graphormer mathematical operations

minasmz

May 2023

1 Initialization

(graph_encoder): GraphormerGraphEncoder(			
	(dropout_module): FairseqDropout()	p = 0.0	
	(graph_node_feature): GraphNodeFeature(		
		(atom_encoder): Embedding (4609, 80, padding_id	Weights: {Parameter:(4609, 80)} normal_ (mean=0.0, std=0.02), then Weights [padding_idx].zeros_()
		(in_degree_encoder): Embedding (512, 80, padding_idx	Weights: {Parameter:(512, 80)} normal_ (mean=0.0, std=0.02), then Weights [padding_idx].zeros_()
		(out_degree_encoder): Embedding (512, 80, padding_idx	Weights: {Parameter:(512, 80)} normal_ (mean=0.0, std=0.02), then Weights [padding_idx].zeros_()

		(graph_token): Embedding(1, 80)	Weights: {Parameter:(1, 80)} normal_ (mean=0.0, std=0.02)
	)		
	(graph_attn_bias): GraphAttnBias(		
		(edge_encoder): Embedding (1537, 8, padding_idx	Weights: {Parameter:(1537, 8)} normal_ (mean=0.0, std=0.02), then Weights [padding_idx].zeros_()
		(edge_dis_encoder): Embedding(8192, 1)	Weights: {Parameter:(8192, 1)} normal_ (mean=0.0, std=0.02)
		(spatial_pos_encoder) Embedding (512, 8, padding_idx=	Weights: {Parameter:(512, 8)} normal_ (mean=0.0, std=0.02), then Weights [padding_idx].zeros_()
		(graph_token_virtual_ Embedding(1, 8)	Weights: {Parameter:(1, 8)} normal_ (mean=0.0, std=0.02)
	)		
	(emb_layer_norm):	LayerNorm((80,), eps=1e-05, elementwise_affine=T	Weights: {Parameter: (80,)}, Already weights.ones_() Bias: {Parameter: (80,)}, Already bias.zeros_()
	(layers): ModuleList(0-11)		
)			

(masked_lm_pooler): Linear (in_features=80, out_features=80, bias=True)	Weights: {Parameter:(80, 80)} normal_(mean=0.0, std=0.02) Bias: {Parameter: (80,)}, bias.zeros_() (it is already zero)		
(lm_head_transform_w Linear (in_features=80, out_features=80, bias=True)	Weights: {Parameter:(80, 80)} normal_(mean=0.0, std=0.02) Bias: {Parameter: (80,)}, bias.zeros_() (it is already zero)		
(layer_norm): LayerNorm((80,), eps=1e-05, elementwise_affine=T	Weights: {Parameter: (80,)}, Already weights.ones_() Bias: {Parameter: (80,)}, Already bias.zeros_()		
(embed_out): Linear (in_features=80, out_features=1, bias=False)	Weights: {Parameter:(1, 80)} normal_(mean=0.0, std=0.02)		

The ModuleList is initialized as:

(0-11): Graphormer GraphEncoderLayer(			
	(dropout_module): FairseqDropout()	p=0.0	
	(activation_dropout _module): FairseqDropout()	p=0.1	
	(self_attn): MultiheadAttention(		
		(dropout_module): FairseqDropout()	p=0.0
		(k_proj): Linear(in_features=8(out_features=80, bias=True)	Weights: {Parameter:(80, 80)} normal_(mean=0.0, std=0.02) Bias: {Parameter: (80,)}, bias.zeros_() (it is already zero)

		(v_proj): Linear(in_features=8 out_features=80, bias=True)	Weights: {Parameter:(80, 80)} normal_(mean=0.0, std=0.02) Bias: {Parameter: (80,)}, bias.zeros_() (it is already zero)
		(q_proj): Linear(in_features=8 out_features=80, bias=True)	Weights: {Parameter:(80, 80)} normal_(mean=0.0, std=0.02) Bias: {Parameter: (80,)}, bias.zeros_() (it is already zero)
		(out_proj): Linear(in_features=8 out_features=80, bias=True)	Weights: {Parameter:(80, 80)} normal_(mean=0.0, std=0.02) Bias: {Parameter: (80,)}, bias.zeros_() (it is already zero)
	)	Again initialize weights of k_proj, v_proj, q_proj by sampling from normal_(mean=0, std=0.02)	
	(self_attn_layer_norm) LayerNorm((80,), eps=1e-05, elementwise_ affine=True)	Weights: {Parameter: (80,)}, Already weights.ones_() Bias: {Parameter: (80,)}, Already bias.zeros_()	
	(fc1): Linear(in_features=8 out_features=80, bias=True)	Weights: {Parameter:(80, 80)} normal_(mean=0.0, std=0.02) Bias: {Parameter: (80,)}, bias.zeros_() (it is already zero)	
	(fc2): Linear(in_features=8 out_features=80, bias=True)	Weights: {Parameter:(80, 80)} normal_(mean=0.0, std=0.02) Bias: {Parameter: (80,)}, bias.zeros_() (it is already zero)	
	(final_layer_norm): LayerNorm((80,), eps=1e-05, elementwise_ affine=True)	Weights: {Parameter: (80,)}, Already weights.ones_() Bias: {Parameter: (80,)}, Already bias.zeros_()	
)			

2 Layers Function

torch.layer_norm normalizes the input tensor by subtracting the mean along the specified dimension(s) and dividing by the standard deviation along the same dimension(s), where the mean and standard deviation are computed over all the remaining dimensions of the input tensor. If weight and bias are provided, they are applied elementwise to the normalized tensor before returning the output.

In the documentation of torch.LayerNorm we have:

Applies Layer Normalization over a mini-batch of inputs as described in the paper [?].

$$y = \frac{x - E[x]}{\sqrt{var[x] + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated over the last D dimensions, where D is the dimension of normalized_shape. γ and β are learnable affine transform parameters of normalized_shape if elementwise_affine is True.

3 Data processing

We registered our dataset "hamiltonian_cycle" as a dgl_dataset object. Thus, to process it, the __preprocess_dgl_graph() function in graphormer/data/dgl_datasets/dgl_dataset.py is called.

```
node_int_feature # {Tensor:(num_node, 1)}
edge_int_feature #{Tensor:(num_edge, 1)}
N = graph_data.num_nodes()
edge_index = graph_data.edges()
#it is a {tuple: 2} each tuple is a tensor {Tensor: (num_edges,)}
#edge_index[0], edge_index[1] indicate src and dst
attn_edge_type = [N, N, edge_int_feature.shape[1] = 1] #consider it as a adjacency matrix
dense_adj #a {Tensor:(N, N)} as an adjacency matrix
shortest_path_result, path = algos.floyd_warshall(dense_adj.numpy())
max_dist = np.amax(shortest_path_result)
edge_input = algos.gen_edge_input(max_dist, path, attn_edge_type.numpy())
spatial_pos = torch.from_numpy((shortest_path_result)).long()}
attn_bias = torch.zeros([N + 1, N + 1], dtype=torch.float)}
```

In shortest_path_result, path = algos.floyd_warshall(dense_adj.numpy()), the shortest_path_result is a numpy array as {ndarray: (N, N)} containing the distance between node i and j. The path variable is also {ndarray: (N, N)} where if i be same as j or if i and j directly connect it is equal to -1. Otherwise, it represent the index of the node which is the next node in the shortest path to j.

The edge_input is a {ndarray: (N, N, max_dist, edge_int_feature.shape[1])} where it is filled with -1 or 1. For the edge_input[i][j][:] we have edge_input[i][j][k] = 1 if the shortest path between i and j be greater equal to k, otherwise it is

-1. The spatial_pos is same as the shortest_path_result except it is converted to the Tensor format.

Further,

```
pyg_graph = PYGGraph()
pyg_graph.x = convert_to_single_emb(node_int_feature)
pyg_graph.adj = dense_adj
pyg_graph.attn_bias = attn_bias
pyg_graph.attn_edge_type = attn_edge_type
pyg_graph.spatial_pos = spatial_pos
pyg_graph.in_degree = dense_adj.long().sum(dim=1).view(-1)
pyg_graph.out_degree = pyg_graph.in_degree
pyg_graph.edge_input = torch.from_numpy(edge_input).long()
if y.dim() == 0:
    y = y.unsqueeze(-1)
pyg_graph.y = y
pyg_graph.idx = idx
```

The pyg_graph.x is a Tensor of unit values with a size same as node_int_feature which is {Tensor:(number_of_nodes, 1)}.

Notice that when we create a PYGGraph() object the y is a {Tensor:()} in other words tensor(). To this end, by y=y.unsqueeze(-1) we convert y to a {Tensor: (1,)} in other words, tensor([1]).

4 Forward Pass in the Layers

```
(encoder): GraphormerEncoder(
    (graph_encoder): GraphormerGraphEncoder(
          (dropout_module): FairseqDropout() # dropout nodes → p=0
          (graph_node_feature): GraphNodeFeature(
            (atom_encoder): Embedding(4609, 80, padding_idx=0)
            (in_degree_encoder): Embedding(512, 80, padding_idx=0)
            (out_degree_encoder): Embedding(512, 80, padding_idx=0)
            (graph_token): Embedding(1, 80)
          )
        (graph_attn_bias): GraphAttnBias(
            (edge_encoder): Embedding(1537, 8, padding_idx=0)
            (edge_dis_encoder): Embedding(8192, 1)
            (spatial_pos_encoder): Embedding(512, 8, padding_idx=0)
            (graph_token_virtual_distance): Embedding(1, 8)
          )
        (emb_layer_norm): LayerNorm((80,), eps=1e-05, elementwise_affine=True)
        (layers): ModuleList(0-11)
     )
(masked_lm_pooler): Linear(in_features=80, out_features=80, bias=True)
(lm_head_transform_weight): Linear(in_features=80, out_features=80, bias=True)
```

```
(layer_norm): LayerNorm((80,), eps=1e-05, elementwise_affine=True)
(embed_out): Linear(in_features=80, out_features=1, bias=False)
)
```

Where ModuleList contains 12 similar layers, one layer is as follows:

```
(0): GraphormerGraphEncoderLayer(
          (dropout_module): FairseqDropout()
          (activation_dropout_module): FairseqDropout()
          (self_attn): MultiheadAttention(
            (dropout_module): FairseqDropout()
            (k_proj): Linear(in_features=80, out_features=80, bias=True)
            (v_proj): Linear(in_features=80, out_features=80, bias=True)
            (q_proj): Linear(in_features=80, out_features=80, bias=True)
            (out_proj): Linear(in_features=80, out_features=80, bias=True)
          )
          (self_attn_layer_norm): LayerNorm((80,), eps=1e-05, elementwise_affine=True)
          (fc1): Linear(in_features=80, out_features=80, bias=True)
          (fc2): Linear(in_features=80, out_features=80, bias=True)
          (final_layer_norm): LayerNorm((80,), eps=1e-05, elementwise_affine=True)
        )
```

GraphormerEncoder outputs (from the embed_out) a probability indicating the probability of the class for the input graph. The input passed through several layers namely, (graph_encoder), (masked_lm_pooler), (lm_head_transform_weight), (layer_norm), and (embed_out) layers.

The (graph_encoder) layer is from the class of GraphormerGraphEncoder. We explain it in the next section

4.1 GraphormerGraphEncoder

In GraphormerGraphEncoder, we have several layers as follows,

(dropout_module), (graph_node_feature), (graph_attn_bias), (emb_layer_norm), (ModuleList(0-11))

The p in the specification of (dropout_module) equals zero, so it does nothing.

The (graph_node_feature) is from the class GraphNodeFeature. The specification is in the next section.

4.1.1 (GraphNodeFeature)

Once again see the GraphNodeFeature encoding layers:

```
(graph_node_feature): GraphNodeFeature(
        (atom_encoder): Embedding(4609, 80, padding_idx=0)
        (in_degree_encoder): Embedding(512, 80, padding_idx=0)
        (out_degree_encoder): Embedding(512, 80, padding_idx=0)
```

```
        (graph_token): Embedding(1, 80)
      )
```

It will be encoded by the following variables.

```
self.graph_node_feature =
    GraphNodeFeature(
    num_heads=8,
    num_atoms=9*512,
    num_in_degree=512,
    num_out_degree=512,
    hidden_dim=80,
    n_layers=12
    )
```

Where we have following:

```
node_feature = self.atom_encoder(x).sum(dim=-2)  # [n_graph, n_node, n_hidden]
node_feature = (
        node_feature
        + self.in_degree_encoder(in_degree)
        + self.out_degree_encoder(out_degree)
            )

graph_token_feature = self.graph_token.weight.unsqueeze(0).repeat(n_graph, 1, 1)
graph_node_feature = torch.cat([graph_token_feature, node_feature], dim=1)
return graph_node_feature
```

Consider N = max number of nodes in batch of graphs, The x is a {Tensor: (4, N, 1)}, out_degree and in_degree are the same with {Tensor:(4, N)}.

Applying the **(atom_encoder)** layer, it is a lookup table of 4609 nodes to an 80 dim feature. The result is Tensor:(4, N, 1, 80). The dim over the -2 dim makes it a Tensor:(4, N, 80).

(in_degree_encoder) and **(out_degree_encoder)** Embedding layers are applied on two tensors of size Tensor: (4, N), and the result is two Tensor:(4, N, 80).

node_feature = node_feature + encoded_in_degree + encoded_out_degree

(graph_token) The initialized self.graph_token.weight (can be considered as Embedding layer producing graph_token_feature) with size (1, 80) will be (1, 1, 80), Then repeated 4 times. Thus, it will be (4, 1, 80).

graph_node_feature = cat([graph_token_feature, node_feature], dim=1)

The result is a Tensor:(4, N+1, 80)

4.2 (GraphAttnBias)

The graph_attn_bias at first is a copy of attn_bias. Then at the second index (unsqueeze(1)) we add a dimension. As a result, the graph_attn_bias which was equal to Tensor:(batch size, N + 1, N + 1) will be converted to size Tensor:(batch size, 1, N + 1, N + 1). Then it will be repeated for the number of heads. As a result the dimension of graph_attn_bias will be Tensor:(batch size = 4, number of heads = 8, N + 1, N + 1)

(spatial_pos_encoder)

```
# spatial pos
# [n_graph, n_node, n_node, n_head] -> [n_graph, n_head, n_node, n_node]
spatial_pos_bias = self.spatial_pos_encoder(spatial_pos).permute(0, 3, 1, 2)
graph_attn_bias[:, :, 1:, 1:] = graph_attn_bias[:, :, 1:, 1:] + spatial_pos_bias
```

spatial_pos contains the shortest path between two nodes and the size is Tensor:(4, N, N) spatial_encoder is an Embedding for 512 nodes to an 8 dimension encoder. So, spatial_pos_bias will be equal to (4, 8, N, N).

spatial_pos_bias encodes the shortest paths. Then the graph_attn_bias with the size (4, 8, N+1, N+1) is updated in the following way. Except for the first index of (N+1, N+1), to the rest of the (N, N) the spatial_pos_bias is added to it. Remember that graph_attn_bias has zero or -inf values.

(graph_token_virtual)

```
# reset spatial pos here
t = self.graph_token_virtual_distance.weight.view(1, self.num_heads, 1)
graph_attn_bias[:, :, 1:, 0] = graph_attn_bias[:, :, 1:, 0] + t
graph_attn_bias[:, :, 0, :] = graph_attn_bias[:, :, 0, :] + t
```

graph_token_virtual_distance.weight (Embedding of graph_token_virtual_distance) from tensor with the shape of {Tensor: (1, 8)} to a tensor with the shape of {Tensor:(1, 8, 1)}.

The two last lines add the t with the (1, 8, 1) dimension to all the first rows and the first columns of (4, 8, N+1, N+1) dimensions. In other words, considering a (N+1, N+1) the first row and the first column will be changed by adding t to them.

(edge encoder)

```
#edge_input : [n_graph, n_node, n_node, max_dist, n_head]
edge_input = self.edge_encoder(edge_input).mean(-2)
```

edge_input is an embedding vector of the size Tensor: (4, N, N, 5, 1, 8). Where applying .mean(-2) on it convert it to Tensor:(4, N, N, 5, 8).

(edge_dis_encoder)

```
edge_input_flat = edge_input.permute(3, 0, 1, 2, 4).reshape(max_dist, -1, self.num_heads)
edge_input_flat = torch.bmm(
                edge_input_flat,
                self.edge_dis_encoder.weight.reshape(
                    -1, self.num_heads, self.num_heads
                )[:max_dist, :, :],
            )
```

To calculate edge_input_flat we have Tensor:(4, N, N, 5, 8) by permutation it will be Tensor:(5, 4, N, N, 8), by reshaping will be Tensor:(5, 4*N*N, 8).

For the second matrix, we reshaping a tensor with .reshape(-1, 8, 8) on a tensor of size (8192, 1) would transform it into a tensor of shape (1024, 8, 8).

The first dimension is inferred by using -1, which means that the number of elements in that dimension will be calculated automatically based on the other dimensions and the total number of elements in the tensor. In this case, -1 is inferred as 1024 because 8 * 8 = 64, and 8192 / 64 = 1024.

by the [:max_dist, :, :], only the first 5 matrices will be kept. Then it multiply a Tensor:(5, N*N*4, 8) with Tensor:(5, 8, 8).

Then,

```
edge_input = edge_input_flat.reshape(
            max_dist, n_graph, n_node, n_node, self.num_heads
        ).permute(1, 2, 3, 0, 4)
```

The edge_input_flat will be reshaped from a Tensor: (5, 4 * N * N, 8) to a Tensor:(5, 4, N, N, 8). Then, it will permute and convert to the following tensor with shape Tensor:(4, N, N, 5, 8).

Then the edge_input will be:

```
edge_input = (
            edge_input.sum(-2) / (spatial_pos_.float().unsqueeze(-1))
            ).permute(0, 3, 1, 2)
```

Intuitivly?

As the edge_input is a Tensor:(4, N, N, 5, 8), by summing over the 5 dimensions in index -2, we convert it to Tensor:(4, N, N, 8) by summing over the 5 dimensions. The spatial_pos_.float().unsqueeze(-1) will convert the Tensor:(4, N, N) to Tensor:(4, N, N, 1). The result size is the nominator. The reshaping process will convert the result to Tensor:(4, 8, N, N). Thus, edge_input will be a Tensor:(4, 8, N, N).

At last, we have the following lines,

```
graph_attn_bias[:, :, 1:, 1:] = graph_attn_bias[:, :, 1:, 1:] + edge_input
graph_attn_bias = graph_attn_bias + attn_bias.unsqueeze(1)  # reset
return graph_attn_bias
```

Where the graph_attn_bias is Tensor:(4, 8, N+1, N+1). We add the edge_input to all the Tensors except the first rows and columns of the N+1 and N+1 matrices. Then the attn_bias as the Tensor:(4, n+1, N+1) will converted to Tensor:(4, 1, N+1, N+1). and it is added to the graph_attn_bias which was previously updated by adding the edge_input to all the N*N elements form the N+1*N+1 elements (did not add to the first row and first column). We return this value as the result of GraphAttnBias.

4.2.1 (emb layer norm)

We have a emb_layer_norm (80,), where x which is a GraphNodeFeature Tensor:(4, N+1, 80) will pass through it and will update.

```
if self.emb_layer_norm is not None:
    x = self.emb_layer_norm(x)
x = self.dropout_module(x))  # do nothing (p=0)
# B x T x C -> T x B x C
x = x.transpose(0, 1)
```

The x is a Tensor:(4, N+1, 80), which goes through a dropout layer with probability 0.0, which means it does nothing. The x will switch the first and second dimensions and become of the form Tensor:(N+1, 4, 80).

Then we create a list to save the x (node features).

```
inner_states = []
if not last_state_only:
    inner_states.append(x)
```

4.3 (layers) ModuleList(0-11) (GraphormerGraphEncoderLayer)

```
(0): GraphormerGraphEncoderLayer(
(dropout_module): FairseqDropout()
(activation_dropout_module): FairseqDropout()
(self_attn): MultiheadAttention(
  (dropout_module): FairseqDropout()
  (k_proj): Linear(in_features=80, out_features=80, bias=True)
  (v_proj): Linear(in_features=80, out_features=80, bias=True)
  (q_proj): Linear(in_features=80, out_features=80, bias=True)
  (out_proj): Linear(in_features=80, out_features=80, bias=True)
)
(self_attn_layer_norm): LayerNorm((80,), eps=1e-05, elementwise_affine=True)
(fc1): Linear(in_features=80, out_features=80, bias=True)
(fc2): Linear(in_features=80, out_features=80, bias=True)
(final_layer_norm): LayerNorm((80,), eps=1e-05, elementwise_affine=True)
```

The layer function gets the x (normalized encoded nodes) which is Tensor:(N+1, 4, 80), self_attn_padding_mask=padding_mask which is a Tensor:(4, N+1) and contains False for the nodes that exist (True for the nodes that do not exist), self_attn_mask=attn_mask which is the None, self_attn_bias=attn_bias which is a GraphAttnBias object (containing the encoded edges) and has the form of Tensor:(4, 8, N+1, N+1).

```
residual = x # normalized GraphNodeFeature
x, attn = self.self_attn(
```

```
            query=x,
            key=x,
            value=x,
            attn_bias=self_attn_bias,
            key_padding_mask=self_attn_padding_mask,
            need_weights=False,
            attn_mask=self_attn_mask,
        )
x = self.dropout_module(x)
x = residual + x
if not self.pre_layernorm:
    x = self.self_attn_layer_norm(x)
residual = x
x = self.activation_fn(self.fc1(x))
x = self.activation_dropout_module(x)
x = self.fc2(x)
x = self.dropout_module(x)
x = residual + x
if not self.pre_layernorm:
    x = self.final_layer_norm(x)
return x, attn
```

It calculates the self_attention first. It gets the x as the normalized GraphNodeFeature, self_attn_bias=attn_bias which is a GraphAttnBias object (containing the encoded edges), and some other parameters.

4.3.1 (self_attn): MultiheadAttention

the query, key, value which are the same as the x (copies of x) with the size Tensor:(N+1, 4, 8), and attn_bias which is a Tensor:(4, 8, N+1, N+1), key_padding_mask which is a Tensor:(4, N+1), need_weight which is False, attn_mask which is None.

We have:

(q_proj), **(k_proj)**, **(v_proj)**

```
q = self.q_proj(query)
k = self.k_proj(query)
v = self.v_proj(query)
q *= self.scaling
```

self.q_proj(query), self.k_proj(query), self.v_proj(query) are Linear(in_feature=80, out_feature=80, bias=True) transformation on query (which is x as the normalized GraphNodeFeature Tensor:(N+1, 4, 80)). To be specific, weight is a {Parameter:(80, 80)}, bias is a {Parameter:(80,)}, input is a {Tensor:(N+1, 4, 80)}.

We have a scaling variable as,

```
self.scaling = self.head\_dim ** -0.5 = 0.31622776601683794
```

Which will scale the q.
Then we call,

```
q = (
    q.contiguous()
    .view(tgt_len = N+1, bsz = 4 * self.num_heads = 8, self.head_dim = 10)
    .transpose(0, 1)
    )
```

The q is a Tensor:(N+1, 4, 80) The view converts it to Tensor:(N+1, 4*8, 10). Transposing 0 with 1, it will convert to Tensor:(4*8, N+1, 10). For the k and v, we have the same procedure.

Then in the following lines, we have:

```
attn_weights = torch.bmm(q, k.transpose(1, 2))
```

torch.bmm will multiply q and k.transpose(1, 2). In this way a Tensor:(32, N+1, 10) * Tensor:(32, 10, N+1) which is a valid multiplication. The result in attn_weights will be a Tensor:(4*8, N+1, N+1).

Then we have what follows in the forward call in multihead_attention.py,

```
if attn_bias is not None:
    attn_weights += attn_bias.view(bsz * self.num_heads, tgt_len, src_len)
```

The attn_bias is a Tensor:(4, 8, N+1, N+1), the view function convert it to Tensor:(4*8, N+1, N+1). Then it will be added to attn_weights.

Then we have,

```
if key_padding_mask is not None:
    # don't attend to padding symbols
    attn_weights = attn_weights.view(bsz, self.num_heads, tgt_len, src_len)
    attn_weights = attn_weights.masked_fill( key_padding_mask.unsqueeze(1).
    unsqueeze(2).to(torch.bool), float("-inf"),
        )
    attn_weights = attn_weights.view(bsz * self.num_heads, tgt_len, src_len)
```

attn_weights is Tensor:(4*8, N+1, N+1). It will converted to Tensor:(4, 8, N+1, N+1). key_padding_mask is a Tensor:(4, N+1), then, key_padding_mask is converted to Tensor:(4, 1, 1, N+1). So on all heads and all nodes it will change the float("-inf") values to True. At last, the attn_weights will converted to Tensor:(4*8, N+1, N+1).

Then we will go through the following,

```
    attn_weights_float = utils.softmax(
            attn_weights, dim=-1, onnx_trace=self.onnx_trace
        )
    attn_weights = attn_weights_float.type_as(attn_weights)
    attn_probs = self.dropout_module(attn_weights)
```

utils.softmax() is a function that applies the softmax function along a specified dimension of a tensor. In this case, the input tensor has shape (4, 8, N+1, N+1), and the dim=-1 argument indicates that the softmax function will be applied along the last dimension of the tensor.

In this case, the input tensor attn_weights has dimensions (4 * 8, N+1, N+1), where the softmax operation is applied along the last dimension (dim=-1), which means that the softmax is applied independently to each spatial location of each head for each element in the batch.

$softmax(x_i) = exp(x_i)/sum_j(exp(x_j))$

The self.dropout_module(attn_weights) apply a dropout on attn_weights as Tensor:(4 * 8, N+1, N+1) with p=0.1.

After this dropout layer on attn_probs, we have:

```
attn = torch.bmm(attn_probs, v)
```

multiply the attn_probs of q and k to v. Where attn_prob is a Tensor:(32, N+1, N+1) and v is Tensor:(32, N+1, 10). The result is assigned to attn variable as a Tensor:(32, N+1, 10). Then,

(out_proj)

```
attn = attn.transpose(0, 1).contiguous().view(tgt_len, bsz, embed_dim)
attn = self.out_proj(attn)
```

Where the attn will be converted to a Tensor:(N+1, 4, 80). self.out_proj value is Linear(in_features=80, out_features=80, bias=True) applied on the attn Tensor:(N+1, 4, 80).

Where bias is {Parameter:(80,)}, with all zero values. weight as a {Parameter:(80, 80)} uniformlt sampled from normal distribution with mean=0, std=0.02.

Finally,

```
return attn, attn_weights  # attn = Linear(attn = attn_weights * v), None
```

4.3.2 rest of GraphormerGraphEncoderLayer

We have,

```
(self_attn_layer_norm): LayerNorm((80,), eps=1e-05, elementwise_affine=True)
(fc1): Linear(in_features=80, out_features=80, bias=True)
(fc2): Linear(in_features=80, out_features=80, bias=True)
(final_layer_norm): LayerNorm((80,), eps=1e-05, elementwise_affine=True)
```

The x, attn will be assigned by attn, None in the following code.

```
residual = x # normalized GraphNodeFeature
x, attn = self.self_attn(
            query=x,
            key=x,
            value=x,
```

```
            attn_bias=self_attn_bias,
            key_padding_mask=self_attn_padding_mask,
            need_weights=False,
            attn_mask=self_attn_mask,
        )
x = self.dropout_module(x)
x = residual + x
if not self.pre_layernorm:
    x = self.self_attn_layer_norm(x)
residual = x
x = self.activation_fn(self.fc1(x))
x = self.activation_dropout_module(x)
x = self.fc2(x)
x = self.dropout_module(x)
x = residual + x
if not self.pre_layernorm:
    x = self.final_layer_norm(x)
return x, attn
```

(self_attn_layer_norm) x (output of attention Tensor:(N+1, 4, 80)) goes through a dropout layer with p=0. Then, we add the normalized GraphNodeFeature to it. The x will go through a layerNorm layer along the normalization dimension, which is the last dimension (the dimension with the length of 80). It then applies the following formula to compute the normalized output:

$$output = \frac{(input - mean)}{\sqrt{(variance + eps)}}$$

Then,

$$output = weight * output + bias$$

The function returns the normalized and scaled tensor with the same shape as the input tensor.

The new x will copy to the residual variable. Then, it will pass through the following:

(fc1)

```
x = self.activation_fn(self.fc1(x))
```

where self.fc1 is Linear(in_features=80, out_features=80, bias=True), Input is the x Tensor:(N+1, 4, 80). The weight is a {Parameter: (80, 80)}, bias is a {Parameter:(80, 0)}. Then, input transform linearly. The result x Tensor:(N+1, 4, 80) will go through an activation function gelu. gelu function as follows:

$$GELU(x) = 0.5 * x * (1 + Tanh(\sqrt{(2/\pi)} * (x + 0.044715 * x^3)))$$

Then it will go through

```
x = self.activation_dropout_module(x)
```

When dropout layer used after an activation function like ReLU or GELU, dropout can be especially effective. This is because activation functions tend to produce outputs that are either very large or very small, and dropout can help to prevent the network from becoming too dependent on these extreme values. By randomly dropping out some of the activations, dropout can encourage the network to learn more smoothly varying features that are less sensitive to individual data points.

Here the dropout layer has p=0.1. Then, we have:

(fc2)

```
x = self.fc2(x)
```

self.fc2 is a Linear(in_features=80, out_features=80, bias=True). In self, the bias Tensor:(80,) is all zeroes. The linear transformation of x as Tensor:(N+1, 4, 80)will be returned. Notice that the weight and bias are as type Parameter which are learnable.

Then we have:

```
x = self.dropout_module(x)
```

Which do nothing. Next, we have:

```
x = residual + x
```

Which adds the x which went through linear → gelu → activation_dropout → linear transformation to the first x. I believe not passing x (and save it in residual) through activation is preventing it from getting 0 values when passing through the activation dropout, in this way, even when x goes through the activation and dropout activation which makes some indices as zero, it will be added to the first x, so none of the indices will become zero.

(final_layer_norm) The combined x will go through a final normalization layer as follows,

```
if not self.pre_layernorm:
    x = self.final_layer_norm(x)
```

Where the self.pre_layernorm is LayerNorm((80,), eps=1e-05, elementwise_affine=True), input is a Tensor:(N+1, 4, 80). The normalized x will be assigned to x.

Finally,

```
return x, attn
```

where the attn is None (the attn_weight of the self_attn).

It will jump back to graphormer_graph_encoder.py file in this part.

```
for layer in self.layers:
    x, _ = layer(
        x,
        self_attn_padding_mask=padding_mask,
        self_attn_mask=attn_mask,
        self_attn_bias=attn_bias,
        )
    if not last_state_only:
    inner_states.append(x)
```

The result of x, attn will assign to x, -. Since the last_state_only is False. The return x will append to the inner_states. Notice that the inner_states[0] is the node feature. This will pass through all the 12 layers. Then we have:

```
    graph_rep = x[0, :, :]
```

As x is a Tensor:(N+1, 4, 80), the first index contain the information of virtual node in 4 graphs represented in 80 dim. The x as the output of the last layer (layer 11) will be considered as the graph_rep.

Finally, we have following line which will finish encoding the Graphormer-GraphEncoder.

```
    return inner_states, graph_rep
```

This returns the inner_states containing the x of the 12 layers and the graph node feature as the first element, (len(inner_states)=13), and the graph_rep to the GraphormerEncoder.

```
def forward(self, batched_data, perturb=None, masked_tokens=None, **unused):
    inner_states, graph_rep = self.graph_encoder(
            batched_data,
            perturb=perturb,
        )
```

5 rest of GraphormerModel

The rest of forward function in GraphormerEncoder is as follows:

```
    x = inner_states[-1].transpose(0, 1)
    # project masked tokens only
    if masked_tokens is not None:
        raise NotImplementedError
    x = self.layer_norm(self.activation_fn(self.lm_head_transform_weight(x)))
    # project back to size of vocabulary
    if self.share_input_output_embed and hasattr(
        self.graph_encoder.embed_tokens, "weight"
        ):
```

```
        x = F.linear(x, self.graph_encoder.embed_tokens.weight)
    elif self.embed_out is not None:
        x = self.embed_out(x)
    if self.lm_output_learned_bias is not None:
        x = x + self.lm_output_learned_bias
    return x
```

It takes the x as the GraphormerGraphEncoderLayer of the last layer out of 12 layers with the size of Tensor:(N+1, 4, 80).

Then, we have the following line,

```
x = self.layer_norm(self.activation_fn(self.lm_head_transform_weight(x)))
```

self.lm_head_transform_weight(x) is Linear(in_features=80, out_features=80, bias=True) where the weight and bias are Parameter with size (80, 80), and (80,). The input containing x as a Tensor:(4, N+1, 80). The x will linearly transform. The result is a x with the same size Tensor:(4, N+1, 80).

Next, the self.activation_fn will apply on x where a Gelu activation function will apply on x. Then a layer_norm layer will apply on x which is a Tensor:(4, N+1, 80). The normalized tensor will be assigned to x as a Tensor:(4, N+1, 80).

```
x = self.embed_out(x)
x = x + self.lm_output_learned_bias

return x
```

x = self.embed_out(x) is called which is a Linear(in_features=80, out_features=1, bias=False). The input is tuple:1 as a Tensor:(4, n+1, 80). As a result, the x is updated by a linear transformation and a Tensor:(4, N+1, 1) assign to x.

Next, self.lm_output_learned_bias which is a learnable {Parameter: (1,)} and is initialized by 0 is added to x.

Finally, the x will be returned as Tensor:(4, N+1, 1) will be assigned the the logits.

```
logits = logits[:, 0, :]
targets = model.get_targets(sample, [logits])
loss = nn.L1Loss(reduction="sum")(logits, targets[: logits.size(0)])
return loss, sample_size
```

Since the first element in N+1 dimension represent the virtual node (a representative node of all nodes in graph), the logits take just that value as a Tensor:(4, 1, 1).

nn.L1Loss is a predefined loss function in PyTorch, which computes the mean absolute error between the predicted logits and the ground truth targets.

In this case, reduction is set to "sum", so the sum of absolute errors is returned instead of the mean. targets[: logits.size(0)] is used to make sure that only the first logits.size(0) elements of targets are used for the computation.

So, nn.L1Loss(reduction=”sum”)(logits, targets[: logits.size(0)]) will return the sum of absolute errors between logits and targets for the first logits.size(0) elements.

$$L1Loss(\hat{y}, y) = \frac{1}{n}\sum_{i=1}^{n} |\hat{y}_i - y_i| \quad (1)$$

In the case of nn.L1Loss(reduction=”sum”), the individual loss values for each element are summed across all elements to give a single scalar value. Thus, the loss will be assigned as a Tensor: () with a single value.

This loss will be backpropagate through the layers and update the Parameters

6 Summary

Consider input in a batch of 4:

x = { 5} as the GraphNodeFeature

1. Input x as a Tensor:(4, N, 1) goes through an embedding layer with size (4609, 80), results in Tensor:(4, N, 1, 80). It will be summed over the -2 dim.
2. Calculate in_degree and out_degree embedding with size Tensor:(512, 80), results in encoded in_degree and out_degree with size Tensor:(4, N, 80).
3. Calculate the summation of the 1 and 2
4. Calculate the embedding of graph_token for all the 4 graphs in batch data results in Tensor:(4, 1, 80)
5. Concatenate 4,3 results in Tensor:(4, N+1, 80)

Next,

attn_bias is a Tensor:(4, N+1, N+1) which goes through a transformation attn_bias = { 13 } as the GraphAttnBias

1. graph_attn_bias : copy of attn_bias
2. graph_attn_bias : unsqueeze(1) the graph_attn_bias. Therefore graph_attn_bias is Tensor:(4, 1, N+1, N+1), repeat for all number_head = 8. As a result, graph_attn_bias is Tensor:(4, 8, N+1, N+1)
3. spatial_pos (shortest paths) is Tensor:(4, N, N). Its embedding is calculated by (spatial_pos_encoder) (512, 8). The result is Tensor:(4, N, N, 8), it will permute to Tensor: (4, 8, N, N).
4. The graph_attn_bias with the size (4, 8, N+1, N+1) is updated in the following way. Except for the first index of (N+1, N+1), to the rest of the (N, N) the spatial_pos_bias is added to it. Remember that graph_attn_bias has zero or -inf values.

5. t: the embedding of (Embedding of graph token virtual distance as a Tensor: (1, 8)) converted to Tensor:(1, 8, 1)

6. update the graph_attn_bias by adding the t with the (1, 8, 1) dimension to all the first rows and the first columns of (4, 8, N+1, N+1) dimensions. In other words, considering a (N+1, N+1) the first row and the first column will be changed by adding t to them.

7. edge_input Tensor:(4, N, N, 5, 1) is encoded by (edge encoder) layer is Tensor:(1537, 8). Thus it will be Tensor:(4, N, N, 5, 1, 8). Calculating the mean over the dim -2, converts it to Tensor:(4, N, N, 5, 8).

8. To calculate the edge_input flat, we have edge_input as Tensor:(4, N, N, 5, 8) by permutation will be Tensor:(5, 4, N, N, 8), by reshaping will be Tensor:(5, 4*N*N, 8).

9. edge_input flat updates. Batch matrix-matrix multiplication is perform on (1) endge_input_flat Tensor:(5, 4*N*N, 8) and (2) we reshape a tensor by reshape(-1, 8, 8) on a embedding of edge_dist_encoder tensor of size (8192, 1). Resulting in a tensor of shape (1024, 8, 8). Where, by the [:max_dist, :, :], only the first 5 matrices will be kept. Then it multiply a Tensor:(5, N*N*4, 8) with Tensor:(5, 8, 8).

10. update edge_input. The edge_input_flat will be reshaped from a Tensor: (5, 4 * N * N, 8) to a Tensor:(5, 4, N, N, 8). Then, it will permute and convert to the following tensor with shape Tensor:(4, N, N, 5, 8). It will assign to the edge_input.

11. We add the edge_input Tensor:(4, 8, N, N) to all the Tensors except the first rows and columns of the N+1, N+1 matrices of graph_attn_bias Tensor:(4, 8, N+1, N+1).

12. Then the attn_bias as the Tensor:(4, N+1, N+1) will converted to Tensor:(4, 1, N+1, N+1). and it is added to the graph_attn_bias which previously updated by adding the edge_input to all the N*N elements form the N+1*N+1 elements (did not add to first row and first column). We return this value as the result of the GraphAttnBias.

Next, x (GraphNodeFeature) will pass through emb_layer_norm as LayerNorm((80), eps=1e-05, elementwise_affine=True) and x is updated. (Tensor: (4, N+1, 80))

x = x.transpose(0, 1) thus x is a Tensor:(N+1, 4, 80)

inner_states = [x]

Then, we go through 12 same layers. The layer function gets the x (normalized encoded nodes) which is Tensor:(N+1, 4, 80), in the following layers x is the output of the previous layer, also, it gets self_attn_padding_mask=padding_mask which is a Tensor:(4, N+1) and contains False for the nodes that exist (True for the nodes that do not exist), self_attn_mask=attn_mask which is the None, self_attn_bias=attn_bias which is a GraphAttnBias object (containing the encoded edges) and has the form of Tensor:(4, 8, N+1, N+1).

x is { 15}

1. It sets the residual as a the x which can be seen as the encoded nodes.
2. Calculate x {K} from self_attention, where input containing the query, key, value which are the same as the x (copies of x) Tensor:(N+1, 4, 80), and attn_bias which is a Tensor:(4, 8, N+1, N+1), key_padding_mask which is a Tensor:(4, N+1), need_weight which is False, attn_mask which is None.
   (a) q = q_proj(query), k = k_proj(query), v = v_proj(query), where these proj are Linear(in_features=80, out_features=80, bias=True).
   (b) the q is scaled by a constant calculated as $self.head\_dim ** -0.5 = 0.31622776601683794$
   (c) the q, k, and v converted to Tensor:(4*8, N+1, 10)
   (d) attn_weight initialized by the batch matrix multiplication of q, and k.transpose(1, 2).
   (e) The attn_bias is a Tensor:(4, 8, N+1, N+1), the view function convert it to Tensor:(4*8, N+1, N+1). Then it will be added to attn_weights.
   (f) attn_weights is Tensor:(4*8, N+1, N+1). It will be converted to Tensor:(4, 8, N+1, N+1). key_padding_mask is a Tensor:(4, N+1), then, key_padding_mask is converted to Tensor:(4, 1, 1, N+1). So on all heads and all nodes it will change the float("-inf") values to True. At last, the attn_weights will converted to Tensor:(4*8, N+1, N+1).
   (g) the input tensor attn_weights has dimensions (4 * 8, N+1, N+1), where the softmax operation is applied along the last dimension (dim=-1), which means that the softmax is applied independently to each spatial location of each head for each element in the batch.
   (h) attn_probs is assigned as the attn_weights will go through a dropout layer with probability 0.1.
   (i) we calculate the attn as batch matrix-matrix multiplication of attn_probs and v. Where attn_prob is a Tensor:(32, N+1, N+1) and v is Tensor:(32, N+1, 10). The result is assigned to attn variable as a Tensor:(32, N+1, 10).
   (j) the attn will be converted to a Tensor:(N+1, 4, 80).
   (k) attn is updated by self.out_proj which is a Linear(in_features=80, out_features=80, bias=True) and is applied on attn.
3. result of k goes through a dropout layer which do nothing.
4. x updated by addition of residual from (1) to it.
5. x Tensor:(N+1, 4, 80) goes through a layer normalization which is LayerNorm((80,), eps=1e-05, elementwise_affine=True)

6. set the residual as the x
7. x Tensor:(N+1, 4, 80) goes through a layer normalization which is LayerNorm((80,), eps=1e-05, elementwise_affine=True)
8. a Linear(in_features=80, out_features=80, bias=True) function apply on the x Tensor:(N+1, 4, 80).
9. The result x Tensor:(N+1, 4, 80) will go through an activation function gelu.
10. x will go through a dropout layer with p=0.1
11. x goes through fc2(), which is a Linear(in_features=80, out_features=80, bias=True).
12. x goes through a dropout layer with p=0
13. x updated by adding the residual in (6) to it
14. A LayerNorm((80,), eps=1e-05, elementwise_affine=True) applies on x
15. return x

The above procedure repeat for 12 times and x as Tensor:(N+1, 4, 80) is added to inner_states. The first index Tensor:(0, 4, 80) represent the representation of the virtual node (graph representation)

As the result graph_rep goes through several steps to predict the class of the graph

The probability of belonging to a class for each 4 graphs in the batch is as {8}.

1. graph_rep = x[0, :, :]
2. x = x.transpose(0, 1) as a result x is Tensor:(4, N+1, 80)
3. A Linear(in_features=80, out_features=80, bias=True) apply on x
4. A Gelu activation function will apply on x
5. A LayerNorm((80,), eps=1e-05, elementwise_affine=True) apply on x.
6. A Linear(in_features=80, out_features=1, bias=False) apply on x, as a result the x from a Tensor:(4, N+1, 80) converts to Tensor:(4, N+1, 1)
7. The learnable self.lm_output_learned_bias parameter {Parameter: (1,)} is added to x.
8. return x

Based on the true class of the graphs in the batch, the L1Loss with ”sum” reduction is calculated. To do so, we have:

1. Since the first element in N+1 dimension represent the virtual node (a representative node of all nodes in graph), the logits take just that value as a Tensor:(4, 1, 1) from previous x
2. The l1Loss of logits and true classes are calculated as the sum of absolute errors.
3. return the loss
4. backpropagate the loss over the network to train the parameters

Do it for all batches for the number of epochs. In case we set the early stop as true, do for all batches for till the validation loss of the first batch dropped.

7 mathematical notations

unsqueeze(k) Let x be a tensor of size $(d_1, d_2, \ldots, d_n)$, where d_i represents the size of the i-th dimension. Then, $unsqueeze(k)$ on x can be represented as:

$$y_{i_1,i_2,\ldots,i_{k-1},k,i_k,i_{k+1},\ldots,i_n} = x_{i_1,i_2,\ldots,i_{k-1},i_k,i_{k+1},\ldots,i_n} \tag{2}$$

for all $i_1, i_2, \ldots, i_{k-1}, i_k, i_{k+1}, \ldots, i_n$ and y is the resulting tensor of size $(d_1, d_2, \ldots, d_{k-1}, 1, d_k, d_{k+1}, \ldots, d_n)$.

Embedding

In mathematical notation, let x be a discrete categorical variable that takes on values from a finite set $V = v_1, v_2, \ldots, v_{|V|}$. An embedding function $\text{Emb} : V \to R^d$ maps each discrete variable v_i to a d-dimensional vector in the continuous vector space, such that:

$$\text{Emb}(v_i) = e_i \in R^d \tag{3}$$

where e_i is the embedding vector for variable v_i, and d is the size of the embedding dimension.

The embedding function can be represented by a matrix $E \in R^{|V|\times d}$, where the i-th row of E corresponds to the embedding vector for variable v_i. In other words, the embedding vector for v_i can be obtained by indexing into the i-th row of the embedding matrix:

$$\text{Emb}(v_i) = E_{i,:} \tag{4}$$

The embedding matrix E is typically learned during training using backpropagation, such that the embedding vectors for similar variables are close to each other in the continuous vector space.

Linear Transformation

To represent a linear transformation of a tensor with bias mathematically, you can use the following equation:

$$\mathbf{Y} = \mathbf{W} \cdot \mathbf{X} + \mathbf{b} \tag{5}$$

where:

$\mathbf{Y}$ is the output tensor resulting from the linear transformation. $\mathbf{X}$ is the input tensor. $\mathbf{W}$ is the weight tensor representing the linear transformation. $\mathbf{b}$ is the bias tensor. The operation $\mathbf{W} \cdot \mathbf{X}$ represents the matrix multiplication between the weight tensor $\mathbf{W}$ and the input tensor $\mathbf{X}$. The result of this multiplication is a new tensor.

The term $\mathbf{b}$ represents the bias tensor, which is added element-wise to the matrix multiplication result.

Note that the dimensions of the tensors should match appropriately for the matrix multiplication and element-wise addition to be valid.

the equation for $Linear(input = x, W, b)$ can be represented as:

$$\mathbf{Y} = \mathbf{W} \cdot \mathbf{X} + \mathbf{b} \tag{6}$$

sum(-2)

$$Y_{i,j,k} = \sum_{l=1}^{80} X_{i,j,1,l} \tag{7}$$

8 Calculate the GraphNodeFeature

8.1 Requirement

Weights in $E_{atom\_encoder}$ is a matrix $X_{atom\_encoder} \in R^{4609 \times 80}$. Weights in $E_{in\_degree\_encoder}$ is a matrix $X_{in\_degree\_encoder} \in R^{512 \times 80}$. Weights in $E_{out\_degree\_encoder}$ is a matrix $X_{out\_degree\_encoder} \in R^{512 \times 80}$. Weights in E_{graph_token} is a matrix $X_{graph\_token} \in R^{1 \times 80}$.

Input: x as Tensor:(4, N, 1), in_degree and out_degree as Tensor:(4, N)

8.2 Mathematical Flow

1. $x = E_{atom\_encoder}(x)$
2. $x = x_{i,j,k} = \sum_{l=1}^{80} x_{i,j,1,l}$
3. $in\_degree\_encoder = E_{in\_degree\_encoder}(in\_degree)$
4. $out\_degree\_encoder = E_{out\_degree\_encoder}(out\_degree)$
5. $y = x + in\_degree\_encoder + out\_degree\_encoder$
6. $$graph\_token\_feature_{k,i,j} = \begin{cases} X_{graph\_token_{i,j}} & \text{if } k = 0 \\ 0 & \text{otherwise} \end{cases}$$
7. $$graph\_token\_feature = Y_{i,j,k} = graph\_token\_feature_{1,k}$$

   for $1 \leq i \leq 4$, $1 \leq j \leq 1$, and $1 \leq k \leq 80$

8.
$$x = Z_{i,j,k} = \begin{cases} \text{graph\_token\_feature}_{i,1,k} & \text{if } j = 1 \\ y_{i,j-1,k} & \text{if } 1 < j \leq N \end{cases}$$

for $1 \leq i \leq 4$, $1 \leq j \leq N+1$, and $1 \leq k \leq 80$, where Z is the resulting tensor.

9 Calculate AttnBias

9.1 Requirement

Weights in $E_{edge\_encoder}$ is a matrix $X_{edge\_encoder} \in R^{1537 \times 8}$. Weights in $E_{edge\_dis\_encoder}$ is a matrix $X_{edge\_dis\_encoder} \in R^{8192 \times 1}$. Weights in $E_{spatial\_pos\_encoder}$ is a matrix $X_{spatial\_pos\_encoder} \in R^{512 \times 8}$. Weights in $E_{graph\_token\_virtual\_distance}$ is a matrix $X_{graph\_token\_virtual\_distance} \in R^{1 \times 8}$.

$$GELU(x) = 0.5 * x * (1 + Tanh(\sqrt{(2/\pi)} * (x + 0.044715 * x^3)))$$

Input: attn_bias as a Tensor:(4, N+1, N+1). spatial_pos (shortest paths) is Tensor:(4, N, N). edge_input is a Tensor:(4, N, N, 5, 1).

9.2 Mathemtical Flow

1. $graph\_pos\_bias = E_{spatial\_pos\_encoder}(spatial\_pos)$

2.
$$graph\_pos\_bias_{i,j,k,l} = graph\_pos\_bias_{i,k,l,j}$$

where $1 \leq i \leq 4$, $1 \leq j \leq 8$, $1 \leq k \leq N$, and $1 \leq l \leq N$.

3. $graph\_attn\_bias = attn\_bias$

4.
$$graph\_attn\_bias_{i,j,k,l} = graph\_attn\_bias_{i,k,l}$$

where $1 \leq i \leq 4$, $1 \leq j \leq 1$, $1 \leq k \leq N+1$, and $1 \leq l \leq N+1$.

5.
$$graph\_attn\_bias_{i,j,k,l} = graph\_attn\_bias_{i,1,k,l}$$

where $1 \leq i \leq 4$, $1 \leq j \leq 8$, $1 \leq k \leq N+1$, and $1 \leq l \leq N+1$.

6.
$$graph\_attn\_bias_{i,j,k,l} = graph\_attn\_bias_{i,j,k,l} + spatial\_pos\_bias_{i,j,k-1,l-1}$$

for all $1 \leq i \leq 4$, $1 \leq j \leq 8$, $2 \leq k \leq N+1$, and $2 \leq l \leq N+1$.

7. $t = X_{graph\_token\_virtual\_distance}$

8.
$$t_{i,j,k} = t_j, \text{ for all } 1 \le i \le 1, 1 \le j \le 8, 1 \le k \le 1$$

9.
$$\begin{gathered} graph\_attn\_bias_{i,j,k,l} = graph\_attn\_bias_{i,j,k,l} + t_{1,j,1}, \\ \text{for all } 1 \le i \le 4, 1 \le j \le 8, 1 \le k \le N, 1 \le l \le 1 \\ graph\_attn\_bias_{i,j,k,l} = graph\_attn\_bias_{i,j,k,l} + t_{1,j,1}, \\ \text{for all } 1 \le i \le 4, 1 \le j \le 8, 1 \le k \le 1, 1 \le l \le N+1 \end{gathered}$$

10. $edge\_input = E_{edge\_encoder}(edge\_input)$

11.
$$\text{mean}(\textbf{edge\_input}, -2)ijklm = \frac{1}{1}\sum_{n=1}^{1} \textbf{edge\_input}_{ijklnm}$$

12.
$$\begin{aligned} &edge\_input\_flat = \\ &\text{reshape}(\textbf{edge\_input}_{\mathbf{l,i,j,k,m}}, (5, -1, 8))ijk = \\ &\textbf{edge\_input}_{\mathbf{l,i,j,k,m}}\left\lfloor \frac{jk}{N^2} \right\rfloor, \frac{jk \bmod N^2}{8}, jk \bmod 8 \end{aligned}$$

This results in a Tensor:(5, 4*N*N, 8)

13.
$$\textbf{edge\_dis\_encoder\_reshaped}_{ijk} = \mathbf{X}_{\textbf{edge\_dis\_encoder}(64i+8j+k,0)}$$

for $0 \le i < \frac{8192}{64}$, $0 \le j < 8$, and $0 \le k < 8$.

14.
$$\textbf{edge\_dis\_encoder\_reshaped}_{ijk} = \textbf{edge\_dis\_encoder\_reshaped}_{ijk} \quad \text{for } 0 \le i < 5,\ 0 \le j < 8,\ 0 \le k < 8$$

15.
$$\textbf{edge\_input\_flat}_{ijk} = \sum_{l=0}^{7} \textbf{edge\_input\_flat}_{ijl}\textbf{edge\_dis\_encoder\_reshaped}_{ilk} \quad \text{for } 0 \le i < 5,\ 0 \le j < 4N^2,\ 0 \le k < 8$$

16.
$$\begin{gathered} \textbf{edge\_input}_{jklim} = \textbf{edge\_input\_flat}_{ij(kN+l),m} \\ \text{for } 0 \le i < 5,\ 0 \le j < 4,\ 0 \le k < N,\ 0 \le l < N,\ 0 \le m < 8 \end{gathered}$$

17.

$$\textbf{nominator}_{ijklm} = \sum_{p=0}^{4} \textbf{edge\_input}_{ijkp,lm}$$
$$\text{for } 0 \le i < 4,\ 0 \le j < N,\ 0 \le k < N,\ 0 \le l < 8,\ 0 \le m < 5$$

18.

$$\textbf{spatial\_pos}_{-ijk} = \begin{cases} \textbf{spatial\_pos}_{ijk} - 1 & \text{if } \textbf{spatial\_pos}_{ijk} > 1 \\ \textbf{spatial\_pos}_{ijk} + 1 & \text{if } \textbf{spatial\_pos}_{ijk} = 0 \\ \textbf{spatial\_pos}_{ijk} & \text{otherwise} \end{cases} \tag{8}$$

for $0 \le i < 4, 0 \le j < N,\ 0 \le k < N$

19.

$$\textbf{denominator}_{ijk\ell} = \textbf{spatial\_pos}_{-ijk} \qquad \text{for } 0 \le i < 4,\ 0 \le j < N,\ 0 \le k < N,\ 0 \le \ell < 1 \tag{9}$$

20. $edge\_input_{i,l,j,k} = \frac{nominator}{denominator}_{i,j,k,l}$ The result is Tensor:(4, 8, N, N)

21.

$$\textbf{graph\_attn\_bias}_{ij(k+1)(\ell+1)} = \textbf{graph\_attn\_bias}_{ij(k+1)(\ell+1)} + \textbf{edge\_input}_{ijk\ell}$$
$$\text{for } 0 \le i < 4,\ 0 \le j < 8,\ 1 \le k \le N,\ 1 \le \ell \le N$$

22.

$$\textbf{attn\_bias\_squeezed}_{ij(k+1)(\ell+1)} = \textbf{attn\_bias}i(k+1)(\ell+1)$$
$$\text{for } 0 \le i < 4,\ 0 \le j < 1,\ 0 \le k \le N,\ 0 \le \ell \le N$$

23. $graph\_attn\_bias = graph\_attn\_bias + attn\_bias\_squeezed$

24. return graph_attn_bias

10 Calculate GraphEncoder Cont.

10.1 requirement

There is a $l_{(emb\_layer\_norm)}$ with two parameters where Weights:{Parameter: (80,)} and Bias:{Parameter: (80,)}. When the function is applied to the input tensor, it first computes the mean and variance of the tensor along the normalization dimension, which is the last dimension (the dimension with the length of 80) in this case. It then applies the following formula to compute the normalized output:

$$output = \frac{(input - mean)}{\sqrt{(variance + eps)}}$$

Then,

$$output = weight * output + bias$$

The function returns the normalized and scaled tensor with the same shape as the input tensor. the $eps = 1e - 05$.

10.2 Mathematical Flow

1. $x = l_{(emb\_layer\_norm)}(x)$ x is output of GraphNodeFeature
2. $x = dropout(x, p = 0.0)$
3. $x = x.transpose(0, 1)$ in other words,

$$(\mathbf{T})ijk \rightarrow (\mathbf{T}^T)jik$$
$$x = (\mathbf{T}^T)jik$$

4. save x as the first index in a list inner_states

11 Calculate ModuleList (layers(0-11)) as Graphormer-GraphEncoderLayer

11.1 Requirement

Then, we go through 12 same layers. The layer function gets the x (normalized encoded nodes) which is Tensor:(N+1, 4, 80), in the following layers x is the output of the previous layer, also, it gets self_attn_padding_mask=padding_mask which is a Tensor:(4, N+1) and contains False for the nodes that exist (True for the nodes that do not exist), self_attn_mask=attn_mask which is the None, self_attn_bias=attn_bias which is a GraphAttnBias object (containing the encoded edges) and has the form of Tensor:(4, 8, N+1, N+1).

We have, $Linear_{k\_proj}(input, W : (80, 80), b : (80,))$, $Linear_{v\_proj}(input, W : (80, 80), b : (80,))$, $Linear_{q\_proj}(input, W : (80, 80), b : (80,))$, and $Linear_{out\_proj}(input, W : (80, 80), b : (80,))$. $Linear_{fc1}(input, W : (80, 80), b : (80,))$

$l_{(self\_attn\_layer\_norm)}(input)$ with two parameters where Weights:{Parameter: (80,)} and Bias:{Parameter: (80,)} and $l_{final\_layer\_norm}(input)$ with two parameters where Weights:{Parameter: (80,)} and Bias:{Parameter: (80,)}.

11.2 Mathematical Flow

1. residual = x
2. Calculate { $x = (s)$} from self_attention, where input containing the query, key, value which are the same as the x (copies of x) Tensor:(N+1, 4, 80), and attn_bias which is a Tensor:(4, 8, N+1, N+1), key_padding_mask which is a Tensor:(4, N+1), need_weight which is False, attn_mask which is None.

   (a) $q = Linear_{k\_proj}(query, W : (80, 80), b : (80,))$

   (b) $k = Linear_{k\_proj}(query, W : (80, 80), b : (80,))$

   (c) $v = Linear_{k\_proj}(query, W : (80, 80), b : (80,))$

   (d) $scaling = head\_dim ** - 0.5 = 0.31622776601683794$

   (e) $q = q * scaling$

   (f)
   $$\mathbf{q}jik = \mathbf{q}ij(k+1) \qquad \text{for } 0 \le i \le N,\ 0 \le j < 4 \times 8,\ 0 \le k < 10$$

   (g)
   $$\mathbf{v}jik = \mathbf{v}ij(k+1) \qquad \text{for } 0 \le i \le N,\ 0 \le j < 4 \times 8,\ 0 \le k < 10$$

   (h)
   $$\mathbf{k}jik = \mathbf{k}ij(k+1) \qquad \text{for } 0 \le i \le N,\ 0 \le j < 4 \times 8,\ 0 \le k < 10$$

   (i)
   $$\mathbf{attn\_weights}_{ijk\ell} = \sum_{m=0}^{n} (\mathbf{q}_{ijm} \cdot \mathbf{k}_{imk})$$
   $$\text{for } 0 \le i < 4 \times 8,\ 0 \le j \le n,\ 0 \le k \le n,\ 0 \le \ell \le n$$

   (j)
   $$\mathbf{attn\_bias}'ijk\ell = \mathbf{attn\_bias}i(j/8)(k+1)(\ell+1) \qquad \text{for } 0 \le i < 4{\times}8,\ 0 \le j \le N,\ 0 \le k, \ell \le N$$

   (k) $attn\_weights = attn\_weights + attn\_bias'$

   (l)
   $$\mathbf{attn\_weights}ijkl = \mathbf{attn\_weights}(i \times 8 + j)k\ell \qquad \text{for } 0 \le i < 4,\ 0 \le j < 8,\ 0 \le k, \ell \le N$$

   (m) $\text{attn\_weights} = \begin{cases} \text{float}(\text{"} - inf\text{"}) & \text{if } \mathbf{key\_padding\_mask}_{ij} = \text{True} \\ \mathbf{attn\_weights}_{ijkl} & \text{if } \mathbf{key\_padding\_mask}_{ij} = \text{False} \end{cases}$

   for $0 \le i < 4,\ 0 \le j < 8,\ 0 \le k, \ell \le N.$

(n)

$$\mathbf{attn\_weights}_{ijkl} = \frac{exp(attn\_weights_{ijkl})}{\sum_{m=0}^{N} exp(attn\_weights_{ijkm})}$$
$$\text{for } 0 \leq i \leq 4 \times 8, 0 \leq j, k \leq N, \text{and } 0 \leq \ell \leq N$$

(o) $attn\_prob = dropout(attn\_weights, p = 0.1)$

(p)

$$\mathbf{attn}_{ijkl} = \sum_{m=0}^{N} \left(\mathbf{attn\_prob}_{ijkm} \cdot \mathbf{v}_{klm}\right)$$

for $0 \leq i < 4 \times 8$, $0 \leq j \leq N$, and $0 \leq \ell \leq 10$.

(q)

$$\mathbf{attn}_{jik\ell} = \mathbf{attn}_{\left(\left\lfloor \frac{i\cdot 4+j}{8} \right\rfloor \cdot 8 + \left(\frac{i\cdot 4+j}{8} \mod 8\right), i \mod (N+1), \ell\right)} \tag{10}$$

(r) $attn = Linear_{out\_proj}(attn)$

(s) return $attn$

3. $x = dropout(x, 0.0)$
4. $x = x + residual$
5. $x = l_{(self\_attn\_layer\_norm)}(x)$
6. $residual = x$
7. $x = l_{final\_layer\_norm}(x)$
8. $x' = Linear_{fc1}(x, W, b)$
9. $x = gelu(x)$
10. $x = dropout(x, p = 0.1)$
11. $x = residual + x$
12. $x = l_{(self\_attn\_layer\_norm)}(x)$
13. return x

This process in 11.2 repeats for 12 times. The returned x of each step is append to *inner_states* and is the input for the new 11.2 call.

12 Calculate GraphEncoder Cont.

12.1 Requirement

$Linear_{lm\_head\_transform\_weight}(x, W : (80, 80), b : (80,))$. $l_{(layer\_norm)}$ with two parameters where Weights:{Parameter: (80, 80)} and Bias:{Parameter: (80,)}. $Linear_{embed\_out}(x, W : (80,)bias = False)$ with one parameters where Weights:{Parameter: (80)}. $b_{lm\_output\_learned\_bias}$ as a Parameter:(1,)

12.2 Mathematical Flow

1. $graph\_rep_{i,j} = x_{0,i,j}$
2. $x_{j,i,k} = x_{i,j,k}$
3. $x = l_{layer\_norm}(gelu(Linear_{lm\_head\_transform\_weight}(x)))$
4. $x = Linear_{embed\_out}(x)$
5. $x = x + b_{lm\_output\_learned\_bias}$
6. return x

13 Calculating the loss

13.1 Requirement

targets is a Tensor:(4,) containing the True class of the graphs in $bsz = 4$

1. $logits_{i,j} = x_{i,0,j}$
2. $$\text{loss} = \sum_{i=0}^{3} |\mathbf{logits}_i - \mathbf{targets}_i|$$
3. backpropagate loss through the network