Training speedups via batching for geometric learning: an analysis of static and dynamic algorithms

Daniel T. Speckhard[1,2] **Tim Bechtel**[1,2] **Sebastian Kehl**[3] **Jonathan Godwin**[2,4]

Claudia Draxl[2]

[1]Humboldt-Universität zu Berlin [2]Max Planck Institute for Solid State Research
[3]Max Planck Computing and Data Facility [4]Orbital Materials
dts@physik.hu-berlin.edu

Abstract

Graph neural networks (GNN) have shown promising results for several domains such as materials science, chemistry, and the social sciences. GNN models often contain millions of parameters, and like other neural network (NN) models, are often fed only a fraction of the graphs that make up the training dataset in batches to update model parameters. The effect of batching algorithms on training time and model performance has been thoroughly explored for NNs but not yet for GNNs. We analyze two different batching algorithms for graph based models, namely static and dynamic batching for two datasets, the QM9 dataset of small molecules and the AFLOW materials database. Our experiments show that changing the batching algorithm can provide up to a 2.7x speedup, but the fastest algorithm depends on the data, model, batch size, hardware, and number of training steps run. Experiments show that for a select number of combinations of batch size, dataset, and model, significant differences in model learning metrics are observed between static and dynamic batching algorithms.

1 Introduction

Graph neural networks (GNN) models have recently shown great promise in regression and classification tasks, where the input data can be represented as graphs (19) (36). These methods have been applied to predict molecular and material behavior from nanometer to millimeter scale (32; 18; 29; 25). These tasks are of upmost importance to society, since they have opened up a research path towards exploring new molecules for drug design and carbon capture (9) and finding new materials for energy storage and generation. (30).

Similar to neural network (NN) models, GNN models typically contain millions of parameters and require large training datasets to achieve sufficient predictive power. In order to effectively train GNNs on large datasets, the graph structured data needs to be batched, otherwise each update over the entire dataset takes too long for the model parameters to converge in a reasonable amount of time. In this paper, we examine two different batching techniques for graph networks, static and dynamic batching. The two algorithms differ in that static batching always grabs the same number of graphs whereas dynamic batching adds graphs to the batch conditionally and seeks to ensure that the memory occupied by the batch of graphs is constant. For static-batching we look at three algorithms, the static-64, static-2^N and static-constant algorithms.

With the advent of neural architecture searches in different fields, including with graph structured data, it is often the case that researchers train model candidate architectures several thousand times (38; 14;

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

34). End-users are likely to re-train the resulting model and re-tune hyper-parameters on new datasets. The total costs involved with the GNN model search and re-training pose a significant computational, financial and environmental burden (21; 33; 26). Therefore, any savings in training time offered by the batching algorithm are significant.

We examine the effect of the algorithms on training time and metrics. For different batch sizes, we compare the computation time required to assemble the graphs in a batch. We also analyze the time to update the model parameter weights using a batch of graphs. Finally, we monitor the effect of the batching techniques on the test metrics of the models. We do this for different batch sizes, models, and datasets. Our main contributions presented in this paper are:

- We formally introduce the static and dynamic batching algorithms and several variants thereof.
- The majority of experiments show no significant difference in model learning between the algorithms, but for some combinations of the dataset, model and batch size, the dynamic batching algorithm yields significantly different test metrics than the static algorithms.
- We find the static-constant algorithm to be the slowest algorithm and recommend not using it unless the goal is to maximize the robustness of the training pipeline. Across our experiments, we observe speedups of up to 12.5x when switching from the static-constant algorithm to another algorithm.
- Excluding the static-constants batching algorithm, we observe at most a 2.7x speedup in mean time per training step when switching from the slowest algorithm to the fastest.
- We demonstrate that the performance of the algorithm is dependent on the graph distribution in the dataset, model used, batch size, and the number of training steps run.
- The static-64 and dynamic algorithms generally outperform the static-2^N algorithms.
- We recommend using the dynamic algorithm, whose performance is similar to the static-64 algorithm, but is faster for fewer number of training steps.

2 Related Work

Several different batching methods, i.e., full-batching, mini-batching (3), stochastic learning (5), have been thoroughly analyzed for NNs in the literature (4). For instance, NN learning performance when using mini-batching has been examined as a function of the batch size. Computation time as a function of batch size has also been explored on GPUs (20). In (7), large batch sizes were found to typically slow down the convergence of the model parameters. In practice, researchers typically set the batch size as a hyperparameter that is found via cross validation (CV).

For NNs, updating the model weights can take up the bulk of training time through backpropagation (24). Typically, NNs operate on numeric data (or data that has been transformed into numeric values). When training the NN with a fixed mini-batch size of numeric data, batches have constant shape and memory requirements. Using JAX, this enables highly efficient training, since the gradient update step can be compiled once at the start of the training. For convolutional neural networks (CNNs), this is not the case, and images of different pixel dimensions are often padded before being fed into the network (35). Similarly, for graph neural networks, the graphs in the dataset typically contain a wide variety of number of nodes and edges. To this end, batching algorithms have emerged, which pad batches of graphs to constant shapes. In this way, the gradient update step for GNNs on batches can be compiled on GPU. Two such methods have become popular, static batching and dynamic batching. The static batching methods always collect a fixed number of graphs while dynamic methods add graphs incrementally to a batch until some padding budget/target is reached.

That said, not all GNN models and libraries pad their data. M3GNet (8), a GNN trained for interatomic potentials (IAP) is written in TensorFlow and performs batching in a similar way to NNs. Its batching procedure collects a number of graphs corresponding to the batch size, and concatenates the atoms, bonds, states and indices into larger lists for the batch. However, it does not perform any padding after batching data. The pyTorch GNN (ptgnn) library implements a dynamic batching algorithm that also does not use padding (1). The algorithm adds graphs until either the batch has the desired batch size (i.e., number of graphs) or some safety limit on the number of nodes in the batch has been reached. This safety limit ensures that the batch will fit into memory.

The pyTorch geometric library (13) implements a similar dynamic batching algorithm. The user specifies whether to use either (but not both) a total number of nodes as the batch cutoff/limit or total number of edges. The algorithm then adds graphs incrementally to the batch until the target number of graphs are in the batch (i.e., the target batch size) or the cutoff has been reached. It also allows the user to skip adding single graphs to the batch that would by themselves exceed the node/edge cutoff.

The Jraph library, which is built on JAX (6), implements a dynamic batching algorithm (15). Given a batch size, it performs a one-time estimate of the next largest multiple of 64 to which the sum of the nodes/edges in the batch should be padded. We can think of these estimates as the node/edge padding targets for each batch (i.e., the number of nodes/edges in a batch after padding). This estimation is done by sampling a random subset of the data. It then iteratively adds one graph at a time to the batch and stops if adding another graph would exceed the node/edge padding targets or the maximum number of graphs (i.e., the batch size) is already in the batch.

The Tensorflow GNN library (12) implements a static and dynamic batching algorithm. The static batching adds a fixed number of graphs to the batch and then pads to a constant padding value. The dynamic batching method, similar to Jraph, estimates a padding budget (they call it a size constraint) for the batch based on a random subset of data, and then adds graphs incrementally to the batch as long as they do not break this budget.

The static and dynamic algorithms have, to our knowledge, not been described in the literature, but solely within code repositories. Experiments to measure the training time as a function of the algorithm, batch size, model, or dataset are also to our knowledge not found in the literature. Our work seeks to describe the static and dynamic batching algorithms in sufficient detail and perform the aforementioned timing experiments using an implementation based on the Jraph library.

3 Preliminaries

Static-constant batching. In static batching, the method grabs a fixed number of graphs for each batch. It then checks how many nodes/edges are present in the batch. It then adds a dummy graph and adds dummy edges/nodes to this graph to serve as padding. Typically, the number of graphs included in the batch is the batch size minus one in order to leave space for the dummy graph in the batch. The algorithm pads the number of nodes/edges up to the next power of two to try to ensure that most batches have a similar number of nodes/edges, and thereby a similar memory footprints. In the static-constant algorithm, implemented in TensorFlow's GNN (TF-GNN) library, the algorithm first finds the graph with the most nodes/edges in the dataset and multiplies this number by the batch size to get the padding target. The padding target dictates how many nodes/edges are expected in each batch. For example, for batch size 32, the algorithm grabs 31 graphs, and then adds a dummy graph with enough dummy nodes/edges so that the batch has the same nodes/edges as the padding target. In this way, each static-constant batch occupies the same memory footprint. This means that the gradient-update step, which updates the model parameters, can always expect a batch size of a constant memory input, and can be compiled for this input size. The downside to this approach, is that for some datasets and batch sizes the target is extremely large since it is based off the largest graph in the dataset.

Static-2^N and static-64 algorithms. In the static-2^N algorithm, which we introduce here, the algorithm grabs batch size minus one number of graphs into a batch, and then pads the batch's total number of nodes/edges to the next largest power of two. Over the course of training, one might encounter a batch that requires a larger power of two than previously used in order to contain all of the nodes and edges in the batch. This requires the gradient-update function to recompile for a different, i.e., larger, input size. The static-2^N algorithm is shown in the appendix Algorithm 4. Similarly, in the static-64 algorithm, which we also introduce here, the total number of nodes/edges in the batch are padded to the next largest multiple of 64.

Dynamic batching. The dynamic batching algorithm, as implemented in Jraph and TF-GNN, first grabs a random subset of the data (e.g., 1000 graphs). It then estimates the mean nodes per graph and mean edges per graph. It uses this to create a padding target (or memory budget), which in effect is the maximum number of nodes and edges allowed to be stored in the batch. The edge (node) padding target is created by multiplying the mean number of edges (nodes) by the batch size and then rounding to the next largest multiple of 64. The algorithm to estimate the padding target is shown in

the appendix in Algorithm 5. It is used to determine the padding required for each batch to ensure that each batch uses the same amount of memory.

The dynamic algorithm, given the padding target, appends one graph at a time to the initially empty batch. Before the addition of each graph, it checks if adding the graph would put the batch over the node or edge padding target. If it does, the graph is not added to the batch. Instead, the algorithm adds a dummy graph with fake nodes/edges, the number of which is determined so as to pad the number of nodes/edges to the padding target. It then adds, if necessary, dummy graphs with no nodes/edges to ensure that the number of graphs in the batch is equal to the batch size. A pseudocode representation is shown in Algorithm 1. The batch method called in the algorithm, concatenates all of the nodes/edges in the list of graphs and creates a single super-graph out of the list of graphs (Pseudocode for the method is found in the Appendix, Section B).

Algorithm 1 dynamic batching

```
Input: set of training graphs G
b ← (n_n, n_e, n_g) = estimate_budget (G)
B ← []
s ← (0, 0, 0)
n_n ← 0 {Accumulated # of nodes}
n_e ← 0
for g in G do
    s_i ← graph_size (g)
    if s + s_i > b − (0, 0, 1) then
        yield batch(pad-to-target(B, n_n, n_e))
    else
        B += g
        s += s_i
        n_n += s_i[0] {Update node sum}
        n_e += s_i[1]
    end if
end for
```

If, however, during the course of training, a single graph is encountered with more nodes/edges than the padding target, the program terminates. The algorithm must be restarted with a larger node/edge padding target. This issue can be minimized by looking at a larger sample of graphs when estimating the targets. Alternatively, the user can loop through the entire dataset of graphs and find the graph with the largest number of nodes/edges to determine suitable budgets. Note that the algorithm we present here is quite different from the pyTorch-Geometric implementation, whose implementation does not perform a budget estimation (has to be entered by the user) and it does not perform a check on the number of edges, but only the nodes. We find the Jraph/TF-GNN implementation, for which a simplified version is shown in Algorithm 1, to be the more sophisticated method, and so it was used for our tests.

4 Datasets

We run profiling experiments on two different datasets. The QM9 dataset (27) contains chemical properties (e.g., internal energies, molecular orbital energy levels) of small organic molecules. The AFLOW dataset (10) is a collection of relevant material properties (e.g., formation energies, band gaps, elastic properties). We choose these datasets for two reasons. First, they have served as benchmark datasets for GNN models (32; 23; 2). Second, they have real world applications. For instance, learning on the QM9 dataset may help models better perform targeted drug discovery, while AFLOW data may help models discover more efficient solar cell semiconductors. For QM9 we target the molecular internal energy and for AFLOW we focus on learning the enthalpy of the crystal.

The datasets are not provided in a graph format. To convert them to graphs, each atom is represented as a node in the graph. For QM9 the edges are added by fully connecting the nodes. For AFLOW, the edges are added with the K-nearest neighbor algorithm based on pairwise distance and setting K equal to 24. This KNN value was chosen from CVstudies studies in the literature (18). From the AFLOW dataset, we remove duplicates and keep only calculations that contain both the enthalpy and

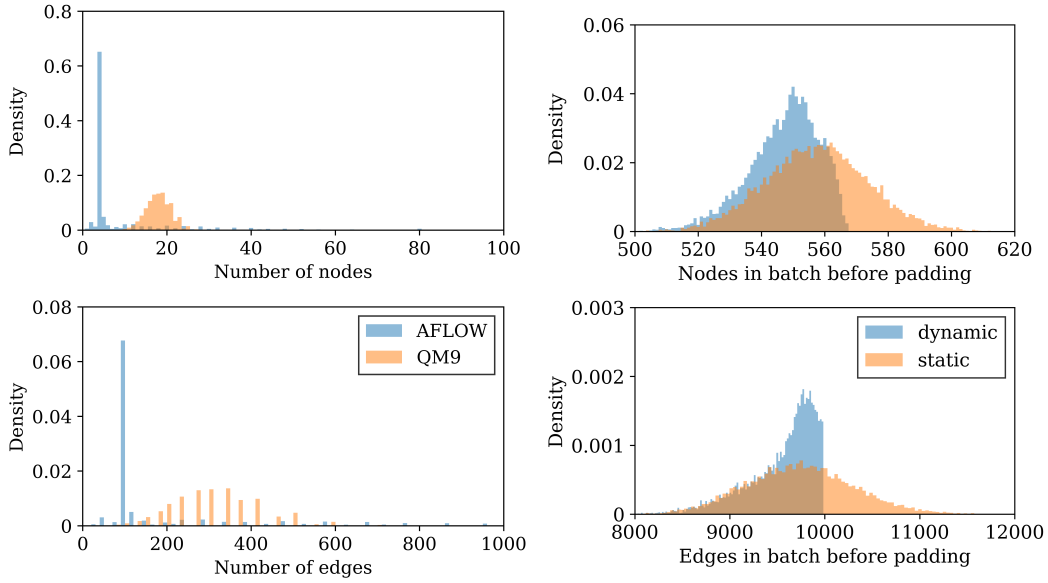

Figure 1: Left: Histograms of the number of nodes (top) and edges (bottom) in the AFLOW and QM9 datasets. Right: Histograms of the number of nodes (top) and edges (bottom) in a batch of size 32 before padding for the static-2^N and dynamic batching algorithms running on the QM9 dataset.

the band structure, following the procedure in (2). We visualize the variety in the graph structures in the datasets in Fig. 1. From the figure we can see that the QM9 dataset node distribution appears Gaussian and centered around 17 nodes per graph. AFLOW's node distribution has long tails despite the mean of the distribution being smaller than QM9's. The distribution of edges for AFLOW and QM9 are reflective of the distribution of the nodes, this is because the number of edges are dependent on the number of nodes when using the KNN algorithm or fully connecting the graphs.

5 Implementation

There are several open-source software libraries to choose from to perform these experiments. We opt to use the Jraph library (15), which is built using the JAX library (6). The JAX library traces out computations into a computation graph that it uses to optimize and compile the code. This compiled auto-differentiation code is optimized to run fast, especially on GPU. To run our experiments, we extend Jraph to run the static-2^N, static-64, and static-constant algorithms, log necessary profiling information and run the models we have selected for our tests.

6 Models

We evaluate three models to get insight into any model dependent effects on our profiling results. Specifically, the SchNet model (32), a message passing model with edge updates (MPEU) model (18; 2) and the PaiNN model (31) which uses equivariant features. The three models were chosen, since the SchNet model is often used as a benchmark, the MPEU has shown better results on the AFLOW dataset in a benchmark study (2), and the PaiNN model is widely used (22). These models vary in size. The SchNet model is the smallest with 84,768 trainable parameters. The next largest is the PaiNN model with 177,568 trainable parameters, and the MPEU model contains 2,553,472.

7 Test setup and compute hardware

As mentioned, the benefit of dynamic batching and static-constant batching is that the memory allocated to a batch of graphs is always the same for each batch. This allows the user to compile the update functions only once and optimize the compiled binary to run on GPU. Note that with static-64

and static-2^N batching, the update function can also be compiled, but needs to be recompiled for different multiples of 64 / powers of two (the static-constant algorithm does not have this problem). We time the training of models for different batching algorithms.

Batching and weight update timing experimental setup. We perform timing experiments on both algorithms for four batch sizes (16, 32, 64, 128) using both datasets and the three models for two million training steps. This number of training steps was chosen since it resulted in converged models (2). Experiments with fewer steps are run and the results can be found in the Appendix. We run each combination of batch size, algorithm, model, and dataset ten times to limit noise effects from the hardware in the profiling results. Each experiment reports the mean batching time, mean gradient update step time (update time), and the sum of the two (combined time).

Hardware used. We run experiments on a cluster that comprises two Intel Xeon IceLake Platinum 8360Y CPUs and 4 NVIDIA A100 GPUs connected via NVLINK. We ran experiments with using a single CPU core and single GPU. This is done so that we can understand the algorithms performance in a controlled setting. Results from running only on CPU and multi-gpu setups are described in the appendix.

8 Profiling results

Batch statistics To understand how the batching algorithms work in practice, we analyze the statistics of the number of nodes in a batch before padding occurs. The histograms for both batching algorithms are shown for the QM9 dataset in Fig. 1 for 10,000 batches with a batch size of 32. As stated above, the dynamic batching algorithm checks the node and edge padding target, as well as the number of graphs already in the batch, before adding a graph to the batch. For this dataset and batch size, the dynamic algorithm's node padding target is 576. We can see that all dynamic batches have fewer than 576 nodes before padding. This results in a one-sided distribution, roughly a truncated Gaussian distribution, where the right-hand side is cutoff near the budget. For static batching, however, no such check exists, and we observe a roughly Gaussian distribution of nodes. Similar effects are seen in the figure for the number of edges before padding.

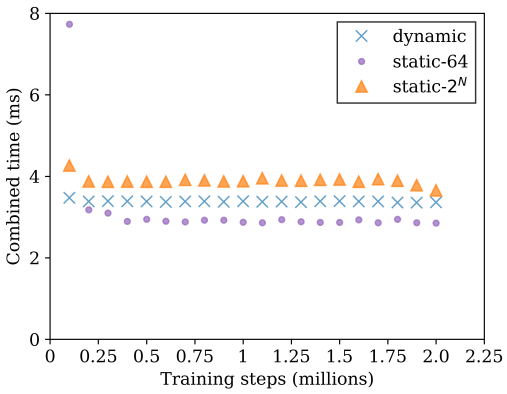

Figure 2: The running average of the combined time (sum of the batching step and gradient-update step and) required per training step as a function of the total number of training steps run. Here only a single iteration is run for the batch size 32, MPEU model and QM9 dataset for both the dynamic, static-64 and static-2^N algorithms.

Recompilation. For fewer training steps, the number of recompilations required in the gradient-update step can be the dominant factor defining the ranking of the algorithms. The number of recompilations is shown in Fig. 3 for the QM9 data for two million training steps. The number of recompilations is model and hardware independent. We see that the static-64 algorithm performs several orders of magnitude more recompilations than the static-2^N and dynamic algorithms. This is to be expected, since the dynamic algorithm compiles only once, and if it encounters a graph, which is bigger than the budget, the program terminates and needs to be restarted with a larger pardding target. Moreover, for static-64 batching, JAX recompiles the gradient-update step function every time a new nearest multiple of 64 is encountered when padding, and for static-2^N batching every time a new nearest power of two is encountered, which is less likely for larger powers of two. For example, for the batch size of 32, the static-2^N algorithm recompiles four times while the static-64 algorithm recompiles 89 times.

Most of the recompilations for the QM9 dataset happen in the first hundred thousand training steps. From training step one hundred thousand to two million, the static-2^N algorithm does not recompile again and the static-64 algorithm recompiles only an additional twenty times. The effect of recompilations is seen in Fig. 6 and Fig. 7 in the Appendix

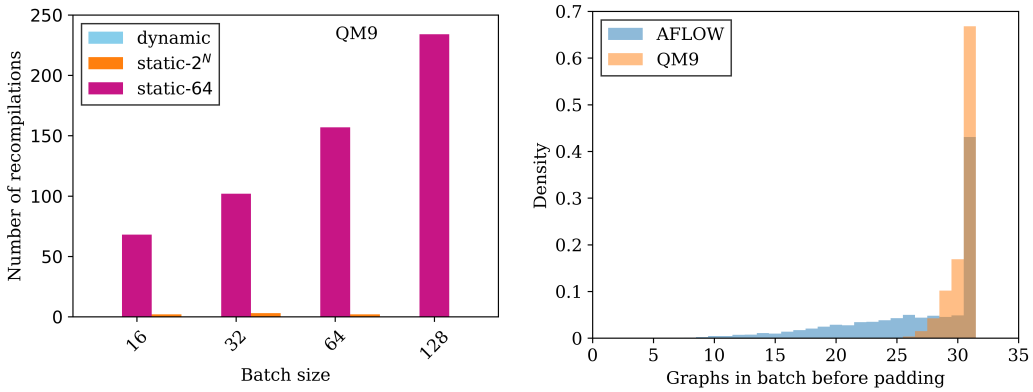

Figure 3: Left: number of recompilations on the QM9 dataset after two million training steps in the gradient-update step as a function of batch size for different batching algorithms. Note, that these numbers are model and hardware independent. Right: histogram of the number of graphs before padding in a batch size of 32 for the AFLOW and QM9 dataset.

which show the mean and median combined times after running only one hundred thousand training steps respectively. In the figures, the static-64 algorithm has the lowest median update times while the static-2^N has the lowest mean update times. The mean is affected by outliers while the median is not. This tells us that when removing the effect of outliers, i.e., slow gradient-update steps which are caused by recompilations, the static-64 algorithm is fastest to update the gradients. This tells us that the algorithm training time rankings are dependent on the dataset (where more variability in graph sizes causes more recompilations) and the number of training steps run.

Static-constant batching results. The static-constant algorithm was also evaluated for a subset of data, models and batch sizes. The results are shown in the Appendix Section H. The static-constant model performs poorly compared to the other static algorithms. As shown in Figure 1, the AFLOW dataset contains a long-tailed distribution in terms of the node/edges (the estimated excess kurtosis of the two histograms are greater than one (11)). Moreover, the mean number of nodes/edges is 10-20 times smaller than the maximum number. As a result, for AFLOW, the static-constant model uses a large padding target that slows down the batching and update step. For QM9, the histograms of the dataset show that the distribution of nodes and edges resemble more closely a normal distribution. The number of edges varies from roughly one hundred to six hundred edges, and as a result, for the QM9 data, we see a smaller yet still significant slowdown using the static-constant batching algorithm compared to other methods.

Static-64, static-2^N and dynamic algorithm results Fig. 4 displays the breakdown of batching time per training step. We see the batching times appear roughly model and dataset independent and scale linearly with the batch size. This reflects the expectation that the time complexity of the batching algorithms is governed by the batch size. The dynamic batching is slower than the static-2^N and static-64 algorithms. In all cases, the dynamic algorithm is slower due to the added overhead from the bookkeeping of the padding targets.

For the SchNet and MPEU models, the timing results of the gradient-update step at small batch sizes are similar for all algorithms. For the PaiNN model, the static-2^N algorithm has a slower gradient update for all batch sizes. For larger batch sizes, across all models and datasets, the dynamic batching algorithm performs the gradient update the fastest. The dynamic batching algorithm is the fastest, since the gradient update code is never recompiled. As seen in the histograms in Fig. 1, the number of nodes in a batch before padding is typically smaller for the dynamic algorithm. This is also true after padding. GNNs like the models we examine here, typically have a node update equation, which means that inference, and therefore the gradient update step, scales roughly linearly with the number of nodes. When the difference between the number of nodes is small between the dynamic and static-64 algorithm we do not expect much difference in the gradient update step time. The static-2^N algorithm, however, typically adds a large number of padded nodes, which explains why the algorithm is typically slower than the dynamic algorithm in the gradient update step. The static-64 and dynamic algorithm show roughly linear scaling in the update time with the batch size. The static-2^N algorithm

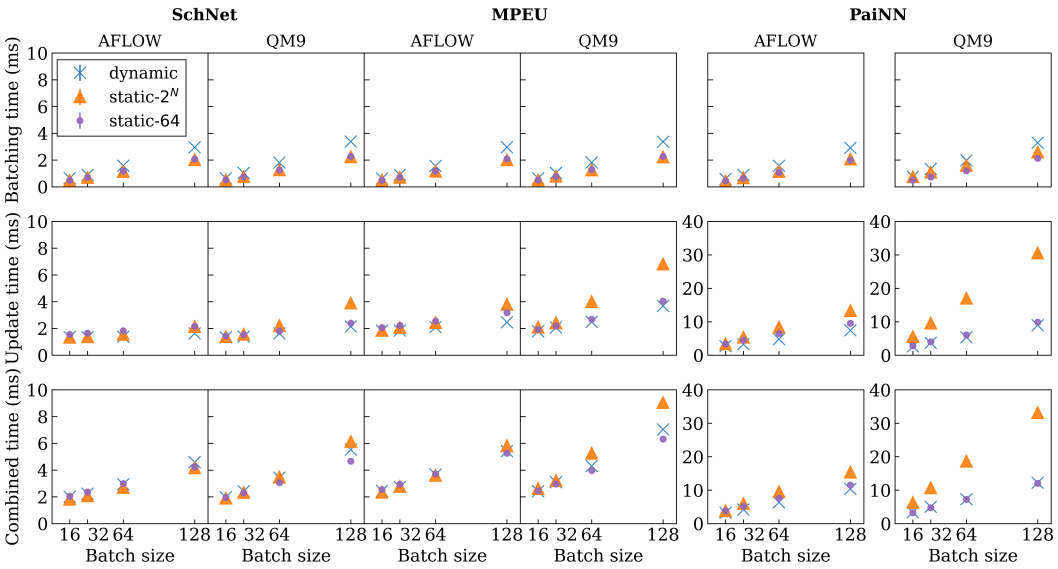

Figure 4: Timing measurements while varying the batch size for SchNet (left two columns), MPEU (middle columns) and PaiNN (right two columns) Results for the AFLOW and QM9 dataset are shown for each model. For each datapoint, ten iterations of two million training steps are run.

shows poor relative performance especially for larger batch sizes. More work needs to be done to understand the static-2^N scaling behavior.

The sum of the batching and gradient-update step mean timings, i.e., the mean combined time, gives us the mean time per training step. The speedup is defined as the ratio between the slowest algorithm combined time per step, T_s, and the faster algorithm combined time per step, T_x, as $\frac{T_s}{T_x}$ (17). For the PaiNN model and batch size 128, the dynamic batching and static-64 algorithm have a speedup of over 2.7 when compared to the static-2^N algorithm. Similarly, for the PaiNN model for AFLOW and batch sizes of 32, 64 and 128, the dynamic algorithm has a speedup of roughly 1.5x when compared to the static-2^N algorithm. For the MPEU model on the QM9 dataset, we observe 1.5x speedup when using the static-64 algorithm compared to the static-$2N$ algorithm. These observations over different datasets and models point to the large effect that changing the batching algorithm can have on the training time. The speedup from switching algorithms for the PaiNN model on the QM9 and AFLOW datasets can be seen in the Fig. 5. If we include the static-constant algorithm, which is always the slowest algorithm for the parameter space we explored here, the largest speedup we see is 3.64 for SchNet, batch size 32, and AFLOW dataset.

Effect on learning. The batching algorithm may also affect the learning of the model. Dynamic batching does not always include the same number of graphs in a batch, since one or more fake padded graphs are added when the node or edge budget is reached. These padded graphs do not contribute to the gradient in the update step. The histogram in Fig. 3 depicts the distribution of the number of graphs before padding for the AFLOW and QM9 dataset after running 10,000 iterations with the dynamic algorithm. While most batches contain 31 graphs for a batch size of 32, some batches for the AFLOW dataset contain as few as ten graphs from the training data. The fact that the QM9 graph node and edge distribution have shorter tails than the AFLOW distribution, as seen in Fig. 1, may explain why the distribution of the number of graphs before padding has shorter tails for QM9 than for AFLOW. The effect of the algorithm on learning can be discerned by training each model to convergence. The test loss (RMSE) as a function of the number of training steps on the QM9 data is shown in the Appendix Fig. 12. Visually, we see general agreement across models but for some batch sizes there are significant discrepancies.

To quantify the difference in test metrics between two batching methods, we perform a pairwise student t-test across the final RMSE test loss. Fig. 14 in the Appendix shows a heatmap of pairwise t-test values and Fig. 15 shows the associated p-values across batching methods for the SchNet

model for the QM9 dataset and batch size 64. The complete set of plots for the other combinations of datasets, models and batch sizes can be found in the Data Availability section.

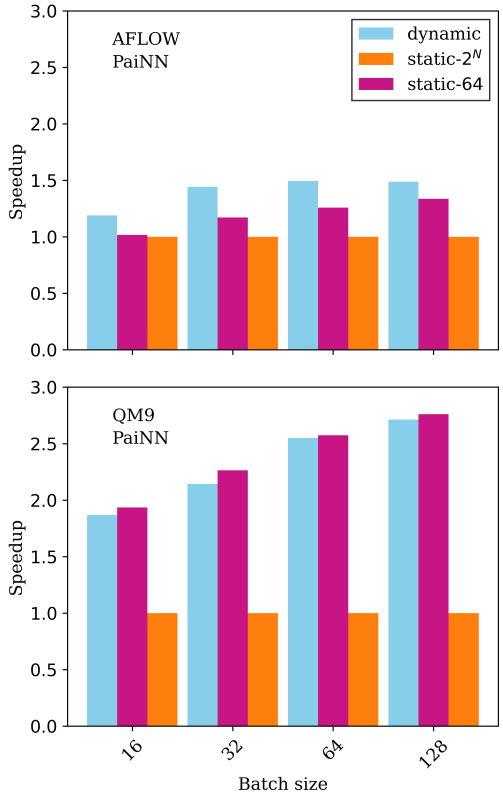

Figure 5: Speedup, on GPU, when switching from the slowest algorithm in terms of combined training time per step for the PaiNN model (not including the static-constant model) for the AFLOW (top) and QM9 (bottom) datasets. For both datasets the slowest algorithm is the static-2^N algorithm. If the static-constant algorithm is included the speedup increases to a maximum of to 12.5

One can either use the t-test statistic (critical value approach) or the associated p-value to determine whether to reject the null hypothesis (16) that the test metrics after two million training steps from different batching methods belong to the same underlying distribution. We adopt a p-value criteria from the literature (28), that argues that there is little evidence that the distribution of the test metrics differ significantly if the p-values are greater than 0.10. For p-values between 0.05 to 0.10, there is moderate evidence that the distributions are different. For lower p-values the evidence is strong. Ignoring the p-values that are greater than 0.10, we can look at several interesting results. For the QM9 data, SchNet model, and batch size 64, the static-2^N and dynamic methods have a p-value of 0.033. For the QM9 data, SchNet model, and batch size 32, the static-2^N and static-64 algorithms have p-values with the dynamic algorithm of 0.067 and 0.053, respectively. For QM9 data, MPEU model, and batch size 64, the static-2^N and dynamic methods have a p-value of 0.071. These results show that for some datasets, models, and batch sizes, the dynamic algorithm gives significantly different test metrics than the static algorithm. For the majority of combinations, however, the results are similar, which may be due to use of the Adam optimizer whose adaptive estimation of first and second moments may reduce the effect of occasionally having smaller batch sizes.

9 Discussion

We find the static-constant algorithm to perform the worst in terms of timing performance per training step. We do not recommend using the algorithm unless the user has a dataset where the maximum graph size is very similar to the mean graph size, and/or the priority is to have robustness in the training pipeline. We see in some cases a speedup of up to 12.5x when switching from the static constant algorithm to the other algorithms. If we discard the static-constant algorithm, choosing one algorithm versus another can result in more than a 2.7x speedup in training time without observable differences in learning metrics. For larger hyperparameter searches or neural architecture searches (NAS), this speedup can save a very large amount of computational resources. Across, the parameter space explored in this work, the static-64 and dynamic algorithm appear the fastest. In general, we would recommend the dynamic algorithm since it performs faster than the static-64 algorithm for fewer number of training steps, which can be important for NAS.

Diving deeper into the timing performance, the static algorithms perform the batching step faster. This is due to the added overhead from the bookkeeping of the padding targets in the dynamic algorithm. All of the algorithms show linear scaling in the batching time with respect to the batch size. The static-64 and dynamic algorithms show linear scaling in the gradient-update step, while the static-2^N algorithm shows exponential scaling. This is due to the exponentially larger memory sizes required when using powers of two for padding. The gradient-update step is performed fastest by the dynamic

algorithm. This is partially because the algorithm is never recompiled. Moreover, the static-2^N algorithm typically has a larger number of nodes/edges in the batch, which slows down the gradient update step.

The learning curves from the static and dynamic algorithms are mostly similar across batching algorithms, but for some batch sizes, datasets, and models they differ significantly between static and dynamic algorithms. This is because the dynamic batching algorithm contains, on average, fewer labeled graphs per batch.

We hope that this work serves to aide users to reduce the computational cost of training graph based models. Further work might focus on predicting the fastest algorithm as a function of the dataset, batch size, and model. Finally, this work could in the future be extended to combine the batching algorithms with batching schemes that partition large graph networks across GPUs.

Software and data availability

The software to perform the batching, profiling experiments, parsing of experiments, and plotting is found in `https://github.com/speckhard/gnn_batching/`. The complete set of student t-test heatmaps can be found at `https://drive.google.com/drive/folders/1W4Fc0a7ne0z5vUDYgNc1_YuUNcjxwX3W?usp=sharing_`.

References

[1] ALLAMANIS, M., MIR, A., AND PATI, S. ptgnn: A pytorch gnn library, 2022.

[2] BECHTEL, T., SPECKHARD, D. T., GODWIN, J., AND DRAXL, C. Band-gap regression with architecture-optimized message-passing neural networks. *arXiv preprint arXiv:2309.06348* (2023).

[3] BISHOP, C. M., AND NASRABADI, N. M. *Pattern recognition and machine learning*, vol. 4. Springer, 2006.

[4] BOTTOU, L., AND BOUSQUET, O. The tradeoffs of large scale learning. *Advances in neural information processing systems 20* (2007).

[5] BOTTOU, L., ET AL. Stochastic gradient learning in neural networks. *Proceedings of Neuro-Nımes 91*, 8 (1991), 12.

[6] BRADBURY, J., FROSTIG, R., HAWKINS, P., JOHNSON, M. J., LEARY, C., MACLAURIN, D., NECULA, G., PASZKE, A., VANDERPLAS, J., WANDERMAN-MILNE, S., AND ZHANG, Q. JAX: composable transformations of Python+NumPy programs, 2018.

[7] BYRD, R. H., CHIN, G. M., NOCEDAL, J., AND WU, Y. Sample size selection in optimization methods for machine learning. *Mathematical programming 134*, 1 (2012), 127–155.

[8] CHEN, C., AND ONG, S. P. A universal graph deep learning interatomic potential for the periodic table. *Nature Computational Science 2*, 11 (2022), 718–728.

[9] CHOUDHARY, K., YILDIRIM, T., SIDERIUS, D. W., KUSNE, A. G., MCDANNALD, A., AND ORTIZ-MONTALVO, D. L. Graph neural network predictions of metal organic framework co2 adsorption properties. *Computational Materials Science 210* (2022), 111388.

[10] CURTAROLO, S., SETYAWAN, W., HART, G. L., JAHNATEK, M., CHEPULSKII, R. V., TAYLOR, R. H., WANG, S., XUE, J., YANG, K., LEVY, O., ET AL. Aflow: An automatic framework for high-throughput materials discovery. *Computational Materials Science 58* (2012), 218–226.

[11] DECARLO, L. T. On the meaning and use of kurtosis. *Psychological methods 2*, 3 (1997), 292.

[12] FERLUDIN, O., EIGENWILLIG, A., BLAIS, M., ZELLE, D., PFEIFER, J., SANCHEZ-GONZALEZ, A., LI, W. L. S., ABU-EL-HAIJA, S., BATTAGLIA, P., BULUT, N., HALCROW, J., DE ALMEIDA, F. M. G., GONNET, P., JIANG, L., KOTHARI, P., LATTANZI, S., LINHARES,

A., MAYER, B., MIRROKNI, V., PALOWITCH, J., PARADKAR, M., SHE, J., TSITSULIN, A., VILLELA, K., WANG, L., WONG, D., AND PEROZZI, B. TF-GNN: graph neural networks in tensorflow. *CoRR abs/2207.03522* (2023).

[13] FEY, M., AND LENSSEN, J. E. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428* (2019).

[14] GAO, Y., YANG, H., ZHANG, P., ZHOU, C., AND HU, Y. Graph neural architecture search. In *International joint conference on artificial intelligence* (2021), International Joint Conference on Artificial Intelligence.

[15] GODWIN*, J., KECK*, T., BATTAGLIA, P., BAPST, V., KIPF, T., LI, Y., STACHENFELD, K., VELIČKOVIĆ, P., AND SANCHEZ-GONZALEZ, A. Jraph: A library for graph neural networks in jax., 2020.

[16] HARTMANN, K., KROIS, J., AND RUDOLPH, A. Statistics and geodata analysis using r, 2023.

[17] HENNESSY, J. L., AND PATTERSON, D. A. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[18] JØRGENSEN, P. B., JACOBSEN, K. W., AND SCHMIDT, M. N. Neural message passing with edge updates for predicting properties of molecules and materials. *arXiv preprint arXiv:1806.03146* (2018).

[19] KIPF, T. N., AND WELLING, M. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[20] KOCHURA, Y., GORDIENKO, Y., TARAN, V., GORDIENKO, N., ROKOVYI, A., ALIENIN, O., AND STIRENKO, S. Batch size influence on performance of graphic and tensor processing units during training and inference phases. In *International Conference on Computer Science, Engineering and Education Applications* (2019), Springer, pp. 658–668.

[21] KOROLEV, V., AND MITROFANOV, A. The carbon footprint of predicting co2 storage capacity in metal-organic frameworks within neural networks. *Iscience 27*, 5 (2024).

[22] KUBEČKA, J., AYOUBI, D., TANG, Z., KNATTRUP, Y., ENGSVANG, M., WU, H., AND ELM, J. Accurate modeling of the potential energy surface of atmospheric molecular clusters boosted by neural networks. *Environmental Science: Advances 3*, 10 (2024), 1438–1451.

[23] LI, S.-C., WU, H., MENON, A., SPIEKERMANN, K. A., LI, Y.-P., AND GREEN, W. H. When do quantum mechanical descriptors help graph neural networks to predict chemical properties? *Journal of the American Chemical Society 146*, 33 (2024), 23103–23120.

[24] LISTER, R., AND STONE, J. V. An empirical study of the time complexity of various error functions with conjugate gradient backpropagation. In *Proceedings of ICNN'95-International Conference on Neural Networks* (1995), vol. 1, IEEE, pp. 237–241.

[25] NEUMANN, M., GIN, J., RHODES, B., BENNETT, S., LI, Z., CHOUBISA, H., HUSSEY, A., AND GODWIN, J. Orb: A fast, scalable neural network potential. *arXiv preprint arXiv:2410.22570* (2024).

[26] PATTERSON, D., GONZALEZ, J., LE, Q., LIANG, C., MUNGUIA, L.-M., ROTHCHILD, D., SO, D., TEXIER, M., AND DEAN, J. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350* (2021).

[27] RAMAKRISHNAN, R., DRAL, P. O., RUPP, M., AND VON LILIENFELD, O. A. Quantum chemistry structures and properties of 134 kilo molecules. *Scientific data 1*, 1 (2014), 1–7.

[28] ROSS, S. M. *Introductory statistics*. Academic Press, 2017.

[29] SANCHEZ-GONZALEZ, A., GODWIN, J., PFAFF, T., YING, R., LESKOVEC, J., AND BATTAGLIA, P. Learning to simulate complex physics with graph networks. In *International conference on machine learning* (2020), PMLR, pp. 8459–8468.

[30] SCHAARSCHMIDT, M., RIVIERE, M., GANOSE, A. M., SPENCER, J. S., GAUNT, A. L., KIRKPATRICK, J., AXELROD, S., BATTAGLIA, P. W., AND GODWIN, J. Learned force fields are ready for ground state catalyst discovery. *arXiv preprint arXiv:2209.12466* (2022).

[31] SCHÜTT, K., UNKE, O., AND GASTEGGER, M. Equivariant message passing for the prediction of tensorial properties and molecular spectra. In *International Conference on Machine Learning* (2021), PMLR, pp. 9377–9388.

[32] SCHÜTT, K. T., SAUCEDA, H. E., KINDERMANS, P.-J., TKATCHENKO, A., AND MÜLLER, K.-R. Schnet–a deep learning architecture for molecules and materials. *The Journal of Chemical Physics 148*, 24 (2018).

[33] SPECKHARD, D., BECHTEL, T., GHIRINGHELLI, L. M., KUBAN, M., RIGAMONTI, S., AND DRAXL, C. How big is big data? *Faraday Discussions* (2025).

[34] SPECKHARD, D. T., MISIUNAS, K., PEREL, S., ZHU, T., CARLILE, S., AND SLANEY, M. Neural architecture search for energy-efficient always-on audio machine learning. *Neural Computing and Applications 35*, 16 (2023), 12133–12144.

[35] TANG, H., ORTIS, A., AND BATTIATO, S. The impact of padding on image classification by using pre-trained convolutional neural networks. In *Image Analysis and Processing–ICIAP 2019: 20th International Conference, Trento, Italy, September 9–13, 2019, Proceedings, Part II 20* (2019), Springer, pp. 337–344.

[36] XU, K., HU, W., LESKOVEC, J., AND JEGELKA, S. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826* (2018).

[37] ZHAO, H., LIU, F., LI, L., AND LUO, C. A novel softplus linear unit for deep convolutional neural networks. *Applied Intelligence 48* (2018), 1707–1720.

[38] ZOPH, B. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).

A Model descriptions

In this section we describe the three models used to evaluate the batching algorithms in greater detail. The models take in a graph, G, composed of nodes (or vertices) and edges $G(n, e)$. The nodes are represented by feature vectors h_n^t, where the subscript represents the specific node n, and the superscript t represents the number of times the vector has been updated.

The SchNet model (32) is a graph convolutional NN. The node feature vectors are updated with a node update equation (or convolution) that convolves the feature vector of the node, h_i^t, and feature vectors in neighborhood, N_i, of the node, i. It uses a convolutional kernel, $W(\vec{r_j} - \vec{r_i})$, which depends on the euclidean distance between nodes:

$$h_i^{t+1} = \sum_{j \in N_i} h_j^t \odot W^t(\vec{r_j} - \vec{r_i}). \tag{1}$$

We also look at a message passing NN model with edge updates (MPEU) (18; 2). Message passing NNs make use of a message function M_{ij}^t, which is defined for a pair of nodes, i and j, and their corresponding edge e_{ij}^t.

$$M_{ij}^t = f_m(h_i^t, h_j^t, e_{ij}^t) \tag{2}$$

Here, the messages are aggregated for each node with a permutation-invariant operator. The MPEU aggregates the messages with a sum,

$$m_i^{t+1} = \sum_{j \in N_i} M_{ij}^t. \tag{3}$$

In the MPEU, the edges between any two connected nodes, i and j, are represented by edge vectors e_{ij}^t. The message function is a function of the edge vector,

$$M_{ij}^t = f_m^t(h_j^t, e_{ij}^t) = (W_1^t h_j^t) \odot \sigma(W_3^t \sigma(W_2^t e_{ij}^t)). \tag{4}$$

where, the feature vectors, h_i, are updated in the MPEU as:

$$h_n^{t+1} = S_t(h_n^t, m_n^{t+1}) = h_n^t + W_5^t \sigma(W_4^t m_n^{t+1}). \tag{5}$$

$\odot$ denotes element-wise multiplication and σ represents the shifted soft plus function (37).

Finally, we evaluate the algorithms on the equivariant PaiNN model (31). This model differs from the MPEU and SchNet by learning not only scalar features, $s_i \in \mathbb{R}^{Fx1}$, for the representation of each atom but also equivariant features, $\vec{v_i} \in \mathbb{R}^{Fx3}$. The two representations are used to update each other. The scalar representation is updated with the following equation:

$$\Delta s_i^u = a_{ss}(s_i, \|V\vec{v}\|) + a_{sv}(s_i, \|V\vec{v}\|)\langle U\vec{v}, V\vec{v}\rangle, \tag{6}$$

where a_{ss} and a_{sv} represent neural networks with non-linear activation functions and V and U weight matrices that need to be learned. The update function for the equivariant representation is:

$$\Delta \vec{v}_i^u = a_{vv}(s_i, \|V\vec{v}\|)U\vec{v} \tag{7}$$

B Batch and pad algorithms

For completeness, we provide pseudocode for the batch algorithm mentioned in Section 3. The pseudocode provided is a simplified version of the method found in Jraph, although the TF-GNN, ptgnn, and pyTorch libraries all perform something similar. The basic principle to create a single disconnected graph that contains all of the information of the individual graphs in the batch. The batch method in Algorithm 2, works by creating six empty lists to store information about the batch. It creates a list to collect information about the individual graphs. The lists collect node/edge feature vectors, number of node/edges, and sender/receiver node indices. It then loops over the input graphs list, and concatenates the graph's node features, edge features, number of nodes/edges and sender/receiver indices into the respective batch lists. Finally, the method creates a new graph object where the constructor is fed the concatenated lists. Thereby, from a list of graphs, we have created one super-graph containing all of the information about the smaller graphs. The individual graphs can be rebuilt from the super-graph if desired. Note that the sender and receiver node indices need to be offset by a running counter of the number of nodes already added into the batch.

The static algorithm's padding method, pad-nearest-power-2, mentioned in Section 3 works by adding a fake dummy graph to the list of graphs. It adds a graph object with the number of nodes and edges being equal to the number of nodes/edges required to bring the sum of all the nodes/edges in the list of graphs to the nearest power of two. The algorithm is shown in pseudocode in Algorithm 3. When running the static-64 algorithm the padding is done to the nearest multiple of 64.

For the static-constant algorithm, the method first finds the graph with the largest number of nodes and the graph with the largest number of edges. These values are saved and when rounded to the nearest multiple of 64 serve as targets for padding. For the dynamic algorithm, the padding method is slightly more complex than Algorithm 3. The padding needs to add more dummy graphs if after adding a single dummy graph, the batch has less graphs than the batch size.

The static batching algorithm is shown in Algorithm 4.

The padding budget estimation is shown in the Algorithm 5.

C Timing experiments setup

The timing experiments made use of python's time library. Timing statements were executed before batching, before the gradient-update step and after the gradient-update step. Block until ready commands were executed to ensure operations on the GPU had finished before the timing measurements were taken. We also experimented with placing timing measurements before the training loop and after the training loop (i.e., after 2 million steps in some cases). We then averaged

Algorithm 2 Batch

```
Input: set of training graphs G
l_n ← [] {list of node features}
l_e ← [] {list of edge features}
l_s ← [] {list of receivers}
l_r ← [] {list of senders}
n_n ← [] {list of number of nodes}
n_e ← [] {list of number of edges}
o ← 0 {offset}
for g in G do
    l_n = concatenate(l_n, g.nodes_list)
    l_e = concatenate(l_e, g.edges_list)
    n_n = concatenate(n_n, g.num_nodes)
    n_e = concatenate(n_e, g.num_edges)
    for i = 0 to len(g.senders) do
        l_r.append(g.receivers[i] + o)
        l_s.append(g.senders[i] + o)
    end for
    o += g.num_nodes
end for
return Graph(l_n, l_e, n_n, n_e)
```

Algorithm 3 pad-nearest-power-2

```
Input: set of training graphs G, batch size B
s_n ← 0 {sum of nodes}
s_e ← 0 {sum of edges}
m_n ← 0 {missing nodes}
m_e ← 0 {missing edges}
for g in G do
    s_n += g[i].num_nodes
    s_e += g[i].num_edges
end for
t_n = next_power_2(s_n)
t_e = next_power_2(s_e)
m_n = t_n - s_n
m_e = t_e - s_e
G' = Graph(m_n, m_e)
return G.append(G')
```

Algorithm 4 Static batching

```
Input: batch size B_G, training graphs G, start point s
A ← [] {Accumulated list of graphs}
n_n ← 0 {Accumulated # of nodes}
n_e ← 0 {Accumulated # of edges}
for i = s to s + B_G do
    A ← A.append(G[i])
    n_n += G[i].num-nodes
    n_e += G[i].num-edges
    if len(A) == (B_G − 1) then
        return batch(pad-nearest-power-2(A, n_n, n_e))
    end if
end for
```

Algorithm 5 Estimate dynamic batching padding target

Input: subset size N, list of graphs G, batch size B_G
$n \leftarrow 0$ {number of nodes in N graphs}
$e \leftarrow 0$ {number of edges in N graphs}
$g \leftarrow G[1:N]$ {Grab N graphs from training data}
for $i = 1$ **to** $i = N$ **do**
  n += $G[i]$.num-nodes
  e += $G[i]$.num-edges
end for
$B_n \leftarrow$ next-multiple-64$(n * B_G/N)$ {padding target}
$B_e \leftarrow$ next-multiple-64$(e * B_G/N)$
return B_n, B_e, B_G

this time by the number of training steps executed. We found this number to be within a standard deviation of the sum of our batching and gradient-update step timing results. This points to the fact that the library runs batch creation and update-kernel execution consecutively. It is possible to run these steps synchronously but this optimization was not done in this study. More details on the implementation can be found in the code repository.

D Mean versus median in timing measurements

For each timing experiment ten experiments are run and either the mean or median is taken. The median timing results for the MPEU model after running one hundred thousand steps is seen in Fig. 6. The mean results for the same experiments are shown in Fig. 7. The difference in the two figures shows the importance of the number of recompilations in the gradient-update method. This is because the mean is affected by outlier measurements, such as recompilations, while the median is not. The fact that for one hundred thousand training steps, the static-64 algorithm has the best median performance but not the best mean performance is due to the number of recompilations which is highest for this algorithm.

The mean and median timing results for two million steps are shown for the MPEU model in Fig. 8 and in Fig. 9 respectively. We see the mean static-64 gradient-update step times are shifted higher than the median results, showing that the recompilations still affect the mean results. However, the rankings of the mean combined times are the same for the median combined times across algorithms, suggesting that for longer training time the effect of recompilations is no longer as significant as we saw earlier for one hundred thousand steps.

E CPU only timing results

The algorithm performances are different when running only on CPU. The batching (inclusive of padding) performance is the same as when running on a system that has both a GPU and a CPU which indicates that the batching is executed only on CPU. We experimented with trying to run part of the batching and padding code on GPU using JAX commands but found no speedup. The gradient-update step, however, is much slower on CPU. The mean timing results, from ten iterations of one hundred thousand training steps, are shown in Fig. 10 for the MPEU model. The median is shown in Fig. 11 for the same model. We ran one hundred thousand training steps instead of two million training steps since the computer cluster we used had a time limit of twelve hours for experiments. From the mean results, we see the dynamic batching algorithm is fastest. The median results, however, show that when the effect of recompilations are reduced, the static-64 algorithm is the fastest except for batch size 128 on the AFLOW dataset. This suggests that for longer training times the static-64 algorithm will be the fastest. The results for the SchNet model can be seen in the accompanying code repository.

F Multi-GPU behavior

We used JAX parallel map (pmap) function to run the batching algorithms on a multi-GPUs of four GPUs. In this case, we saw that the static-64 and static-2^n algorithms did not work out of the box

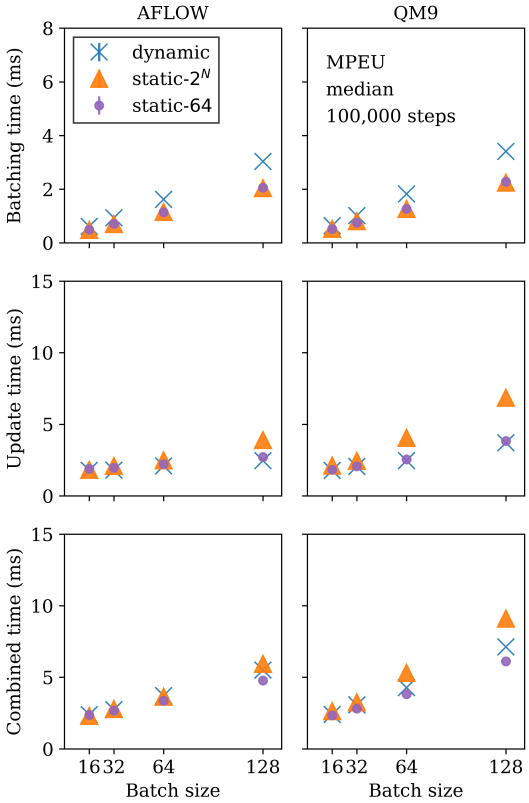

Figure 6: Median batching (top) and median gradient-update times (middle), and median combined time (bottom) for varying batch sizes on the AFLOW (left) and QM9 (right) data using the MPEU model. For each datapoint, ten iterations of one hundred thousand training steps are run.

with pmap since the algorithm expects the batches sent to each GPU to be the same size. For the AFLOW dataset, we saw roughly that the four batches had different shapes after padding when using the static-2^n algorithm roughly 90% of the time, and close to 99% when using the static-64 algorithm. It's possible to alter the static-64 and static-2^N to work by padding, with another round of padding, each batch to the largest of the four padded batches. The results, will be highly implementation dependent, more-so than in the single-GPU case. To simplify the comparison of the algorithms, we evaluate the algorithms only on single-GPU setups and note that the dynamic algorithm is easily ported to a multi-GPU setup using JAX's pmap function.

G Test metric results

The test metric (RMSE) curves for each of the ten iterations are averaged for each batch size, model and dataset combination. The resulting mean curve is shown for the MPEU and SchNet models using the QM9 data in Fig. 12. We do not see significant differences in the test performance for the static-2^N or dynamic batching algorithm. Note that we do not expect nor do we see any noticeable difference in learning from the static-64 to static-2^N algorithms since the difference between the methods is the padding scheme which does not affect the loss. Therefore, the static-64 curve is left out of Fig. 12 for visual clarity. The results for AFLOW can be seen in the accompanying code repository.

H Static-constant batching

The TF-GNN library tutorial suggests running the static batching algorithm with a constant padding target. As described in Section 3, the algorithm first iterates through all the graphs in the dataset and saves the maximum number of nodes/edges seen in a graph and uses this as a padding target. We

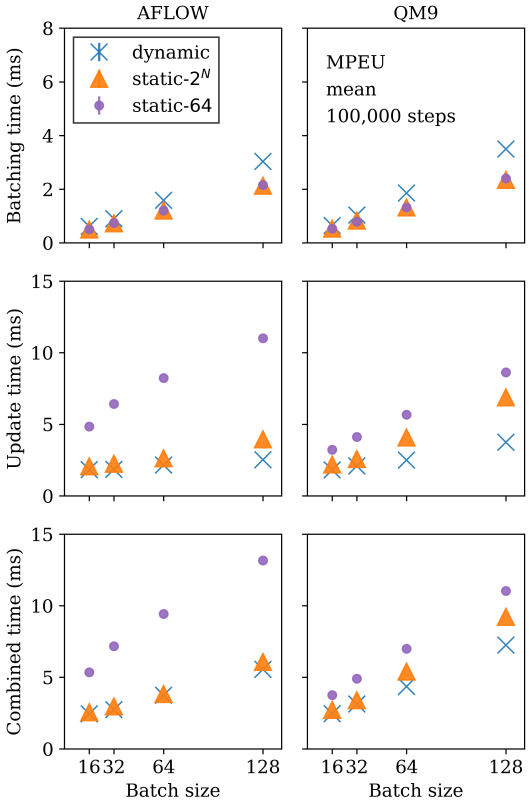

Figure 7: Mean batching (top) and mean gradient-update times (middle), and mean combined time (bottom) for varying batch sizes on the AFLOW (left) and QM9 (right) data using the MPEU model. For each datapoint, ten iterations of one hundred training steps are run.

evaluated this method on a subset of batch sizes, dataset and model combinations, and compared it to the static-2^N, static-64 and dynamic algorithms. The results are shown in Table 1. Our results show that the static-constant algorithm performs poorly in comparison to the other algorithms, and as a result it is not used in the main text.

I Effect on rankings of the number of training steps

The effect of the number of training steps run on the rankings of the fastest algorithms is depicted in Fig. 13. We can clearly see that for a smaller number of training steps, the dynamic batching algorithm is faster than the static-64 algorithm which is due to the number of recompilations that happen during the gradient-update step. For more training steps, this effect is subdued and the static-64 algorithm is faster.

J Student t-tests on test metrics

The student t-test was performed in a pairwise fashion between two batching algorithms upon their distribution of final test RMSE values. For the SchNet model on the QM9 dataset for batch size 64, a heatmap of the t-test values is shown in Fig. 14 and the associated p-values in Fig. 15.

Figures for other datasets and models can be found in the Software and Data section.

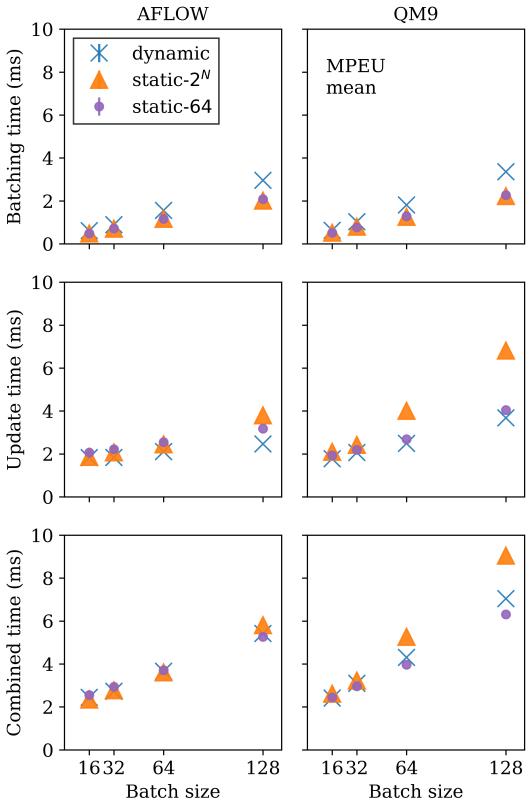

Figure 8: Mean batching time (upper row), mean gradient-update time (middle row), and the mean combined time (bottom row) for varying batch sizes on the AFLOW (left) and QM9 (right) data using the MPEU model. For each datapoint, ten iterations of two million training steps are run.

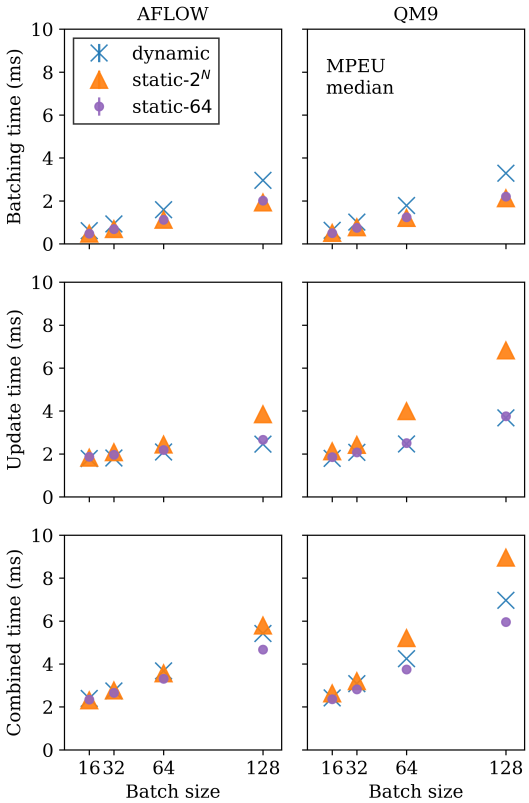

Figure 9: Median batching time (upper row), median gradient-update time (middle row), and the median combined time (bottom row) for varying batch sizes on the AFLOW (left) and QM9 (right) data using the MPEU model. For each datapoint, ten iterations of two million training steps are run.

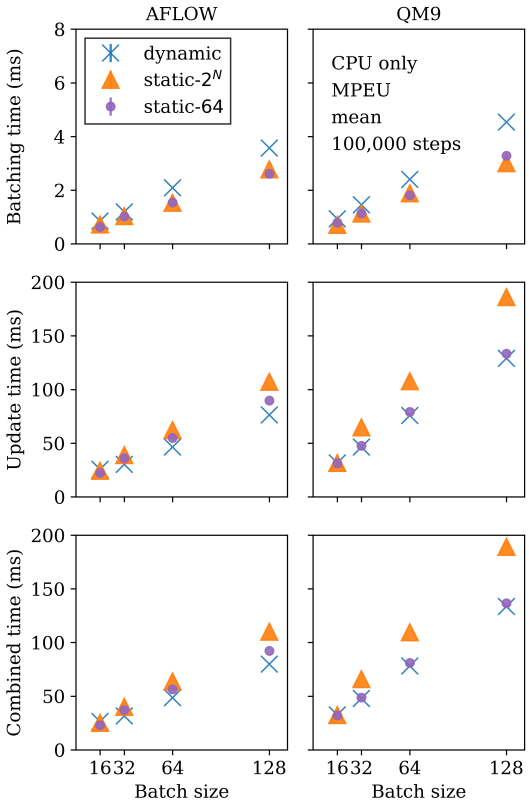

Figure 10: Mean batching time (upper row), mean gradient-update time (middle row), and the mean combined time (bottom row) on CPU for varying batch sizes on the AFLOW (left) and QM9 (right) data using the MPEU model. The experiments were run for one hundred thousand training steps.

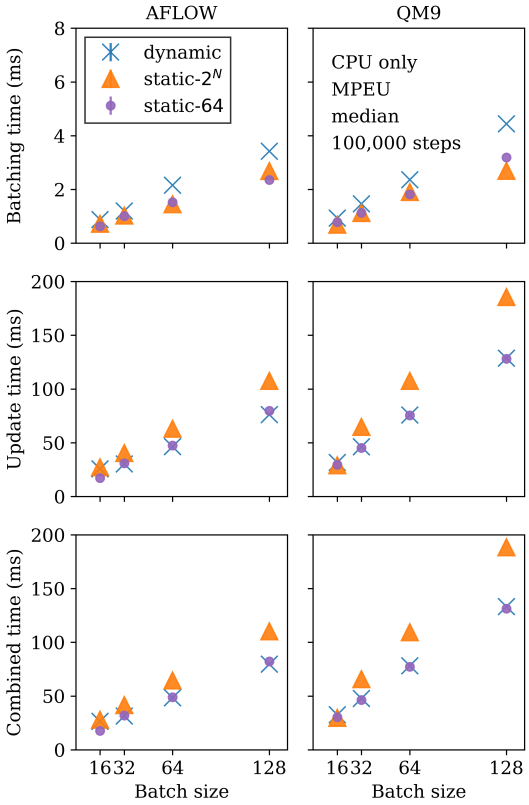

Figure 11: Median batching time (upper row), median gradient-update time (middle row), and the median combined time (bottom row) on CPU for varying batch sizes on the AFLOW (left) and QM9 (right) data using the MPEU model. The experiments were run for one hundred thousand training steps.

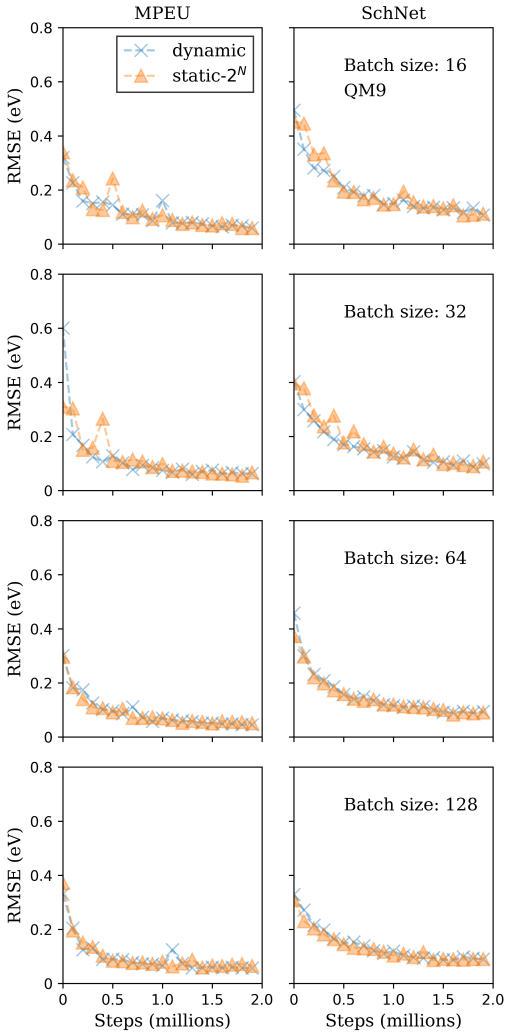

Figure 12: Mean test RMSE curves for the MPEU model (left) and SchNet (right) on QM9 test for batch sizes of 16, 32, 64 and 128 (from top to bottom).

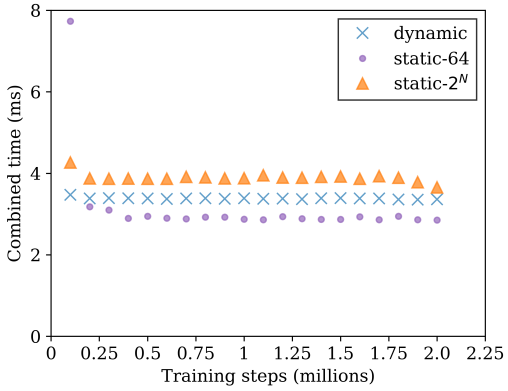

Figure 13: Running average combined time (gradient-update step and batching) as a function of the total number of training steps run. Here only a single iteration is run for the batch size 32, MPEU model and QM9 dataset for both the dynamic, static-64 and static-2^N algorithms.

Table 1: Static-constant timing results compared with the batching algorithms.

Algorithm	Dataset	Model	Batch size	Batch time (ms)	Update time (ms)	Combined time (ms)
static-constant	AFLOW	MPEU	16	0.63	6.34	7.30
static-2^N	AFLOW	MPEU	16	0.49	1.89	2.38
static-64	AFLOW	MPEU	16	0.49	1.89	2.38
dynamic	AFLOW	MPEU	16	0.60	2.41	3.01
static-constant	AFLOW	SchNet	32	0.99	6.34	7.33
static-2^N	AFLOW	SchNet	32	0.71	1.38	2.01
static-64	AFLOW	SchNet	32	0.71	1.64	2.35
dynamic	AFLOW	SchNet	32	0.87	1.30	2.17
static-constant	AFLOW	PaiNN	16	0.83	46.56	47.39
static-2^N	AFLOW	PaiNN	16	0.45	3.82	4.27
static-64	AFLOW	PaiNN	16	0.46	3.88	4.34
dynamic	AFLOW	PaiNN	16	0.59	3.20	3.79
static-constant	QM9	SchNet	32	0.78	2.02	2.80
static-2^N	QM9	SchNet	32	0.74	1.55	2.29
static-64	QM9	SchNet	32	0.77	1.38	2.36
dynamic	QM9	SchNet	32	1.03	1.38	2.41
static-constant	QM9	SchNet	128	2.31	5.19	7.50
static-2^N	QM9	SchNet	128	2.20	3.88	6.08
static-64	QM9	SchNet	128	2.24	2.39	4.63
dynamic	QM9	SchNet	128	3.38	2.13	5.51
static-constant	QM9	PaiNN	16	0.94	11.67	12.61
static-2^N	QM9	PaiNN	16	0.71	6.43	7.14
static-64	QM9	PaiNN	16	0.50	3.30	3.80
dynamic	QM9	PaiNN	16	0.68	3.33	4.01

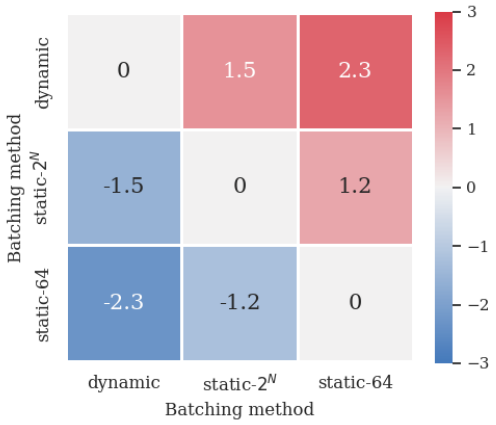

Figure 14: Heatmap of pairwise student t-test values on the distribution of test RMSE values for the SchNet model, QM9 data and a batch size of 64.

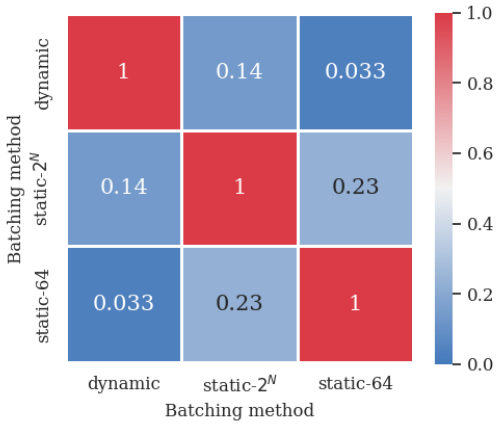

Figure 15: Heatmap of p-values from the pairwise student t-test on the distribution of test RMSE values for the SchNet model, QM9 data and a batch size of 64.