.shape[0]
273:
             sdfs = torch.full((N, M), 1.0, device=device)
274:
275:
             valid_indices = torch.where(validity_mask)[0]
276:
            if len(valid_indices) == 0:
277:
                 return sdfs
278:
279:
             valid_polygons = polygons_batch[valid_indices]
280:
             for i, poly_idx in enumerate(valid_indices):
281:
                 poly = valid_polygons[i]
282:
                 vertex_mask = (poly.sum(dim=1) != 0.0)
283:
                 if vertex_mask.sum().item() >= 3:
284:
                     valid_poly = poly[vertex_mask]
285:
                     sdf = self.polygon_sdf(valid_poly)
286:
                     sdfs[poly_idx] = sdf
287:
288:
            return sdfs
289:
        def polygon_sdf(self, polygon_xy):
290:
291:
            device = polygon_xy.device
292:
             self._ensure_coords(device)
            pts = self._coords
293:
294:
            P = polygon_xy.shape[0]
295:
296:
            if P < 2:
297:
                 return torch.full((pts.shape[0],), 1.0, device=device)
298:
299:
             # Sanitize polygon coordinates
300:
             polygon_xy = _sanitize_tensor(polygon_xy, default_value=0.0, name="polygon_xy_fast")
301:
302:
             v0 = polygon xy.unsqueeze(1)
303:
             v1 = torch.roll(polygon_xy, shifts=-1, dims=0).unsqueeze(1)
304:
            pts_exp = pts.unsqueeze(0)
305:
306:
             e = v1 - v0
307:
             v = pts_exp - v0
308:
             e_norm_sq = (e^**2).sum(dim=2, keepdim=True) + 1e-8
309:
             t = (v * e).sum(dim=2, keepdim=True) / e_norm_sq
310:
            t_{clamped} = t.clamp(0.0, 1.0)
311:
312:
            proj = v0 + t_clamped * e
            diff = pts_exp - proj
313:
314:
            dists = torch.norm(diff, dim=2)
315:
316:
             # Sanitize distances before min operation
```

```
317:
             dists = _sanitize_tensor(dists, default_value=1.0, name="distances_fast")
318:
             min_dist_per_point, _ = dists.min(dim=0)
319:
320:
            x_pts = pts[:, 0].unsqueeze(0)
             y_pts = pts[:, 1].unsqueeze(0)
321:
322:
            x0, y0 = v0[..., 0], v0[..., 1]
323:
            x1, y1 = v1[..., 0], v1[..., 1]
324:
325:
             y\_crosses = ((y0 \le y\_pts) & (y1 > y\_pts)) | ((y1 \le y\_pts) & (y0 > y\_pts))
             inter_x = x0 + (x1 - x0) * ((y_pts - y0) / (y1 - y0 + 1e-8))
326:
327:
             crossings = (inter_x > x_pts) & y_crosses
328:
             crossing_count = crossings.sum(dim=0)
329:
             inside = (crossing_count % 2 == 1)
330:
331:
             sdf = min_dist_per_point.clone()
332:
             sdf[inside] = -sdf[inside]
333:
334:
             # Final sanitization of SDF output
             sdf = _sanitize_tensor(sdf, default_value=1.0, name="sdf_fast")
335:
336:
             return sdf
337:
338:
        def forward(self, polygons: torch.Tensor, attributes: torch.Tensor, validity_scores:
                  torch.Tensor) -> torch.Tensor:
339:
             device = polygons.device
340:
             B, N, P, \_ = polygons.shape
            D = H = W = self.voxel_size
341:
342:
343:
             # Sanitize input tensors
344:
            polygons = _sanitize_tensor(polygons, default_value=0.0, name="input_polygons_fast")
345:
             attributes = _sanitize_tensor(attributes, default_value=0.6, name="input_attributes_fast")
346:
             validity_scores = _sanitize_tensor(validity_scores, default_value=0.0,
                      name="input_validity_scores_fast")
347:
348:
             voxels = torch.zeros((B, D, H, W), device=device)
349:
350:
             for b in range(B):
351:
                 validity_mask = validity_scores[b] >= 0.5
352:
                 if not validity_mask.any():
353:
                     continue
354:
355:
                 sdfs = self.batch_polygon_sdf(polygons[b], validity_mask)
356:
357:
                 # Sanitize SDFs before sigmoid
                 {\tt sdfs = \_sanitize\_tensor(sdfs, default\_value=1.0, name=f"batch\_sdfs\_b\{b\}")}
358:
359:
360:
                 sharpness = 100.0
361:
                masks = torch.sigmoid(-sdfs * sharpness)
362:
                 masks_2d = masks.view(N, H, W)
363:
                 # Sanitize masks
364:
                masks_2d = _sanitize_tensor(masks_2d, default_value=0.0, name=f"masks_2d_b{b}")
365:
366:
367:
                 # Sanitize height
368:
                 wall height normalized = attributes[b, 0]
369:
                 sanitized_norm = _sanitize_normalized_height(wall_height_normalized, sample_id=b,
                         default=0.6)
370:
                 wall_height_m = sanitized_norm * 5.0
371:
                 height_frac = wall_height_m / 5.0
372:
                 height_voxels = int(round(height_frac * D))
373:
                 height_voxels = max(1, min(D, height_voxels))
374:
375:
                 combined_mask = torch.zeros((H, W), device=device)
376:
                 for n in range(N):
377:
                     if validity_mask[n]:
378:
                         combined_mask = torch.maximum(combined_mask, masks_2d[n])
379:
380:
                 mask_3d = combined_mask.unsqueeze(0).expand(height_voxels, -1, -1)
381:
382:
                 # Sanitize final mask before assignment
                 mask_3d = _sanitize_tensor(mask_3d, default_value=0.0, name=f"final_mask_3d_fast_b{b}")
383:
                 voxels[b, :height_voxels] = mask_3d
384:
385:
386:
             # Final sanitization of output
```

■ File: models\heads.py

```
_____
 1: """
 2: Multi-task prediction heads for the Neural-Geometric 3D Model Generator
 3: ""'
 4:
 5: import torch
 6: import torch.nn as nn
 7: import torch.nn.functional as F
 9:
10: class SegmentationHead(nn.Module):
 11:
        """Semantic segmentation head with multi-scale fusion"""
12:
       def __init__(self, feature_dim=512, num_classes=5, dropout=0.1):
14:
           super().__init__()
15:
16:
           # Multi-scale fusion
17:
           self.fusion = nn.Sequential(
               nn.Conv2d(feature_dim * 4, feature_dim, 3, 1, 1),
18:
19:
               nn.BatchNorm2d(feature_dim),
 20:
                nn.ReLU(),
21:
                nn.Dropout2d(dropout),
22:
          )
23:
 24:
           # Segmentation decoder
 25:
           self.decoder = nn.Sequential(
               nn.Conv2d(feature_dim, feature_dim // 2, 3, 1, 1),
 26:
 27:
               nn.BatchNorm2d(feature_dim // 2),
 28:
               nn.ReLU(),
 29:
               nn.Conv2d(feature_dim // 2, feature_dim // 4, 3, 1, 1),
 30:
                nn.BatchNorm2d(feature_dim // 4),
31:
               nn.ReLU(),
32:
                nn.Conv2d(feature_dim // 4, num_classes, 1),
33:
            )
 34:
35:
       def forward(self, features):
            # Fuse multi-scale features
37:
            p1, p2, p3, p4 = features["p1"], features["p2"], features["p3"], features["p4"]
 38:
 39:
            # Upsample all to p1 resolution
            p2_up = F.interpolate(
40:
 41:
                p2, size=p1.shape[-2:], mode="bilinear", align_corners=False
 42:
            )
 43:
           p3_up = F.interpolate(
                p3, size=p1.shape[-2:], mode="bilinear", align_corners=False
44:
 45:
 46:
            p4_up = F.interpolate(
 47:
               p4, size=p1.shape[-2:], mode="bilinear", align_corners=False
 48:
 49:
 50:
           fused = torch.cat([p1, p2_up, p3_up, p4_up], dim=1)
51:
           fused = self.fusion(fused)
 52:
53:
            # Final segmentation
 54:
           seg = self.decoder(fused)
 55:
           return F.interpolate(seg, scale_factor=4, mode="bilinear", align_corners=False)
 56:
 57:
58: class AttributeHead(nn.Module):
 59:
        """Attribute regression head for geometric parameters"""
60:
        def __init__(self, feature_dim=512, num_attributes=6, dropout=0.2):
 61:
62:
            super().__init__()
63:
 64:
           self.regressor = nn.Sequential(
                nn.Linear(feature_dim, feature_dim),
 65:
 66:
                nn.ReLU(),
```

```
67:
                nn.Dropout(dropout),
 68:
                nn.Linear(feature_dim, feature_dim // 2),
 69:
                nn.ReLU(),
 70:
                nn.Dropout(dropout),
                nn.Linear(feature_dim // 2, num_attributes),
 71:
 72:
                 nn.Sigmoid(), # Output in [0,1] range
 73:
            )
 74:
 75:
        def forward(self, global_features):
 76:
            return self.regressor(global_features)
 77:
 78:
 79: class SDFHead(nn.Module):
 80:
         """Signed Distance Field prediction for sharp boundaries"""
 81:
 82:
        def __init__(self, feature_dim=512, dropout=0.1):
 83:
            super().__init__()
 84:
            self.sdf_decoder = nn.Sequential(
 85:
                nn.Conv2d(feature_dim, feature_dim // 2, 3, 1, 1),
 87:
                 nn.BatchNorm2d(feature_dim // 2),
 88:
                nn.ReLU(),
 89:
                nn.Dropout2d(dropout),
 90:
                nn.Conv2d(feature_dim // 2, feature_dim // 4, 3, 1, 1),
                nn.BatchNorm2d(feature_dim // 4),
 91:
 92:
                nn.ReLU(),
 93:
                 nn.Conv2d(feature_dim // 4, 1, 1),
                 nn.Tanh(), # SDF in [-1, 1]
 94:
 95:
             )
 96:
 97:
       def forward(self, features):
 98:
            # Use highest resolution features
99:
            p1 = features["p1"]
100:
            sdf = self.sdf_decoder(p1)
101:
            return F.interpolate(sdf, scale_factor=4, mode="bilinear", align_corners=False)
```

■ File: models\model.py

```
______
 1: """
 2: Advanced loss functions for multi-task training with dynamic weighting
 3: Enhanced with cross-modal consistency, graph constraints, and GradNorm
 4: Modified to support conditional geometric losses via run_full_geometric flag
 5: """
 6: import torch
 7: import torch.nn as nn
 8: import torch.nn.functional as F
 9: from .encoder import MultiScaleEncoder
 10: from .heads import SegmentationHead, AttributeHead, SDFHead
11: from .dvx import DifferentiableVectorization
12: from .extrusion import DifferentiableExtrusion
13:
14:
15: class L2Normalize(nn.Module):
       """L2 normalization layer"""
16:
17:
18:
       def __init__(self, dim=1):
 19:
           super().__init__()
           self.dim = dim
 20:
 21:
 22:
      def forward(self, x):
 23:
          return F.normalize(x, p=2, dim=self.dim)
 24:
 25:
 26: class LatentEmbeddingHead(nn.Module):
        """Auxiliary head for cross-modal latent consistency"""
 27:
 28:
 29:
       def __init__(self, feature_dim: int, embedding_dim: int = 256):
30:
           super().__init__()
           self.embedding_dim = embedding_dim
 31:
 32:
 33:
           # 2D embedding path
```

```
self.embedding_2d = nn.Sequential(
 34:
 35:
                 nn.AdaptiveAvgPool2d((1, 1)),
 36:
                 nn.Flatten(),
 37:
                 nn.Linear(feature_dim, embedding_dim * 2),
 38:
                nn.ReLU(),
 39:
                nn.Dropout(0.1),
                 nn.Linear(embedding_dim * 2, embedding_dim),
 40:
 41:
                 L2Normalize(dim=1), # L2 normalize for cosine similarity
 42:
             )
 43:
            # 3D embedding path (from voxel features)
 44:
            self.embedding_3d = nn.Sequential(
 45:
                 nn.AdaptiveAvgPool3d((1, 1, 1)),
 46:
 47:
                 nn.Flatten(),
 48:
                nn.Linear(feature_dim, embedding_dim * 2),
 49:
                nn.ReLU(),
 50:
                nn.Dropout(0.1),
 51:
                 nn.Linear(embedding_dim * 2, embedding_dim),
 52:
                 L2Normalize(dim=1),
 53:
             )
 54:
 55:
        def forward(
 56:
            self, features_2d: torch.Tensor, features_3d: torch.Tensor = None
 57:
         ) -> tuple:
 58:
             Generate 2D and 3D embeddings for consistency loss
 59:
 60:
 61:
            Args:
 62:
                 features_2d: [B, C, H, W] - 2D feature maps
 63:
                 features_3d: [B, C, D, H, W] - 3D feature maps (optional)
 64:
 65:
             Returns:
 66:
              tuple: (embedding_2d, embedding_3d)
 67:
             # 2D embedding
 68:
 69:
             emb_2d = self.embedding_2d(features_2d)
 70:
 71:
             # 3D embedding (if available, otherwise use 2D features reshaped)
 72:
             if features_3d is not None:
 73:
                 emb_3d = self.embedding_3d(features_3d)
 74:
             else:
 75:
                # Create pseudo-3D from 2D features
                 B, C, H, W = features_2d.shape
 77:
                 pseudo_3d = features_2d.unsqueeze(2).expand(
 78:
                    B, C, 4, H, W
 79:
                 ) # Duplicate across depth
 :08
                 emb_3d = self.embedding_3d(pseudo_3d)
 81:
 82:
            return emb_2d, emb_3d
 83:
 84:
 85: class GraphStructureHead(nn.Module):
 86:
         """Head for predicting graph structure (room connectivity)"""
 87:
 88:
        def __init__(self, feature_dim: int, max_rooms: int = 10):
 89:
            super().__init__()
 90:
             self.max_rooms = max_rooms
 91:
 92:
             # Room detection branch
 93:
             self.room_detector = nn.Sequential(
                nn.Conv2d(feature_dim, feature_dim // 2, 3, padding=1),
 94:
 95:
 96:
                 nn.Conv2d(feature_dim // 2, max_rooms, 3, padding=1),
 97:
                 nn.Sigmoid(), # Room probability maps
            )
 98:
 99:
100:
            # Room feature extractor
101:
            self.room_features = nn.Sequential(
102:
                 nn.AdaptiveAvgPool2d((8, 8)), # Pool to fixed size
103:
                nn.Flatten(),
104:
                nn.Linear(feature_dim * 64, 256),
105:
                nn.ReLU(),
106:
                 nn.Linear(256, 128), # Room feature vectors
```

```
107:
108:
109:
             # Adjacency predictor
            self.adjacency_net = nn.Sequential(
110:
                nn.Linear(128 * 2, 64), # Pairwise room features
111:
112:
                nn.ReLU(),
113:
                nn.Linear(64, 32),
114:
                nn.ReLU(),
                 nn.Linear(32, 1),
115:
                 nn.Sigmoid(), # Adjacency probability
116:
117:
118:
119:
         def forward(self, features: torch.Tensor) -> dict:
120:
121:
             Predict room graph structure
122:
123:
            Args:
124:
                features: [B, C, H, W] - Feature maps
125:
126:
             dict with 'room_maps', 'room_features', 'adjacency_matrix'
127:
128:
129:
             B = features.shape[0]
130:
131:
             # Detect room probability maps
             room_maps = self.room_detector(features) # [B, max_rooms, H, W]
132:
133:
134:
             # Extract room features
135:
             room_feats = self.room_features(features) # [B, 128]
136:
137:
             # Create adjacency matrix for all room pairs
138:
             adjacency_matrices = []
139:
             for b in range(B):
140:
                 # Get room features for this batch item
141:
142:
                 feat_b = room_feats[b : b + 1] # [1, 128]
143:
144:
                 # Create pairwise combinations
145:
                 adj_matrix = torch.zeros(
146:
                     (self.max_rooms, self.max_rooms), device=features.device
147:
148:
149:
                 for i in range(self.max_rooms):
150:
                     for j in range(i + 1, self.max_rooms):
151:
                         # Concatenate features for room pair
152:
                         pair_feat = torch.cat([feat_b, feat_b], dim=1) # [1, 256]
153:
154:
                         # Predict adjacency
155:
                         adj_prob = self.adjacency_net(pair_feat) # [1, 1]
156:
157:
                         # Fill symmetric matrix
158:
                         adj_matrix[i, j] = adj_prob.squeeze()
159:
                         adj_matrix[j, i] = adj_prob.squeeze()
160:
161:
                 adjacency_matrices.append(adj_matrix)
162:
            return {
163:
                 "room_maps": room_maps,
164:
165:
                 "room_features": room_feats,
166:
                 "adjacency_matrices": torch.stack(adjacency_matrices),
             }
167:
168:
169:
170: class NeuralGeometric3DGenerator(nn.Module):
171:
172:
        Enhanced neural-geometric system with auxiliary heads for novel training strategies:
173:
         - Cross-modal latent consistency
174:
         - Graph structure prediction
         - Multi-view embeddings for dynamic curriculum
175:
176:
         - Conditional geometric computation via run_full_geometric flag
177:
178:
179:
         def __init__(
```

```
180:
            self.
181:
            input_channels=3,
182:
           num classes=5,
183:
            feature_dim=512,
            num_attributes=6,
184:
185:
            voxel_size=64,
186:
            max_polygons=20,
187:
            max_points=50,
188:
             use_latent_consistency=True,
189:
             use_graph_constraints=True,
190:
            latent_embedding_dim=256,
191:
       ):
192:
            super().__init__()
193:
194:
             # Store configuration
195:
             self.use_latent_consistency = use_latent_consistency
             self.use_graph_constraints = use_graph_constraints
196:
197:
             self.feature_dim = feature_dim
198:
199:
             # Core components
200:
             self.encoder = MultiScaleEncoder(input_channels, feature_dim)
201:
             self.seg_head = SegmentationHead(feature_dim, num_classes)
202:
             self.attr_head = AttributeHead(feature_dim, num_attributes)
203:
            self.sdf_head = SDFHead(feature_dim)
204:
             self.dvx = DifferentiableVectorization(max_polygons, max_points, feature_dim)
205:
             self.extrusion = DifferentiableExtrusion(voxel_size)
206:
             # NEW: Auxiliary heads for novel training strategies
207:
208:
             if use_latent_consistency:
209:
                 self.latent_head = LatentEmbeddingHead(feature_dim, latent_embedding_dim)
210:
211:
             if use_graph_constraints:
212:
                 self.graph_head = GraphStructureHead(feature_dim)
213:
214:
             # Enhanced feature processing for multi-stage training
215:
             self.feature_enhancer = nn.Sequential(
216:
                 nn.Conv2d(feature_dim, feature_dim, 3, padding=1),
217:
                 nn.GroupNorm(32, feature_dim),
218:
                 nn.ReLU(),
                 nn.Conv2d(feature_dim, feature_dim, 3, padding=1),
219:
220:
                 nn.GroupNorm(32, feature_dim),
221:
             )
222:
             \# lazy-created 3d voxel processor will be attached on first use
223:
224:
             self._voxel_processor = None
225:
226:
        def _select_spatial_feature(self, features):
227:
             Given encoder output (dict or tensor), select a spatial 4-D feature map
228:
229:
             Prefer high-resolution feature maps (p1) and avoid selecting 'global' vector.
230:
231:
             # If encoder returned a tensor already, make sure it's 4D
232:
             if not isinstance(features, dict):
233:
                if features.dim() == 4:
234:
                     return features
235:
                 else:
236:
                     raise ValueError(
237:
                         f"Encoder returned a tensor with shape {tuple(features.shape)}; "
238:
                         "expected a 4D feature map [B, C, H, W]."
239:
240:
241:
             # Encoder returned dict: prefer pl,p2,p3,p4,high_res,out,main but NEVER 'global'
242:
             preferred_keys = ["p1", "p2", "p3", "p4", "high_res", "out", "main"]
243:
             for k in preferred_keys:
                 if k in features:
244:
245:
                     candidate = features[k]
                     if isinstance(candidate, torch.Tensor) and candidate.dim() == 4:
246:
247:
                         return candidate
248:
249:
             # As a last resort, scan dict values for the first 4D tensor that isn't 'global'
250:
             for k, v in features.items():
251:
                if k == "global":
252:
                     continue
```

```
253:
                 if isinstance(v, torch.Tensor) and v.dim() == 4:
254:
255:
             # If nothing found, raise informative error rather than silently picking wrong shape
256:
257:
             raise RuntimeError(
258:
                 "No spatial 4D feature map found in encoder output. Encoder returned keys: "
259:
                 \texttt{f"\{list(features.keys())\}. Ensure encoder provides at least one [B,C,H,W] tensor "}
260:
                 "under keys like 'p1','p2','p3','p4','out', or 'high_res'."
261:
262:
         def forward(self, image, run_full_geometric=True, return_aux=True):
263:
264:
265:
             Enhanced forward pass with auxiliary outputs and conditional geometric computation
266:
267:
             Aras:
268:
                 image: [B, C, H, W] input images
269:
                 run_full_geometric: Whether to run heavy DVX and extrusion computations
270:
                 return_aux: Whether to compute auxiliary outputs
271:
272:
             Returns:
273:
                dict with predictions, conditionally including geometric outputs
274:
275:
             # Multi-scale feature extraction
276:
             features = self.encoder(image)
277:
278:
             # Enhance features
279:
             spatial_feat = self._select_spatial_feature(features)
280:
             enhanced_features = self.feature_enhancer(spatial_feat)
281:
282:
             # keep structured features dict for heads that expect multi-scale inputs
283:
             if isinstance(features, dict):
284:
                 features["enhanced"] = enhanced_features
285:
                 main_features = enhanced_features
286:
             else:
                 features = {"main": enhanced_features, "enhanced": enhanced_features}
287:
288:
                 main_features = enhanced_features
289:
290:
             # Core predictions (always computed - these are fast)
291:
             segmentation = self.seg_head(features)
292:
             attributes = self.attr_head(
                 features.get("global")
293:
                 if isinstance(features, dict) and "global" in features
294:
295:
                 else main_features.mean(dim=[2, 3])
296:
             )
297:
             sdf = self.sdf_head(features)
298:
299:
             # Base outputs
300:
             outputs = {
301:
                 "segmentation": segmentation,
302:
                 "attributes": attributes,
303:
                 "sdf": sdf,
304:
                 "features": features,
305:
             }
306:
307:
             # Conditional geometric computation (heavy operations)
308:
             if run full geometric:
                 # DVX polygon fitting
309:
310:
                 dvx_output = self.dvx(features, segmentation)
311:
                 polygons = dvx_output.get("polygons", None)
                 validity = dvx_output.get("validity", None)
312:
313:
314:
                 # 3D extrusion (defensive: ensure inputs exist)
315:
                 try:
316:
                     voxels_pred = self.extrusion(polygons, attributes, validity)
317:
                 except Exception as e:
318:
                     # Log or print a helpful message for debugging; avoid crashing training
319:
                     # (Replace print with logger if you have one)
320:
                     print(f"[Warning] extrusion failed: {e}")
321:
                     voxels_pred = None
322:
323:
                 # Add geometric outputs
324:
                 outputs.update({
325:
                     "polygons": polygons,
```

```
"polygon validity": validity,
326:
327:
                      "voxels_pred": voxels_pred,
328:
                 })
329:
                 # NEW: Auxiliary outputs for novel training strategies (only when geometric is enabled)
330:
331:
                 if return aux:
332:
                     # Cross-modal consistency embeddings
                     if self.use_latent_consistency:
333:
334:
                         if voxels_pred is not None:
                             voxel_features = self._create_3d_features_from_voxels(voxels_pred)
335:
336:
                             latent_2d, latent_3d = self.latent_head(main_features, voxel_features)
337:
                         else:
338:
                              # Fall back to pseudo-3D built from 2D features if voxels not available
339:
                              latent_2d, latent_3d = self.latent_head(main_features, None)
340:
                         outputs["latent_2d_embedding"] = latent_2d
341:
                         outputs["latent_3d_embedding"] = latent_3d
342:
             else:
343:
                 # Geometric path explicitly skipped for this stage
344:
                 outputs.update({
345:
                     "polygons": None,
                     "polygon_validity": None,
346:
347:
                      "voxels_pred": None,
348:
                 })
349:
350:
                 # Still compute some auxiliary outputs that don't depend on geometry
351:
                 if return_aux and self.use_latent_consistency:
352:
                     # Use pseudo-3D features for 2D-only consistency inside latent head
353:
                     latent_2d, latent_3d = self.latent_head(main_features, None)
354:
                     outputs["latent_2d_embedding"] = latent_2d
355:
                     outputs["latent_3d_embedding"] = latent_3d
356:
357:
             # Graph structure predictions (independent of geometric computation)
358:
             if return_aux and self.use_graph_constraints:
359:
                 graph_output = self.graph_head(main_features)
360:
                 outputs.update(graph_output)
361:
362:
             return outputs
363:
364:
         def get_latent_embeddings(self, image):
365:
366:
             Convenience method to get just the latent embeddings
367:
             Used by trainer for consistency loss
368:
             if not self.use_latent_consistency:
369:
370:
                 return None, None
371:
372:
             with torch.no grad():
373:
                 features = self.encoder(image)
                 spatial_feat = self._select_spatial_feature(features)
374:
375:
                 main_features = self.feature_enhancer(spatial_feat)
376:
377:
                 # Quick forward to get segmentation/attributes
378:
                 segmentation = self.seg_head(features)
379:
                 attributes = self.attr_head(
380:
                     features.get("global")
                     if isinstance(features, dict) and "global" in features
381:
382:
                     else main_features.mean(dim=[2, 3])
383:
                 )
384:
385:
                 # Attempt DVX + extrusion, but be defensive (may be expensive)
386:
                 dvx_output = self.dvx(features, segmentation)
387:
                 polygons = dvx_output.get("polygons", None)
388:
                 validity = dvx_output.get("validity", None)
389:
390:
                 try:
391:
                     voxels_pred = self.extrusion(polygons, attributes, validity)
392:
                 except Exception as e:
                     print(f"[Warning] \ get\_latent\_embeddings: \ extrusion \ failed: \ \{e\}")
393:
394:
                     voxels_pred = None
395:
396:
                 # If voxels not available, latent_head will fall back to pseudo-3D
397:
                 if voxels pred is not None:
398:
                     voxel_features = self._create_3d_features_from_voxels(voxels_pred)
```

```
399:
                 else:
400:
                     voxel features = None
401:
402:
             return self.latent_head(main_features, voxel_features)
403:
404:
         def _create_3d_features_from_voxels(self, voxels):
405:
406:
             Create 3D feature representation from voxel predictions
407:
408:
             Arqs:
409:
                voxels: [B, D, H, W] voxel predictions
410:
411:
             Returns:
                [B, C, D, H, W] 3D features
412:
413:
             # Defensive check
414:
             if voxels is None:
415:
416:
                 raise ValueError(
                     "Received voxels=None in _create_3d_features_from_voxels(). "
417:
418:
                     "This indicates that the geometric pipeline was skipped or extrusion failed. "
419:
                     "Call this method only when voxels are available, or use latent_head(..., None) to "
420:
                     "compute pseudo-3D features from 2D."
421:
                 )
422:
423:
             # Ensure expected shape
424:
             if voxels.dim() != 4:
425:
                 raise ValueError(f"voxels must have shape [B,D,H,W], got {tuple(voxels.shape)}")
426:
427:
             B, D, H, W = voxels.shape
428:
429:
             # Expand voxels to have feature channels
430:
             # Simple approach: repeat voxel values across feature dimension
431:
             rep_ch = max(1, self.feature_dim // 4)
             voxel_features = voxels.unsqueeze(1).expand(B, rep_ch, D, H, W).contiguous()
432:
433:
434:
             # Add some learned 3D processing
435:
             if self._voxel_processor is None:
436:
                 # Build with correct device
437:
                 device = voxels.device
438:
                 self._voxel_processor = nn.Sequential(
439:
                     nn.Conv3d(rep_ch, max(rep_ch, self.feature_dim // 2), 3, padding=1),
440:
                     nn.ReLU(),
441:
                     nn.Conv3d(max(rep_ch, self.feature_dim // 2), self.feature_dim, 3, padding=1),
442:
                 ).to(device)
443:
444:
             return self._voxel_processor(voxel_features)
445:
446:
         def get_stage_parameters(self, stage: int):
447:
448:
             Get parameters for specific training stage
449:
             Useful for stage-specific optimization
450:
             . . .
451:
             if stage == 1:
452:
                 # Stage 1: 2D components only
453:
                 params = []
454:
                 params.extend(list(self.encoder.parameters()))
                 params.extend(list(self.seg_head.parameters()))
455:
456:
                 params.extend(list(self.attr_head.parameters()))
457:
                 params.extend(list(self.sdf_head.parameters()))
458:
                 params.extend(list(self.feature_enhancer.parameters()))
459:
460:
                 if self.use_latent_consistency:
461:
                     params.extend(list(self.latent_head.parameters()))
462:
             elif stage == 2:
463:
464:
                 # Stage 2: DVX components
                 params = list(self.dvx.parameters())
465:
466:
467:
             else: # stage == 3
                 # Stage 3: All parameters
468:
469:
                 params = list(self.parameters())
470:
             return params
471:
```

```
472:
473:
         def freeze_stage_parameters(self, stages_to_freeze: list):
474:
475:
             Freeze parameters for specific stages
476:
477:
             Arqs:
478:
                stages_to_freeze: List of stage numbers to freeze
479:
480:
             for stage in stages_to_freeze:
                 stage_params = self.get_stage_parameters(stage)
481:
482:
                 for param in stage_params:
483:
                     param.requires_grad = False
484:
485:
         def unfreeze_stage_parameters(self, stages_to_unfreeze: list):
486:
487:
             Unfreeze parameters for specific stages
488:
489:
             Arqs:
490:
               stages_to_unfreeze: List of stage numbers to unfreeze
491:
492:
             for stage in stages_to_unfreeze:
                 stage_params = self.get_stage_parameters(stage)
493:
494:
                 for param in stage_params:
495:
                     param.requires_grad = True
496:
497:
         def get_curriculum_metrics(self):
498:
             Get metrics useful for curriculum learning decisions
499:
500:
501:
             metrics = {}
502:
503:
             # Parameter counts per stage
504:
             for stage in [1, 2, 3]:
505:
                 stage_params = self.get_stage_parameters(stage)
506:
                 metrics[f"stage_{stage}_params"] = sum(p.numel() for p in stage_params)
507:
508:
             # Feature dimensions
509:
            metrics["feature_dim"] = self.feature_dim
510:
            metrics["has_latent_consistency"] = self.use_latent_consistency
511:
            metrics["has_graph_constraints"] = self.use_graph_constraints
512:
513:
            return metrics
```

File: train.py

```
1: """
2: Enhanced training script for the Neural-Geometric 3D Model Generator
3: Implements novel training strategies: dynamic curriculum, adaptive weighting, cross-modal
           consistency
4: """
5:
6: import argparse
7: import torch
8: from torch.utils.data import DataLoader
9: from pathlib import Path
10: import json
11: import matplotlib.pyplot as plt
12: import numpy as np
13:
14: from dataset import AdvancedFloorPlanDataset
15: from models.model import NeuralGeometric3DGenerator
16: from training.trainer import AdaptiveMultiStageTrainer, MultiStageTrainer
17: from utils.visualization import plot_training_history, plot_curriculum_analysis
18: from config import (
19:
       DEFAULT_DATA_CONFIG,
20:
       DEFAULT_MODEL_CONFIG,
21:
       DEFAULT_TRAINING_CONFIG,
22:
      DEFAULT_LOSS_CONFIG,
23:
      TrainingConfig,
24:
       CurriculumConfig
25: )
```

```
26:
27:
28: def create_enhanced_config(args):
        """Create enhanced training configuration with novel strategies"""
29:
30:
        config = TrainingConfig()
31:
32:
        # Basic settings
33:
        config.device = args.device or ("cuda" if torch.cuda.is_available() else "cpu")
34:
        # Dynamic curriculum settings
35:
36:
        if args.dynamic_curriculum:
            config.curriculum = CurriculumConfig()
37:
38:
            config.curriculum.use_dynamic_curriculum = True
39:
            config.curriculum.stage_switch_patience = args.patience
40:
            config.curriculum.min_improvement_threshold = args.min_improvement
41:
42:
            # Adjust epoch limits for dynamic training
43:
            config.max_stagel_epochs = args.max_stagel_epochs
44:
            config.max_stage2_epochs = args.max_stage2_epochs
45:
            config.max_stage3_epochs = args.max_stage3_epochs
46:
47:
            print("Dynamic curriculum learning enabled")
48:
            print(f" Stage switch patience: {config.curriculum.stage_switch_patience}")
49:
           \verb|print(f" Min improvement threshold: {config.curriculum.min_improvement\_threshold}"|)|
50:
        else:
            # Disable dynamic curriculum for traditional training
51:
52:
            config.curriculum.use_dynamic_curriculum = False
            print("Using traditional fixed-epoch training")
53:
54:
55:
        # GradNorm dynamic weighting
56:
       if args.gradnorm:
57:
            config.curriculum.use_gradnorm = True
58:
            config.curriculum.gradnorm_alpha = args.gradnorm_alpha
            config.curriculum.gradnorm_update_freq = args.gradnorm_freq
59:
            print(f"GradNorm dynamic weighting enabled (alpha={args.gradnorm_alpha})")
60:
61:
62:
        # Topology-aware scheduling
63:
        if args.topology_schedule != "static":
64:
            config.curriculum.topology_schedule = args.topology_schedule
65:
            config.curriculum.topology_start_weight = args.topology_start
66:
            config.curriculum.topology_end_weight = args.topology_end
67:
            print(f"Topology-aware scheduling: {args.topology_schedule}")
68:
           print(f" Weights: {args.topology_start} -> {args.topology_end}")
69:
70:
       return config
71:
73: def create_enhanced_model(args):
74:
        """Create enhanced model with auxiliary heads"""
75:
        model = NeuralGeometric3DGenerator(
76:
           input_channels=args.input_channels,
77:
           num_classes=args.num_classes,
78:
            feature_dim=args.feature_dim,
79:
           num attributes=args.num attributes,
:08
            voxel_size=args.voxel_size,
81:
           max_polygons=args.max_polygons,
82:
           max_points=args.max_points,
           use_latent_consistency=args.latent_consistency,
83:
84:
            use_graph_constraints=args.graph_constraints,
85:
            latent_embedding_dim=args.embedding_dim
86:
87:
88:
       print(f"Enhanced model created:")
89:
        print(f" Feature dim: {args.feature_dim}")
       print(f" Latent consistency: {args.latent_consistency}")
90:
91:
       print(f" Graph constraints: {args.graph_constraints}")
92:
93:
        # Print parameter counts
94:
        total_params = sum(p.numel() for p in model.parameters())
95:
        trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
96:
        print(f" Total parameters: {total_params:,}")
97:
        print(f" Trainable parameters: {trainable_params:,}")
98:
```

```
99:
         return model
100:
101:
102: def visualize_training_results(history, output_dir):
         """Create comprehensive training visualizations"""
103:
104:
         output_dir = Path(output_dir)
105:
         output_dir.mkdir(exist_ok=True)
106:
107:
         # Traditional loss curves
         plot_training_history(history, save_path=str(output_dir / "training_history.png"))
108:
109:
110:
         # Novel curriculum analysis plots
111:
         if "stage_transitions" in history and history["stage_transitions"]:
112:
             plot_curriculum_analysis(history, save_path=str(output_dir / "curriculum_analysis.png"))
113:
114:
         # Dynamic weight evolution
         if "dynamic_weights" in history and history["dynamic_weights"]:
115:
116:
             plt.figure(figsize=(12, 8))
117:
118:
             # Extract weight evolution data
119:
             epochs = [entry["epoch"] for entry in history["dynamic_weights"]]
120:
             weight_names = list(history["dynamic_weights"][0]["weights"].keys())
121:
122:
             for weight name in weight names:
                 weights = [entry["weights"].get(weight_name, 0) for entry in history["dynamic_weights"]]
123:
                 if any(w > 0.001 for w in weights): \# Only plot significant weights
124:
125:
                     plt.plot(epochs, weights, label=weight_name, linewidth=2)
126:
127:
             plt.xlabel("Global Epoch")
128:
             plt.ylabel("Loss Weight")
129:
             plt.title("Dynamic Loss Weight Evolution")
130:
             plt.legend()
131:
             plt.grid(True, alpha=0.3)
132:
             plt.tight_layout()
             plt.savefig(output_dir / "weight_evolution.png", dpi=300)
133:
134:
             plt.close()
135:
136:
         # Component loss breakdown
137:
         fig, axes = plt.subplots(1, 3, figsize=(18, 5))
         stage_names = ["stage1", "stage2", "stage3"]
138:
139:
140:
         for idx, stage_name in enumerate(stage_names):
141:
             if stage_name in history and "component_losses" in history[stage_name]:
142:
                 component_data = history[stage_name]["component_losses"]
143:
                 if component_data:
144:
                     # Get component names from first entry
145:
                     component_names = list(component_data[0].keys())
146:
147:
                     for comp_name in component_names:
148:
                         if comp_name in ['seg', 'dice', 'polygon', 'voxel', 'topology',
149:
                                         'latent_consistency', 'graph']:
                             values = [entry.get(comp_name, 0) for entry in component_data]
150:
151:
                             if any(v > 0.001 \text{ for } v \text{ in values}): # Only plot significant losses
152:
                                 axes[idx].plot(values, label=comp_name, linewidth=2)
153:
                     axes[idx].set_title(f"{stage_name.upper()} Component Losses")
154:
                     axes[idx].set_xlabel("Epoch")
155:
                     axes[idx].set_ylabel("Loss Value")
156:
157:
                     axes[idx].legend()
158:
                     axes[idx].grid(True, alpha=0.3)
159:
160:
         plt.tight_layout()
161:
         plt.savefig(output_dir / "component_losses.png", dpi=300)
162:
         plt.close()
163:
164:
         print(f"Training visualizations saved to {output_dir}")
165:
166:
167: def save_training_summary(history, config, output_dir):
         """Save comprehensive training summary"""
168:
169:
         output_dir = Path(output_dir)
170:
171:
         summary = {
```

```
172:
             "training_config": {
                 "dynamic_curriculum": config.curriculum.use_dynamic_curriculum,
173:
174:
                 "gradnorm_enabled": config.curriculum.use_gradnorm,
175:
                 "topology_schedule": config.curriculum.topology_schedule,
176:
                 "max_epochs": [config.max_stage1_epochs, config.max_stage2_epochs,
                          config.max_stage3_epochs]
177:
178:
             "training_results": {},
179:
             "novel_strategies_summary": {}
         }
180:
181:
182:
         # Training results
183:
         for stage_name, data in history.items():
             if isinstance(data, dict) and "val_loss" in data and data["val_loss"]:
184:
185:
                 summary["training results"][stage name] = {
186:
                     "final_val_loss": data["val_loss"][-1],
                     "best_val_loss": min(data["val_loss"]),
187:
188:
                     "epochs_trained": len(data["val_loss"])
                 }
189:
190:
191:
         # Novel strategies summary
192:
         if "stage_transitions" in history:
193:
             summary["novel_strategies_summary"]["adaptive_transitions"] =
                      len(history["stage_transitions"])
194:
195:
         if "dynamic_weights" in history:
196:
             summary["novel_strategies_summary"]["weight_updates"] = len(history["dynamic_weights"])
197:
198:
         if "curriculum_events" in history:
199:
             summary["novel_strategies_summary"]["curriculum_events"] = len(history["curriculum_events"])
200:
201:
         # Save as JSON
         with open(output_dir / "training_summary.json", 'w') as f:
202:
203:
             json.dump(summary, f, indent=2)
204:
205:
         print(f"Training summary saved to {output_dir / 'training_summary.json'}")
206:
207:
208: def main():
209:
        parser = argparse.ArgumentParser(description="Enhanced Neural-Geometric 3D Model Generator
                  Training")
210:
211:
         # Basic arguments
         parser.add_argument("--data_dir", type=str, default="./data/floorplans",
212:
213:
                           help="Path to dataset directory")
214:
         parser.add_argument("--batch_size", type=int, default=2, help="Batch size")
        parser.add_argument("--num_workers", type=int, default=4, help="Number of data workers")
215:
216:
        parser.add_argument("--device", type=str, default=None, help="Training device")
217:
         \verb|parser.add_argument("--resume", type=str, default=None, help="Resume from checkpoint")| \\
218:
         parser.add_argument("--output_dir", type=str, default="./checkpoints",
                            help="Output directory for checkpoints")
219:
220:
221:
         # Training mode selection
222:
         \verb|parser.add_argument("--training_mode", type=str, choices=["traditional", "adaptive"]|, \\
223:
                            default="adaptive", help="Training mode (traditional fixed epochs vs
                                     adaptive)")
224:
         parser.add_argument("--stage", type=str, choices=["1", "2", "3", "all"], default="all",
225:
                            help="Training stage to run (only for traditional mode)")
226:
227:
         # Novel training strategies
228:
         parser.add_argument("--dynamic-curriculum", action="store_true", default=True,
229:
                            help="Enable adaptive stage transitioning")
230:
         parser.add_argument("--patience", type=int, default=5,
231:
                            help="Epochs without improvement before stage transition")
232:
         parser.add_argument("--min-improvement", type=float, default=0.001,
233:
                            help="Minimum relative improvement threshold")
234:
235:
         parser.add_argument("--gradnorm", action="store_true", default=True,
236:
                            help="Enable GradNorm dynamic loss weighting")
        parser.add_argument("--gradnorm-alpha", type=float, default=0.12,
237:
238:
                            help="GradNorm restoring force parameter")
239:
         parser.add argument("--gradnorm-freg", type=int, default=5,
240:
                            help="GradNorm update frequency (batches)")
```

```
241:
242:
         parser.add_argument("--topology-schedule", type=str,
243:
                            choices=["static", "progressive", "linear_ramp"],
                            default="progressive", help="Topology loss scheduling strategy")
244:
         parser.add_argument("--topology-start", type=float, default=0.1,
245:
246:
                            help="Starting weight for topology loss")
247:
        parser.add_argument("--topology-end", type=float, default=1.0,
248:
                            help="Ending weight for topology loss")
249:
         # Model enhancements
250:
        parser.add_argument("--latent-consistency", action="store_true", default=True,
251:
252:
                            help="Enable cross-modal latent consistency loss")
253:
        parser.add_argument("--graph-constraints", action="store_true", default=True,
254:
                            help="Enable graph-based topology constraints")
255:
        parser.add argument("--embedding-dim", type=int, default=256,
256:
                            help="Latent embedding dimension")
257:
258:
         # Model architecture
         parser.add_argument("--input_channels", type=int, default=3, help="Input image channels")
259:
        parser.add_argument("--num_classes", type=int, default=5, help="Number of segmentation classes")
260:
        \verb|parser.add_argument("--feature_dim", type=int, default=768, help="Feature dimension")| \\
261:
262:
        parser.add_argument("--num_attributes", type=int, default=6, help="Number of attribute
                  predictions")
        parser.add_argument("--voxel_size", type=int, default=64, help="3D voxel grid size")
263:
264:
        parser.add_argument("--max_polygons", type=int, default=30, help="Maximum number of polygons")
265:
         parser.add_argument("--max_points", type=int, default=100, help="Maximum points per polygon")
266:
267:
         # Dynamic epoch limits
268:
        parser.add_argument("--max-stagel-epochs", type=int, default=50, help="Max epochs for Stage 1")
269:
        \verb|parser.add_argument("--max-stage2-epochs", type=int, default=35, help="Max epochs for Stage 2")| \\
270:
        parser.add_argument("--max-stage3-epochs", type=int, default=100, help="Max epochs for Stage 3")
271:
         parser.add_argument("--persistent_workers",action="store_true",default=False,help="Keep
272:
                  DataLoader workers alive between epochs (requires num_workers > 0).")
273:
274:
         parser.add_argument("--prefetch_factor",type=int,default=2,help="Number of batches preloaded by
                  each worker.")
275:
276:
277:
         args = parser.parse_args()
278:
279:
         # Setup device
280:
         device = args.device or ("cuda" if torch.cuda.is_available() else "cpu")
        print(f"Using device: {device}")
281:
282:
283:
         import torch.backends.cudnn as cudnn
284:
        if device == "cuda":
285:
             cudnn.benchmark = True
286:
287:
         # Create output directory
         output_dir = Path(args.output_dir)
288:
289:
         output_dir.mkdir(exist_ok=True)
290:
291:
         # Create enhanced configuration
292:
         config = create_enhanced_config(args)
293:
        print("\n" + "="*80)
294:
295:
        print("NEURAL-GEOMETRIC 3D MODEL GENERATOR - ENHANCED TRAINING")
296:
        print("="*80)
297:
         print("Novel Training Strategies Enabled:")
298:
        if config.curriculum.use_dynamic_curriculum:
299:
            print("? Adaptive Stage Transitioning (Dynamic Curriculum)")
300:
         if config.curriculum.use gradnorm:
301:
            print("? Multi-objective Optimization with GradNorm")
         if config.curriculum.topology_schedule != "static":
302:
303:
            print("? Topology-aware Loss Scheduling")
304:
         if args.latent_consistency:
305:
            print("? Cross-modal Latent Consistency Learning")
306:
         if args.graph_constraints:
307:
            print("? Graph-based Topology Constraints")
308:
        print("="*80)
309:
310:
         # Create datasets
```

```
print("\nLoading datasets...")
311:
312:
        train_dataset = AdvancedFloorPlanDataset(
313:
             args.data_dir, split="train", augment=True
314:
315:
        val_dataset = AdvancedFloorPlanDataset(
             args.data_dir, split="val", augment=False
316:
317:
        )
318:
319:
        print(f"Train samples: {len(train_dataset)}")
         print(f"Validation samples: {len(val_dataset)}")
320:
321:
322:
         if len(train_dataset) == 0:
323:
             print("Error: No training samples found!")
             return
324:
325:
326:
         # Create data loaders
327:
         train_loader = DataLoader(
328:
             train_dataset,
329:
            batch_size=args.batch_size,
330:
            shuffle=True,
331:
            num_workers=args.num_workers,
332:
            pin memory=True,
333:
            drop_last=True,
            persistent_workers=args.persistent_workers if args.num_workers > 0 else False,
334:
335:
            prefetch_factor=args.prefetch_factor if args.num_workers > 0 else None
336:
        )
337:
        val_loader = DataLoader(
338:
339:
             val_dataset,
340:
            batch_size=max(1, args.batch_size),
341:
            shuffle=False,
342:
            num_workers=max(1, args.num_workers // 2),
343:
            pin_memory=True,
344:
            drop_last=False,
            persistent_workers=args.persistent_workers if args.num_workers > 0 else False,
345:
346:
            prefetch_factor = args.prefetch_factor if args.num_workers > 0 else None
347:
348:
349:
         # Create enhanced model
350:
        print("\nInitializing enhanced model...")
351:
        model = create_enhanced_model(args)
352:
353:
         # Create appropriate trainer
354:
         if args.training_mode == "adaptive":
355:
            print("\nUsing Adaptive Multi-Stage Trainer with Novel Strategies")
356:
             trainer = AdaptiveMultiStageTrainer(
357:
                model=model,
358:
                 train_loader=train_loader,
359:
                 val_loader=val_loader,
360:
                 device=device,
361:
                 config=config
362:
             )
363:
         else:
364:
           print("\nUsing Traditional Multi-Stage Trainer")
365:
            trainer = MultiStageTrainer(
366:
                model=model,
367:
                 train_loader=train_loader,
368:
                 val_loader=val_loader,
369:
                 device=device,
370:
                 config=config
371:
             )
372:
373:
         # Resume from checkpoint if specified
374:
         if args.resume:
375:
             print(f"Resuming from checkpoint: {args.resume}")
376:
             trainer.load_checkpoint(args.resume)
377:
378:
         # Run training
         if args.training_mode == "adaptive" or args.stage == "all":
379:
             print("\nStarting adaptive multi-stage training with novel strategies...")
380:
381:
             history = trainer.train_all_stages()
382:
         else:
383:
             # Traditional single-stage training
```

```
384:
            stage num = int(args.stage)
385:
            print(f"Training Stage {stage_num} only...")
386:
            if stage_num == 1:
387:
                 trainer.train_stage1()
            elif stage_num == 2:
388:
389:
                trainer.train_stage2()
390:
             elif stage_num == 3:
391:
                trainer.train_stage3()
392:
            history = trainer.history
393:
         # Save final model
394:
        final_model_path = output_dir / "final_enhanced_model.pth"
395:
        if hasattr(trainer, '_save_checkpoint'):
397:
             trainer._save_checkpoint(str(final_model_path))
398:
        print(f"Final model saved to: {final_model_path}")
399:
400:
        # Create comprehensive visualizations
401:
        print("\nGenerating training analysis...")
402:
        visualize_training_results(history, output_dir)
404:
         # Save training summary
        save_training_summary(history, config, output_dir)
405:
406:
407:
        print(f"\n? Enhanced training completed successfully!")
        print(f"? Results saved to: {output_dir}")
408:
        print("\nNovel contributions implemented:")
409:
410:
        print("- Dynamic curriculum learning with adaptive stage transitions")
        print("- Multi-objective optimization with gradient-based reweighting")
411:
        print("- Topology-aware progressive constraint injection")
412:
413:
       print("- Cross-modal latent consistency learning")
414:
        print("- Graph-based architectural constraint learning")
415:
416:
417: if __name__ == "__main__":
418:
        main()
```

■ File: training\losses.py

```
______
 1: """
 2: Advanced loss functions for multi-task training with dynamic weighting
 3: Enhanced with cross-modal consistency, graph constraints, and GradNorm
 4: Modified to support conditional geometric losses via run_full_geometric flag
 5: FIXED: Dynamic loss component initialization for stage transitions
 6: """
 7:
 8: import torch
 9: import torch.nn as nn
10: import torch.nn.functional as F
11: import cv2
12: import numpy as np
13: from typing import Dict, Optional, Tuple, List
14: import networkx as nx
15:
16:
17: class DynamicLossWeighter:
     def __init__(self, loss_names: List[str], alpha: float = 0.12, device: str = 'cuda'):
18:
 19:
           self.loss_names = loss_names
 20:
           self.alpha = alpha
 21:
           self.device = device
 22:
 23:
            # Initialize weights for all known loss components
 24:
            self.weights = {name: 1.0 for name in loss_names}
            self.initial_task_losses = {}
 25:
 26:
            # Add running normalization to prevent raw magnitude issues
 27:
            self.running_mean_losses = {name: 0.0 for name in loss_names}
 28:
            self.running_std_losses = {name: 1.0 for name in loss_names} # NEW
 29:
            self.update_count = 0
30:
 31:
            print(f"[DynamicWeighter] Initialized with loss components: {loss_names}")
 32:
 33:
        def update_weights(self, task_losses: Dict[str, torch.Tensor],
```

```
34:
                           shared_parameters, update_freq: int = 10):
 35:
             """Update loss weights using GradNorm with stability improvements"""
 36:
             if self.update_count % update_freq != 0:
 37:
                 self.update_count += 1
 38:
                 return self.weights
 39:
 40:
             # Initialize tracking for new loss components
 41:
             newly_initialized = []
 42:
             for name, loss in task_losses.items():
 43:
                 if name not in self.initial_task_losses:
 44:
                     loss_val = loss.item() if torch.is_tensor(loss) else float(loss)
 45:
                     # Use log-scale initialization for stability
 46:
                     self.initial_task_losses[name] = max(np.log(abs(loss_val) + 1e-6), -10.0)
 47:
 48:
                     if name not in self.weights:
 49:
                         self.weights[name] = 1.0
                     if name not in self.running_mean_losses:
 50:
 51:
                         self.running_mean_losses[name] = loss_val
 52:
                     if name not in self.running_std_losses:
 53:
                         self.running_std_losses[name] = max(abs(loss_val), 1e-3)
 54:
 55:
                     newly_initialized.append(name)
 56:
 57:
             # Update running statistics with EMA
             for name, loss in task_losses.items():
 58:
                 loss_val = loss.item() if torch.is_tensor(loss) else float(loss)
 59:
 60:
                 if name in self.running_mean_losses:
 61:
                     # Exponential moving average for mean and std
 62:
                     self.running_mean_losses[name] = 0.9 * self.running_mean_losses[name] + 0.1 *
                              loss_val
 63:
 64:
                     # Update running std using Welford's algorithm
                     delta = loss_val - self.running_mean_losses[name]
 65:
                     self.running_std_losses[name] = 0.9 * self.running_std_losses[name] + 0.1 *
 66:
                              abs(delta)
 67:
                     self.running_std_losses[name] = max(self.running_std_losses[name], 1e-3)
 68:
 69:
             # Calculate normalized relative decrease rates
 70:
             loss_ratios = {}
 71:
             for name, loss in task_losses.items():
 72:
                 if name in self.initial_task_losses and self.initial_task_losses[name] > -9.0:
 73:
                     # Normalize current loss by running statistics
 74:
                     current_loss = self.running_mean_losses.get(name, loss.item())
 75:
                     normalized_current = current_loss / (self.running_std_losses[name] + 1e-6)
 76:
 77:
                     initial_loss = self.initial_task_losses[name]
 78:
                     # Use log-space ratios for stability
 79:
                     loss_ratios[name] = np.exp(min(max(normalized_current - initial_loss, -5.0), 5.0))
 80:
 81:
             if not loss_ratios:
 82:
                 self.update_count += 1
 83:
                 return self.weights
 84:
 85:
             # Calculate gradient norms with improved stability
 86:
             grad_norms = {}
 87:
             for name, loss in task_losses.items():
 88:
                 if name in loss_ratios:
 89:
                     if not torch.is_tensor(loss) or not loss.requires_grad:
 90:
                         continue
                     if not torch.isfinite(loss):
 91:
 92:
                         continue
 93:
 94:
                     try:
                         grads = torch.autograd.grad(
 95:
 96:
                             loss, shared_parameters,
 97:
                             retain_graph=True, create_graph=False, allow_unused=True
 98:
 99:
100:
                         grad_norm_sq = 0.0
101:
                         valid_grads = False
102:
                         for grad in grads:
103:
                              if grad is not None and torch.isfinite(grad).all():
104:
                                  # Apply gradient norm stabilization
```

```
105:
                                 clipped_grad = torch.clamp(grad, -10.0, 10.0)
106:
                                 grad_norm_sq += clipped_grad.norm().item() ** 2
107:
                                 valid_grads = True
108:
109:
                         if valid_grads and grad_norm_sq > 0:
110:
                             # Use log-scale gradient norms
111:
                             grad_norms[name] = np.log(grad_norm_sq ** 0.5 + 1e-8)
112:
113:
                     except Exception as e:
114:
                         continue
115:
116:
             if not grad_norms:
117:
                 self.update_count += 1
112:
                 return self.weights
119:
120:
             # Normalize gradient norms
121:
             mean_grad_norm = np.mean(list(grad_norms.values()))
122:
123:
             # Update weights with improved stability
124:
             for name in grad_norms.keys():
125:
                 if name in loss_ratios:
126:
                     # Calculate target gradient in log space
127:
                     target_grad_log = mean_grad_norm + self.alpha * np.log(loss_ratios[name] + 1e-8)
128:
                     current_grad_log = grad_norms[name]
129:
130:
                     # Calculate weight update with damping
131:
                     weight_update_log = target_grad_log - current_grad_log
                     weight_update = np.exp(np.clip(weight_update_log, -1.0, 1.0)) # Stronger clipping
132:
133:
134:
                     # Apply update with momentum and stronger constraints
                     current_weight = self.weights.get(name, 1.0)
135:
136:
                     new_weight = 0.8 * current_weight + 0.2 * weight_update # More conservative
137:
                     self.weights[name] = float(np.clip(new_weight, 0.1, 2.0)) # Tighter bounds
138:
             self.update_count += 1
139:
140:
             return self.weights
141:
142: class GraphTopologyExtractor:
143:
         """Extracts graph structure from segmentation for topology constraints"""
144:
145:
        @staticmethod
146:
         def extract_room_graph(segmentation: torch.Tensor) -> Dict[str, torch.Tensor]:
             """Extract room connectivity graph from segmentation mask"""
148:
             B, C, H, W = segmentation.shape
149:
             device = segmentation.device
150:
151:
             # Get room predictions (assume classes: 0=bg, 1=wall, 2=door, 3=window, 4=room)
152:
             room_probs = F.softmax(segmentation, dim=1)
             \verb|room_mask| = \verb|room_probs[:, 4]| if C > 4 else torch.zeros((B, H, W), device=device)|
153:
154:
             wall_mask = room_probs[:, 1] if C > 1 else torch.zeros((B, H, W), device=device)
155:
156:
             # Simple connectivity: rooms connected if they share wall boundary
157:
             adjacency_matrices = []
             room_features = []
158:
159:
160:
             for b in range(B):
                 room_b = room_mask[b].detach().cpu().numpy()
161:
162:
                 wall_b = wall_mask[b].detach().cpu().numpy()
163:
164:
                 # Find connected components (rooms)
165:
                 trv:
166:
                     from scipy import ndimage
167:
                     labeled_rooms, num_rooms = ndimage.label(room_b > 0.5)
168:
169:
                     # Create adjacency matrix
170:
                     adj_matrix = np.zeros((max(num_rooms, 1), max(num_rooms, 1)))
171:
                     room_centroids = []
172:
173:
                     for i in range(1, num_rooms + 1):
174:
                         room_i_mask = (labeled_rooms == i)
175:
                         if np.sum(room_i_mask) > 0:
176:
                             centroid = ndimage.center of mass(room i mask)
177:
                             room_centroids.append(centroid)
```

```
178:
179:
                             # Check connectivity to other rooms through walls
180:
                             for j in range(i + 1, num_rooms + 1):
                                 room_j_mask = (labeled_rooms == j)
181:
182:
                                 if np.sum(room_j_mask) > 0:
183:
                                     # Check if rooms are connected via wall adjacency
184:
                                      connectivity = GraphTopologyExtractor._check_room_connectivity(
185:
                                         room_i_mask, room_j_mask, wall_b
186:
187:
                                      adj_matrix[i-1, j-1] = connectivity
188:
                                     adj_matrix[j-1, i-1] = connectivity
189:
190:
                     # Convert to tensor
191:
                     adj_tensor = torch.from_numpy(adj_matrix).float().to(device)
192:
                     centroids_tensor = torch.from_numpy(np.array(room_centroids) if room_centroids else
                              np.zeros((0, 2))).float().to(device)
193:
194:
                 except ImportError:
195:
                     # Fallback if scipy not available
196:
                     adj_tensor = torch.zeros((1, 1), device=device)
197:
                     centroids_tensor = torch.zeros((0, 2), device=device)
198:
                 except Exception as e:
199:
                     # General fallback for any other issues
200:
                     print(f"Warning: Graph extraction failed: {e}")
201:
                     adj_tensor = torch.zeros((1, 1), device=device)
                     centroids_tensor = torch.zeros((0, 2), device=device)
202:
203:
204:
                 adjacency_matrices.append(adj_tensor)
205:
                 room_features.append(centroids_tensor)
206:
207:
             return {
208:
                 "adjacency_matrices": adjacency_matrices,
209:
                 "room_features": room_features
             }
210:
211:
212:
         @staticmethod
         def _check_room_connectivity(room1_mask, room2_mask, wall_mask):
213:
214:
             """Check if two rooms are connected through walls"""
215:
216:
                 from scipy.ndimage import binary_dilation
217:
218:
                 # Dilate room masks to check wall adjacency
                 dilated1 = binary_dilation(room1_mask, iterations=2)
220:
                 dilated2 = binary_dilation(room2_mask, iterations=2)
221:
222:
                 # Check overlap through wall areas
223:
                 wall_overlap = (dilated1 & dilated2) & (wall_mask > 0.3)
224:
                 return float(np.sum(wall_overlap) > 0)
225:
             except ImportError:
226:
                 # Simple distance-based fallback
227:
                 return 0.0
228:
229:
230: class ResearchGradeLoss(nn.Module):
231:
232:
        Multi-task loss with stage-aware dynamic weighting:
         - Stage 1: segmentation, dice, sdf, attributes, topology, graph
233:
234:
         - Stage 2: + polygon (DVX)
235:
         - Stage 3: + voxel, latent_consistency (full geometric)
236:
237:
         FIXED: Dynamic initialization handles new loss components during stage transitions
238:
239:
240:
         def __init__(
241:
             self,
242:
             seg_weight: float = 1.0,
243:
             dice_weight: float = 1.0,
244:
            sdf_weight: float = 0.5,
245:
             attr_weight: float = 1.0,
            polygon_weight: float = 1.0,
246:
247:
             voxel_weight: float = 1.0,
248:
             topology weight: float = 0.5,
249:
             latent_consistency_weight: float = 0.5,
```

```
250:
             graph_constraint_weight: float = 0.3,
251:
             enable_dynamic_weighting: bool = True,
252:
             gradnorm_alpha: float = 0.12,
             device: str = 'cuda',
253:
             weight_update_freq: int = 10,
254:
255:
             weight_momentum: float = 0.9,
       ):
256:
257:
             super().__init__()
258:
259:
             # Store initial weights for all possible loss components
260:
             self.initial_weights = {
261:
                 'seg': float(seg_weight),
262:
                 'dice': float(dice_weight),
                 'sdf': float(sdf_weight),
263:
264:
                 'attr': float(attr weight),
265:
                 'polygon': float(polygon_weight),
266:
                 'voxel': float(voxel_weight),
267:
                 'topology': float(topology_weight),
268:
                 'latent_consistency': float(latent_consistency_weight),
269:
                 'graph': float(graph_constraint_weight)
             }
270:
271:
272:
             # Current weights (will be dynamically updated)
273:
             self.weights = self.initial_weights.copy()
274:
275:
             # Core losses
276:
             self.ce_loss = nn.CrossEntropyLoss()
277:
             self.mse_loss = nn.MSELoss()
278:
             self.l1_loss = nn.L1Loss()
279:
             self.cosine_loss = nn.CosineEmbeddingLoss()
280:
281:
             # Dynamic weighting with all possible loss names
282:
             self.enable_dynamic_weighting = enable_dynamic_weighting
283:
             if enable_dynamic_weighting:
284:
                 all_loss_names = list(self.initial_weights.keys())
285:
                 self.loss_weighter = DynamicLossWeighter(
286:
                     all_loss_names, alpha=gradnorm_alpha, device=device,
287:
288:
                 self.update_freq = weight_update_freq
289:
                 self.momentum = weight_momentum
290:
                 print(f"[ResearchGradeLoss] Dynamic weighting enabled for: {all_loss_names}")
291:
292:
             self.device = device
293:
294:
         def update_loss_weights(self, new_weights: Dict[str, float]):
295:
             """Update loss weights (called by trainer for curriculum scheduling)"""
296:
             for key, value in new weights.items():
297:
                 if key in self.weights:
298:
                     self.weights[key] = float(value)
299:
300:
         def forward(self, predictions: dict, targets: dict, shared_parameters=None,
                 run_full_geometric=True):
             """Compute multi-task loss with proper normalization and aggregation"""
301:
302:
             # Input validation and sanitization
303:
             predictions = self._sanitize_predictions(predictions)
304:
             targets = self._sanitize_targets(targets)
305:
306:
             device = self._get_device_from_inputs(predictions, targets)
307:
             losses = {}
308:
             total_loss = torch.tensor(0.0, device=device, requires_grad=True)
309:
310:
             # ---- STAGE 1 LOSSES with proper scaling ----
311:
             if "segmentation" in predictions and "mask" in targets:
312:
                 seg_pred = predictions["segmentation"]
                 seg_target = targets["mask"].long()
313:
314:
315:
                 # Scale CE loss by number of pixels for consistency
316:
                 ce_loss = self.ce_loss(seg_pred, seg_target)
317:
                 losses["seg"] = ce_loss
318:
319:
                 dice_loss = self._dice_loss(seg_pred, seg_target)
320:
                 losses["dice"] = dice_loss
321:
```

```
if "sdf" in predictions and "mask" in targets:
322:
323:
                 sdf_pred = predictions["sdf"]
324:
                 sdf_pred = torch.clamp(sdf_pred, -1.0, 1.0)
325:
                 sdf_target = self._mask_to_sdf(targets["mask"])
326:
                 sdf_target = sdf_target.to(sdf_pred.device).type_as(sdf_pred)
327:
                 # Normalize SDF loss by spatial dimensions
328:
                 sdf_loss = self.mse_loss(sdf_pred, sdf_target)
329:
                 losses["sdf"] = sdf_loss
330:
             if "attributes" in predictions and "attributes" in targets:
331:
                 pred_attr = predictions["attributes"].float()
332:
333:
                 tgt_attr = targets["attributes"].float().to(pred_attr.device)
334:
                 # Normalize attribute loss by number of attributes
335:
                 attr_loss = self.l1_loss(pred_attr, tgt_attr) / pred_attr.shape[-1]
336:
                 losses["attr"] = attr_loss
337:
338:
             # Apply proper scaling to topology losses
339:
             if "segmentation" in predictions:
340:
                 topology_loss = self._topology_loss(predictions["segmentation"])
341:
                 # Scale topology loss to reasonable magnitude
342:
                 losses["topology"] = topology_loss * 0.5
343:
344:
                 graph_loss = self._graph_topology_loss(predictions["segmentation"])
345:
                 # Graph loss is already scaled in the function above
                 losses["graph"] = graph_loss
346:
347:
348:
             # ---- GEOMETRIC LOSSES with normalization ----
349:
             if run_full_geometric:
350:
                 if ("polygons" in predictions and predictions["polygons"] is not None and
351:
                     "polygons_gt" in targets):
352:
                     poly_loss = self._polygon_loss(predictions, targets["polygons_gt"])
353:
                     # Normalize polygon loss by number of polygons and points
354:
                     if "polygons" in predictions and predictions["polygons"] is not None:
355:
                         B, P, N, _ = predictions["polygons"].shape
                         poly_loss = poly_loss / (P * N) # Normalize by polygon complexity
356:
357:
                     losses["polygon"] = poly_loss
                 else:
358:
359:
                     losses["polygon"] = torch.tensor(0.0, device=device)
360:
361:
                 if ("voxels_pred" in predictions and predictions["voxels_pred"] is not None and
362:
                      "voxels_gt" in targets):
                     pred_vox = predictions["voxels_pred"].float()
363:
364:
                     tgt_vox = targets["voxels_gt"].float().to(pred_vox.device)
365:
                     voxel_loss = self._voxel_iou_loss(pred_vox, tgt_vox)
366:
                     losses["voxel"] = voxel_loss
367:
                 else:
368:
                     losses["voxel"] = torch.tensor(0.0, device=device)
369:
370:
                 if ("latent_2d_embedding" in predictions and "latent_3d_embedding" in predictions and
371:
                     predictions["latent_2d_embedding"] is not None and
                              predictions["latent_3d_embedding"] is not None):
372:
                     consistency_loss = self._latent_consistency_loss(
373:
                         predictions["latent_2d_embedding"],
374:
                         predictions["latent_3d_embedding"]
375:
376:
                     losses["latent_consistency"] = consistency_loss
377:
378:
                     losses["latent_consistency"] = torch.tensor(0.0, device=device)
379:
             else:
                 losses["polygon"] = torch.tensor(0.0, device=device)
380:
381:
                 losses["voxel"] = torch.tensor(0.0, device=device)
382:
                 losses["latent_consistency"] = torch.tensor(0.0, device=device)
383:
384:
             # ---- IMPROVED WEIGHTING AND AGGREGATION ----
385:
             active_losses = {
386:
                 name: loss for name, loss in losses.items()
387:
                 if isinstance(loss, torch.Tensor) and loss.requires_grad and loss.item() > 1e-8
388:
             }
389:
390:
             if self.enable_dynamic_weighting and shared_parameters is not None and active_losses:
391:
392:
                     dynamic_weights = self.loss_weighter.update_weights(
393:
                         active_losses, shared_parameters, self.update_freq
```

```
394:
                     )
395:
396:
                     # Apply weights with additional stability checks
397:
                     for name, loss in losses.items():
398:
                         if name in self.weights and isinstance(loss, torch.Tensor) and
                                   torch.isfinite(loss):
399:
                             weight = dynamic_weights.get(name, self.weights[name])
400:
                              # Apply weight with gradient scaling for stability
401:
                              weighted_loss = weight * loss
402:
                             if torch.isfinite(weighted loss):
403:
                                  total_loss = total_loss + weighted_loss
404:
405:
                 except Exception as e:
                     print(f"[ResearchGradeLoss] Dynamic weighting failed: {e}, falling back to static
406:
                              weights")
407:
                     # Fallback to static weights
408:
                     for name, loss in losses.items():
409:
                         if name in self.weights and isinstance(loss, torch.Tensor) and
                                   torch.isfinite(loss):
410:
                              total_loss = total_loss + self.weights[name] * loss
411:
             else:
412:
                 # Static weights with stability
413:
                 for name, loss in losses.items():
414:
                     if name in self.weights and isinstance(loss, torch.Tensor) and torch.isfinite(loss):
                         total_loss = total_loss + self.weights[name] * loss
415:
416:
417:
             # Final loss scaling and validation
             total_loss = torch.clamp(total_loss, 0.0, 100.0) # Prevent explosion
418:
419:
420:
             if not torch.isfinite(total_loss):
421:
                 print("[ResearchGradeLoss] Warning: Non-finite total loss detected, using fallback")
422:
                 total_loss = torch.tensor(1.0, device=device, requires_grad=True)
423:
             losses["total"] = total_loss
424:
425:
             return total_loss, losses
426:
427:
         def __call__(self, predictions: dict, targets: dict, shared_parameters=None,
                 run_full_geometric=True):
             """Trainer compatibility method"""
428:
429:
             return self.forward(predictions, targets, shared_parameters, run_full_geometric)
430:
         def _sanitize_predictions(self, predictions: dict) -> dict:
431:
432:
             """Sanitize prediction tensors"""
433:
             sanitized = {}
434:
             for name, tensor in predictions.items():
435:
                 if torch.is_tensor(tensor):
436:
                     if torch.isnan(tensor).any() or torch.isinf(tensor).any():
437:
                         print(f"WARNING: NaN/Inf in predictions[{name}] - zeroing out")
                         sanitized[name] = torch.zeros_like(tensor)
438:
439:
                     else:
440:
                         sanitized[name] = tensor
441:
                 else:
442:
                     sanitized[name] = tensor
443:
             return sanitized
444:
         def _sanitize_targets(self, targets: dict) -> dict:
445:
446:
             """Sanitize target tensors"""
447:
             sanitized = {}
448:
             for name, tensor in targets.items():
449:
                 if torch.is_tensor(tensor):
450:
                     if torch.isnan(tensor).any() or torch.isinf(tensor).any():
451:
                         print(f"WARNING: NaN/Inf in targets[{name}] - zeroing out")
452:
                         sanitized[name] = torch.zeros_like(tensor)
453:
                     else:
454:
                         sanitized[name] = tensor
455:
                 else:
456:
                     sanitized[name] = tensor
457:
             return sanitized
458:
459:
         def _get_device_from_inputs(self, predictions, targets):
             """Helper to determine device from inputs"""
460:
461:
             for pred_dict in [predictions, targets]:
462:
                 for value in pred_dict.values():
```

```
463:
                     if torch is tensor(value):
                         return value.device
464:
465:
            return self.device
466:
         # ---- LOSS COMPONENT IMPLEMENTATIONS ----
467:
468:
469:
         def _latent_consistency_loss(self, embedding_2d: torch.Tensor, embedding_3d: torch.Tensor) ->
                 torch.Tensor:
470:
             """Cross-modal latent consistency loss"""
             if embedding_2d.shape != embedding_3d.shape:
471:
                 min_dim = min(embedding_2d.shape[-1], embedding_3d.shape[-1])
472:
                 embedding_2d = embedding_2d[..., :min_dim]
473:
474:
                 embedding_3d = embedding_3d[..., :min_dim]
475:
            target = torch.ones(embedding_2d.shape[0], device=embedding_2d.device)
476:
477:
             cosine_loss = self.cosine_loss(embedding_2d, embedding_3d, target)
             12_loss = F.mse_loss(embedding_2d, embedding_3d)
478:
479:
480:
             return 0.7 * cosine_loss + 0.3 * 12_loss
481:
482:
         def _graph_topology_loss(self, segmentation_logits: torch.Tensor) -> torch.Tensor:
483:
              """Graph-based topology constraints with proper normalization"""
484:
485:
                 graph_data = GraphTopologyExtractor.extract_room_graph(segmentation_logits)
486:
                 device = segmentation_logits.device
487:
488:
                 total_graph_loss = torch.tensor(0.0, device=device)
489:
                 batch_size = segmentation_logits.shape[0]
490:
                 valid_batches = 0
491:
492:
                 for b in range(batch_size):
493:
                     if b < len(graph_data["adjacency_matrices"]):</pre>
494:
                         adj_matrix = graph_data["adjacency_matrices"][b]
                         if adj_matrix.numel() == 0:
495:
                             continue
496:
497:
498:
                         # Normalize by matrix size to prevent scale explosion
499:
                         matrix_size = max(adj_matrix.shape[0], 1)
500:
                         norm_factor = 1.0 / (matrix_size + 1e-6)
501:
502:
                         degrees = adj_matrix.sum(dim=1)
503:
                         isolation_penalty = torch.exp(-degrees).mean() * norm_factor
504:
505:
                         max_reasonable_connections = min(4, adj_matrix.shape[0] - 1)
506:
                         over_connection_penalty = F.relu(degrees - max_reasonable_connections).mean() *
                                  norm_factor
507:
508:
                         if b < len(graph_data["room_features"]) and</pre>
                                  graph_data["room_features"][b].numel() > 0:
509:
                             room_features = graph_data["room_features"][b]
                             if room_features.shape[0] > 1:
510:
511:
                                 feature_distances = torch.cdist(room_features, room_features)
512:
                                 # Normalize distance computation
513:
                                 mean distance = feature distances.mean()
514:
                                 normalized_distances = feature_distances / (mean_distance + 1e-6)
                                 smoothness_loss = (adj_matrix * normalized_distances).sum() /
515:
                                          (adj_matrix.sum() + 1e-6)
516:
                                 smoothness_loss = smoothness_loss * norm_factor
                             else:
517:
518:
                                 smoothness_loss = torch.tensor(0.0, device=device)
519:
                         else:
520:
                             smoothness_loss = torch.tensor(0.0, device=device)
521:
522:
                         # Apply strong penalty scaling to keep graph loss in reasonable range
                         batch_graph_loss = (0.4 * isolation_penalty +
523:
524:
                                          0.3 * over_connection_penalty +
                                           0.3 * smoothness_loss) * 0.1 \# Scale down by 10x
525:
526:
527:
                         total_graph_loss = total_graph_loss + batch_graph_loss
528:
                         valid batches += 1
529:
530:
                 # Average over valid batches and apply final normalization
531:
                 if valid_batches > 0:
```

```
532:
                     return total_graph_loss / valid_batches
533:
534:
                     return torch.tensor(0.0, device=segmentation_logits.device)
535:
536:
             except Exception as e:
537:
                 return torch.tensor(0.0, device=segmentation_logits.device)
538:
539:
         def _dice_loss(self, pred: torch.Tensor, target: torch.Tensor, smooth: float = 1e-6) ->
                  torch.Tensor:
             """Dice loss implementation"""
540:
541:
             pred_soft = F.softmax(pred, dim=1)
542:
             B, C = pred_soft.shape[:2]
543:
544:
             dice_losses = []
545:
             for c in range(C):
546:
                 pred_c = pred_soft[:, c, :, :]
547:
                 target_c = (target == c).float().to(pred_c.device)
548:
                 intersection = (pred_c * target_c).view(B, -1).sum(dim=1)
                 \label{eq:union} \verb|union| = pred_c.view(B, -1).sum(dim=1) + target_c.view(B, -1).sum(dim=1) \\
549:
550:
                 dice = (2.0 * intersection + smooth) / (union + smooth)
551:
                 dice_losses.append((1.0 - dice).mean())
552:
553:
             return torch.stack(dice_losses).mean()
554:
         def _mask_to_sdf(self, mask: torch.Tensor) -> torch.Tensor:
555:
556:
              """Convert mask to SDF"""
557:
             device = mask.device if torch.is_tensor(mask) else self.device
558:
             if not torch.is_tensor(mask):
559:
                 mask = torch.tensor(mask, device=device)
560:
561:
             B, H, W = mask.shape
562:
             sdf = torch.zeros((B, 1, H, W), dtype=torch.float32, device=device)
563:
564:
             for b in range(B):
                 mask_np = mask[b].detach().cpu().numpy().astype(np.uint8)
565:
566:
                 try:
567:
                     dist_inside = cv2.distanceTransform((mask_np > 0).astype(np.uint8), cv2.DIST_L2, 5)
568:
                     dist_outside = cv2.distanceTransform((mask_np == 0).astype(np.uint8), cv2.DIST_L2,
                              5)
569:
                     sdf_np = dist_inside.astype(np.float32) - dist_outside.astype(np.float32)
570:
                     sdf_np = np.tanh(sdf_np / 10.0).astype(np.float32)
571:
                     sdf[b, 0] = torch.from_numpy(sdf_np)
572:
                 except Exception:
573:
                     sdf[b, 0] = torch.zeros_like(mask[b].float())
574:
575:
             return sdf
576:
577:
         def _polygon_loss(self, predictions: dict, targets: dict) -> torch.Tensor:
              """Polygon/DVX loss"""
578:
579:
             pred_polys = predictions.get("polygons")
             tgt_polys = targets.get("polygons")
580:
581:
             valid_mask = targets.get("valid_mask")
582:
583:
             if pred_polys is None or tgt_polys is None:
584:
                 return torch.tensor(0.0, device=pred_polys.device if pred_polys is not None else
                          self.device)
585:
586:
             pred_polys = pred_polys.float()
587:
             tgt_polys = tgt_polys.float().to(pred_polys.device)
588:
589:
             point_loss = self.mse_loss(pred_polys, tgt_polys)
590:
591:
             pred_valid = predictions.get("polygon_validity")
592:
             if pred_valid is None or valid_mask is None:
593:
                 validity_loss = torch.tensor(0.0, device=pred_polys.device)
594:
595:
                 pred_valid = pred_valid.float().to(pred_polys.device)
596:
                 valid_mask_f = valid_mask.float().to(pred_polys.device)
597:
                 validity_loss = self.mse_loss(pred_valid, valid_mask_f)
598:
599:
             smoothness_loss = self._polygon_smoothness(pred_polys)
600:
             rect_loss = self._rectilinearity_loss(pred_polys)
601:
```

```
602:
             return point_loss + 0.1 * validity_loss + 0.05 * smoothness_loss + 0.1 * rect_loss
603:
604:
         def _polygon_smoothness(self, polygons: torch.Tensor) -> torch.Tensor:
605:
              """Polygon smoothness loss""
606:
             if polygons is None or polygons.numel() == 0:
607:
                 return torch.tensor(0.0, device=polygons.device if polygons is not None else
                          self.device)
608:
609:
             p1 = polygons
610:
             p2 = torch.roll(polygons, -1, dims=2)
611:
            p3 = torch.roll(polygons, -2, dims=2)
612:
             curvature = torch.norm(p1 - 2.0 * p2 + p3, dim=-1)
613:
             return curvature.mean()
614:
615:
        def _rectilinearity_loss(self, polygons: torch.Tensor) -> torch.Tensor:
616:
             """Encourage axis-aligned structure"""
617:
             if polygons is None or polygons.numel() == 0:
618:
                 return torch.tensor(0.0, device=polygons.device if polygons is not None else
                          self.device)
619:
620:
             edges = torch.roll(polygons, -1, dims=2) - polygons
621:
             edge_norms = torch.norm(edges, dim=-1, keepdim=True)
622:
             edges_normalized = edges / (edge_norms + 1e-6)
623:
624:
             edge1 = edges_normalized
             edge2 = torch.roll(edges_normalized, -1, dims=2)
625:
626:
             cos_angles = (edge1 * edge2).sum(dim=-1)
627:
            cos2 = cos_angles ** 2
628:
629:
            perp_penalty = cos2
             parallel_penalty = (\cos 2 - 1.0) ** 2
630:
631:
             angle_penalty = torch.minimum(perp_penalty, parallel_penalty)
632:
             return angle_penalty.mean()
633:
634:
         def _voxel_iou_loss(self, pred_voxels: torch.Tensor, target_voxels: torch.Tensor) ->
                 torch.Tensor:
             """3D voxel IoU loss"""
635:
636:
             pred_prob = torch.sigmoid(torch.clamp(pred_voxels, -10.0, 10.0))
637:
             target = target_voxels.float().to(pred_prob.device)
638:
639:
             intersection = (pred_prob * target).view(pred_prob.shape[0], -1).sum(dim=1)
             union = (pred_prob.view(pred_prob.shape[0], -1).sum(dim=1) +
640:
641:
                     target.view(target.shape[0], -1).sum(dim=1) - intersection)
642:
643:
             iou = (intersection + 1e-6) / (union + 1e-6)
644:
             return (1.0 - iou).mean()
645:
646:
         def _topology_loss(self, segmentation_logits: torch.Tensor) -> torch.Tensor:
647:
              """Traditional topology loss"""
648:
             seg_soft = F.softmax(segmentation_logits, dim=1)
649:
             C = seq soft.shape[1]
650:
             device = seg_soft.device
651:
             walls = seg_soft[:, 1] if C > 1 else torch.zeros_like(seg_soft[:, 0])
652:
653:
             doors = seg_soft[:, 2] if C > 2 else torch.zeros_like(walls)
             windows = seg_soft[:, 3] if C > 3 else torch.zeros_like(walls)
654:
655:
             door_wall_overlap = doors * walls
656:
657:
             window_wall_overlap = windows * walls
658:
659:
             door_penalty = torch.maximum(doors - door_wall_overlap, torch.zeros_like(doors))
660:
             window_penalty = torch.maximum(windows - window_wall_overlap, torch.zeros_like(windows))
661:
662:
             connectivity_loss = self._connectivity_loss(walls)
663:
664:
             return door_penalty.mean() + window_penalty.mean() + 0.1 * connectivity_loss
665:
666:
         def _connectivity_loss(self, wall_prob: torch.Tensor) -> torch.Tensor:
667:
              """Connectivity loss for walls"""
668:
             if wall_prob is None or wall_prob.numel() == 0:
                 return torch.tensor(0.0, device=wall_prob.device if wall_prob is not None else
669:
                          self.device)
670:
```

```
671:
             kernel = torch.ones((1, 1, 3, 3), device=wall_prob.device, dtype=wall_prob.dtype) / 9.0
672:
             neighbors = F.conv2d(wall_prob.unsqueeze(1), kernel, padding=1).squeeze(1)
673:
             isolation_penalty = wall_prob * torch.exp(-neighbors)
674:
675:
             return isolation_penalty.mean()
676:
677:
678: class LossScheduler:
679:
         """Manages curriculum-based loss weight scheduling"""
680:
681:
        def __init__(self, config):
682:
             self.config = config
683:
             self.loss_schedules = config.loss_schedule
684:
685:
        def get_scheduled_weights(self, current_stage: int, current_epoch: int,
686:
                                 stage_epoch: int, total_stage_epochs: int,
687:
                                 base_weights: Dict[str, float]) -> Dict[str, float]:
688:
             """Calculate loss weights based on curriculum schedule"""
689:
             scheduled_weights = base_weights.copy()
690:
691:
             for loss_name, schedule_type in self.loss_schedules.items():
692:
                 if loss_name not in scheduled_weights:
693:
                     continue
694:
695:
                 base_weight = scheduled_weights[loss_name]
696:
697:
                 if schedule_type == "static":
698:
                     continue
699:
700:
                 elif schedule_type == "progressive":
                     if loss_name == "topology":
701:
702:
                         start_weight = self.config.topology_start_weight
                         end_weight = self.config.topology_end_weight
703:
704:
                         ramp_epochs = self.config.topology_ramp_epochs
705:
                         progress = min(current_epoch / ramp_epochs, 1.0)
706:
                         scheduled_weights[loss_name] = start_weight + progress * (end_weight -
                                  start_weight)
707:
708:
                 elif schedule_type == "linear_ramp":
709:
                     progress = stage_epoch / max(total_stage_epochs, 1)
710:
                     scheduled_weights[loss_name] = base_weight * progress
711:
712:
                 elif schedule_type == "exponential":
713:
                     progress = stage_epoch / max(total_stage_epochs, 1)
714:
                     scheduled_weights[loss_name] = base_weight * (progress ** 2)
715:
                 elif schedule_type == "early_decay":
716:
717:
                     if current_stage > 1:
718:
                         scheduled_weights[loss_name] = base_weight * 0.3
719:
720:
                 elif schedule_type == "staged_ramp":
721:
                     if current_stage == 2:
722:
                         progress = stage_epoch / max(total_stage_epochs, 1)
723:
                         scheduled_weights[loss_name] = base_weight * progress
724:
                     elif current_stage < 2:</pre>
725:
                         scheduled_weights[loss_name] = 0.0
726:
727:
                 elif schedule_type == "late_ramp":
728:
                     if current_stage == 3:
                         progress = stage_epoch / max(total_stage_epochs, 1)
729:
730:
                         scheduled_weights[loss_name] = base_weight * progress
731:
                     elif current_stage < 3:</pre>
732:
                         scheduled_weights[loss_name] = 0.0
733:
                 elif schedule_type == "mid_ramp":
734:
735:
                     if current_stage >= 2:
736:
                         if current_stage == 2:
737:
                             progress = min(stage_epoch / (total_stage_epochs * 0.5), 1.0)
738:
                             scheduled_weights[loss_name] = base_weight * progress
739:
                         else:
740:
                             scheduled_weights[loss_name] = base_weight
741:
                     else:
742:
                         scheduled_weights[loss_name] = 0.0
```

```
743:
744:
                 elif schedule_type == "delayed_ramp":
745:
                     if current_epoch >= self.config.graph_start_epoch:
746:
                         epochs_since_start = current_epoch - self.config.graph_start_epoch
747:
                         ramp_duration = 50
748:
                         progress = min(epochs_since_start / ramp_duration, 1.0)
749:
                         scheduled_weights[loss_name] = self.config.graph_end_weight * progress
750:
                     else:
751:
                         scheduled_weights[loss_name] = 0.0
752:
            return scheduled_weights
```

■ File: training\trainer.py

```
______
 1: """
 2: Dynamic Multi-stage training system with adaptive curriculum learning
 3: Implements novel training strategies: dynamic stage transitions, topology-aware scheduling,
 4: multi-objective optimization, and cross-modal consistency learning
 5: """
 6:
 7: import torch
 8: import torch.nn.utils
 9:
10: # training/trainer.py - Fixed AMP imports
 11: from torch.amp import autocast, GradScaler
 12: import time
13: import numpy as np
14: import random
15: from pathlib import Path
 16: from tqdm import tqdm
17: from typing import Dict, List, Optional, Tuple
18: from collections import deque
 20: from .losses import ResearchGradeLoss, LossScheduler
 21: from config import DEFAULT_TRAINING_CONFIG, DEFAULT_LOSS_CONFIG, StageTransitionCriteria
 23:
 24: class CurriculumState:
 25:
        """Tracks curriculum learning state and metrics"""
 26:
 27:
       def __init__(self, config):
 28:
            self.config = config
 29:
 30:
            # Loss history for plateau detection
 31:
            self.loss_history = {
 32:
                "stagel": deque(maxlen=config.plateau_detection_window * 2),
 33:
                "stage2": deque(maxlen=config.plateau_detection_window * 2),
 34:
                "stage3": deque(maxlen=config.plateau_detection_window * 2),
35:
            }
 37:
            # Component loss tracking
 38:
            self.component losses = {
 39:
                "segmentation": deque(maxlen=20),
                "dice": deque(maxlen=20),
 40:
                "polygon": deque(maxlen=20),
 41:
                "voxel": deque(maxlen=20),
 42:
 43:
                "topology": deque(maxlen=20),
                "latent_consistency": deque(maxlen=20),
 44:
 45:
                "graph": deque(maxlen=20),
            }
 46:
 47:
 48:
            # Gradient magnitude tracking for dynamic weighting
 49:
            self.gradient_norms = {
 50:
                name: deque(maxlen=config.gradient_norm_window)
51:
                for name in self.component_losses.keys()
 52:
            }
53:
 54:
            # Stage transition tracking
 55:
            self.epochs_without_improvement = 0
 56:
            self.best_val_loss = float("inf")
 57:
            self.stage_transition_epochs = []
```

```
58:
 59:
             # Dynamic weights history
 60:
             self.weight_history = []
 61:
         def update_loss_history(self, stage: str, val_loss: float):
 62:
 63:
              """Update validation loss history for plateau detection"""
 64:
             if stage in self.loss_history:
 65:
                 self.loss_history[stage].append(val_loss)
 66:
 67:
             # Update improvement tracking
 68:
             if val_loss < self.best_val_loss:</pre>
 69:
                 self.best_val_loss = val_loss
 70:
                 self.epochs_without_improvement = 0
 71:
             else:
 72:
                 self.epochs_without_improvement += 1
 73:
 74:
         def update_component_losses(self, loss_components: Dict[str, float]):
 75:
              """Update individual loss component history"""
 76:
             for name, loss_val in loss_components.items():
 77:
                 if name in self.component_losses:
 78:
                     self.component_losses[name].append(loss_val)
 79:
 :08
         def should_transition(self, current_stage: int) -> bool:
             """Check if should transition to next stage"""
 81:
 82:
             if current_stage == 1:
                 val_losses = list(self.loss_history["stage1"])
 83:
 84:
                 return StageTransitionCriteria.should_transition_from_stage1(
 85:
                     [], val_losses, self.config
 86:
 87:
             elif current_stage == 2:
 88:
                 polygon_losses = list(self.component_losses["polygon"])
 89:
                 return StageTransitionCriteria.should_transition_from_stage2(
 90:
                     polygon_losses, self.config
 91:
 92:
 93:
             return False
 94:
 96: class AdaptiveMultiStageTrainer:
 97:
 98:
         Advanced multi-stage trainer with dynamic curriculum learning:
 99:
         - Adaptive stage transitioning based on performance plateaus
100:
         - Topology-aware loss scheduling
101:
         - Multi-objective optimization with dynamic weighting
102:
         - Cross-modal latent consistency learning
103:
         - Graph-based topology constraints
104:
105:
         # Class constant for rolling checkpoint path
106:
107:
         ROLLING_CHECKPOINT = "latest_checkpoint.pth"
108:
109:
         def __init__(self, model, train_loader, val_loader, device=None, config=None):
110:
             if config is None:
111:
                 config = DEFAULT_TRAINING_CONFIG
112:
             self.model = model.to(device or config.device)
113:
             self.train_loader = train_loader
114:
115:
             self.val_loader = val_loader
116:
             self.device = device or config.device
117:
             self.config = config
118:
119:
             # Initialize curriculum state
120:
             self.curriculum_state = CurriculumState(config.curriculum)
121:
             self.loss_scheduler = LossScheduler(config.curriculum)
122:
123:
             # Training state tracking for resume functionality
124:
             self.current_stage = 1
125:
             self.current_epoch = 0
126:
             self.global_epoch = 0
127:
             self.stage_epoch = 0
             self.stage_start_time = None
128:
129:
             self.epoch_times = []
130:
```

```
# Add AMP and optimization settings - Updated for new PyTorch API
131:
             self.use_amp = getattr(config, "use_mixed_precision", True)
132:
             self.scaler = GradScaler("cuda", enabled=self.use_amp)
133:
             self.accumulation_steps = getattr(config, "accumulation_steps", 1)
135:
             self.dvx_step_freq = getattr(config, "dvx_step_freq", 1)
136:
             self.voxel_size_stage = getattr(config, "voxel_size_stage", None)
             self.image_size_stage = getattr(config, "image_size_stage", None)
137:
138:
             self.\_step = 0
139:
140:
             # Enhanced optimizers with better hyperparameters
             self.optimizer_2d = torch.optim.AdamW(
142:
                 list(self.model.encoder.parameters())
143:
                 + list(self.model.seg_head.parameters())
144:
                 + list(self.model.attr_head.parameters())
145:
                 + list(self.model.sdf_head.parameters()),
146:
                 lr=config.stage1_lr,
147:
                 weight_decay=config.stagel_weight_decay,
                 betas=(0.9, 0.999),
             )
149:
150:
151:
             self.optimizer_dvx = torch.optim.AdamW(
152:
                 self.model.dvx.parameters(),
153:
                 lr=config.stage2_lr,
154:
                 weight_decay=config.stage2_weight_decay,
                 betas=(0.9, 0.999),
155:
             )
156:
157:
             self.optimizer_full = torch.optim.AdamW(
158:
159:
                 self.model.parameters(),
160:
                 lr=config.stage3_lr,
161:
                 weight_decay=config.stage3_weight_decay,
162:
                 betas=(0.9, 0.999),
163:
             # Enhanced learning rate schedulers
165:
166:
             if config.use_cosine_restarts:
167:
                 self.scheduler_2d = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(
168:
                     self.optimizer_2d, T_0=20, T_mult=1
169:
170:
                 self.scheduler_dvx = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(
171:
                     self.optimizer_dvx, T_0=15, T_mult=1
172:
173:
                 self.scheduler_full = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(
174:
                     self.optimizer_full, T_0=30, T_mult=1
175:
                 )
176:
             else:
177:
                self.scheduler_2d = torch.optim.lr_scheduler.CosineAnnealingLR(
178:
                     self.optimizer_2d, T_max=config.max_stage1_epochs
179:
                 self.scheduler_dvx = torch.optim.lr_scheduler.CosineAnnealingLR(
180:
181:
                     self.optimizer_dvx, T_max=config.max_stage2_epochs
182:
183:
                 self.scheduler_full = torch.optim.lr_scheduler.CosineAnnealingLR(
184:
                     self.optimizer_full, T_max=config.max_stage3_epochs
185:
186:
             # Enhanced loss function with dynamic weighting
187:
188:
             base_loss_kwargs = {
189:
                 k: v
190:
                 for k, v in DEFAULT_LOSS_CONFIG.__dict__.items()
191:
                 if k != "enable_dynamic_weighting"
192:
193:
             self.loss fn = ResearchGradeLoss(
194:
                 **base_loss_kwargs,
195:
                 \verb|enable_dynamic_weighting=bool(config.curriculum.use\_gradnorm)|,\\
196:
                 gradnorm_alpha=float(config.curriculum.gradnorm_alpha),
197:
                 device=self.device,
198:
             )
199:
200:
             self.history = {
                 "stage1": {"train_loss": [], "val_loss": [], "component_losses": []},
201:
                 "stage2": {"train_loss": [], "val_loss": [], "component_losses": []},
202:
203:
                 "stage3": {"train_loss": [], "val_loss": [], "component_losses": []},
```

```
204:
                 "stage transitions": [].
205:
                 "dynamic_weights": [],
206:
                 "curriculum_events": [],
207:
208:
209:
        def _get_eta_string(self, epoch, total_epochs):
210:
             """Calculate and format ETA string"""
             if len(self.epoch_times) == 0:
211:
212:
                 return "ETA: calculating..."
213:
214:
            avg_epoch_time = sum(self.epoch_times) / len(self.epoch_times)
215:
             remaining_epochs = total_epochs - epoch - 1
216:
             eta_seconds = avg_epoch_time * remaining_epochs
217:
218:
            if eta seconds < 60:
219:
                 return f"ETA: {int(eta_seconds)}s"
             elif eta_seconds < 3600:
220:
221:
                return f"ETA: {int(eta_seconds // 60)}m {int(eta_seconds % 60)}s"
222:
             else:
223:
                hours = int(eta_seconds // 3600)
224:
                 minutes = int((eta_seconds % 3600) // 60)
225:
                 return f"ETA: {hours}h {minutes}m"
226:
227:
         def _get_shared_parameters(self):
228:
             """Get shared parameters for GradNorm weighting"""
229:
             # Return encoder parameters as shared across tasks
230:
             return list(self.model.encoder.parameters())
231:
232:
        def _update_loss_weights_for_curriculum(
233:
             self, current_stage: int, stage_epoch: int, total_stage_epochs: int
234:
235:
             """Update loss weights based on curriculum schedule"""
             base_weights = {
236:
                 "seg": self.loss_fn.initial_weights["seg"],
237:
                 "dice": self.loss_fn.initial_weights["dice"],
238:
239:
                 "sdf": self.loss_fn.initial_weights["sdf"],
240:
                 "attr": self.loss_fn.initial_weights["attr"],
241:
                 "polygon": self.loss_fn.initial_weights["polygon"],
242:
                 "voxel": self.loss_fn.initial_weights["voxel"],
243:
                 "topology": self.loss_fn.initial_weights["topology"],
244:
                 "latent_consistency": self.loss_fn.initial_weights["latent_consistency"],
245:
                 "graph": self.loss_fn.initial_weights["graph"],
246:
            }
247:
248:
            scheduled_weights = self.loss_scheduler.get_scheduled_weights(
249:
                 current_stage,
250:
                 self.global_epoch,
251:
                 stage_epoch,
252:
                 total_stage_epochs,
253:
                 base_weights,
254:
255:
256:
             self.loss_fn.update_loss_weights(scheduled_weights)
257:
258:
             # Log weight changes
             self.history["dynamic_weights"].append(
259:
260:
                {
261:
                     "epoch": self.global_epoch,
262:
                      "stage": current_stage,
                     "weights": scheduled_weights.copy(),
263:
264:
                 }
265:
             )
266:
267:
         def _train_epoch(self, mode="stage1"):
             """Enhanced training epoch with improved stability and speed"""
268:
269:
            self.model.train()
270:
            total_loss = 0
271:
            component_loss_sums = {}
272:
             # Select optimizer and apply gradient scaling
273:
274:
             if mode == "stage1":
275:
                 optimizer = self.optimizer_2d
276:
             elif mode == "stage2":
```

```
277:
                 optimizer = self.optimizer_dvx
278:
             else:
279:
                 optimizer = self.optimizer_full
280:
281:
             # Improved progress tracking
282:
             train_pbar = tqdm(
283:
                 self.train_loader, desc=f"Training {mode.upper()}", leave=False, ncols=120
284:
             )
285:
             batch count = 0
286:
287:
             epoch_start_time = time.time()
288:
289:
             # Add gradient accumulation tracking
290:
             accumulated_loss = 0.0
291:
292:
             for batch_idx, batch in enumerate(train_pbar):
293:
                 self._step += 1
294:
                 batch = {
                    k: v.to(self.device, non_blocking=True) if torch.is_tensor(v) else v
295:
296:
                     for k, v in batch.items()
297:
                 }
298:
299:
                 # Smart geometric computation gating
300:
                 run full geometric = (
301:
                     mode == "stage3" or # Always run in final stage
302:
                     (mode == "stage2" and self._step % 2 == 0) or # Every other step in stage 2
303:
                     (mode == "stage1" and self._step % 4 == 0) # Every 4th step in stage 1
304:
305:
306:
                 with autocast("cuda", enabled=self.use_amp):
307:
                     predictions = self.model(
308:
                         batch["image"], run_full_geometric=run_full_geometric
309:
310:
311:
                     targets = self._prepare_targets(batch, mode)
312:
313:
                     shared_params = (
314:
                         self._get_shared_parameters()
315:
                         if self.config.curriculum.use_gradnorm
316:
                         else None
317:
318:
319:
                     loss, loss_components = self.loss_fn(
320:
                         predictions,
321:
                         targets,
322:
                         shared_params,
                         run_full_geometric=run_full_geometric,
323:
324:
325:
326:
                     # Scale for accumulation
                     loss = loss / self.accumulation_steps
327:
328:
                     accumulated_loss += loss.item()
329:
330:
                 # Backward pass with stability
331:
                 self.scaler.scale(loss).backward()
332:
                 # Gradient accumulation and update
333:
334:
                 if ((batch_idx + 1) % self.accumulation_steps) == 0:
335:
                     # Enhanced gradient clipping
336:
                     self.scaler.unscale_(optimizer)
337:
338:
                     # Adaptive gradient clipping based on loss magnitude
339:
                     max_grad_norm = min(self.config.grad_clip_norm * (1.0 + accumulated_loss), 2.0)
340:
                     torch.nn.utils.clip_grad_norm_(
341:
                         self.model.parameters(), max_grad_norm
342:
343:
344:
                     self.scaler.step(optimizer)
345:
                     self.scaler.update()
346:
                     optimizer.zero_grad()
347:
348:
                     # Reset accumulation
349:
                     accumulated_loss = 0.0
```

```
350:
351:
                 current_loss = loss.item() * self.accumulation_steps
352:
                 total_loss += current_loss
353:
                  # Track components with better averaging
354:
355:
                 for name, component_loss in loss_components.items():
356:
                      if name != "total":
                          loss_val = (
357:
358:
                              component_loss.item()
359:
                              if torch.is_tensor(component_loss)
360:
                              else component_loss
361:
362:
                          if name not in component_loss_sums:
363:
                              component_loss_sums[name] = []
364:
                          component_loss_sums[name].append(loss_val)
365:
366:
                 batch_count += 1
367:
368:
                  # Less frequent but more informative logging
369:
                 if (batch_idx + 1) % 100 == 0:
370:
                      elapsed = time.time() - epoch_start_time
                      avg_time_per_batch = elapsed / (batch_idx + 1)
371:
372:
373:
                      # Show meaningful component averages
374:
                      recent_components = {}
375:
                      for name, vals in component_loss_sums.items():
376:
                          if len(vals) >= 10: # Only show if we have enough samples
377:
                              recent_avg = np.mean(vals[-10:]) # Last 10 batches
378:
                              if recent_avg > 0.01: # Only show significant components
379:
                                  recent_components[name] = recent_avg
380:
381:
                      \texttt{comp\_str} = \texttt{", ".join}([\texttt{f"}\{k\}:\{v:.3f\}" \texttt{ for } k, \texttt{ v in recent\_components.items()]})
                      print(f"[Epoch {self.global_epoch}] Batch {batch_idx+1} | "
382:
                            f"{avg_time_per_batch:.2f}s/batch | loss:{total_loss/batch_count:.4f} |
383:
                                     {comp_str}")
384:
                  # Update progress with meaningful info
385:
386:
                 train_pbar.set_postfix({
387:
                      "loss": f"{current_loss:.4f}",
388:
                      "lr": f"{optimizer.param_groups[0]['lr']:.6f}"
389:
                 })
390:
              # Calculate proper component averages
392:
             avg_component_losses = {}
393:
             for name, loss_list in component_loss_sums.items():
394:
                 if loss_list:
                     avg_component_losses[name] = np.mean(loss_list)
395:
396:
                  else:
397:
                      avg_component_losses[name] = 0.0
398:
399:
             return total_loss / batch_count, avg_component_losses
401:
         def _prepare_targets(self, batch, mode):
402:
               ""Prepare targets based on training mode"""
403:
             if mode == "stage1":
                 return {"mask": batch["mask"], "attributes": batch["attributes"]}
404:
              elif mode == "stage2":
405:
406:
                 return {
407:
                      "polygons_gt": {
                          "polygons": batch["polygons_gt"]["polygons"].to(self.device),
408:
409:
                          "valid_mask": batch["polygons_gt"]["valid_mask"].to(self.device),
410:
411:
                 }
412:
             else: # stage3
                 return {
413:
414:
                      "mask": batch["mask"],
                      "attributes": batch["attributes"],
415:
416:
                      "voxels_gt": batch["voxels_gt"],
417:
                      "polygons_gt": {
418:
                          "polygons": batch["polygons_gt"]["polygons"].to(self.device),
419:
                          "valid_mask": batch["polygons_gt"]["valid_mask"].to(self.device),
420:
                      },
421:
                  }
```

```
422:
423:
         def _validate(self, mode="stage1"):
424:
             """Enhanced validation with consistent loss computation"""
425:
             self.model.eval()
            total_loss = 0
426:
427:
             component_loss_sums = {}
428:
429:
            val_pbar = tqdm(
430:
                 self.val_loader, desc=f"Validating {mode.upper()}", leave=False, ncols=120
431:
432:
433:
             batch_count = 0
434:
             with torch.no_grad():
435:
                 for batch in val_pbar:
436:
                     batch = {
437:
                         k: v.to(self.device, non_blocking=True) if torch.is_tensor(v) else v
438:
                         for k, v in batch.items()
439:
440:
441:
                     with autocast("cuda", enabled=self.use_amp):
442:
                         # ALWAYS run full geometric in validation for consistency
443:
                         predictions = self.model(batch["image"], run_full_geometric=True)
444:
445:
                         targets = self._prepare_targets(batch, "stage3") # Use full targets
446:
447:
                         # Use same loss computation as training but without dynamic weighting
448:
                         loss, loss_components = self.loss_fn(
449:
                             predictions, targets, shared_parameters=None, run_full_geometric=True
450:
451:
452:
                     current_loss = loss.item()
453:
                     total_loss += current_loss
454:
                     # Track component losses properly
455:
456:
                     for name, component_loss in loss_components.items():
457:
                         if name != "total":
458:
                             loss_val = (
459:
                                 component_loss.item()
460:
                                  if torch.is_tensor(component_loss)
461:
                                 else component_loss
462:
463:
                             if name not in component_loss_sums:
464:
                                 component_loss_sums[name] = []
465:
                             component_loss_sums[name].append(loss_val)
466:
467:
                     batch_count += 1
                     val_pbar.set_postfix({"loss": f"{current_loss:.4f}"})
468:
469:
470:
             # Calculate proper averages
471:
             avg_component_losses = {}
472:
             for name, loss_list in component_loss_sums.items():
473:
                 if loss_list:
474:
                     avg_component_losses[name] = np.mean(loss_list)
475:
                 else:
476:
                     avg_component_losses[name] = 0.0
477:
478:
             return total_loss / batch_count, avg_component_losses
479:
480:
         def train_stage_adaptive(self, stage: int, max_epochs: int, min_epochs: int):
481:
482:
             Train a stage with adaptive termination based on curriculum learning
483:
484:
             Arqs:
485:
                 stage: Stage number (1, 2, 3)
                 max_epochs: Maximum epochs for this stage
486:
487:
                min_epochs: Minimum epochs before considering transition
488:
489:
             print("=" * 60)
490:
             print(f"STAGE {stage}: Adaptive Training with Dynamic Curriculum")
            print("=" * 60)
491:
492:
493:
             self.current stage = stage
494:
             self.stage_start_time = time.time()
```

```
495:
496:
             # Only reset if not resuming
497:
             if not hasattr(self, "epoch_times") or self.epoch_times is None:
498:
                 self.epoch_times = []
499:
500:
             start_epoch = int(self.stage_epoch or 0)
501:
502:
             # Set parameter gradients for current stage
503:
             self._configure_stage_parameters(stage)
504:
505:
             mode_name = f"stage{stage}"
506:
507:
             for epoch in range(start_epoch, max_epochs):
508:
                 epoch_start_time = time.time()
509:
                 self.stage epoch = epoch
510:
                 self.global_epoch += 1
511:
512:
                 # Update loss weights based on curriculum
513:
                 self._update_loss_weights_for_curriculum(stage, epoch, max_epochs)
514:
                 print(
515:
516:
                     f"\nStage {stage} - Epoch {epoch+1}/{max_epochs} (Global: {self.global_epoch})"
517:
518:
519:
                 # Training and validation
                 train_loss, train_components = self._train_epoch(mode_name)
520:
521:
                 val_loss, val_components = self._validate(mode_name)
522:
523:
                 # Record epoch time
524:
                 epoch_time = time.time() - epoch_start_time
525:
                 self.epoch_times.append(epoch_time)
526:
527:
                 if len(self.epoch_times) > 10:
528:
                     self.epoch_times.pop(0)
529:
530:
                 # Update curriculum state
531:
                 self.curriculum_state.update_loss_history(mode_name, val_loss)
532:
                 self.curriculum_state.update_component_losses(val_components)
533:
534:
                 # Store training history
535:
                 self.history[mode_name]["train_loss"].append(train_loss)
                 self.history[mode_name]["val_loss"].append(val_loss)
536:
537:
                 self.history[mode_name]["component_losses"].append(val_components)
538:
539:
                 # Update learning rate
540:
                 if stage == 1:
                     self.scheduler_2d.step()
541:
542:
                 elif stage == 2:
543:
                     self.scheduler_dvx.step()
544:
                 else:
545:
                     self.scheduler_full.step()
546:
547:
                 # Display comprehensive results
548:
                 self._display_epoch_results(
549:
                     epoch,
550:
                     max_epochs,
551:
                     train_loss,
552:
                     val_loss,
553:
                     train_components,
554:
                     val_components,
555:
                     epoch_time,
556:
557:
558:
                 # Check for adaptive stage transition
559:
                 if epoch >= min_epochs:
560:
                     should_transition = self.curriculum_state.should_transition(stage)
561:
                     if should_transition:
562:
                         print(
563:
                             f"\n? ADAPTIVE TRANSITION: Stage {stage} converged after {epoch+1} epochs"
564:
                         )
565:
566:
                                 Detected performance plateau - transitioning to next stage"
567:
```

```
568:
569:
                          self.history["stage_transitions"].append(
570:
571:
                                  "from_stage": stage,
572:
                                  "epoch": epoch + 1,
                                  "global_epoch": self.global_epoch,
573:
574:
                                  "reason": "performance_plateau",
575:
                              }
576:
577:
578:
                          self.history["curriculum_events"].append(
579:
                                  "type": "stage_transition",
580:
581:
                                  "stage": stage,
                                  "epoch": self.global_epoch,
582:
583:
                                  "details": f"Converged after {epoch+1} epochs",
584:
                              }
585:
586:
                          break
587:
588:
                 # Save rolling checkpoint
589:
                 if (epoch + 1) % self.config.checkpoint_freq == 0:
590:
                     self._save_rolling_checkpoint()
591:
592:
             print(f"\nStage {stage} completed after {epoch+1} epochs")
593:
594:
         def _configure_stage_parameters(self, stage: int):
              """Configure which parameters require gradients for each stage"""
595:
596:
             # First freeze everything
597:
             for param in self.model.parameters():
598:
                 param.requires_grad = False
599:
600:
             if stage == 1:
                 # Stage 1: Segmentation + Attributes (2D only)
601:
602:
                 for param in self.model.encoder.parameters():
603:
                     param.requires_grad = True
604:
                 for param in self.model.seg_head.parameters():
605:
                     param.requires_grad = True
606:
                 for param in self.model.attr_head.parameters():
607:
                     param.requires_grad = True
608:
                 for param in self.model.sdf_head.parameters():
609:
                     param.requires_grad = True
610:
             elif stage == 2:
611:
612:
                 # Stage 2: DVX training (polygon fitting) - keep encoder frozen initially
613:
                 for param in self.model.dvx.parameters():
                     param.requires_grad = True
614:
615:
                 # Optionally unfreeze encoder in later epochs
616:
                 if self.stage_epoch > 10:
617:
                     for param in self.model.encoder.parameters():
                         param.requires_grad = True
618:
619:
620:
             else: # stage == 3
621:
                 # Stage 3: End-to-end fine-tuning (all parameters)
622:
                 for param in self.model.parameters():
623:
                     param.requires_grad = True
624:
625:
         def _display_epoch_results(
626:
             self,
             epoch: int,
627:
628:
             total_epochs: int,
629:
             train_loss: float,
630:
             val_loss: float,
631:
             train_components: Dict,
632:
             val components: Dict,
633:
             epoch_time: float,
634:
635:
             """Display comprehensive epoch results with curriculum information"""
636:
             eta_str = self._get_eta_string(epoch, total_epochs)
637:
638:
             print(f"Train Loss: {train_loss:.4f}, Val Loss: {val_loss:.4f}")
639:
             print(f"Epoch time: {epoch_time:.1f}s, {eta_str}")
640:
```

```
641:
             # Show significant component losses
642:
             significant_components = {
643:
                 k: v
                 for k, v in val_components.items()
644:
                 if v > 0.01
645:
646:
                 and k
647:
                 in [
648:
                      "sea",
649:
                      "dice",
                      "polygon",
650:
651:
                      "voxel",
652:
                      "topology",
653:
                      "latent_consistency",
654:
                      "graph",
655:
                 ]
656:
             if significant_components:
657:
658:
                 comp_str = ", ".join(
                     [f"\{k\}: \{v:.3f\}" for k, v in significant_components.items()]
659:
660:
661:
                 print(f"Components: {comp_str}")
662:
663:
             # Show current loss weights for active components
664:
             active\_weights = \{k: v for k, v in self.loss\_fn.weights.items() if v > 0.001\}
665:
             if active weights:
                 weight\_str = ", ".join([f"{k}: {v:.3f}" for k, v in active\_weights.items()])
666:
667:
                 print(f"Weights: {weight_str}")
668:
669:
             # Show curriculum status
670:
             plateau_epochs = self.curriculum_state.epochs_without_improvement
671:
             if plateau_epochs > 0:
672:
                 print(f"Plateau: {plateau_epochs} epochs without improvement")
673:
674:
         def _save_rolling_checkpoint(self):
675:
              """Enhanced checkpoint saving with curriculum state, RNG state, and scaler state"""
676:
             checkpoint = {
                 "model_state_dict": self.model.state_dict(),
677:
678:
                 "optimizer_2d_state_dict": self.optimizer_2d.state_dict(),
679:
                 "optimizer_dvx_state_dict": self.optimizer_dvx.state_dict(),
                 "optimizer_full_state_dict": self.optimizer_full.state_dict(),
680:
681:
                 "scheduler_2d_state_dict": self.scheduler_2d.state_dict(),
                 "scheduler_dvx_state_dict": self.scheduler_dvx.state_dict(),
682:
683:
                 "scheduler_full_state_dict": self.scheduler_full.state_dict(),
684:
                 "scaler_state_dict": self.scaler.state_dict(),  # Add AMP scaler state
685:
                 "loss_fn_state": {
686:
                      "weights": self.loss_fn.weights,
                      "initial_weights": self.loss_fn.initial_weights,
687:
688:
689:
                 "history": self.history,
690:
                 "config": self.config,
                  "current_stage": self.current_stage,
691:
                 "current_epoch": self.current_epoch,
692:
693:
                 "global_epoch": self.global_epoch,
694:
                  "stage_epoch": self.stage_epoch,
695:
                  "epoch_times": self.epoch_times,
                  "step_counter": self._step,  # Save step counter for DVX gating
696:
697:
                 "curriculum_state": {
                      "loss_history": dict(self.curriculum_state.loss_history),
698:
699:
                      "component_losses": dict(self.curriculum_state.component_losses),
700:
                      "epochs_without_improvement": self.curriculum_state.epochs_without_improvement,
701:
                      "best_val_loss": self.curriculum_state.best_val_loss,
702:
                      "stage_transition_epochs": self.curriculum_state.stage_transition_epochs,
703:
                 },
704:
                 "rng_state": {
                      "torch": torch.get_rng_state(),
705:
706:
                      "cuda": torch.cuda.get_rng_state_all()
707:
                     if torch.cuda.is_available()
708:
                     else None,
709:
                      "numpy": np.random.get_state(),
710:
                      "python": random.getstate(),
711:
                 },
712:
             }
713:
```

```
checkpoint_path = self.ROLLING_CHECKPOINT
714:
715:
             torch.save(checkpoint, checkpoint_path)
716:
             print(f"Rolling checkpoint saved: {checkpoint_path}")
717:
         def load_checkpoint(self, filename):
718:
719:
720:
             Enhanced checkpoint loading with architecture compatibility handling
721:
             Safely handles model architecture changes by filtering incompatible parameters
722:
             print(f"Loading checkpoint: {filename}")
723:
724:
             checkpoint = torch.load(filename, map_location=self.device, weights_only=False)
725:
726:
             # === SAFE MODEL STATE LOADING ===
727:
             model_state = checkpoint["model_state_dict"]
728:
             current_model_keys = set(self.model.state_dict().keys())
729:
730:
             # Filter parameters into compatible and incompatible
731:
             compatible_state = {}
             incompatible_keys = []
732:
733:
             missing_keys = []
734:
735:
             # Check each parameter in the checkpoint
736:
             for key, value in model_state.items():
737:
                 if key in current_model_keys:
738:
                     # Check if tensor shapes match
739:
                     current_param = self.model.state_dict()[key]
740:
                     if current_param.shape == value.shape:
741:
                        compatible_state[key] = value
742:
743:
                         incompatible_keys.append(f"{key} (shape mismatch: {value.shape} ->
                                  {current_param.shape})")
744:
                 else:
745:
                     incompatible_keys.append(f"{key} (parameter not found in current model)")
746:
747:
             # Check for missing parameters in checkpoint
748:
             for key in current_model_keys:
749:
                 if key not in model_state:
750:
                     missing_keys.append(key)
751:
752:
             # Load compatible parameters only
753:
             loaded_keys, unexpected_keys = self.model.load_state_dict(compatible_state, strict=False)
754:
755:
             # Report parameter loading status
756:
             print(f"? Successfully loaded {len(compatible_state)} compatible parameters")
757:
758:
             if incompatible_keys:
759:
                 print(f"? Skipped {len(incompatible_keys)} incompatible parameters:")
760:
                 for key in incompatible_keys[:10]: # Show first 10
761:
                     print(f"
                                 - {key}")
762:
                 if len(incompatible_keys) > 10:
763:
                                ... and {len(incompatible_keys) - 10} more")
                     print(f"
764:
765:
             if missing_keys:
766:
                 print(f"? {len(missing_keys)} parameters will use random initialization:")
767:
                 for key in missing_keys[:10]: # Show first 10
                     print(f" - {key}")
768:
                 if len(missing_keys) > 10:
769:
770:
                     print(f"
                                 ... and {len(missing_keys) - 10} more")
771:
             # === OPTIMIZER STATES LOADING ===
772:
773:
             try:
774:
                 self.optimizer_2d.load_state_dict(checkpoint["optimizer_2d_state_dict"])
775:
                 print("? Loaded optimizer_2d state")
776:
             except Exception as e:
777:
                 print(f"? Could not load optimizer_2d state: {e}")
778:
779:
             trv:
780:
                 self.optimizer_dvx.load_state_dict(checkpoint["optimizer_dvx_state_dict"])
781:
                 print("? Loaded optimizer_dvx state")
782:
             except Exception as e:
783:
                 print(f"? Could not load optimizer_dvx state: {e}")
784:
785:
             trv:
```

```
786:
                 self.optimizer_full.load_state_dict(checkpoint["optimizer_full_state_dict"])
787:
                 print("? Loaded optimizer_full state")
788:
             except Exception as e:
789:
                 print(f"? Could not load optimizer_full state: {e}")
790:
791:
             # === AMP SCALER STATE ===
792:
             if "scaler_state_dict" in checkpoint:
793:
                 try:
794:
                     self.scaler.load_state_dict(checkpoint["scaler_state_dict"])
                     print("? Loaded AMP scaler state")
795:
796:
                 except Exception as e:
797:
                     print(f"? Could not load scaler state: {e}")
             # === SCHEDULER STATES ===
799:
800:
             scheduler mappings = [
801:
                 ("scheduler_2d_state_dict", self.scheduler_2d),
802:
                 ("scheduler_dvx_state_dict", self.scheduler_dvx),
803:
                 ("scheduler_full_state_dict", self.scheduler_full),
             1
804:
805:
806:
             for state_key, scheduler_obj in scheduler_mappings:
807:
                 if state_key in checkpoint:
808:
                     try:
809:
                         scheduler obj.load state dict(checkpoint[state key])
810:
                         print(f"? Loaded {state_key.replace('_state_dict', '')} scheduler")
811:
                     except Exception as e:
812:
                         print(f"? Could not load {state_key}: {e}")
813:
814:
             # === LOSS FUNCTION STATE ===
815:
             if "loss_fn_state" in checkpoint:
816:
                 try:
817:
                     loaded_weights = checkpoint["loss_fn_state"]["weights"]
818:
                     if isinstance(loaded weights, dict):
819:
                         # Handle device transfer for tensor weights
820:
                         self.loss_fn.weights = {
821:
                             k: (v.to(self.device) if torch.is_tensor(v) else v)
822:
                             for k, v in loaded_weights.items()
823:
                         }
824:
                     else:
825:
                         self.loss_fn.weights = loaded_weights
826:
827:
                     self.loss_fn.initial_weights = checkpoint["loss_fn_state"]["initial_weights"]
828:
                     print("? Loaded loss function weights")
829:
                 except Exception as e:
830:
                     print(f"? Could not load loss function state: {e}")
831:
832:
             # === TRAINING HISTORY ===
833:
             if "history" in checkpoint:
                 self.history = checkpoint["history"]
834:
835:
                 print("? Loaded training history")
836:
837:
             # === TRAINING STATE VARIABLES ===
838:
             state_variables = [
839:
                ("current_stage", "current_stage"),
840:
                 ("current_epoch", "current_epoch"),
                 ("global_epoch", "global_epoch"),
841:
                 ("stage_epoch", "stage_epoch"),
842:
                 ("epoch_times", "epoch_times"),
843:
844:
                 ("step_counter", "_step"),
             1
845:
846:
847:
             for checkpoint_key, attr_name in state_variables:
848:
                 if checkpoint_key in checkpoint:
849:
                     setattr(self, attr_name, checkpoint[checkpoint_key])
850:
                     print(f"? Restored {checkpoint_key}: {getattr(self, attr_name)}")
851:
             # === CURRICULUM STATE RESTORATION ===
852:
853:
             if "curriculum_state" in checkpoint:
854:
                 try:
855:
                     cs = checkpoint["curriculum_state"]
856:
857:
                     # Restore loss history deques
858:
                     for key, history in cs.get("loss_history", {}).items():
```

```
859:
                         self.curriculum state.loss history[kev] = deque(
860:
                             history, maxlen=self.config.curriculum.plateau_detection_window * 2
861:
862:
863:
                     # Restore component loss deques
864:
                     for key, history in cs.get("component_losses", {}).items():
865:
                         self.curriculum_state.component_losses[key] = deque(history, maxlen=20)
866:
867:
                     # Restore curriculum metrics
868:
                     self.curriculum_state.epochs_without_improvement =
                              cs.get("epochs_without_improvement", 0)
869:
                     self.curriculum_state.best_val_loss = cs.get("best_val_loss", float("inf"))
870:
                     self.curriculum_state.stage_transition_epochs = cs.get("stage_transition_epochs",
                              []
871:
872:
                     print("? Restored curriculum learning state")
873:
                 except Exception as e:
874:
                     print(f"? Could not restore curriculum state: {e}")
875:
876:
             # === RNG STATE RESTORATION ===
877:
             if "rng_state" in checkpoint:
878:
                 try:
879:
                     rs = checkpoint["rng_state"]
880:
                     # Torch RNG (CPU)
881:
                     if "torch" in rs and rs["torch"] is not None:
882:
883:
                         torch_state = rs["torch"]
884:
                         if torch.is_tensor(torch_state) and torch_state.dtype == torch.uint8:
885:
                             torch.set_rng_state(torch_state)
886:
                         else:
887:
                             torch.set_rng_state(torch.tensor(torch_state, dtype=torch.uint8))
888:
889:
                     # CUDA RNG (all devices)
                     if "cuda" in rs and rs["cuda"] is not None and torch.cuda.is_available():
890:
891:
                         cuda_state = rs["cuda"]
892:
                         cuda_tensors = []
893:
                         for s in cuda_state:
894:
                             if torch.is_tensor(s) and s.dtype == torch.uint8:
895:
                                 cuda_tensors.append(s)
896:
                             else:
897:
                                 cuda_tensors.append(torch.tensor(s, dtype=torch.uint8))
898:
                         torch.cuda.set_rng_state_all(cuda_tensors)
899:
900:
                     # NumPy RNG
901:
                     if "numpy" in rs and rs["numpy"] is not None:
902:
                         np.random.set_state(rs["numpy"])
903:
904:
                     # Python random RNG
905:
                     if "python" in rs and rs["python"] is not None:
906:
                         random.setstate(rs["python"])
907:
908:
                     print("? Restored RNG states for reproducibility")
909:
                 except Exception as e:
910:
                     print(f"? Could not restore RNG states: {e}")
911:
             # === DATALOADER STATE (if available) ===
912:
             if "dataloader_state" in checkpoint:
913:
914:
                 try:
915:
                     dl_state = checkpoint["dataloader_state"]
916:
                     if (dl_state.get("train_sampler_state") is not None and
                         hasattr(self.train_loader.sampler, "__dict__")):
917:
918:
                         self.train_loader.sampler.__dict__.update(dl_state["train_sampler_state"])
919:
920:
                     if (dl_state.get("val_sampler_state") is not None and
                         hasattr(self.val_loader.sampler, "__dict__")):
921:
922:
                         self.val_loader.sampler.__dict__.update(dl_state["val_sampler_state"])
923:
924:
                     print("? Restored dataloader sampler states")
925:
                 except Exception as e:
926:
                     print(f"? Could not restore dataloader states: {e}")
927:
928:
             # === FINAL REPORT ===
929:
             print("\n" + "="*60)
```

```
930:
             print("CHECKPOINT LOADING SUMMARY")
931:
            print("="*60)
932:
             print(f"? Checkpoint loaded: {filename}")
             print(f"? Resuming from Stage {self.current_stage}, Global Epoch {self.global_epoch}")
933:
             print(f"? Model parameters: {len(compatible_state)}/{len(model_state)} loaded successfully")
934:
935:
936:
             if hasattr(self, 'curriculum_state'):
937:
                 print(f"? Curriculum state: {self.curriculum_state.epochs_without_improvement} epochs
                          without improvement")
938:
939:
             if incompatible keys:
940:
                 print(f"? Architecture changes detected: {len(incompatible_keys)} parameters skipped")
941:
                 print(" This is normal after model architecture updates.")
942:
943:
             if missing keys:
944:
                 print(f"? New parameters detected: {len(missing_keys)} will use random initialization")
                 print(" These will be learned quickly during resumed training.")
945:
946:
             print("="*60)
947:
948:
             print("Ready to resume adaptive multi-stage training!")
949:
             print("="*60)
950:
951:
         def train_all_stages(self):
952:
             Run complete adaptive multi-stage training pipeline
953:
954:
955:
             This is the main entry point that orchestrates the dynamic curriculum learning
956:
957:
             if Path(self.ROLLING_CHECKPOINT).exists():
958:
                 print(f"Found existing checkpoint: {self.ROLLING_CHECKPOINT}")
959:
                 print("Resuming adaptive training from checkpoint...")
960:
                 self.load_checkpoint(self.ROLLING_CHECKPOINT)
961:
             else:
                 print("Starting fresh adaptive training pipeline...")
962:
963:
                 self.current_stage = 1
964:
                 self.current_epoch = 0
965:
                 self.global_epoch = 0
966:
967:
            print("\n" + "=" * 80)
            print("ADAPTIVE MULTI-STAGE TRAINING WITH DYNAMIC CURRICULUM")
968:
969:
             print("Novel Training Strategies:")
970:
            print("? Adaptive Stage Transitioning (Dynamic Curriculum)")
971:
            print("? Topology-aware Loss Scheduling")
972:
            print("? Multi-objective Optimization with Dynamic Weighting")
973:
             print("? Cross-modal Latent Consistency Learning")
974:
             print("? Graph-based Topology Constraints")
975:
             print("=" * 80)
976:
977:
             # Stage 1: Adaptive 2D training
978:
             if self.current stage <= 1:
979:
                 print("\n? STAGE 1: Adaptive 2D Segmentation + Attributes Training")
980:
                 self.train_stage_adaptive(
981:
982:
                     max_epochs=self.config.max_stagel_epochs,
983:
                     min_epochs=self.config.min_stagel_epochs,
984:
                 self.current_stage = 2
985:
986:
                 self.stage_epoch = 0
987:
                 print("\nStage 1 completed. Transitioning to Stage 2...")
988:
989:
             # Stage 2: Adaptive DVX training
990:
             if self.current_stage <= 2:</pre>
991:
                 print("\n? STAGE 2: Adaptive DVX Polygon Fitting Training")
992:
                 self.train_stage_adaptive(
993:
                     stage=2,
994:
                     max_epochs=self.config.max_stage2_epochs,
995:
                     min_epochs=self.config.min_stage2_epochs,
996:
997:
                 self.current_stage = 3
998:
                 self.stage epoch = 0
                 print("\nStage 2 completed. Transitioning to Stage 3...")
999:
1000:
1001:
              # Stage 3: Adaptive end-to-end fine-tuning
```

```
1002:
              if self.current stage <= 3:
1003:
                  print("\n? STAGE 3: Adaptive End-to-End Fine-tuning with Full Loss Suite")
1004:
                  self.train_stage_adaptive(
1005:
                      stage=3,
                      max_epochs=self.config.max_stage3_epochs,
1006:
1007:
                      min_epochs=self.config.min_stage3_epochs,
1008:
1009:
                  print("\nStage 3 completed!")
1010:
              print("\n" + "=" * 80)
1011:
              print("? ALL ADAPTIVE TRAINING STAGES COMPLETED!")
1012:
              print("=" * 80)
1013:
1014:
1015:
              # Generate training report
1016:
              self._generate_training_report()
1017:
1018:
              # Save final model
1019:
              self._save_checkpoint("final_adaptive_model.pth")
1020:
1021:
              # Clean up rolling checkpoint
1022:
              if Path(self.ROLLING_CHECKPOINT).exists():
1023:
                  Path(self.ROLLING_CHECKPOINT).unlink()
1024:
                  print(f"Cleaned up rolling checkpoint: {self.ROLLING_CHECKPOINT}")
1025:
1026:
              return self.history
1027:
1028:
          def _generate_training_report(self):
               """Generate comprehensive training report with curriculum insights"""
1029:
1030:
              print("\n" + "=" * 60)
1031:
              print("ADAPTIVE TRAINING REPORT")
1032:
              print("=" * 60)
1033:
1034:
              # Stage transition summary
1035:
              if self.history["stage_transitions"]:
1036:
                  print("\n? Stage Transitions:")
1037:
                  for transition in self.history["stage_transitions"]:
                      print(
1038:
1039:
                              ? Stage {transition['from_stage']} ? {transition['from_stage']+1}: "
1040:
                          f"Epoch {transition['epoch']} (Global: {transition['global_epoch']})"
1041:
                      )
1042:
                                  Reason: {transition['reason']}")
                      print(f"
1043:
1044:
              # Dynamic weight evolution
1045:
              if self.history["dynamic_weights"]:
1046:
                  print(
1047:
                      f"\n?? Dynamic Weight Updates: {len(self.history['dynamic_weights'])} updates"
1048:
1049:
                  final_weights = self.history["dynamic_weights"][-1]["weights"]
1050:
                  print(" Final loss weights:")
1051:
                  for name, weight in final_weights.items():
1052:
                      if weight > 0.001:
1053:
                          print(f"
                                      {name}: {weight:.3f}")
1054:
1055:
              # Curriculum events
1056:
              if self.history["curriculum_events"]:
1057:
                  print(
1058:
                      f"\n? Curriculum Events: {len(self.history['curriculum_events'])} events"
1059:
1060:
                  for event in self.history["curriculum_events"][-5:]: # Show last 5 events
                      print(
1061:
1062:
                          f" ? {event['type']} at global epoch {event['epoch']}: {event['details']}"
1063:
                      )
1064:
1065:
              # Performance summary
              print("\n? Final Performance:")
1066:
1067:
              for stage_name, data in self.history.items():
                  if isinstance(data, dict) and "val_loss" in data and data["val_loss"]:
1068:
1069:
                      final_loss = data["val_loss"][-1]
1070:
                      best_loss = min(data["val_loss"])
1071:
                      print(
1072:
                          f" ? {stage_name.upper()}: Final={final_loss:.4f}, Best={best_loss:.4f}"
1073:
                      )
1074:
```

```
1075:
              print("\n? Training completed with novel adaptive curriculum strategies!")
1076:
              print("=" * 60)
1077:
1078:
          def _save_checkpoint(self, filename):
              """Save final training checkpoint"""
1079:
1080:
              checkpoint = {
1081:
                  "model_state_dict": self.model.state_dict(),
                  "optimizer_2d_state_dict": self.optimizer_2d.state_dict(),
1082:
                  "optimizer_dvx_state_dict": self.optimizer_dvx.state_dict(),
1083:
                  "optimizer_full_state_dict": self.optimizer_full.state_dict(),
1084:
                  "scheduler_2d_state_dict": self.scheduler_2d.state_dict(),
1085:
                  "scheduler_dvx_state_dict": self.scheduler_dvx.state_dict(),
1086:
                  "scheduler_full_state_dict": self.scheduler_full.state_dict(),
1087:
1088:
                  "scaler_state_dict": self.scaler.state_dict(),
1089:
                  "loss fn state": {
1090:
                      "weights": self.loss_fn.weights,
                      "initial_weights": self.loss_fn.initial_weights,
1091:
1092:
                  },
                  "history": self.history,
1093:
                  "config": self.config,
1095:
                  "final_stage": self.current_stage,
1096:
                  "total_epochs": self.global_epoch,
1097:
                  "training_complete": True,
1098:
                  "curriculum_summary": {
1099:
                      "stage_transitions": len(self.history["stage_transitions"]),
1100:
                      "weight_updates": len(self.history["dynamic_weights"]),
1101:
                      "curriculum_events": len(self.history["curriculum_events"]),
                  },
1102:
1103:
             }
1104:
             torch.save(checkpoint, filename)
1105:
             print(f"Final model saved: {filename}")
1106:
1107:
1108: # Legacy compatibility class
1109: class MultiStageTrainer(AdaptiveMultiStageTrainer):
1110:
1111:
          Legacy wrapper for backward compatibility
1112:
          Redirects to the new adaptive trainer
1113:
1114:
1115:
          def __init__(self, *args, **kwargs):
1116:
              super().__init__(*args, **kwargs)
1117:
              print("Note: Using enhanced AdaptiveMultiStageTrainer with dynamic curriculum")
1118:
1119:
         def train_stage1(self, epochs=None):
1120:
              """Legacy method - redirects to adaptive training"""
1121:
              max_epochs = epochs or self.config.max_stage1_epochs
1122:
              min_epochs = self.config.min_stage1_epochs
1123:
              return self.train_stage_adaptive(1, max_epochs, min_epochs)
1124:
1125:
          def train_stage2(self, epochs=None):
              """Legacy method - redirects to adaptive training"""
1126:
1127:
              max_epochs = epochs or self.config.max_stage2_epochs
1128:
              min epochs = self.config.min stage2 epochs
1129:
              return self.train_stage_adaptive(2, max_epochs, min_epochs)
1130:
          def train_stage3(self, epochs=None):
1131:
              """Legacy method - redirects to adaptive training"""
1132:
1133:
              max_epochs = epochs or self.config.max_stage3_epochs
1134:
              min_epochs = self.config.min_stage3_epochs
1135:
             return self.train_stage_adaptive(3, max_epochs, min_epochs)
```

■ File: utils\visualization.py

1: """

2: Visualization and utility functions

3: """

5: import matplotlib.pyplot as plt6: import numpy as np

7: import cv2

4:

```
8: import torch
9: from pathlib import Path
10: from evaluation.metrics import compute_iou
12:
13: def plot_training_history(history, save_path="training_history.png"):
14:
        """Plot training curves for all stages"""
        fig, axes = plt.subplots(1, 3, figsize=(15, 5))
15:
16:
17:
        for idx, (stage, data) in enumerate(history.items()):
18:
           if isinstance(data, dict) and "train_loss" in data and data["train_loss"]: # Only plot if
                     stage was executed
19:
                axes[idx].plot(data["train_loss"], label="Train", linewidth=2)
                axes[idx].plot(data["val_loss"], label="Validation", linewidth=2)
20:
                axes[idx].set_title(f"{stage.upper()} Training")
21:
22:
                axes[idx].set_xlabel("Epoch")
23:
                axes[idx].set_ylabel("Loss")
24:
                axes[idx].legend()
25:
                axes[idx].grid(True, alpha=0.3)
27:
       plt.tight_layout()
28:
       plt.savefig(save_path, dpi=300, bbox_inches="tight")
29:
        plt.show()
30:
31:
32: def plot_curriculum_analysis(history, save_path="curriculum_analysis.png"):
33:
        """Plot curriculum learning analysis including stage transitions and adaptive behavior"""
34:
        fig, axes = plt.subplots(2, 2, figsize=(15, 10))
36:
        # Plot 1: Stage transition timeline
37:
        if "stage_transitions" in history and history["stage_transitions"]:
38:
            transitions = history["stage_transitions"]
39:
40:
            # Extract transition epochs and reasons
41:
            transition_epochs = [t["epoch"] for t in transitions]
            transition_stages = [t["from_stage"] + " ? " + t["to_stage"] for t in transitions]
42:
            transition_reasons = [t.get("reason", "threshold") for t in transitions]
43:
44:
45:
            # Create timeline
46:
            y_positions = range(len(transition_epochs))
47:
            colors = ['red' if 'patience' in reason else 'green' for reason in transition_reasons]
48:
            axes[0, 0].barh(y_positions, transition_epochs, color=colors, alpha=0.7)
50:
            axes[0, 0].set_yticks(y_positions)
51:
            axes[0, 0].set_yticklabels(transition_stages)
52:
            axes[0, 0].set_xlabel("Epoch")
53:
            axes[0, 0].set_title("Stage Transition Timeline")
54:
            axes[0, 0].grid(True, alpha=0.3)
55:
56:
            # Add legend
57:
            axes[0, 0].legend(['Patience-based', 'Threshold-based'], loc='lower right')
58:
        else:
59:
            axes[0, 0].text(0.5, 0.5, "No stage transitions recorded",
                           ha='center', va='center', transform=axes[0, 0].transAxes)
60:
61:
            axes[0, 0].set_title("Stage Transition Timeline")
62:
        # Plot 2: Loss component evolution
63:
         \  \, \text{if "dynamic\_weights" in history and history["dynamic\_weights"]:} \\
64:
65:
            weight_data = history["dynamic_weights"]
66:
            epochs = [entry["epoch"] for entry in weight_data]
67:
68:
            # Plot each loss component weight
69:
            weight_names = list(weight_data[0]["weights"].keys()) if weight_data else []
70:
            for weight_name in weight_names[:5]: # Limit to top 5 for readability
71:
                weights = [entry["weights"].get(weight_name, 0) for entry in weight_data]
72:
                if any(w > 0.001 \text{ for } w \text{ in weights}): # Only plot significant weights
73:
                    axes[0, 1].plot(epochs, weights, label=weight_name, linewidth=2, marker='o',
                             markersize=3)
74:
75:
           axes[0, 1].set_xlabel("Global Epoch")
76:
            axes[0, 1].set_ylabel("Loss Weight")
            axes[0, 1].set_title("Dynamic Loss Weight Evolution")
77:
78:
            axes[0, 1].legend()
```

```
79:
             axes[0, 1].grid(True, alpha=0.3)
 80:
 81:
             axes[0, 1].text(0.5, 0.5, "No dynamic weights recorded",
 82:
                            ha='center', va='center', transform=axes[0, 1].transAxes)
 83:
             axes[0, 1].set_title("Dynamic Loss Weight Evolution")
 84:
 85:
         # Plot 3: Curriculum progress indicators
         if "curriculum_events" in history and history["curriculum_events"]:
 86:
 87:
             events = history["curriculum_events"]
 88:
             event_types = {}
 89:
 90:
             for event in events:
 91:
                 event_type = event.get("type", "unknown")
 92:
                 if event_type not in event_types:
 93:
                     event_types[event_type] = []
 94:
                 event_types[event_type].append(event["epoch"])
 95:
 96:
             # Plot event timeline
 97:
             y\_offset = 0
 98:
             for event_type, epochs in event_types.items():
 99:
                 axes[1, 0].scatter(epochs, [y_offset] * len(epochs),
100:
                                   label=event_type, s=50, alpha=0.7)
101:
                 y_offset += 1
102:
             axes[1, 0].set_xlabel("Epoch")
103:
104:
             axes[1, 0].set_ylabel("Event Type")
105:
             axes[1, 0].set_title("Curriculum Learning Events")
106:
             axes[1, 0].legend()
107:
             axes[1, 0].grid(True, alpha=0.3)
108:
         else:
109:
             axes[1, 0].text(0.5, 0.5, "No curriculum events recorded",
110:
                            ha='center', va='center', transform=axes[1, 0].transAxes)
             axes[1, 0].set_title("Curriculum Learning Events")
111:
112:
         # Plot 4: Stage performance comparison
113:
114:
         stage_names = ["stage1", "stage2", "stage3"]
115:
         stage_performance = {}
116:
117:
         for stage_name in stage_names:
118:
             if stage_name in history and isinstance(history[stage_name], dict):
119:
                 stage_data = history[stage_name]
                 if "val_loss" in stage_data and stage_data["val_loss"]:
120:
121:
                     stage_performance[stage_name] = {
                         "final_loss": stage_data["val_loss"][-1],
122:
123:
                         "best_loss": min(stage_data["val_loss"]),
124:
                         "epochs": len(stage_data["val_loss"])
125:
                     }
126:
127:
         if stage_performance:
128:
             stages = list(stage_performance.keys())
129:
             final_losses = [stage_performance[s]["final_loss"] for s in stages]
130:
             best_losses = [stage_performance[s]["best_loss"] for s in stages]
131:
132:
             x = np.arange(len(stages))
133:
             width = 0.35
134:
             axes[1, 1].bar(x - width/2, final_losses, width, label='Final Loss', alpha=0.8)
135:
136:
             axes[1, 1].bar(x + width/2, best_losses, width, label='Best Loss', alpha=0.8)
137:
             axes[1, 1].set_xlabel("Training Stage")
138:
139:
             axes[1, 1].set_ylabel("Validation Loss")
140:
             axes[1, 1].set_title("Stage Performance Comparison")
141:
             axes[1, 1].set_xticks(x)
142:
             axes[1, 1].set_xticklabels([s.upper() for s in stages])
143:
             axes[1, 1].legend()
144:
             axes[1, 1].grid(True, alpha=0.3)
145:
146:
             # Add epoch count annotations
147:
             for i, stage in enumerate(stages):
148:
                 epochs = stage_performance[stage]["epochs"]
149:
                 axes[1, 1].text(i, max(final_losses) * 0.9, f'{epochs} epochs',
150:
                                ha='center', va='bottom', fontsize=9)
151:
         else:
```

```
axes[1, 1].text(0.5, 0.5, "No stage performance data",
152:
                            ha='center', va='center', transform=axes[1, 1].transAxes)
153:
154:
             axes[1, 1].set_title("Stage Performance Comparison")
155:
156:
        plt.tight_layout()
157:
        plt.savefig(save_path, dpi=300, bbox_inches="tight")
158:
        plt.close()
159:
160:
         print(f"Curriculum analysis saved to {save_path}")
161:
162:
163: def visualize_predictions(image, predictions, targets=None, save_path=None):
164:
         """Visualize model predictions"""
         fig, axes = plt.subplots(2, 3, figsize=(15, 10))
165:
166:
167:
         # Original image
168:
         if len(image.shape) == 4:
169:
             img_np = image[0].permute(1, 2, 0).cpu().numpy()
170:
         else:
171:
             img_np = image.permute(1, 2, 0).cpu().numpy()
172:
173:
         axes[0, 0].imshow(img_np)
174:
         axes[0, 0].set_title("Input Image")
175:
         axes[0, 0].axis('off')
176:
         # Predicted segmentation
177:
178:
         if "segmentation" in predictions:
             seg_pred = torch.argmax(predictions["segmentation"], dim=1)[0].cpu().numpy()
179:
180:
             axes[0, 1].imshow(seg_pred, cmap='tab10')
181:
             axes[0, 1].set_title("Predicted Segmentation")
182:
             axes[0, 1].axis('off')
183:
184:
         # Ground truth segmentation (if available)
         if targets and "mask" in targets:
185:
             gt_mask = targets["mask"][0].cpu().numpy()
186:
187:
             axes[0, 2].imshow(gt_mask, cmap='tab10')
188:
             axes[0, 2].set_title("Ground Truth Segmentation")
189:
             axes[0, 2].axis('off')
190:
191:
         # SDF prediction
192:
         if "sdf" in predictions:
193:
             sdf_pred = predictions["sdf"][0, 0].cpu().numpy()
194:
             im = axes[1, 0].imshow(sdf_pred, cmap='RdBu', vmin=-1, vmax=1)
195:
             axes[1, 0].set_title("Predicted SDF")
196:
             axes[1, 0].axis('off')
197:
             plt.colorbar(im, ax=axes[1, 0])
198:
199:
         # Polygon visualization
200:
        if "polygons" in predictions:
201:
            poly_vis = visualize_polygons(
                 predictions["polygons"][0],
202:
203:
                 predictions["polygon_validity"][0],
                 image_size=(256, 256)
204:
205:
206:
             axes[1, 1].imshow(poly_vis)
207:
             axes[1, 1].set_title("Predicted Polygons")
208:
             axes[1, 1].axis('off')
209:
210:
         # 3D voxel slice
         if "voxels_pred" in predictions:
211:
212:
             voxels = torch.sigmoid(predictions["voxels_pred"][0]).cpu().numpy()
213:
             # Show middle slice
214:
            mid_slice = voxels[voxels.shape[0]//2]
             axes[1, 2].imshow(mid_slice, cmap='viridis')
215:
             axes[1, 2].set_title("3D Voxels (Mid Slice)")
216:
217:
             axes[1, 2].axis('off')
218:
219:
        plt.tight_layout()
220:
221:
        if save path:
222:
             plt.savefig(save_path, dpi=300, bbox_inches="tight")
223:
224:
         plt.show()
```

```
225:
226:
227: def visualize_polygons(polygons, validity, image_size=(256, 256), threshold=0.5):
         """Visualize predicted polygons""
         vis_img = np.zeros((*image_size, 3), dtype=np.uint8)
229:
230:
231:
        for poly_idx, (polygon, valid_score) in enumerate(zip(polygons, validity)):
232:
             if valid_score > threshold:
                 # Convert to image coordinates
233:
                 points = polygon.cpu().numpy() * np.array(image_size)
234:
235:
236:
                 # Remove zero-padded points
237:
                 valid_points = points[points.sum(axis=1) > 0]
238:
239:
                 if len(valid_points) >= 3:
240:
                     points_int = valid_points.astype(np.int32)
241:
242:
                     # Different colors for different polygons
243:
                     color = plt.cm.tab10(poly_idx % 10)[:3]
244:
                     color = tuple(int(c * 255) for c in color)
245:
246:
                     cv2.polylines(vis_img, [points_int], True, color, 2)
247:
248:
                     # Add polygon index
249:
                     center = points_int.mean(axis=0).astype(int)
                     cv2.putText(vis_img, str(poly_idx), tuple(center),
250:
251:
                                cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 1)
252:
253:
        return vis_img
254:
255:
256: def save_model_outputs(predictions, output_dir, sample_id):
257:
         """Save all model outputs for detailed analysis"""
258:
         output_dir = Path(output_dir)
259:
         output_dir.mkdir(exist_ok=True)
260:
261:
         sample_dir = output_dir / sample_id
262:
         sample_dir.mkdir(exist_ok=True)
263:
264:
         # Save segmentation
265:
         if "segmentation" in predictions:
266:
             seg_pred = torch.argmax(predictions["segmentation"], dim=1)[0].cpu().numpy()
267:
             cv2.imwrite(str(sample_dir / "segmentation.png"), seg_pred * 50)
268:
269:
         # Save SDF
         if "sdf" in predictions:
270:
271:
             sdf_pred = predictions["sdf"][0, 0].cpu().numpy()
272:
             sdf_normalized = ((sdf_pred + 1) * 127.5).astype(np.uint8)
             cv2.imwrite(str(sample_dir / "sdf.png"), sdf_normalized)
273:
274:
275:
         # Save attributes
276:
         if "attributes" in predictions:
277:
             attrs = predictions["attributes"][0].cpu().numpy()
278:
             np.save(sample_dir / "attributes.npy", attrs)
279:
280:
         # Save polygons
         if "polygons" in predictions:
281:
             polygons = predictions["polygons"][0].cpu().numpy()
282:
283:
             validity = predictions["polygon_validity"][0].cpu().numpy()
284:
285:
             np.save(sample_dir / "polygons.npy", polygons)
286:
             np.save(sample_dir / "polygon_validity.npy", validity)
287:
288:
         # Save voxels
         if "voxels_pred" in predictions:
289:
290:
             voxels = torch.sigmoid(predictions["voxels_pred"][0]).cpu().numpy()
291:
             np.save(sample_dir / "voxels.npy", voxels)
292:
293:
294: def create_comparison_grid(input_images, predictions, targets=None, num_samples=4):
295:
         """Create a comparison grid showing inputs, predictions, and targets"""
296:
         fig, axes = plt.subplots(num_samples, 4, figsize=(16, 4 * num_samples))
297:
```

```
298:
         for i in range(min(num_samples, len(input_images))):
299:
             # Input image
300:
             img = input_images[i].permute(1, 2, 0).cpu().numpy()
301:
             axes[i, 0].imshow(img)
             axes[i, 0].set_title(f"Sample {i+1}: Input")
302:
303:
             axes[i, 0].axis('off')
304:
305:
             # Predicted segmentation
306:
             seg_pred = torch.argmax(predictions["segmentation"][i], dim=0).cpu().numpy()
307:
             axes[i, 1].imshow(seg_pred, cmap='tab10')
             axes[i, 1].set_title("Predicted Seg")
308:
309:
             axes[i, 1].axis('off')
310:
311:
             # Ground truth segmentation (if available)
312:
             if targets and "mask" in targets:
313:
                 gt_mask = targets["mask"][i].cpu().numpy()
314:
                 axes[i, 2].imshow(gt_mask, cmap='tab10')
315:
                 axes[i, 2].set_title("GT Segmentation")
316:
             else:
317:
                 axes[i, 2].text(0.5, 0.5, "No GT", ha='center', va='center',
318:
                                transform=axes[i, 2].transAxes)
319:
                 axes[i, 2].set_title("GT Segmentation")
320:
             axes[i, 2].axis('off')
321:
             # Polygon overlay
322:
323:
             poly_vis = visualize_polygons(
324:
                 predictions["polygons"][i],
325:
                 predictions["polygon_validity"][i]
326:
327:
             axes[i, 3].imshow(poly_vis)
328:
             axes[i, 3].set_title("Predicted Polygons")
329:
             axes[i, 3].axis('off')
330:
331:
        plt.tight_layout()
332:
         return fig
333:
334:
335: def analyze_failure_cases(predictions, targets, threshold_iou=0.5):
336:
         """Analyze failure cases for debugging"""
337:
         failure_indices = []
338:
339:
         for i, (pred_seg, gt_mask) in enumerate(zip(predictions["segmentation"], targets["mask"])):
340:
             seg_pred = torch.argmax(pred_seg, dim=0)
341:
             iou = compute_iou(seg_pred, gt_mask)
342:
343:
             if iou < threshold_iou:</pre>
                 failure_indices.append({
344:
345:
                     "index": i,
                      "iou": iou,
346:
347:
                      "pred_classes": torch.unique(seg_pred).tolist(),
                      "gt_classes": torch.unique(gt_mask).tolist()
348:
349:
                 })
350:
351:
         return failure_indices
352:
353:
354: class ProgressiveVisualization:
355:
         """Track and visualize training progress"""
356:
              _init__(self, save_dir="./training_progress"):
357:
358:
             self.save_dir = Path(save_dir)
359:
             self.save_dir.mkdir(exist_ok=True)
360:
361:
         def log_epoch_results(self, epoch, stage, predictions, targets, sample_image):
             """Log results for a specific epoch""
362:
363:
             epoch_dir = self.save_dir / f"{stage}_epoch_{epoch}"
             epoch_dir.mkdir(exist_ok=True)
364:
365:
366:
             # Save prediction visualization
             fig = plt.figure(figsize=(12, 8))
367:
368:
             visualize_predictions(sample_image, predictions, targets)
369:
             plt.savefig(epoch_dir / "predictions.png", dpi=150, bbox_inches="tight")
370:
             plt.close()
```

```
371:
372:
             # Save individual outputs
373:
             save_model_outputs(predictions, epoch_dir, "sample")
374:
        def create_training_animation(self, stage, metric_name="total_loss"):
375:
376:
             """Create animated GIF showing training progress"""
377:
             # This would create an animation of training progress
378:
             # Implementation depends on having saved epoch results
379:
             pass
380:
381:
382: def compute_architectural_metrics(predictions, image_size=(256, 256)):
383:
         """Compute architecture-specific metrics""
384:
        metrics = {}
385:
386:
        if "segmentation" in predictions:
387:
             seg_pred = torch.argmax(predictions["segmentation"], dim=1)[0]
388:
389:
             # Room count.
390:
             room_mask = (seg_pred == 0).cpu().numpy().astype(np.uint8)
391:
             contours, _ = cv2.findContours(room_mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
392:
             room_count = len([c for c in contours if cv2.contourArea(c) > 100])
393:
             metrics["room_count"] = room_count
394:
395:
             # Wall connectivity
396:
            wall_mask = (seg_pred == 1).cpu().numpy().astype(np.uint8)
397:
             wall_components = cv2.connectedComponents(wall_mask)[0] - 1 # Subtract background
398:
             metrics["wall_components"] = max(0, wall_components)
399:
400:
             # Door and window counts
401:
            door_pixels = (seg_pred == 2).sum().item()
402:
             window_pixels = (seg_pred == 3).sum().item()
            metrics["door_pixels"] = door_pixels
403:
            metrics["window_pixels"] = window_pixels
404:
405:
406:
         if "polygons" in predictions:
407:
             validity = predictions["polygon_validity"][0]
408:
             valid_polygons = (validity > 0.5).sum().item()
409:
             metrics["valid_polygon_count"] = valid_polygons
410:
411:
             # Average polygon area
412:
            polygons = predictions["polygons"][0]
413:
             areas = []
414:
             for poly_idx, (polygon, valid) in enumerate(zip(polygons, validity)):
415:
                 if valid > 0.5:
416:
                     # Compute polygon area using shoelace formula
                     points = polygon.cpu().numpy() * np.array(image_size)
417:
418:
                     valid_points = points[points.sum(axis=1) > 0]
419:
                     if len(valid_points) >= 3:
420:
                         area = compute_polygon_area(valid_points)
421:
                         areas.append(area)
422:
423:
             metrics["avg_polygon_area"] = np.mean(areas) if areas else 0.0
424:
425:
         return metrics
426:
427:
428: def compute_polygon_area(points):
429:
         """Compute polygon area using shoelace formula"""
         if len(points) < 3:</pre>
430:
431:
            return 0.0
432:
433:
        x = points[:, 0]
        y = points[:, 1]
434:
435:
436:
         # Shoelace formula
        area = 0.5 * abs(sum(x[i] * y[i+1] - x[i+1] * y[i] for i in range(-1, len(x)-1)))
437:
438:
        return area
439:
440:
441: def create_model_summary_report(model, sample_input, save_path="model_summary.txt"):
442:
         """Create detailed model summary report""
443:
         with open(save_path, "w") as f:
```

```
444:
            f.write("Neural-Geometric 3D Model Generator - Model Summary\n")
445:
            f.write("=" * 60 + \sqrt{n}")
446:
447:
            # Model architecture
            f.write("MODEL ARCHITECTURE:\n")
448:
449:
            f.write("-" * 20 + "\n")
450:
451:
            total_params = sum(p.numel() for p in model.parameters())
452:
            trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
453:
            f.write(f"Total parameters: {total_params:,}\n")
454:
455:
            f.write(f"Trainable parameters: {trainable_params:,}\n")
            f.write(f"Model size: {total_params * 4 / 1024 / 1024:.2f} MB\n\n")
457:
458:
            # Component breakdown
459:
            f.write("COMPONENT PARAMETERS:\n")
            f.write("-" * 25 + "n")
460:
461:
            encoder_params = sum(p.numel() for p in model.encoder.parameters())
462:
463:
            seg_params = sum(p.numel() for p in model.seg_head.parameters())
464:
            attr_params = sum(p.numel() for p in model.attr_head.parameters())
465:
            sdf_params = sum(p.numel() for p in model.sdf_head.parameters())
466:
            dvx_params = sum(p.numel() for p in model.dvx.parameters())
467:
            ext_params = sum(p.numel() for p in model.extrusion.parameters())
468:
469:
            f.write(f"Encoder: {encoder_params:,} ({encoder_params/total_params*100:.1f}%)\n")
470:
            f.write(f"Segmentation Head: {seg_params:,} ({seg_params/total_params*100:.1f}%)\n")
            f.write(f"Attribute Head: {attr_params:,} ({attr_params/total_params*100:.1f}%)\n")
471:
472:
            f.write(f"SDF Head: {sdf_params:,} ({sdf_params/total_params*100:.1f}%)\n")
473:
            474:
            f.write(f"Extrusion Module: {ext_params:,} ({ext_params/total_params*100:.1f}%)\n\n")
475:
476:
            # Forward pass analysis
            f.write("FORWARD PASS ANALYSIS:\n")
477:
            f.write("-" * 25 + "\n")
478:
479:
480:
            model.eval()
481:
            with torch.no_grad():
482:
                predictions = model(sample_input)
483:
484:
                for key, value in predictions.items():
485:
                    if torch.is tensor(value):
486:
                        f.write(f"{key}: {list(value.shape)} - {value.dtype}\n")
487:
                    else:
488:
                        f.write(f"{key}: {type(value)}\n")
489:
        print(f"Model summary saved to {save_path}")
490:
491:
492:
493: def debug_gradient_flow(model, loss):
        """Debug gradient flow through the model"""
494:
495:
        print("Gradient Flow Analysis:")
496:
        print("-" * 30)
497:
498:
        total_norm = 0
499:
        component_norms = {}
500:
501:
        for name, param in model.named_parameters():
502:
            if param.grad is not None:
503:
                param_norm = param.grad.norm().item()
504:
                total_norm += param_norm ** 2
505:
506:
                # Group by component
507:
                component = name.split('.')[0]
508:
                if component not in component_norms:
509:
                    component_norms[component] = 0
                component_norms[component] += param_norm ** 2
510:
511:
512:
        total_norm = total_norm ** 0.5
513:
514:
        print(f"Total gradient norm: {total_norm:.4f}")
515:
        print("Component gradient norms:")
516:
```

```
517:
        for component, norm in component_norms.items():
518:
            norm = norm ** 0.5
519:
            print(f" {component}: {norm:.4f} ({norm/total_norm*100:.1f}%)")
520:
521:
522: def create_3d_visualization(voxels, output_path="3d_preview.png"):
523:
        """Create 3D visualization of voxel prediction"""
524:
        try:
525:
             import matplotlib.pyplot as plt
            from mpl_toolkits.mplot3d import Axes3D
526:
527:
            # Convert to binary
528:
529:
            if isinstance(voxels, torch.Tensor):
530:
                voxels = voxels.cpu().numpy()
531:
532:
            binary_voxels = voxels > 0.5
533:
534:
             # Get occupied voxel coordinates
535:
            occupied = np.where(binary_voxels)
536:
537:
            if len(occupied[0]) == 0:
538:
                print("No occupied voxels to visualize")
539:
                return
540:
541:
             # Create 3D plot
            fig = plt.figure(figsize=(10, 8))
542:
            ax = fig.add_subplot(111, projection='3d')
543:
544:
            # Plot occupied voxels
545:
546:
            ax.scatter(occupied[0], occupied[1], occupied[2],
547:
                      c=occupied[2], cmap='viridis', s=1, alpha=0.6)
548:
549:
           ax.set_xlabel('X')
550:
           ax.set_ylabel('Y')
            ax.set_zlabel('Z')
551:
552:
            ax.set_title('3D Voxel Occupancy')
553:
554:
           plt.savefig(output_path, dpi=150, bbox_inches="tight")
555:
           plt.close()
556:
557:
            print(f"3D visualization saved to {output_path}")
558:
559:
         except ImportError:
560:
            print("3D visualization requires matplotlib with 3D support")
```
