Author: Zach Ghera
Created: Nov 18, 2021

Mesh R-CNN Mesh Sampler

Paper Summary

The mesh losses introduced in Mesh R-CNN are based on a differentiable mesh sampling operation to sample points and their normal vectors uniformly from the surface of the ground truth and predicted meshes. Points are uniformly sampled from the surface of the mesh by sampling a face f = (v1, v2, v3) from the mesh where the probability distribution of faces defined as:

$$P(f) = \frac{area(f)}{\sum_{f' \in F} area(f')}$$

Once a face is selected. We sample a point p uniformly from the interior of f by $p = \sum_i w_i v_i$

where

$$w_1 = 1 - \sqrt{\xi_1}, \quad w_2 = (1 - \xi_2)\sqrt{\xi_1}, w_3 = \xi_2\sqrt{\xi_1}, \text{ and } \xi_1, \xi_2 \sim U(0,1)$$

This use of the reparameterization trick (like VAEs) allows gradients of the loss wrt. points p can propagate back to gradients wrt. the face verticesv_i.

Original Implementation

Structure

The code for mesh sampling lies primarily in `sample_points_from_meshes()` of pytorch3d/pytorch3d/ops/sample_points_from_meshes.py.

This function is called from `_sample_meshes()` of meshrcnn/meshrcnn/utils/metrics.py. This function is called for each of the predicted and ground truth meshes that are passed into `compare_meshes()`, the function used to compute the mesh losses / metrics.
The only additional work performed in `_sample_meshes()` is handling the option to use the vertices of the mesh rather than sampling from the mesh. I was unable to find an instance where this option was used within the Mesh RCNN repo.

Back to `sample_points_from_meshes()`. There are three potential return tensors. The two that we are interested in for mesh sampling are `samples` and `normals` representing the coordinates of sampled points from the `Meshes` instance and corresponding normal vectors for each point. The other optional return tensor is `textures` which Mesh RCNN does not use.

Walkthrough

This section breaks down the different subcomponents of the mesh sampler implementation and provides some inline examples.

Dimensions

- B = batch size
- B_V = batch size - `num_empty_meshes`
- N = number of points to sample from mesh
- sum({V/F}_n) = Number of vertices/faces in the packed representation
- max({V/F}_n) = Max number of vertices/faces out of all meshes in the batch

Inputs

- **meshes**: A `Meshes` instance representing a batch of B meshes.
- **num_samples**: Integer giving the number of point samples per mesh. (N above)

Outputs

- **samples**: FloatTensor[B, N, 3] holding the coordinates of sampled points from each mesh in the batch. A samples matrix for a mesh will be 0 (i.e. samples[i, :, :] = 0) if the mesh is empty.
- **normals**: FloatTensor[B, N, 3] holding the normal vector for each sampled point. Like `samples`, an empty mesh will correspond to a 0 normals matrix.

Top-level pseudocode

```
faces ← IntTensor[sum(F_n), 3]: faces in packed representation.
verts ← FloatTensor[sum(V_n), 3]: vertices in packed representation.
mesh_to_face ← IntTensor[B,]: Index of the first face in the packed
      representation of each mesh.
samples ← Zero tensor with shape [B, N, 3].

sample_face_idxs ← sample_faces() # FloatTensor[Bv, N].

face_verts ← verts[faces] # FloatTensor[sum(F_n), 3, 3]: Tensor containing
      a matrix of the three vertices for each face.
v0, v1, v2 ← face_verts[:, 0], face_verts[:, 1], face_verts[:, 2]
      # Each FloatTensor[sum(F_n), 3] where each contains all of the ith
      (0, 1 or 2) vertices for each face of the mesh.

w0, w1, w2 ← rand_barycentric_coords() # Each FloatTensor[Bv, N] such that
```

```
        w0[i,j] + w1[i,j] + w2[i,j] == 1 for all i,j (i.e. barycentric
        coordinates).
a, b, c ← v0[sample_face_idxs], v1[sample_face_idxs], v2[sample_face_idxs]
        # Each FloatTensor[Bv, N, 3] where each represents the ith (0, 1 or
        2) vertices of the N sampled faces for the valid (non-zero) meshes.

samples[meshes.valid] = w0 * a + w1 * b + w2 * c # FloatTensor[B, N, 3]:
        Each vertex (x,y,z) for each sampled face is element wise multiplied
        with the respective barycentric ‘weight’. These are added together to
        produce the N vertices sampled from the triangle face of each sampled
        face of each mesh. Indexing with `meshes.valid` sets batch elements
        in the sample Tensor that correspond to non-empty meshes.
normals ← get_normals() # FloatTensor[B, N, 3].
return samples, normals
```

get_normals()

When mesh areas are calculated in `sample_faces()`, `mesh_face_areas_normals` calculates both the face areas AND the face normal vectors as both computations rely on the face normal vector computed from the cross product of two edges of the triangle. However, in the original implementation of the mesh sampler, the authors do not use the normals obtained from `mesh_face_areas_normals` and instead recalculate the normal vectors. I am not sure why.

Pseudocode

```
normals ← Zero tensor with shape [B, N, 3].
vert_normals ← cross_product((v1 - v0), (v2 - v1), dim=1) #
        FloatTensor[sum(F_n), 3]: Tensor containing the unnormalized normal
        vectors for each face of the mesh
vert_normals = vert_normals / norm(vert_normals, dim=1) # Normalize the
        face normal vectors
vert_normals = vert_normals[sample_face_idxs] # FloatTensor[Bv, N, 3]:
        Tensor that represents the normal vectors of the N sampled faces for
        the valid (non-zero) meshes.
normals[meshes.valid] = vert_normals # FloatTensor[B, N, 3]: Sets the
        Tensor that holds the normal vectors of the N sampled faces for all
        of the meshes. Indexing with `meshes.valid` sets batch elements in
        the Tensor that correspond to non-empty meshes.
```

sample_faces()

Pseudocode

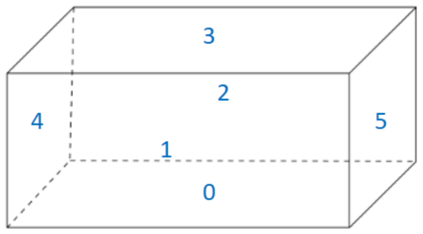
```
areas ← mesh_face_areas_normals()** # FloatTensor[sum(F_n),]: Packed
```

```
        version of the face areas. areas[i] is the area of faces[i].
areas_padded ← packed_to_padded()** # FloatTensor[Bv, max(F_n)]: Get
        padded version of `areas`.
sample_faces_idxs ← multinomial(areas_padded, num_samples) #
        FloatTensor[Bv, N]: Samples from the probability distribution over
        faces for each mesh in the batch give by PDF P(f) = area(f) /
        sum(area(f)). This is equivalent to sampling from a multinomial
        distribution for each mesh where the unnormalized probabilities are
        given by the face areas.
sample_faces_idxs += mesh_to_face[valid_meshes_idxs] # elementwise add the
        starting face idx to each row (face probabilities for each mesh).
        This changes the index from a local index relative to that mesh to
        the index that refers to that mesh face in the packed representation.
return sample_faces_idxs
```

** Implemented as custom operations in the original implementation.

Example

Take the example of a rectangular prism mesh. The actual mesh would have 12 faces since each face is a triangle, but for simplicity let's just say there are 6 rectangular faces:



```
areas_padded = [..., [6, 6, 6, 6, 2, 2], ...]
```

`sample_faces_idxs` is created by sampling N points from a multinomial distribution where the unnormalized probabilities (don't need to sum to 1) are given by the vector for each mesh in `areas_padded`. Let's say N = 10. Then we may get a `sample_faces_idxs` like this for the `areas_padded` above:

```
sample_faces_idxs= [..., [0, 1, 2, 1, 0, 3, 4, 4, 3, 6], ...]
```

Finally, we element wise add the index of the first face in the packed representation to each mesh/element in `sample_faces_idxs`. This can be thought of as transforming the face idx from an index local to the mesh to an index relative to the first dim of the packed faces tensor. In this example, let's say the index of face 0 for this rectangular prism mesh had index 9 in the face packed representation, then:

```
sample_faces_idxs = [..., [0, 1, 2, 1, 0, 3, 4, 4, 3, 6], ...]
```

```
                                  + [9, 9, 9, 9, 9, 9, 9, 9, 9, 9]

sample_faces_idxs = [..., [9, 10, 11, 10, 9, 12, 13, 13, 12, 15], ...]
```

TensorFlow Implementation

This section breaks down the different subcomponents of the TensorFlow mesh sampler implementation.

Dimensions

- B = batch size
- Nv = number of vertices. Calculated as `(vox_grid_dim + 1)^3`
- Nf = number of vertices. Calculated as `12 * (vox_grid_dim)^3`
- Ns = number of points to sample from mesh
- sum({V/F}_n) = Number of vertices/faces in the packed representation
- max({V/F}_n) = Max number of vertices/faces out of all meshes in the batch

Inputs

- **verts**: FloatTensor[B, Nv, 3], where the last dimension contains all (x,y,z) vertex coordinates in the initial mesh mesh.
- **verts_mask**: IntTensor[B, Nv], a mask for valid vertices in the watertight mesh.
- **faces**: IntTensor[B, Nf, 3], where the last dimension contains the verts indices that make up the face. This may include duplicate faces.
- **faces_mask**: IntTensor[B, Nf], a mask for valid faces in the watertight mesh.
- **num_samples**: Integer giving the number of point samples per mesh. (Ns above)

Outputs

- **samples**: FloatTensor[B, Ns, 3] holding the coordinates of sampled points from each mesh in the batch. A samples matrix for a mesh will be 0 (i.e. samples[i, :, :] = 0) if the mesh is empty.
- **normals**: FloatTensor[B, Ns, 3] holding the normal vector for each sampled point. Like `samples`, an empty mesh will correspond to a 0 normals matrix.

Top-level pseudocode

I am going to do this in code first rather than in this doc so I can print and inspect the tensors as I have ideas. I will make sure to comment heavily. And if requested, I can convert that code into pseudocode and put it here once I am done.