ChatGPT

# Handling Single vs. Multi-Ligand Cases in the Metal-pKa GNN Model

## Code Functionality for a Single Ligand

For a single metal–ligand complex, the provided model code constructs a graph representation with one metal node and one ligand. It uses the following steps (summarizing the initial code snippet):

- **Metal embedding and projection:** The metal ion (given by `batch.metal_id`) is embedded and projected to the hidden dimension (`metal_proj_feature`) [1]. This yields a 1×384 feature for the metal (for batch size 1).
- **Ligand node projection:** All atoms of the ligand (`batch.x` features of shape [num_atoms, node_dim]) are projected to the hidden dimension via `ligand_proj` to get `h1_L` (shape [num_atoms, hidden_dim]) [2].
- **Binding vs. backbone split:** The code identifies **binding atom indices** (`binding_atom_indices`) which correspond to the ligand atoms that coordinate the metal. In the simple case, `batch.pred_pos` provides this index (e.g., one index if a single ligand) [1]. A boolean mask marks binding atoms vs. the rest. Using this mask, it splits ligand features into binding atom features `h_B` (e.g., shape [1, hidden_dim] if one binding atom) and backbone features `h_BB` (all other ligand atoms, shape [N-1, hidden_dim]) [2]. The average backbone feature $\bar{h}_{BB}$ is computed as the mean of `h_BB` [2]. This $\bar{h}_{BB}$ represents the overall context of the ligand's non-binding atoms (environment).
- **Gating mechanism:** A small neural network (`gate_network`) produces a gate weight between 0 and 1 by looking at the backbone context $\bar{h}_{BB}$. The binding atom feature is then updated as `h2_L = h_B + gate * h̄_BB` [2]. Intuitively, this gate allows the model to incorporate the influence of the ligand's overall environment on the binding site representation. If the backbone context is important, the gate will be high and add more of $\bar{h}_{BB}$ to the binding site feature. If not, the gate stays low, leaving `h_B` mostly unchanged.
- **Metal–ligand graph construction:** The model then concatenates the metal's feature and the ligand's binding-site representation: `h2_mL = [metal_proj_feature; h2_L_repr]`, where `h2_L_repr` is the mean of `h2_L` (in a single binding-atom case, it's just that atom's feature in a 1×384 vector) [2]. This forms a tiny graph of 2 nodes: index 0 = metal, index 1 = ligand representation. An edge index tensor `ei_mL = [[0,1],[1,0]]` connects the two nodes in both directions, and a dummy edge feature matrix (`eq_mL` of zeros) is created since the TransformerConv expects an edge attribute (though here no meaningful edge features are used).
- **Graph transformer convolution:** A `TransformerConv` layer is applied on this 2-node graph (`h3_mL = transformer_conv(h2_mL, ei_mL, edge_attr=eq_mL)`). This allows the metal and ligand node to exchange information via an attention mechanism. After this convolution, each node's feature is updated based on its neighbor's information [3]. In particular, the metal node's new feature will attend to the ligand node's feature, and vice versa.
- **Graph pooling and output:** The updated node features `h3_mL` (shape [2, hidden_dim]) are averaged (`V_mL = mean(h3_mL)`) to produce a single graph-level representation [2]. Finally, a predictor MLP generates the pKa prediction (`pka_pred`) from this representation. If a true pKa value is available (`batch.true_pka` or in `batch.pka_labels` at the binding atom index), a

loss (Smooth L1 / Huber) is computed between the predicted and true value (after normalization) for training.

For a **single-ligand scenario**, this pipeline makes sense – it identifies the one binding site and produces one pKa prediction. The code as given can handle a single metal with a single ligand attached, yielding one output value.

## Challenge of Multiple Ligands (1–3 Identical Ligands)

In many metal complexes, a metal ion can coordinate multiple identical ligand molecules (up to 3 in our case). The dataset example given (e.g., `Ag+, CCC(N)CC, 2, [3.53, 7.73]`) suggests: one $Ag^+$ ion, the ligand with SMILES `CCC(N)CC`, and "2" indicating two ligand molecules attached, with two pKa values `[3.53, 7.73]` corresponding to stepwise dissociation. In such a scenario:

- **Data interpretation:** "ligand_num = 2" means there are two identical ligand molecules coordinating the metal. The list of pKa values likely represents the stepwise pKa for each ligand dissociating. For example, $pKa_1$ might be the dissociation constant for removing the first ligand from the di-ligand complex (leaving a mono-ligand complex), and $pKa_2$ for removing the second ligand (from mono-ligand to free metal). These values are different due to cooperative effects – the environment changes after one ligand is removed.
- **Graph representation needed:** To model a state with multiple ligand molecules, the representation should somehow include **multiple binding sites**. Each ligand will contribute a binding atom (or atoms) and its own chemical environment. The model must capture that a metal with two ligands attached is not the same as a metal with one ligand attached. In other words, the input to the model for an "Ag(L)_2" complex should reflect two coordinating groups influencing the metal, whereas "Ag(L)_1" has one.

The challenge is that the current implementation only builds the graph with one ligand's atoms. The code does not inherently account for multiple copies of the ligand. Simply looping over the number of ligands and repeating the same single-ligand features will not correctly simulate the multi-ligand environment – we need a way to represent all ligands simultaneously or sequentially update the representation to reflect multiple attachments.

## Issues Identified in the Current Code for Multi-Ligand Cases

The provided code attempts to handle multiple ligands by looping over `range(max_ligands)` (where `max_ligands` is the number of ligand molecules, up to 3). However, several issues prevent it from correctly modeling and predicting multi-ligand complexes:

- **No true multi-ligand representation:** In the loop, the code **does not actually build a combined graph with multiple ligand nodes**. It recomputes `h1_L` for the same single ligand (`batch.x`) each iteration, and tries to adjust `binding_atom_indices` based on the iteration index. However, it never creates additional ligand nodes in the graph. As a result, each iteration is still effectively using the same single-ligand graph. This means the model isn't aware that a second or third ligand is present; there is no representation of a second ligand's atoms or features being added to the metal's neighborhood. The loop just produces multiple outputs from the same input, which is conceptually wrong.

- `binding_atom_indices` **logic flaw:** The code tries to select a number of binding atoms equal to the current ligand count (`max_binding_atoms = eq_type_idx + 1`). This makes sense if,

for example, the ligand had multiple distinct donor atoms that could coordinate simultaneously. The logic attempts to:

- Identify all possible binding atoms in the ligand (`binding_atom_indices = binding_atom(mol)`, presumably using RDKit to find donor atoms).
- If there are more binding atoms than needed, it picks a subset equal to the required count (preferring those that have labels in `batch.pka_labels`).

- **However, if there are fewer binding atoms than the required count (e.g., a monodentate ligand with only 1 donor, but `max_binding_atoms` = 2 or 3), the code does nothing to augment or replicate them.** It does not add any new indices or duplicates. It simply proceeds with whatever is available. For a monodentate ligand, `binding_atom_indices` might be a single index (say `[i]`). For two ligands needed, ideally we'd want two binding sites – but the code will still have just `[i]` (length 1) because it doesn't handle the "< needed" case. There's no second index to represent the second ligand's binding site. This means in iteration for 2 ligands, `h_B` will still contain only one atom's feature. No additional binding atom is added, leading to the same `h_B` and `h_BB` as in the one-ligand case. The output for the second iteration will end up identical to the first (aside from any randomness or minor differences), since the model input hasn't actually changed.

- **Lack of state update or sequential dependency:** The code loops through ligand counts but does not carry over any state from one iteration to the next. In a true sequential removal scenario, after predicting the first pKa (for removing one ligand), the second prediction should consider that one ligand has been removed (i.e., the environment is now a mono-ligand complex). The model could achieve this by updating or reusing the result of the first step as input to the second. However, in the current implementation each iteration starts from scratch with the original ligand graph and metal feature, unaware of any previous ligand having been removed. This again means all iterations are effectively identical for monodentate ligands (same input state each time). The code does not decrement `ligand_num` or modify the input features based on previous outputs.

- **Attention aggregator and identical features:** Even if we attempted to represent multiple ligands by duplicating features, the use of a TransformerConv attention layer may dilute the impact of multiple identical neighbors. The attention mechanism in `TransformerConv` normalizes contributions from neighbors via a softmax [4]. For example, if the metal node has two identical ligand nodes as neighbors with the same feature vector, the attention would likely assign each neighbor ~50% weight. The sum of contributions might end up similar to having one neighbor (each gets half the weight) [4]. In other words, simply duplicating a node with identical features doesn't necessarily double the influence on the metal node under an attention aggregator. The model might treat two identical ligands almost like one, unless there's some differentiating feature. This is a limitation of using attention/mean aggregation – it handles variable neighbor counts gracefully, but **does not inherently encode "two of the same thing" as a stronger effect than one** [4]. (By contrast, a sum-based aggregation would yield a larger summed message with two neighbors versus one.) This means that if we don't adjust the model, a multi-ligand complex could be under-represented in the metal's features.

- **No feature indicating ligand count:** The model never explicitly uses `batch.ligand_num` (except to control loop range). There is no input feature that tells the neural network how many ligand molecules are present. Ideally, the model should either construct the graph in such a way that the number of ligands is reflected (e.g. multiple ligand nodes as neighbors), or include a feature/embedding for the coordination number. In the current code, if the graph input remains

the same single-ligand structure, the network has no way of knowing whether it's supposed to be predicting for an $ML_1$ vs $ML_2$ vs $ML_3$ complex. This will lead to confusion or identical predictions for each case.

- **Potential ordering mismatch:** The code as written collects predictions in ascending order of `eq_type_idx` (starting from 1 ligand up to `max_ligands`). However, if the expected output list `[3.53, 7.73]` corresponds to the stepwise pKa values for removing ligands from a di-ligand complex, typically the first value (3.53) would be the pKa for the first ligand dissociating from the full (two-ligand) complex, and the second value (7.73) for the second ligand dissociating from the mono-ligand complex. In other words, the full complex ($AgL_2$) has one pKa, and the AgL (mono) state has another. If so, the model should output pKa for the 2-ligand state first, then the 1-ligand state. But the loop currently does it in reverse (first $ML_1$, then $ML_2$). This might produce an output list in the wrong order relative to the data. It's important to verify how the dataset is ordered, but it's likely the code needs to output in the proper sequence of dissociation.

In summary, **the current prototype does not yet correctly incorporate multiple ligands in the graph representation or the model logic**, which would result in incorrect or identical predictions for multi-ligand cases. We need to fix the representation of multiple ligands and ensure the model's output corresponds properly to each ligand's dissociation step.

## Correct Approach to Model Multiple Ligand Attachments

To properly handle a single metal with 1–3 ligand molecules, the model should represent the metal–ligand complex in a way that reflects all ligands present. There are two conceptual strategies to achieve this:

**1. Explicit Multi-Ligand Graph Representation:**
The most direct approach is to construct a graph that includes the metal and multiple ligand nodes. Rather than only one ligand node connected to the metal, we would have one node per ligand (each representing that ligand's binding site contribution). For example, for a metal with two identical ligands:

- We create **two ligand representation nodes** (in addition to the metal node). Each could use the same ligand's features as a base, since the ligands are identical. Practically, we could take the single-ligand's `h2_L_repr` and duplicate it for the second ligand. These two nodes (call them L1 and L2) would initially have the same feature vector (if the ligands are identical and symmetrically bound).
- Construct the edge index such that the metal node (M) is connected to L1 and L2, and vice versa. For instance, nodes: 0 = M, 1 = L1, 2 = L2. Edges would be (0–1, 1–0, 0–2, 2–0). This can be represented as `ei_mL = [[0,0,1,2],[1,2,0,0]]` (source->target pairs). No ligand–ligand edges are needed if we assume ligands don't directly influence each other except through the metal.
- Apply the TransformerConv on this graph. Now the metal node will attend to two neighbor nodes instead of one. The ligand nodes will also receive messages from the metal. This setup allows the model to consider contributions from multiple ligands simultaneously. Importantly, using a GNN inherently handles a variable number of neighbors – the formulation (sum/attention over neighbors) naturally extends to any number [5]. The metal's updated representation after convolution will integrate information from both ligand nodes. If there are differences (e.g., different ligand types or different donor atoms), the attention could weigh them differently. If they are identical, the metal node will still know it has two neighbors, although as noted, attention

alone might not fully emphasize the count. However, the presence of two nodes could still subtly influence the metal's feature (for example, the metal gets two messages instead of one, and there could be some learned bias or nonlinearity to distinguish that).

- Graph pooling: after convolution, we need a single representation to feed to the predictor for a pKa. In the single-ligand case, we averaged metal and ligand node features. For multiple nodes, a similar approach can be taken: average (or sum) over **all nodes (metal + all ligand nodes)** to get `V_mL`. This acts as a permutation-invariant pooling to summarize the whole complex. (Alternatively, one could choose just the metal node's output as the summary, since the metal now contains info from all ligands. But averaging over all nodes is a simple symmetric choice that was already used for one ligand.)
- The predictor then outputs a single pKa value for that multi-ligand state.

By doing this for each possible ligand count, we can generate predictions for each coordination number. For example: build a graph for the di-ligand complex (M + 2L nodes) -> predict $pKa_1$; build a graph for the mono-ligand complex (M + 1L node) -> predict $pKa_2$. This would correctly simulate the sequence: the first predicted pKa corresponds to the full complex losing one ligand, the second corresponds to the remaining complex losing the last ligand. If `max_ligands = 3`, we would do it for tri-, di-, and mono-ligand states similarly.

**2. Sequential Update (Iterative) Representation:**
An alternative is to simulate the effect of removing ligands one by one within a single forward pass, updating the representation as you go. The code attempted a form of this (looping and computing `h2_L` each time) but, as discussed, it didn't truly update the state. A better iterative approach could be:

- Start with the **full ligand set**: determine features for all binding sites (as if all ligands are present). This might again require identifying multiple binding atoms or duplicating the ligand's binding atom feature if the ligand is monodentate. For example, if the ligand has one donor atom and there are 3 ligands, we could take that donor's feature and treat it as 3 separate vectors representing 3 donors. One way is to tile or repeat the binding atom feature 3 times (creating a [3, hidden_dim] matrix for `h_B`). The backbone (`h_BB`) in this case might be ambiguous – if each ligand contributes a backbone, the combined "backbone" could be considered the union of all ligand atoms minus all binding atoms. If the ligands are identical and non-interacting, one might approximate that each ligand's backbone contributions are similar. The code's current gating design doesn't straightforwardly extend to multiple separate ligand backbones, so this approach is a bit hacky with the existing structure.
- Compute a combined representation for the full complex (similar to multi-node graph method internally), get a pKa prediction for the first dissociation.
- **Update state:** Remove one ligand's influence and then compute the next. For instance, after predicting the first pKa, you could "remove" one ligand node or exclude one binding site and its backbone from the next iteration. In practice, this might mean adjusting `binding_mask` to remove one of the binding atoms (or if duplicates were used, drop one), and perhaps recompute a new `h_bar_BB` for the remaining backbone. Essentially, you'd simulate the new state (metal with one less ligand) and then run the model again to get the next pKa. This requires the loop to carry information forward (the current code does not do this – it always starts from scratch).

While the sequential update approach can work, it's more complex to implement correctly. It's easier and more transparent to construct the graph for each coordination state explicitly (as in approach 1), especially since the maximum number of ligands is small (<=3). The overhead of constructing a tiny 2-4 node graph a few times is negligible.

**3. Incorporating Ligand Count as a Feature (Auxiliary approach):**
In addition to the above, one can give the model an explicit clue about how many ligands are present. This can be done by:
- Adding a feature to the metal node encoding the coordination number (e.g., a one-hot or scalar embedded input for "1 ligand vs 2 vs 3"). For instance, the `metal_emb` could be extended to include coordination environment: instead of just metal type embedding, concatenate or sum it with an embedding for ligand count. This way, the metal's initial feature differs for Ag with 2 ligands vs Ag with 1 ligand.
- Another way is to modify the predictor to take the ligand count into account (e.g., concatenate `V_mL` with a one-hot [1,0,0] for mono-, [0,1,0] for di-, etc., before the final linear layers). This would inform the network which state it's predicting, if the graph alone isn't making it clear.

However, if we properly construct the graph with multiple ligand nodes, the network can potentially infer the count from the graph structure itself (number of neighbor nodes). Still, due to the attention normalization issue discussed, providing an explicit count feature could help the model distinguish, ensuring that (for example) a metal with two identical ligand nodes isn't treated exactly the same as one ligand case.

# Proposed Code Adjustments for Correct Multi-Ligand Handling

To implement the above ideas, we can adjust the `forward` method as follows:

- **Determine ligand count:** Use `batch.ligand_num.item()` (or the length of `pka_values`) to know how many ligand molecules (N) are present for this complex. This is already done in the code (`max_ligands`). Ensure this is the actual number of ligands, not exceeding `self.max_ligands` (which is 3). Let's call this `L = max_ligands`.

- **Loop over coordination states in the correct order:** It's logical to predict starting from the full complex and going down to one ligand. So iterate `for n in range(L, 0, -1):` (from L down to 1). This `n` represents the current number of ligands in the complex state we're considering. (If the dataset expects the outputs in the order of $pKa_1$, $pKa_2$, ... corresponding to removing one ligand at a time, this descending order will align with that sequence).

- **Construct binding atom features for n ligands:** We need `h_B` that represents all n binding sites. Two scenarios:

- If the ligand has at least n distinct donor atoms (binding sites) identified by `binding_atom_indices`, we can take the first n of those indices. (The code already partially handles the case of more binding atoms than needed by selecting a subset).
- If the ligand has fewer than n donor sites (e.g., monodentate ligand with one donor), we will **duplicate** the available binding site feature to create n copies. For example, if `binding_atom_indices = [i]` but we need 2, we can use `[i, i]` (the same index twice). In practice, we can gather the feature `h1_L[i]` twice. This yields an `h_B` of shape [2, hidden_dim] where both rows are the same donor's features. (This is a proxy for two identical ligand molecules each contributing a similar donor atom. It's not perfect but gives the model two neighbor vectors to work with, which the TransformerConv can at least sum/attend over.)

- If there are multiple donor types and fewer donors than ligands, one could take all donors and then reuse some. But since most cases will be monodentate or having enough donors, a simple

approach is: if needed_count > len(binding_atom_indices), just pad by repeating the last (or primary) index until we have n indices.

- **Compute backbone context appropriately:** For multiple ligands, each ligand has its own set of non-binding atoms. If we only have one ligand's structure, we might approximate the combined backbone as just the same set of atoms repeated, but since we cannot explicitly repeat the graph, another approach is to use the single ligand's backbone as a proxy for each. In practice, when duplicating binding atoms, we **do not duplicate backbone atoms in** `h_BB` – we keep using the single ligand's backbone feature `h̄_BB`. This is a limitation (the model doesn't see that there would be twice as many total atoms), but it's a reasonable approximation that each ligand's environment looks similar. The gate network will produce a weight based on this single-ligand backbone. Ideally, if multiple ligands are present, perhaps the backbone influence per binding site could be a bit smaller (since each ligand's own backbone is just one of the contributions), but the model can potentially learn this via the coordination count feature if provided.

- **Apply gating:** Use the same formula `h2_L = h_B + gate_weight * h̄_BB.expand_as(h_B)`. If `h_B` has n rows (n binding atoms), we first compute `h̄_BB` as before (1×hidden_dim). The gate network outputs a scalar weight (1×1) for that backbone. We expand that to n×hidden_dim and multiply with each row of `h̄_BB`, then add to each binding atom row. This yields an `h2_L` with n rows, each row being the enriched binding atom feature for one ligand. (If `h_B` rows were identical due to duplication, they will remain identical after this addition, since the same `h̄_BB` is added to each. So all ligand nodes initially have the same feature – which is expected if the ligands are identical.)

- **Metal + multi-ligand node graph:** Now concatenate metal and these n ligand features: `h2_mL = torch.cat([metal_node_feature, h2_L_repr1, h2_L_repr2, ..., h2_L_reprN], dim=0)`. Here each `h2_L_reprX` would just be the row from `h2_L` (or if we want each ligand as a single node, we can use the row directly rather than averaging – since each row already corresponds to one ligand's binding site). In fact, since we expanded one binding atom per ligand, `h2_L` already has one row per ligand, so we can treat those as the ligand node features. No further averaging per ligand is needed (we would only average if a single ligand had multiple binding atoms itself, which could happen in polydentate case, but usually we'd pick just one per ligand in this context). So simply take `metal_proj_feature` (which is 1×hidden_dim for this complex) and stack it with `h2_L` (n×hidden_dim). This yields a (n+1)×hidden_dim matrix of node features.

- **Edge index for metal–ligand connections:** We need to dynamically create `ei_mL` for a star graph connecting the metal node (index 0) to each ligand node (indices 1..n). This can be done by: for ligand index `j` from 1 to n, add edges (0 -> j) and (j -> 0). The resulting edge_index will have 2n entries (two per ligand). For example, if n=3, `edge_index = [[0,0,0, 1,2,3],[1,2,3, 0,0,0]]` (metal to 1,2,3 and back). In code, this can be generated easily with PyTorch tensor operations. Edge attributes can remain zeros as before (we'll have 2n rows of edge features, each of dimension `bond_dim`).

- **TransformerConv and pooling:** Run the `transformer_conv` on this graph. The output `h3_mL` now has n+1 rows. We then compute `V_mL = mean(h3_mL, dim=0)` (averaging over the metal and all ligand node features). Alternatively, using `V_mL = h3_mL[0]` (metal node's feature) could be considered, but averaging all nodes treats the metal and ligand contributions

equally in forming the final representation, as was done in the single-ligand case. This pooled vector is [1, hidden_dim].

- **Predict and record output:** Pass `V_mL` to the predictor to get a scalar `pKa_pred`. Append it to a list of predictions. If the true pKa for this state is known, compute loss. Now proceed to the next iteration (next smaller n).

- **Loss targeting:** We need the correct true pKa for the state with n ligands. If `batch.pka_labels` was set up to mark binding atom indices with pKa values, it might not directly support multiple outputs. It might be easier if the data object provided a list of true pKa values (like `batch.true_pka` as an array of length L). If, for instance, `batch.true_pka = [pKa1, pKa2]`, we could index into it by something like `true_pka_value = batch.true_pka[L-n]` (if the first value corresponds to full complex). Ensuring the ordering is correct is important. The code currently tries to extract `true_pka` via `batch.pka_labels` by checking nonzero labels at binding atom positions, which is a bit cumbersome for multiple values. A cleaner way is to simply store the list of pKa values and use the loop index to pick the right one. Since the question provides `"[3.53, 7.73]"` as data, presumably that list is accessible. For example, `batch.true_pka` might be such a list or `batch.pka_values`. If not, one can populate `pka_labels` such that each binding atom in a multi-ligand scenario has one of the pKa values (but if the binding atoms are identical, they probably didn't do that). We may need to adjust how targets are retrieved.

- **Batch processing:** Note that the above assumes a single complex in the batch. If multiple complexes were batched, one would have to loop over each graph in the batch and do this process (since each graph could have different ligand counts). The current implementation doesn't fully support multi-graph batches either (it picks `batch.metal_id[0]` and such). Ensuring compatibility with batching would require indexing by graph, but we can simplify by handling one graph at a time for now, as the question does.

By implementing these changes, the model will explicitly account for multiple ligands. Essentially, we convert the one-ligand representation into a star graph of metal with N ligand nodes. This leverages the GNN's ability to handle a variable number of neighbors [5]. Each additional ligand node provides an extra neighbor message to the metal during message passing, informing the metal's representation. Even though the ligand nodes might start with identical features, the metal node's aggregation can sum or average them (the TransformerConv's attention will distribute focus across them; if identical, it might give equal attention, but the presence of two nodes still results in two terms in the sum before softmax normalization). If necessary, one could switch to a simpler GCN or GraphConv (which sums messages) to ensure multiple neighbors increase the signal, but this is a hyperparameter choice.

Finally, the predictor will output a list of pKa predictions, one per ligand dissociation step. We should output them in the correct order matching the dataset. If the dataset lists pKa for $ML_3$, $ML_2$, $ML_1$ in that order, our descending loop already matches it. If the dataset lists them as [pKa1, pKa2, ...] in stepwise removal order (which is likely full complex first), that also matches the descending loop (full complex = first output). It's important to double-check but this approach is flexible once we know the convention.

## Conclusion

The current code needs significant modifications to handle multi-ligand scenarios correctly. The primary correction is to **represent multiple ligand attachments either by constructing multiple ligand nodes**

**in the graph or by sequentially updating the state** to reflect ligand removal. By doing so, the model will be aware of the number of ligands in the complex and can predict distinct pKa values for each scenario. The explicit multi-node graph approach is recommended for clarity: it naturally extends the single-ligand model by leveraging GNN aggregation for a variable number of ligand neighbors [5]. Additionally, incorporating the ligand count as a feature can help the model learn the quantitative effect of multiple ligands. With these changes, the model should be able to execute correctly for a single metal ion with 1 to 3 ligands and output the corresponding pKa values for each coordination state.

---


[1] [2] (PDF) Accurate and rapid prediction of p K a of transition metal complexes: semiempirical quantum chemistry with a data-augmented approach
https://www.researchgate.net/publication/347911507_Accurate_and_rapid_prediction_of_p_K_a_of_transition_metal_complexes_semiempirical_quantum_chemistry_with_a_data-augmented_approach

[3] [4] torch_geometric.nn.conv.TransformerConv — pytorch_geometric documentation
https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.nn.conv.TransformerConv.html

[5] [D] Handle varying output dimension in Graph Neural Networks? : r/MachineLearning
https://www.reddit.com/r/MachineLearning/comments/1h4uyn9/d_handle_varying_output_dimension_in_graph_neural/