My House, My Rules: Learning Tidying Preferences with Graph Neural Networks

Supplementary Material

Ivan Kapelyukh and Edward Johns

1 Result Visualisations

In our paper, we detailed a series of experiments to test the capabilities of our `NeatNet` method. In this section, we visualise a selection of arrangements generated in these experiments to complement the quantitative results and analysis provided in the paper.

1.1 Can NeatNet Tidy a Known Scene?

This experiment tests the tidiness of `NeatNet`'s reconstructed arrangements for known scenes (Table 1 in the paper). The example arrangement provided by the user contains noise and imperfect alignment of objects, as shown in Figure 1. `NeatNet` is able to reduce the noise by leveraging prior knowledge from other users when it reconstructs the scene.

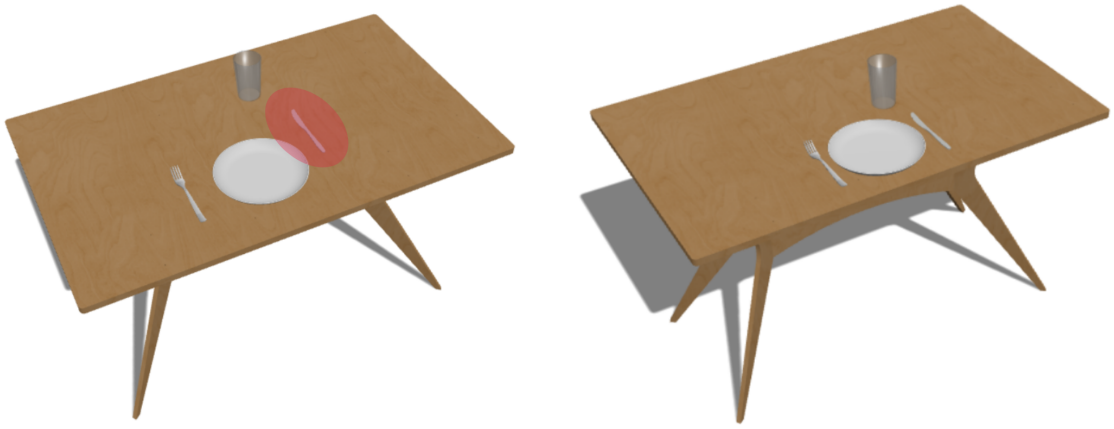

Figure 1: Tidying a known scene. **Left:** example arrangement supplied by a user. **Right:** reconstructed arrangement generated by `NeatNet`.

1.2 Can NeatNet Generalise to Objects Unseen During Training?

Table 3 in the paper shows the results of an experiment to place an object never seen before during training. Sample arrangements produced are visualised in Figures 2 and 3.

In the office scene, the network has never seen any examples of how a laptop should be placed, but it does know how this test user placed a computer, monitor, keyboard and mouse in their example arrangement of the office scene. Since these objects share linguistic and semantic features with the laptop, `NeatNet` is able to predict a reasonable position.

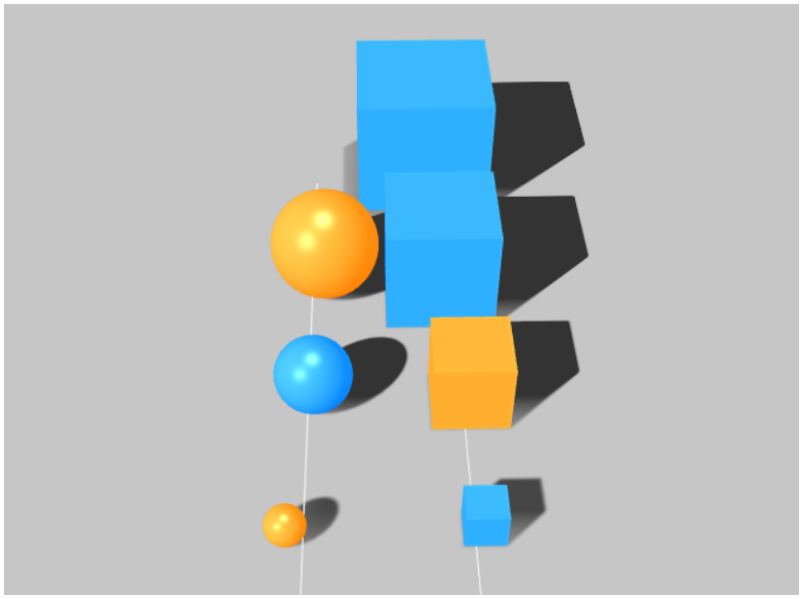

Figure 2: Placing a new abstract object (the largest blue box). Its size is outside the range seen by the network during training. The network is able to correctly place it on the right-hand side, since the user prefers to group objects by shape. It also correctly extrapolates the order-of-size pattern. This demonstrates an important spatial reasoning capability: for example, stacking plates or books in order of size would be a common scenario for a household robot.

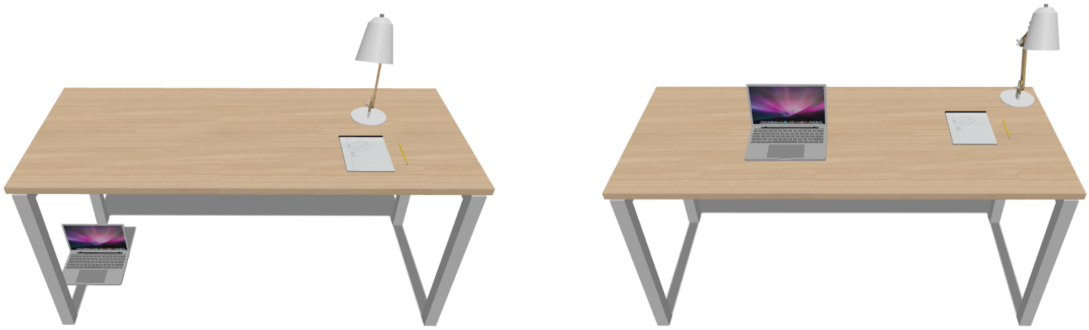

Figure 3: Placing a new object (the laptop). **Left:** `Nearest-Neighbour` baseline method. **Right:** `NeatNet` method, predicting a satisfactory position for the new laptop.

1.3 Can NeatNet Predict a Personalised Arrangement for a New Scene?

In this experiment (Table 4 in the paper) the network has not seen how this test user likes this new scene to be arranged. It predicts this based on this user's preferences, inferred from another scene, and prior knowledge from how similar training users arranged this new scene. Sample generated arrangements are shown in Figure 4.

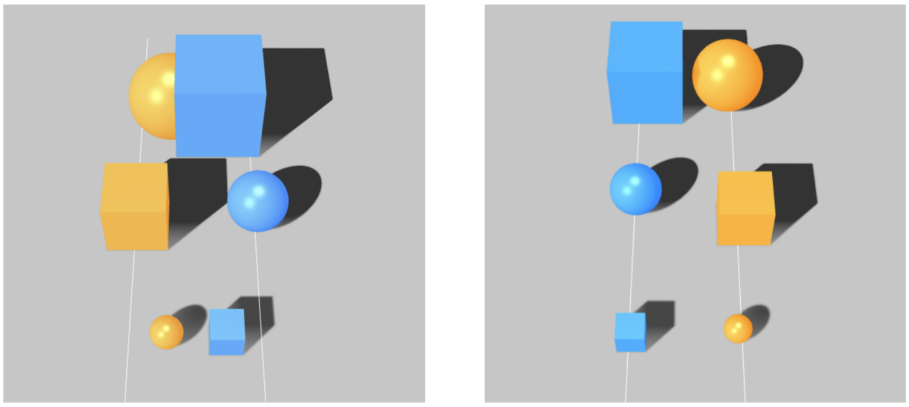

Figure 4: Arranging a new scene. **Left:** `kNN-Scene-Projection` correctly groups objects by colour, but does not line them up neatly, with some objects clipping into each other. **Right:** arrangement generated by `NeatNet`, using learned prior knowledge from how these objects were arranged by training users to neatly arrange them into lines.

1.4 Can the User Latent Space be Interpreted?

The NeatNet encoder maps each user to a vector in the shared latent space of user preferences. We visualise a batch of users in this latent space to investigate its interpretability in Figure 5. To aid visualisation, the user dimension hyperparameter was set to 2. A clear separation can be seen in the user latent space for the abstract scenes, where a cluster of users chose to group objects by colour, and another chose to do so by shape, showing that preferences which generalise across scenes can be discovered by the network. The user preference space for tidying the dining table can also be readily interpreted, with separate clusters emerging for left-handed and right-handed users. This shows that `NeatNet` can successfully learn high-level, interpretable preference characteristics such as handedness, which influence the placement of several objects in a scene.

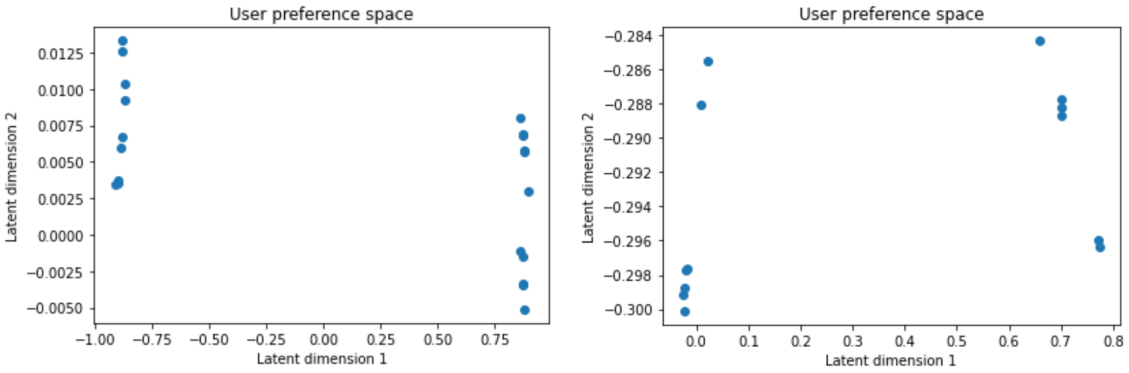

Figure 5: User latent space visualisation. Each point represents one inferred user vector. **Left:** abstract scene preferences. **Right:** dining scene preferences.

2 Implementation Details

In this section, we highlight several key points of implementation detail for the reader's convenience. Further low-level detail can be found in the code and inline documentation.

2.1 Libraries Used

The network architecture was implemented with the PyTorch machine learning library [1]. Additionally, the PyTorch Geometric library [2] was used to implement Graph Neural Network components.

2.2 Optimisation: Batching Scenes & Users

A user's preference vector is inferred from all the example scenes provided by that user, as described in Section 3.3 of the paper. Therefore, all the examples scenes for a user must be passed through the encoder.

A naive implementation would iterate through the scenes and pass each one through the encoder in sequence. However, this would perform $O(n)$ forward passes for n example scenes per user, which would significantly degrade the speed of training and inference.

Instead, we wish to pass all the example scenes for one user through the network in one forward pass. Therefore, we batch these scenes together by stacking them into a *supergraph*: a graph containing all the example scenes as subgraphs. The node matrices are concatenated together, so that the new supergraph contains all the nodes of the individual scene subgraphs. The edge structure is preserved so that each scene subgraph is fully connected but there are no edges between scene subgraphs. This is because each scene is arranged separately by the user, so remains separate in the supergraph structure.

This approach allows multiple scenes arranged by the same user to be processed by the network in one forward pass. A similar technique is applied to combine the graphs of several users into a single supergraph representing a batch of users, using the batch data loader from PyTorch Geometric [2]. This allows us to control the batch size as a hyperparameter, balancing regularisation with the stability of the training process.

2.3 Hyperparameter Settings

The optimiser used to update `NeatNet`'s parameters is the PyTorch Stochastic Gradient Descent implementation with momentum (based on validation performance, this was chosen over Adam [3] for this specific task). Additionally, we use a PyTorch learning rate scheduler which reduces the learning rate when performance plateaus, to fine-tune the model towards the end of the training process.

The network is trained for 2000 epochs, with an initial learning rate of 0.10 for abstract scenes and 0.08 for real scenes. We use a batch size of 4, to introduce sufficient regularisation. The graph encoding dimension is set to 20 for abstract scenes and 24 for real scenes. The encoder's Graph Attention [4] layer with a hidden dimension of 24 is followed by a fully-connected linear layer applied nodewise and 2 further fully-connected layers for the user preference extractor. The position predictor also uses a Graph Attention layer but with a hidden dimension of 32, followed by 2 fully-connected linear layers applied nodewise to predict positions for each node. Experiments on abstract scenes use a VAE β value [5] of 0.08, whereas 0.01 is used for real scenes to allow for further tailoring to individual preferences. The semantic embeddings for abstract objects are vectors which describe their size, RGB colour, and shape. Semantic embeddings for real objects are inferred from the word embeddings of their name. A negative slope of 0.2 is used for LeakyReLU activations [6]. The scheduler reduces the learning rate by a factor of 0.5, with a patience of 100 epochs (monitoring whether the loss is stagnating), and a cooldown

period of an additional 80 epochs after the last halving. However, these vary slightly based on the specific task: further details can be found in the code.

Data augmentation is applied to improve generalisation. Position encodings are normalised by subtracting the mean position across the training examples and downscaling the coordinates based on the maximum distance from the center found in the training dataset. This stabilises the training process. In order to prevent overfitting, Gaussian noise is added to object positions, with a standard deviation of 0.02 for the dining scene and 0.05 for the office scene. For experiments where the task is to predict the positions of missing objects, node masking (with a rate of 0.1) is applied during the training process, so that the network can learn to predict the positions of an object based on other objects. The initial learning rate is increased to 0.12, since the loss is otherwise more likely to stagnate in local optima, and the batch size is increased to 6 to stabilise the learning process. Similarly, a scene masking rate of 0.2 is applied to train the network to predict how a user would arrange a new scene based on their preferences inferred from another scene.

3 Pose Graph Baseline

One of our core contributions is the use of spatial preferences to tidy in a personalised way. To demonstrate that personalization improves user ratings, we include comparisons against strong baseline methods which can produce neat but not personalised arrangements.

We developed a baseline method referred to in results tables as "`Pose-Graph`". It constructs a probabilistic pose graph to model general tidying preferences (not specific to any individual). The parameters of this model are learned from the same example arrangements that `NeatNet` trains on. This model can be used as a tidiness cost function. A sampling & scoring graph optimisation technique is used to find the optimum of this cost function, outputting a tidy arrangement.

3.1 Modelling Arrangements as Pose Graphs

Each node represents an object, including its position. The edge between two nodes represents a probability distribution over possible displacements between those two objects. This distribution can be multi-modal, as shown in Figure 6, and so each edge stores the parameters of a Gaussian Mixture Model.

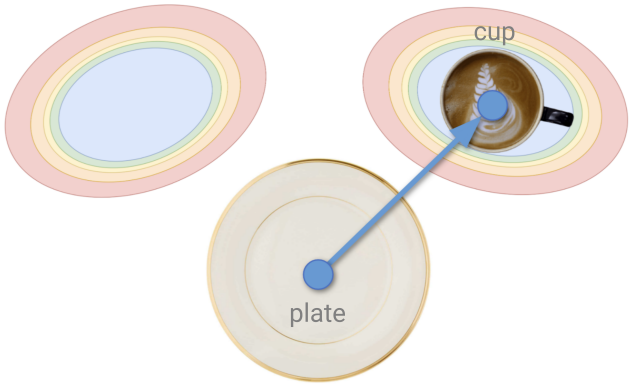

Figure 6: Example pose graph with two nodes and one edge, showing the most likely displacements from the plate to the cup. The Gaussian Mixture Model has two peaks: these positions for the cup as considered tidiest.

3.2 Learning Model Parameters

We now outline an algorithm for learning the parameters of this model from example scenes, so that the model will represent general tidying preferences. The output is a probabilistic pose graph, where each node is an object and each edge holds the parameters of a distribution over the displacements between those two objects.

Algorithm 1: Learning the Cost Function

input : List of training scenes, each a list of n objects
output: $n \times n$ matrix, where entry (i, j) is a distribution over displacements from i to j

```
distributions ← EmptyMatrix(n, n)
for each pair of objects i, j do
    displacements ← [ ]
    for each scene in scenes do
        displacements ← displacements ++ (scene[j] − scene[i])
    end
    // Calls a density estimation algorithm, like EM-GMM.
    distributions [i][j] ← FitDistribution(displacements)
end
```

To fit the distribution in each edge, we apply the Expectation-Maximisation algorithm (using the sckit-learn library [7]), which outputs the parameters of a Gaussian Mixture Model.

3.3 Using the Model as a Cost Function

Here we describe how the probabilistic pose graph can be used as a cost function for tidiness. The input to this function will be an arrangement $(x_1, \ldots, x_n)$, i.e. a position x_i for each object node i. The output will be some scalar tidiness cost, so that tidy arrangements have a low cost and untidy arrangements have a high cost.

Consider the local cost function $c_{ij}(x_i, x_j)$ for an edge between two objects i and j. Suppose that the displacement between them is $z_{ij} = x_j - x_i$. In a tidy arrangement, that displacement is a very likely one, i.e. $p_{ij}(z_{ij})$ is high. Therefore, we set the cost function of this edge to be the negative log-likelihood of the displacement between those two objects:

$$c_{ij}(x_i, x_j) \triangleq -\log p_{ij}(z_{ij}) \tag{1}$$

The global cost function is an aggregation of edge likelihoods, summing over each pair of objects:

$$C(x_i, \ldots, x_n) \triangleq -\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \log p_{ij}(z_{ij}) \tag{2}$$

This aggregation is similar to the global error function in graph-based SLAM literature, which is also the sum of the errors in each edge [8]. This means that arrangements where the distance between each object is likely are considered tidy, and arrangements where the distances between each object are implausible will have a high cost.

3.4 Finding the Optimum of the Cost Function

At this point, we know how we can use a probabilistic pose graph as a cost function, and we know how to learn the parameters of this cost function so that it models human tidying preferences. Given an arrangement of objects, this cost function will tell us whether one arrangement is more or less tidy than another.

Now, we want to find the tidiest possible arrangement, i.e. one which corresponds to the optimum of the cost function. Once we have this optimal arrangement, then the robot will know where each object should be placed to tidy a scene.

3.4.1 Objective Definition

The optimisation objective is to find the tidiest possible arrangement, i.e. we need to find a position for each object such that the overall cost function defined in Equation 2 is minimised. The tidiest arrangement which we are trying to find is therefore given by:

$$x_1^*, \ldots, x_n^* = \underset{x_1,\ldots,x_n}{\operatorname{argmin}} C(x_i, \ldots, x_n) \tag{3}$$

Intuitively, this means we need to find a position for each node such that each edge in the pose graph is as "likely" as possible. An analogy often used in SLAM literature goes as follows: imagine that each edge is an elastic spring attached between nodes. Shifting one node may loosen some springs attached to it but tighten some others instead, which are pulling in a different direction. We want to minimise the strain on the system, i.e. arrange the nodes so that as many of the springs are as loose as possible. This means we need to simultaneously optimise many relative constraints. Therefore, we can apply techniques inspired by SLAM literature to optimise this pose graph, and find a solution with a low cost.

3.4.2 Sampling & Scoring Algorithm

We now detail an algorithm for placing the objects into the scene in a way which simultaneously optimises the edges between all the object nodes. This is based on SLAM techniques for handling multi-modal distributions [9].

The intuitive idea is as follows. We start with a set of candidate arrangements, each containing just an origin node. Then, we add one object at a time. To add the next object, we sample from the distribution in the graph edge which connects it to its parent in the tree. This distribution returns a displacement, allowing us to place this new object into the arrangement based on the position of its parent (already placed). We place this object into every candidate arrangement. Then, we evaluate each candidate, by aggregating the likelihood of all the edges in that arrangement (using the cost functions defined in Section 3.3). Once all the objects have been placed, the candidate with the highest likelihood is the tidied solution. If at any stage we have too many candidates with low scores, we can re-sample: candidates with higher weights are more likely to survive, and unlikely arrangements will be eliminated. However, since our graphs are fully connected, re-sampling is often not necessary because the trees are rarely very deep, and so error does not accumulate as much as it would when performing SLAM along a long corridor.

This sampling & scoring algorithm is illustrated in Figure 7.

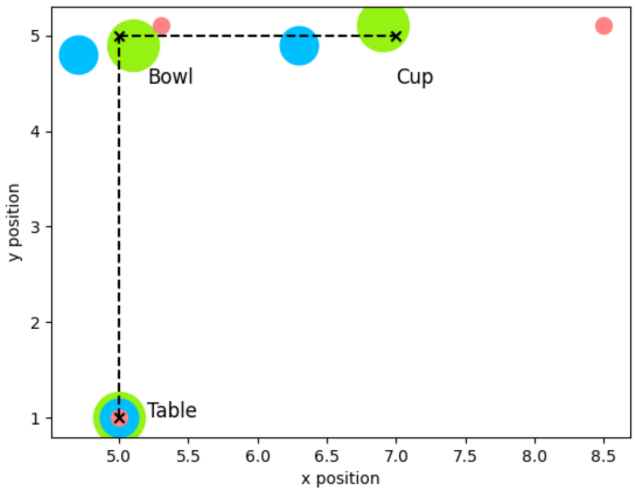

Figure 7: Illustration of our sampling-based method. Each candidate is assigned one colour and represents one full arrangement of the Table, Bowl and Cup. The Table is chosen as the origin node in this example. The green poses represent one possible arrangement, which has the greatest circles and the largest weight. This is because the relative displacements between the green pose estimates are closest to the ground truth optimal relative displacements in this experiment, marked by the dashed lines. The slightly smaller blue poses represent another candidate, slightly less likely but still reasonably highly weighted (note that the relative displacement between the Bowl and Cup is similar to the optimal). The red circles are the smallest because the arrangement is highly improbable.

The order in which we add objects into the scene is determined by a spanning tree of the pose graph: this is passed into this algorithm. Since all edges are used to score arrangements, the algorithm will run correctly regardless of which tree is chosen, but we can improve performance by selecting edges according to some heuristics. We found that prioritising edges with the lowest Bayesian Information Criterion [10] improved performance. Intuitively, if the graph contains a "strict" edge between two objects (e.g. between the fork and the knife), then that edge is likely to be prioritised, and this quickly eliminates untidy arrangements. An example tree is shown in Figure 8.

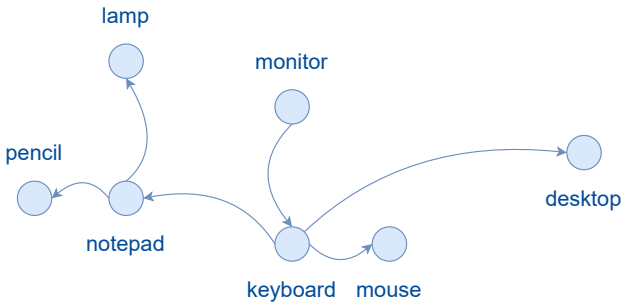

Figure 8: The tree selected for the office scene using BIC. Arrows represent edges in the tree. The notepad is chosen as the parent of both the lamp and the pencil, meaning that the positions of those objects are closely related, as would be expected semantically.

The full algorithm for sampling and scoring arrangements is given below. The output is a list of candidate arrangements, from which we can pick the tidiest (the one with the highest score). This produces an arrangement which is near the optimum of the learned cost function for tidiness.

Algorithm 2: Sampling & Scoring: Minimising the Cost Function

input : Matrix of relative distributions
input : A list of edges defining a minimum spanning tree in the distributions graph
input : Number of solutions, pop_size, defaulting to 1000
output: A list of scores
output: A list of arrangements, each a list of object positions

```
Initialise each score to 1 / pop_size
arrangements ← [ ]
for each edge in edges do
    (start, end) ← edge
    for i in 0..pop_size do
        // Sample a position for the new node.
        // Displacement may need to be flipped to match edge direction.
        displacement ← distributions [start][end].sample()
        start_pos ← arrangements [i][start]
        end_pos ← start_pos + displacement
        arrangements [i][end] ← end_pos
        // Score arrangement based on total likelihood of all edges so
           far.
        likelihood ← ScoreArrangement(distributions, arrangements [i])
        score ← 1 / pop_size + likelihood
    end
    Normalise scores: divide by total score.
    If too many arrangements have a low score, can resample here.
end
Sort arrangements so that highest scores are first.
```

We have shown how these algorithms allow the `Pose-Graph` baseline method to generate arrangements which are generally tidy, but not tailored to any specific user.

References

[1] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[2] M. Fey and J. E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[3] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[4] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018.

[5] I. Higgins, L. Matthey, A. Pal, C. P. Burgess, X. Glorot, M. Botvinick, S. Mohamed, and A. Lerchner. beta-VAE: Learning basic visual concepts with a constrained variational framework. In *ICLR*, 2017.

[6] A. L. Maas, A. Y. Hannun, and A. Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, volume 30, page 3, 2013.

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[8] G. Grisetti, R. Kümmerle, C. Stachniss, and W. Burgard. A tutorial on graph-based SLAM. *IEEE Intelligent Transportation Systems Magazine*, 2(4):31–43, 2010.

[9] M. Pfingsthorn and A. Birk. Simultaneous localization and mapping with multimodal probability distributions. *The International Journal of Robotics Research*, 32:143–171, 02 2013.

[10] A. A. Neath and J. E. Cavanaugh. The bayesian information criterion: Background, derivation, and applications. *WIREs Comput. Stat.*, 4(2):199–203, Mar. 2012. ISSN 1939-5108.