ED-Batch: Efficient Automatic Batching of Dynamic Neural Networks via Learned Finite State Machines

Siyuan Chen [1] **Pratik Fegade** [2] **Tianqi Chen** [2 3] **Phillip B. Gibbons** [2] **Todd C. Mowry** [2]

Abstract

Batching has a fundamental influence on the efficiency of deep neural network (DNN) execution. However, for dynamic DNNs, efficient batching is particularly challenging as the dataflow graph varies per input instance. As a result, state-of-the-art frameworks use heuristics that result in suboptimal batching decisions. Further, batching puts strict restrictions on memory adjacency and can lead to high data movement costs. In this paper, we provide an approach for batching dynamic DNNs based on finite state machines, which enables the automatic discovery of batching policies specialized for each DNN via reinforcement learning. Moreover, we find that memory planning that is aware of the batching policy can save significant data movement overheads, which is automated by a PQ tree-based algorithm we introduce. Experimental results show that our framework speeds up state-of-the-art frameworks by on average 1.15x, 1.39x, and 2.45x for chain-based, tree-based, and lattice-based DNNs across CPU and GPU. The framework is open-sourced at `https://github.com/gulang2019/ED-Batch`.

1. Introduction

Batching accelerates the training and inference for deep neural networks (DNN) because (i) it launches fewer kernels resulting in lower kernel launch and scheduling overhead on the CPU, and (ii) it better utilizes the hardware by exploiting more parallelism. For static DNNs, i.e. DNNs whose dataflow graphs (a.k.a., computation graphs) are identical across every input instance, batched execution is trivial as one can batch corresponding operations for each input together. However, DNNs used to model structured data such as trees (Tai et al., 2015), grids (Chen et al., 2015), and lattices (Zhang & Yang, 2018) in applications like natural language processing and speech recognition, exhibit dynamism in the network structure. In other words, the dataflow graph for these DNNs varies for each input instance. As a result, batching is a non-trivial problem for these DNNs.

Due to the presence of dynamism, batching for dynamic DNNs cannot be done during compilation. As a result, past works on the efficient execution of dynamic DNNs focused on two directions: (i) enabling operation-level batching at runtime (Looks et al., 2017; Neubig et al., 2017a; Zha et al., 2019), i.e. *dynamic batching*, and (ii) extracting static subgraphs (Xu et al., 2018) (e.g., LSTM cells) from the dataflow graph and optimizing them during compilation (Fegade et al., 2021; Fegade, 2023). Because of strict runtime constraints, the former approach relies on simple heuristics to guide batching, leading to suboptimal performance. In the latter approach, techniques dedicated to certain control flow patterns or subgraphs are used for optimization, which is difficult to automate and requires developers with strong expertise in optimizing new DNN models.

Furthermore, due to the dynamic and runtime nature of past techniques, past work is unable to optimize inter-tensor memory layouts during compilation. Past solutions, thus, either (i) emit gather/scatter operations before and after each batch (Xu et al., 2018; Neubig et al., 2017a) or (ii) rely on specially designed and/or hand-optimized kernels to operate on scattered data in-place (Fegade et al., 2021; Fegade, 2023), thus precluding the use of highly-optimized vendor libraries on common hardware.

To address these problems, we propose ED-Batch (Efficient Dynamic Batching), an efficient automatic batching framework for dynamic neural networks via learned finite state machines (FSM) and batching-aware memory planning.

For dynamic batching, we exploit the insight that the optimal batching policy for a wide variety of dynamic DNNs can be represented by an FSM, where each state represents a set of possible operator types on the frontier of the dataflow graph. Unlike the previous algorithms that depend heavily on aggregated graph statistics to guide batching and result in highly suboptimal decisions, our FSM approach learns which decisions are better by examining the entire graph. We find that using FSMs represents a sweet-spot between

[1]School of EECS, Peking University [2]Carnegie Mellon University [3]OctoML. Correspondence to: Siyuan Chen <siyuanc3@andrew.cmu.edu>.

Proceedings of the 40th International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023. Copyright 2023 by the author(s).

expressiveness of batching choices (the same choice for the same state, leveraging the regularity in network topology for a given input) and efficiency. Further, we adopt a reinforcement-learning (RL) algorithm to learn the FSM from scratch. To guide the training of RL, we design a reward function inspired by a sufficient condition for the optimal batching policy.

For the static subgraphs of the dynamic DNN, we take a general approach to optimize it by memory-efficient batching. Our key insight is that the memory operations can be significantly minimized by better planning the inter-tensor memory layouts after batching, which we perform by using a novel PQ tree-based (Booth & Lueker, 1976) algorithm that we have designed.

In summary, this paper makes the following contributions:

- We propose an FSM-based batching algorithm to batch dynamic DNNs that finds a near-optimal batching policy.
- We design a PQ tree-based algorithm with almost linear complexity to reduce memory copies introduced by dynamic batching.
- We compare the performance of ED-Batch with state-of-the-art dynamic DNNs frameworks on eight workloads and achieve on average 1.15x, 1.39x, and 2.45x speedups for chain-based, tree-based, and lattice-based networks across CPU and GPU. Our framework is open-sourced at https://github.com/gulang2019/ED-Batch.

2. FSM-based Dynamic Batching Algorithm

In this section, we identify the shortcomings of current batching techniques and then propose a new FSM-based batching algorithm and an RL approach to learn the FSM.

2.1. Problem Characterization

Dynamic batching was initially proposed in TensorFlow Fold (Looks et al., 2017) and DyNet (Neubig et al., 2017b) to enable batched execution of operations for dynamic DNNs. Specifically, given a mini-batch of input instances, dataflow graphs are generated for each of the input instances in the mini-batch and each operation is given a type (indicating operation class, tensor shape, etc.). Upon execution, the runtime identifies batching opportunities within the dataflow graphs by executing operations of the same type together. To avoid severe runtime overhead, the batching algorithm cannot have high complexity. However, the problem of minimizing the number of launched (batched) kernels is an NP-hard problem with no constant approximation algorithm,[1] making the batching problem extremely challenging.

[1] Proved by reducing from the *shortest common supersequence* problem (Räihä & Ukkonen, 1981) in Appendix A.1.

Algorithm 1 FSM-based Dynamic Batching

1: **Input:** Dataflow Graph G, State Encoding Function F, Batching Policy π
2: **while** G.notEmpty() **do**
3: nextType = $\pi(F(G))$
4: batch = [v for v in Frontier(G) if v.type is nextType]
5: Execute batch and remove nodes from G
6: **end while**

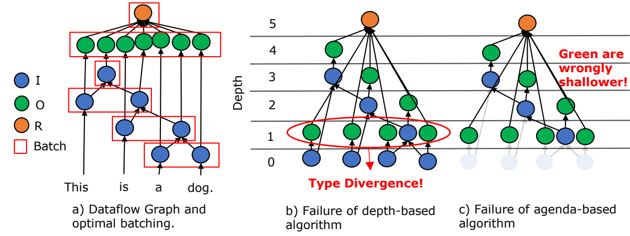

Figure 1. Example demonstrating suboptimal batching using current dynamic batching algorithms.

As a result, the heuristics used for dynamic batching in current frameworks often find a suboptimal policy (up to 3.27x more batches than with our approach (§5.3)).

Specifically, previous state-of-the-art algorithms use heuristics depending on aggregated graph statistics to guide batching. The *depth-based algorithm* in Tensorflow Fold (Looks et al., 2017) batches operations with the same type at the same topological depth (the input operation to the network has depth 0). And the *agenda-based algorithm* in DyNet (Neubig et al., 2017b) executes operations of the type with minimal average topological depth iteratively. However, topological depth cannot always capture the regularity of the dataflow graph and results in sub-optimal batching choices. Fig. 1(a) shows a dataflow graph of the tree-based network, which builds upon the parse tree of a sentence with three types of operations: internal nodes (I), output nodes (O), and reduction nodes (R). The ideal batching policy (red boxes) executes 6 batches, taking all O nodes in one batch. However, the depth-based algorithm (Fig. 1(b)) executes the O nodes in four batches because they have different topological depths, resulting in a total of 9 batches. For the agenda-based algorithm, when it is deciding the next batch after batching the I nodes as its first batch (Fig. 1(c)), because the O nodes have a lower average depth (average depth $= (1+1+1+1+2+3+4)/7 = 1.86$) than the I nodes (average depth $= (1+2+3)/3 = 2$), the algorithm will pick the O nodes for the next batch. As the result, the agenda-based algorithm executes O nodes in both its 2nd and 6th batches, resulting in 7 batches total.

2.2. FSM-based Dynamic Batching

To fully overcome the limitation of specific graph statistics, we found that an FSM-based approach (i) offers the oppor-

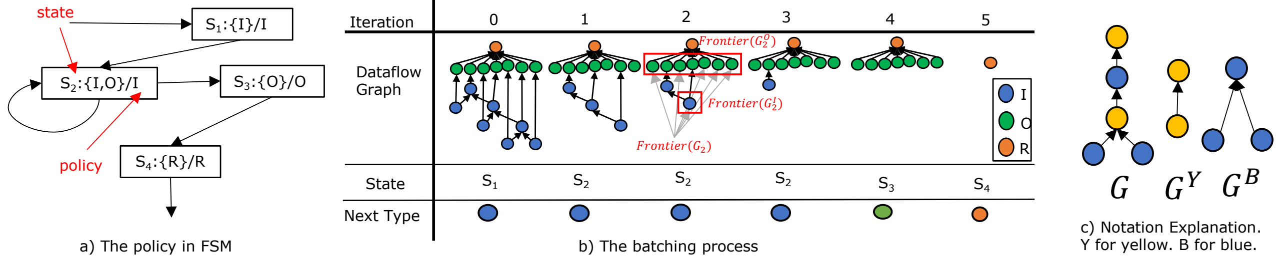

Figure 2. Dynamic Batching Policy by FSM

tunity to specialize for network structure under a rich design space of potential batching policies and (ii) can generalize to any number of input instances, as long as they share the same regularity in topology.

Shown in Algorithm 1, the FSM-based dynamic batching approach is an iterative process of choosing the operation type for the next batch. The process differs from the agenda-based algorithm only in how it computes the next type for batching (line 3). During each iteration, the next type is decided by first encoding the current dataflow graph G into a state $S = F(G)$, and then using a policy π to map the state into an operation type $t = \pi(S)$. Then, the operations of type t on the frontier of the G form the next batch. After they are executed, these operations are removed from G and the next iteration begins.

For the model in Fig. 1(a), an optimal FSM-based batching policy is shown in Fig. 2(a), where we encode the dataflow graph by the set of types on the frontier. State $S_2 : \{I, O\}/I$, for example, encodes graphs where types I and O comprise the frontier, and the policy π is to use I as the next type. Fig. 2(b) shows the batching process. From iterations 1 to 3, the dataflow graph is encoded into $S_2 = \{I, O\}$, thus the policy continues to batch nodes of type $I = \pi(S_2)$, avoiding batching the O nodes as past heuristics would do. At the same time, it is not hard to see that this FSM-based batching policy can be applied to batch multiple input instances of different parse trees.

2.3. Using RL to Learn the FSM

As the FSM provides us with the design space for potential batching policies, we need an algorithm to come up with the best batching policy specialized for a given network structure. In ED-Batch, we adopt an RL-based approach for the design-space-exploration and learn the best FSM by a reward function inspired by a sufficient condition for optimal batching.

In RL, an agent learns to maximize the accumulated reward by exploring the environment. At time t, the environment is encoded into a state S_t, and the agent takes action $a_t = \pi(S_t)$ following the policy π and receives a reward $r_t = r(S_t, a_t)$. After this, the environment transforms to the next state S_{t+1}. This results in a sequence of states, actions, and rewards: $(S_0, a_0, r_0, S_1, a_1, r_1, ..., S_{N-1}, a_{N-1}, r_{N-1}, S_N)$, where N is the number of time steps and S_N is an end state. The agent aims to maximize the accumulated reward $\Sigma_t r_t$ by updating the policy π. For FSM-based dynamic batching, the environment is the dataflow graph, which is encoded into states by the encoding function F. For every iteration, the agent decides on the type for the next batch, receives a reward on that decision, and the environment gets updated according to Algorithm 1. Now we elaborate on the state encoding, reward design, and training respectively.

We use the following notations. For a dataflow graph G, G_t refers to its status at step t, G^a refers to the extracted subgraph of G composed solely of type a operations (as illustrated in Fig. 2(c)), $Frontier(G)$ refers to the set of ready-to-execute operations, and $Frontier_a(G)$ refers to the subset of $Frontier(G)$ with type a. Fig. 2(b) annotates $Frontier(G_2)$, $Frontier(G_2^I)$ and $Frontier(G_2^O)$.

State Encoding: The design of state encoding should carry enough information to capture the network's regularity and yet be as simple as possible to avoid heavy runtime overhead. In practice, we experimented with three ways of encoding:

- $E_{base}(G) = \{v.type | v \in Frontier(G)\}$ is the set of operation types on the frontier
- $E_{max}(G) = (E_{base}(G), argmax_t |Frontier_t(G)|)$ is $E_{base}(G)$ plus the most common type on the frontier
- $E_{sort}(G) = sort(\{v.type | v \in Frontier(G)\}, t : |Frontier_t(G)|)$ is $E_{base}(G)$ sorted by the number of occurrences on the frontier

Empirically, we found that E_{sort} was the best among the three (§5.3).

Reward: We design the reward to minimize the number of batches. The reward function is defined as

$$r(S_t, a_t) = -1 + \alpha * \frac{|Frontier_{a_t}(G_t)|}{|Frontier(G_t^{a_t})|} \quad (1)$$

where α is a positive hyper parameter and $S_t = F(G_t)$. The constant -1 in the reward penalizes every additional batch, thereby helping us minimize the number of batches.

The second term is inspired by a sufficient condition for op-

timal batching (Lemma 2.1 below, proof in Appendix A.2) to prioritize the type such that all operations on the frontier of the subgraph of this type are ready to execute. For the tree-based network, this term prioritizes the batching choice made by the optimal batching policy in Fig. 2(a). For example, at iteration 2, this term is $\frac{5}{7}$ and $\frac{1}{1}$ for the O and I node respectively and the I node is given higher priority for batching. For other networks, like the chained-based networks (Fig. 7), this sufficient condition continues to hold.

Lemma 2.1 (Sufficient Condition for Optimal Batching)**.** *If* $\frac{|Frontier_{a_t}(G_t)|}{|Frontier(G_t^{a_t})|} = 1$*, then there exists a shortest batching sequence starting with* a_t*.*

Training: Training is performed for each new network topology (e.g., chain-based network, tree-based network) at compile time. We adopt the tabular-based Q-learning (Watkins & Dayan, 1992) algorithm to learn the policy. An N-step bootstrapping mechanism is used to increase the current batching choice's influence on longer previous steps. Specifically, the algorithm learns a Q function, which maps each state and action pair to a real number indicating its score.

The key observation for training is that the policy (i.e. FSM) is independent of the input instance and the batch size, and is only dependent on the DNN's type. This observation enables us to train on a single batch and generalize to larger batch sizes of different input instances. During training, the RL agent repeatedly performs batching on a single dataflow graph of a batch (size of 32) of randomly sampled inputs. Empirically, this takes hundreds of trials to converge (§5.3).

During inference, the Q table for the network is loaded by the runtime. At each state S, the runtime selects the operation type with the highest Q value for the next batch, i.e. $\pi(S) = argmax_a Q(S, a)$. This step is done by a lookup into stored Q functions in constant time.

3. Memory-efficient Batching for Static Subgraphs

3.1. Background and Motivation

In order to invoke a batched tensor operator in a vendor library, the source and result operands in each batch are usually required to be contiguous in memory (as per the vendor library specifications). Current batching frameworks such as Cavs and DyNet ensure this by performing explicit memory gather and/or scatter operations, leading to high data movement. On the other hand, Cortex (Fegade et al., 2021) relies on specialized, hand-optimized batched kernels instead of relying on vendor libraries. This approach, however, is unable to reuse the highly performant optimizations available as part of vendor libraries.

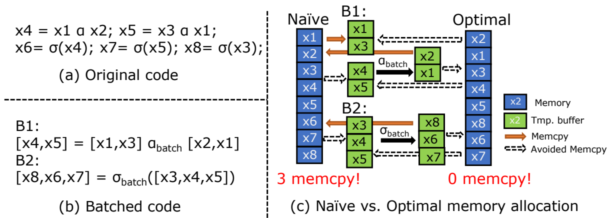

Figure 3. Memory allocation example. α, σ represent operators.

In ED-Batch, we take a different approach to fit the memory layout into the batching policy, where operations in the source and result operands for batched execution are already contiguous in memory.

We illustrate the approach by an example. Fig. 3(a) shows a sample code for a static subgraph and Fig. 3(b) shows its batched version. In Fig. 3(c), we compare two memory layouts. On the left, we directly allocate memory according to the variable's label, then two memory gather for $[x_1, x_3], [x_2, x_1]$ and one scatter for $[x_8, x_6, x_7]$ is performed because they are either not contiguous or aligned in memory. We say an operand of a batch is aligned in memory if the order of its operations matches with the one in memory. Now, consider the memory allocation on the right, which allocates memory following $(x_2, x_1, x_3, x_4, x_5, x_8, x_6, x_7)$. Then, every source and result operand of the batched execution is already contiguous and aligned in memory, saving us from extra memory copies.

3.2. PQ tree-based memory allocation

To find the ideal layout, we designed an almost linear complexity memory allocation algorithm based on the PQ tree (Booth & Lueker, 1976), which is a tree-based structure used to solve the consecutive one property (Meidanis et al., 1998) and is previously applied to DNA-related analysis (Landau et al., 2005) in biology research.

We define the *ideal memory layout* as a sequence of variables satisfying two constraints:

- **Adjacency Constraint:** Result and source operands in every batch should be adjacent in the sequence. E.g., $\{x_4, x_5\}, \{x_1, x_3\}, \{x_2, x_1\}$ for $B1$, $\{x_8, x_6, x_7\}, \{x_3, x_4, x_5\}$ for $B2$ are adjacent in the sequence.
- **Alignment Constraint:** The order of the result and source operands should be aligned in a batch. E.g. for $B1$, $x_4 \prec x_5 \iff x_1 \prec x_3 \iff x_2 \prec x_1$ in the sequence.

The adjacency constraint is satisfied by the PQ tree algorithm. Given several subsets of a set S, the PQ tree algorithm returns a data structure in linear time called a PQ tree, representing potential permutations of S such that elements in each subset are consecutive. Fig. 4(a) shows the

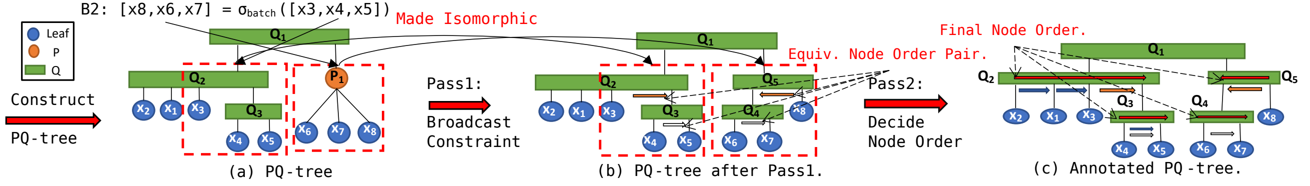

Figure 4. Example for PQ tree-based algorithm. x_1-x_8 are variables.

Algorithm 2 PQ tree Memory Allocation

1: **function** BROADCASTCONSTRAINT($tree$, $\mathcal{B}$)
2: visited = getSet()
3: **for** $batch$ in $\mathcal{B}$ **do**
4: **if** batch in visited **then**
5: continue
6: **end if**
7: Q = Queue()
8: Q.push($batch$)
9: **while** not Q.isEmpty() **do**
10: b = Q.pop();
11: visited.insert(b)
12: $cons$ = ParseConstraints(b)
13: $suc, updatedBatches$ = ApplyConstraints($cons$, $tree$)
14: **if** suc is False **then**
15: $\mathcal{B}$.erase(b)
16: **else**
17: **for** b in $updatedBatches$ **do**
18: Q.push(b)
19: **end for**
20: **end if**
21: **end while**
22: **end for**
23: **end function**
24: **function** DECIDENODESORDER($tree$, $\mathcal{B}$)
25: $POrder$ = getUnionFindSet(tree.PNodes) {A union-find set to decide QNode's direction.}
26: $QOrder$ = getUnionFindSet(tree.QNodes) {A union-find set to decide PNode's permutation.}
27: **for** $batch$ in $\mathcal{B}$ **do**
28: $EquivPairs$ = ParseEquivNodeOrderPair($tree$, $batch$)
29: **for** $EquivPair$ in $EquivPairs$ **do**
30: **if** $EquivPair$ is a P-node pair **then**
31: POrder.Union($EquivPair$)
32: **else if** $EquivPair$ is a Q-node pair **then**
33: QOrder.Union($EquivPair$)
34: **end if**
35: **end for**
36: **end for**
37: **return** $QOrder$, $POrder$
38: **end function**
39: **function** MAIN(X, $\mathcal{B} = (batch_1, ..., batch_n)$)
40: {X the variable set, $\mathcal{B}$ the batches }
41: $tree$ = ConstructPQTree(X, $\mathcal{B}$)
42: BroadcastConstraint($tree$, $\mathcal{B}$)
43: $QOrder$, $POrder$ = DecideNodesOrder($tree$, $\mathcal{B}$)
44: **return** GetLeafOrder($tree$, $QOrder$, $POrder$)
45: **end function**

PQ tree for the example code. The tree has three kinds of nodes: leaf node, P-node, and Q-node. Leaf nodes represent the variables; P-nodes have more than two children, whose appearance in the sequence is contiguous but could be permuted; Q-nodes have more than one child, whose appearance in the sequence follows an order but could be reversed. A depth-first traversal of the leaf nodes gives the sequence. For example, there is one P-node and three Q-nodes in Fig. 4(a). Q_2 indicates the order should only be (x_2, x_1, x_3, Q_3) or (Q_3, x_3, x_1, x_2), while P_2 indicates that one permutation of $\{x_6, x_7, x_8\}$ appears in the sequence. The adjacency of $\{x_4, x_5\}$ is embedded in Q_3, $\{x_1, x_3\}, \{x_2, x_1\}, \{x_4, x_3, x_5\}$ in Q_2, and $\{x_6, x_7, x_8\}$ in P_1. A possible sequence is $(x_2, x_1, x_3, x_4, x_5, x_6, x_7, x_8)$.

To satisfy the alignment constraint, we annotate each node on the PQ tree with an order. An annotated PQ tree is shown in Fig. 4(c), where a direction mark is attached to every Q-node, indicating its traversal order. As a result, any leaf node sequence of legal traversal on this annotated PQ tree indicates a memory allocation order satisfying both constraints.

Shown in Algorithm 2, two passes obtain the order annotation to the PQ tree. The first pass, BROADCASTCONSTRIANT, makes the tree structure of each batch's operands isomorphic. For $B2$'s operands, $\{x_3, x_4, x_5\}$'s tree structure is $Q_2 = (..., x_3, Q_3 = (x_4, x_5))$, and $\{x_6, x_7, x_8\}$'s tree structure is $P_1 = (x_6, x_7, x_8)$. After the pass, they have isomorphic tree structures ($Q_2 = (..., x_3, Q_3 = (x_4, x_5))$ and $Q_5 = (Q_4 = (x_6, x_7), x_8)$). The second pass, DECIDENODESORDER, derives the equivalent class of node-order pairs and searches for an annotation for the direction of Q-nodes and the permutation of P-nodes that is compatible with the equivalence relationship.

We walk through the algorithm by the example in Fig. 4. At first, the PQ tree is constructed by the standard algorithm to satisfy the adjacency constraint (Fig. 4(a)). After that, the BROADCASTCONSTRAINT pass makes the tree-structure of operands in a batch isomorphic by repeatedly parsing adjacency constraints (line 12) and broadcasting them across operands (line 13). For $B2$, the parsed adjacency constraints are $\{x_3, x_4, x_5\}$, $\{x_4, x_5\}$ for subtree $Q_2 = (..., x_3, Q_3 = (x_4, x_5))$, and $\{x_6, x_7, x_8\}$ for subtree $P_1 = (x_6, x_7, x_8)$. After that, $\{x_4, x_5\}$ in the source operand is broadcast into $\{x_6, x_7\}$ for the result operand, and $\{x_6, x_7\}$ is applied to the PQ tree as an adjacency

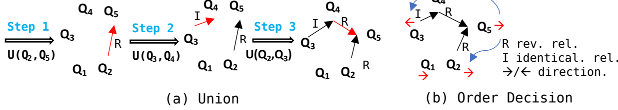

Figure 5. Illustration for order decision in Pass2.

constraint[2], resulting in the replacement of P_1 node by $Q_5 = (Q_4 = (x_6, x_7), x_8)$. Now tree structures for $B2$'s operands are isomorphic, and the algorithm applies this process to other batches in a breadth-first search until no update on the tree structure happens.

In DECIDENODESORDER (line 24), we assign directions for the Q-node and permutations for the P-node. We start by parsing the equivalence relationship (line 28) among $\langle QNode, direction \rangle$ pairs or the $\langle PNode, permutation \rangle$ pairs from the isomorphic tree structures after the first pass, e.g. $\langle Q_2, \leftarrow \rangle \iff \langle Q_3, \leftarrow \rangle$ for $B1$, $\langle Q_3, \leftarrow \rangle \iff \langle Q_4, \leftarrow \rangle$ and $\langle Q_2, \leftarrow \rangle \iff \langle Q_5, \rightarrow \rangle$ for $B2$. After that, we spread the equivalence relationship across batches with the support of a *union–find data structure*. In the algorithm, a graph carrying the equivalence relationship is constructed by iteratively UNIONing equivalent relationships among the node-order pairs (lines 29-35). The nodes are composed of all P/Q-nodes in the tree, and a directed edge $\langle n_s, n_t, f \rangle$ between nodes n_s and n_t with transformation function f indicates that n_s's order after transformation f should be the same as n_t's order.

Fig. 5 shows a graph construction process for the example in Fig. 4. When processing $\langle Q_2, \leftarrow \rangle \iff \langle Q_5, \rightarrow \rangle$ in step 1, a $\langle Q_2, Q_5, R \rangle$ edge is added to the graph, indicating Q_2's direction is determined by the reverse of Q_5's direction. When processing the $\langle Q_2, \leftarrow \rangle \iff \langle Q_3, \leftarrow \rangle$ in step 3, we first find the decider of their order, i.e. Q_5 for Q_2 and Q_4 for Q_3, and add a $\langle Q_4, Q_5, R \rangle$ edge. In this way, Q_2 and Q_3 always have the same order. Finally, as illustrated in Fig. 5(b), the deciders in the graph (Q_5 and Q_1 for the example) are assigned with arbitrary directions, which spread across the graph following the relationship on the edge.

The PQ tree memory allocation algorithm's time complexity is given in Lemma 3.1, showing that under certain constraints, the PQ tree algorithm scales linearly with the size of the dataflow graph and the batch size. Empirically, the algorithm runs in tens of milliseconds for common subgraphs (§5.3).

Lemma 3.1. *PQ tree memory allocation algorithm's time complexity is* $O(\Sigma_{b \in batches}|b| \max^2_{b \in batches}|b|)$*, where* $|.|$ *counts the operations in a batch.*

Currently, PQ tree memory optimization is applied to every static subgraph at compile time because its execution time does not fit into the strict runtime constraint for dynamic DNNs. But the algorithm is applicable to arbitrary dataflow graphs, as well as the idea of better memory planning for any batching problems.

Lastly, there may be conflicts across different sub-graphs' memory layout optimization. When conflict happens between a producer subgraph and a consumer subgraph, memory copy kernels are required to arrange the output of the producer into the layout required by the consumer. We expect the memory overhead caused by the conflicts will not be severe since the I/O ops involved in the conflict is a small portion of the subgraph.

Appendix B provides a detailed explanation of the algorithm.

4. Implementation

The optimizations in ED-Batch are fully automated and implemented as a runtime extension to DyNet in 5k lines of C++ code. The user can optionally enable the batching optimization by passing a flag when launching the application, and enable the static subgraph optimization by defining subgraphs in DyNet's language with a few annotations. Before execution, the RL algorithm learns the batching policy and ED-Batch optimizes the static subgraph by the approach in §3. For the RL algorithm, a tuned and fixed set of hyper parameters is used for training all types of networks. Upon execution, ED-Batch calls DyNet's executor for batched execution, which is supported by vendor libraries.

5. Evaluation

5.1. Experiment Setup

We evaluate our framework against DyNet (Neubig et al., 2017a) and Cavs (Xu et al., 2018), two state-of-the-art runtimes for dynamic DNNs, which are shown to be an order of magnitude faster than traditional frameworks like Pytorch (Paszke et al., 2019) and TensorFlow (Abadi et al., 2016) (see Appendix D). As noted in Fegade, Chen, Gibbons, and Mowry (2021), Cavs' open-sourced version has worse performance than DyNet, because certain optimizations are not included. To make a fair comparison with Cavs, we use an extended version of DyNet with the main optimizations in Cavs enabled as a reference for Cavs' performance (referred to as Cavs DyNet). Namely, the static subgraphs in the network are pre-defined and batching optimization is applied to them. ED-Batch is implemented on top of this extended version, with the RL-based dynamic batching algorithm (E_{sort} for state encoding) and memory optimization on the static subgraphs by PQ Tree. On the other side, the agenda-based algorithm and the depth-

[2] Perform by standard REDUCE step in the Vanilla PQ tree algorithm to satisfy adjacency constraint by restructuring the tree.

Table 1. Models and datasets used in our evaluation

Model	Short name	Dataset
A bi-directional LSTM Named Entity Tagger (Huang et al., 2015)	BiLSTM-Tagger	WikiNER English Corpus (Nothman et al., 2013)
An LSTM-based encoder-decoder model for neural machine translation.	LSTM-NMT	IWSLT 2015 En-Vi
N-ary TreeLSTM (Tai et al., 2015)	TreeLSTM	Penn treebank (Marcus et al., 1994)
N-ary TreeGRU	TreeGRU	
MV-RNN (Socher et al., 2012)	MV-RNN	
An extension to TreeLSTM that contains two types of internal nodes, each with 50% probability	TreeLSTM-2Type	
A lattice-based LSTM network for Chinese NER (Zhang & Yang, 2018)	LatticeLSTM	Lattices generated based on Chinese Weibo Dataset
A lattice-based GRU network for neural machine translation (Su et al., 2017)	LatticeGRU	

based algorithm are used for dynamic batching on Vanilla/-Cavs DyNet. Depending on the workload and configuration, a better-performing algorithm is chosen for Vanilla/Cavs DyNet in the evaluation.

We test the framework on 8 workloads, shown in Table 1. They follow an increase in dynamism, from chains to trees and graphs. Except for lattice-based networks, all workloads appeared as benchmarks for past works. The Chinese Weibo Dataset was retrieved from `https://github.com/OYE93/Chinese-NLP-Corpus/tree/master/NER/Weibo`.

We run our experiments on a Linux server with an Intel Xeon E2590 CPU (28 physical cores) and an Nvidia V100 GPU. The machine runs CentOS 7.7, CUDA 11.1, and cuDNN 8.0. We use DyNet's latest version (Aug 2022, commit c418b09) for our evaluation.

5.2. Overall Performance

Fig. 6 compares ED-Batch's end-to-end inference throughput against Vanilla/Cavs DyNet. We follow past work to evaluate different batch sizes (1, 8, 32, 64, 128, 256) and model sizes (32, 64, 128, 256, 512), which is the size for the hidden vector length and the embedding size. The throughput is calculated as the *maximum* throughput among all batch size choices. For all cases, ED-Batch outperforms Vanilla DyNet significantly due to the reduction in graph construction and runtime overhead by pre-definition of the static subgraph.

We now discuss the comparison with Cavs DyNet. For the chain-based models BiLSTM-tagger and LSTM-NMT, ED-Batch achieved on average 1.20x, 1.11x higher throughput on CPU and 1.20x, 1.12x on GPU. Because the network structure is basically made up of chains, both the agenda-based algorithm and our FSM-based batching algorithm find the optimal batching policy (shown in Fig.7). On the other hand, the LSTMCell is 1.54x higher throughput with the PQ-tree optimization compared to the one with DyNet's memory allocation, which explains the improvements.

For the tree-based models, compared to agenda/depth-based batching heuristics, ED-Batch reduces the number of batches by 37%. This is because the FSM-based algorithm executes the output nodes in one batch (Fig. 1). For TreeLSTM and TreeGRU, ED-Batch achieved on average 1.63x, 1.46x higher throughput on CPU and 1.23x, 1.29x higher throughput on GPU. ED-Batch's performance is close to Cavs DyNet on MVRNN because the execution is bounded by matrix-matrix multiplications, which can hardly benefit from extra batch parallelism and the reduction in runtime overhead.

For the lattice-based models LatticeLSTM and LatticeGRU, ED-Batch increases DyNet Cavs's throughput significantly by 1.32-2.97x on CPU and 2.54-3.71x on GPU, which is attributed to both better dynamic batching and static subgraph optimization. For the lattice-based models' network structure in Fig. 8, the FSM-based algorithm prioritizes the execution of the majority type on the frontier, whereas the depth/agenda-based algorithms batch the character cell and word cell more arbitrarily. As a result, the number of batches is reduced by up to 3.27 times (Fig. 7). For the static subgraph, the used LSTMCell and GRUCell's latency is cut by 34% and 35%, which adds to the higher throughput.

5.3. Analysis

Where does ED-Batch's speedup come from? Fig. 9 shows a breakdown of the inference pass into construction, scheduling and execution time. Construction time is the time to define the dataflow graph. Scheduling time is the time for dynamic batching analysis. Execution time is the rest, mainly composed of executing operations. While having similar construction/scheduling time, ED-Batch speeds up Cavs DyNet by greatly reducing execution time due to better batching and fewer kernels for data movement.

Does the algorithm find a good enough batching policy? Compared to agenda/depth-based batching, our FSM-based batching uniformly executes fewer batches (Fig. 7). Among three state encoding choices, E_{sort} is slightly better because of the stronger expressiveness, finding the optimal batching policy on BiLSTM-tagger, LSTM-NMT, and Tree-based models and executing 23% and 44% more batches on TreeLSTM-2Type and Lattice-Based models.

To demonstrate the efficiency of the reward function, we measure the number of batches executed by a sufficient-

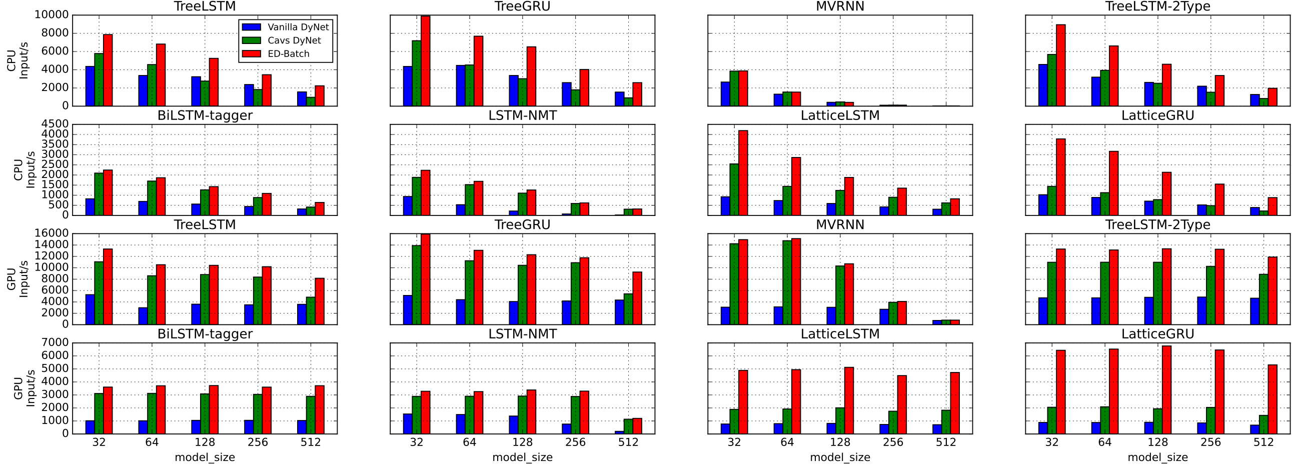

Figure 6. ED-Batch vs. Vanilla/Cavs DyNet: Inference Throughput

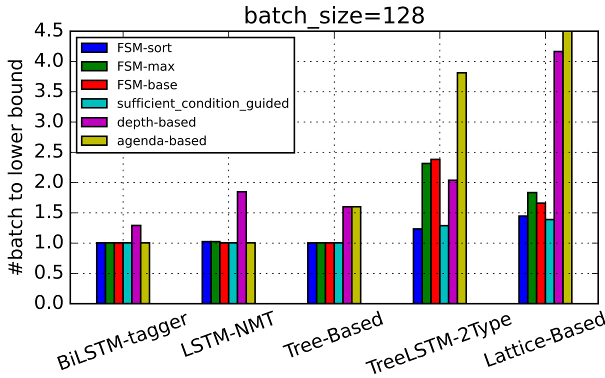

Figure 7. The number of batches for different batching algorithms. FSM-base/sort/max refers to the FSM based algorithm with different state encodings.

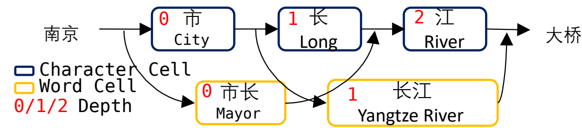

Figure 8. Lattice Network for Chinese NER. The input sentence topology is a chain of character cells with jump links of word cells. Agenda/depth batching fails to batch the word cells together.

condition-guided heuristic, which selects the type for the next batch that maximizes the second term in Eq. 1. Fig. 7 shows that this heuristic executes batches paramount to the best FSM-based algorithm. However, this heuristic has high runtime overhead. Thus, on the evaluated workloads, the FSM-based algorithm can be treated as a time-efficient distiller of this heuristic.

Ablation Study of the Static Subgraph Optimization. In Table 2, we evaluate ED-Batch's memory layout optimization on the static subgraphs. For all evaluated cases, the PQ-tree algorithm finds the *ideal memory allocation order* (the remaining data transfer is caused by broadcasts that

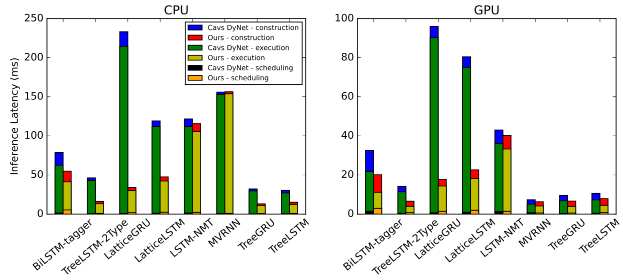

Figure 9. Cavs DyNet vs. ED-Batch: Time Decomposition for model size = 128 and batch size = 64.

Table 2. Batching with DyNet's memory allocation (left) vs. Batching with PQ tree-based memory allocation on static subgraphs (batch size = 8, model size = 64).

Subgraph	Latency (ms) value	ratio	Mem Kernels/Subgraph value	ratio	Memcpy Amount (kB) value	ratio
GRUCell	0.11 / **0.07**	1.54	6 / **2**	3.0	666.0 / **14.0**	47.57
LSTMCell	0.2 / **0.13**	1.52	4 / **1**	4.0	1054.0 / **16.0**	65.88
MVCell	**0.08** / 0.08	0.96	2 / **2**	1.0	260.0 / **260.0**	1.0
TreeGRU-Internal	0.24 / **0.15**	1.6	8 / **2**	4.0	552.0 / **16.0**	34.5
TreeGRU-Leaf	0.09 / **0.07**	1.4	4 / **2**	2.0	268.0 / **8**	33.5
TreeLSTM-Internal	0.19 / **0.12**	1.61	7 / **3**	2.33	1064.0 / **22.0**	48.36
TreeLSTM-Leaf	0.12 / **0.09**	1.27	3 / **1**	3.0	396.0 / **6.0**	66.0

cannot be optimized by better memory layout). Compared to the baseline, ED-Batch reduces the latency of the static subgraph by up to 1.6x, memory kernels by up to 4x, and memory transfer amount by up to 66x. This significant reduction in memory transfer can be attributed to the better arrangement of the weight parameters. For example, there are four gates in the LSTM cell that perform feed-forward arithmetic $y_i = W_i x_i + b_i$, which are executed in a batch. The memory arrangement in ED-Batch makes sure the inputs, parameters, and intermediate results of batched kernels are contiguous in the memory, which is not considered by DyNet's policy. Since the weight matrix occupies memory relative to the square of the problem size, this leads to a huge reduction in memory transfer.

Comparison with a more specialized framework. Cor-

Table 3. ED-Batch vs. Cortex: Inference Latency (ms).

		TreeGRU		TreeLSTM	
batch size	model size	Cortex	Ours	Cortex	Ours
10	256	2.30	**2.27**	**2.244**	2.78
	512	5.60	**3.04**	8.500	**4.70**
20	256	3.73	**3.03**	**3.460**	3.52
	512	11.70	**3.70**	19.210	**4.82**

tex (Fegade et al., 2021) is highly specialized for optimizing a class of recursive neural networks and it requires the user to not only express the tensor computation, but also specify low-level optimizations specific to underlying hardware, both through TVM's domain-specific language (Chen et al., 2018). We compare ED-Batch with Cortex on TreeLSTM and TreeGRU. To make more of an apples-to-apples comparison in terms of user effort in developing the application, we enabled Cortex's automated optimizations like *linearization* and *auto-batching* and used simple policies on optional user-given (manual) optimizations like kernel fusion and loop transformation (details in Appendix C). As shown in Table 3, ED-Batch can speed up Cortex by up to 3.98x.

Table 4. RL Training Time and iterations

	Time (s)	Train Iter.
TreeLSTM	0.154	50
TreeGRU	0.141	50
MVRNN	0.254	50
TreeLSTM-2type	2.217	1000
BiLSTM-tagger	1.629	50
BiLSTM-tagger-withchar	6.268	50
LatticeLSTM	21.733	1000
LatticeGRU	4.911	1000

Compilation overhead. The training of the RL model is empirically efficient. We trained the RL model for up to 1000 trials and stopped early if the number of batches reaches the lower bound (check every 50 iterations). The most onerous task (see Table 4) is to train for the lattice-based network. This takes 22 seconds for 1000 iterations to train on a dataflow graph of 11,626 nodes, and results in 1040 states in the Q-table. On the static subgraph, the batching policy is obtained by the grid search and the PQ tree optimization is applied afterward. Shown in Table 5, it takes tens of milliseconds to optimize the static subgraph.

Table 5. Static Subgraph Compile Time

	Time (ms)
GRUCell	4.82
LSTMCell	12.95
MVCell	1.53
TreeGRU-Internal	10.43
TreeGRU-Leaf	2.91
TreeLSTM-Internal	29.89
TreeLSTM-Leaf	3.64

6. Related Work and Discussion

There are a variety of frameworks specialized for dynamic neural network training (Neubig et al., 2017a; Looks et al., 2017; Xu et al., 2018) and inference (Fegade et al., 2021; Fegade, 2023; Zha et al., 2019; Gao et al., 2018). Concerning batching for the dynamic neural networks, DyNet (Neubig et al., 2017a) and TFFold (Looks et al., 2017) laid the system and algorithm foundation to support *dynamic batching*. However, their batching heuristics are often sub-optimal as we saw above. Nevertheless, their algorithms have been used in other frameworks, like Cavs (Xu et al., 2018) and Acrobat (Fegade, 2023). Apart from batching, another major direction of optimization is to extract the static information from the dynamic DNN and optimize them during compile time. Cavs (Xu et al., 2018) proposed the idea of predefining the static subgraphs, which was later extended in (Zha et al., 2019) to batch on different granularities. ED-Batch adopts this multi-granularity batching idea to perform batching on both the graph level and the subgraph level. For static subgraphs, traditional techniques are used for optimization, like the kernel fusion in Cavs and Cortex (Fegade et al., 2021), the AoT compilation (Kwon et al., 2020), and specialized kernel generation in Acrobat (Fegade, 2023). However, though with high efficiency, these optimizations can hardly be automated because either the developer or the user needs to optimize each subgraph manually. In ED-Batch, fully automated runtime optimizations are used instead to enable both efficient execution and generalization.

Considering the scalability of ED-Batch, increasing the expressiveness of the state encoding would enable our FSM-based method to accommodate increasingly complex networks. Also, the invariance to the feature sizes of the runtime enables ED-Batch to scale with any hidden vector size. Lastly, the used RL training/inference algorithm scales linearly with the size of the dataflow graph, and moreover this overhead is hidden by parallel execution of CPU and GPU at runtime.

7. Conclusion

In ED-Batch, we designed an FSM-based algorithm for batched execution of dynamic DNNs. Also, we mitigated the memory copying introduced by batching through a memory layout optimization based on the PQ-tree algorithm. The experimental results showed that our approach achieved significant speedup compared to current frameworks by reducing the number of batches and data movement.

Acknowledgements

This work is supported in part by grants from the National Science Foundation, and by the member companies of the PDL consortium. We thank the reviewers for very constructive comments that helped improve the paper. We thank the PKU-CMU summer program for providing the research opportunity, Peking University's supercomputing team for providing the hardware to run the experiments, and Kevin Huang for his help in the stages of the research.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P. A., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 265–283, 2016.

Booth, K. S. and Lueker, G. S. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *Journal of Computer and System Sciences*, 13(3):335–379, 1976.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E. Q., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 578–594, 2018.

Chen, X., Qiu, X., Zhu, C., and Huang, X. Gated recursive neural network for Chinese word segmentation. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 1744–1753, 2015.

Fegade, P. *Auto-batching Techniques for Dynamic Deep Learning Computation*. PhD thesis, January 2023. URL https://kilthub.cmu.edu/articles/thesis/Auto-batching_Techniques_for_Dynamic_Deep_Learning_Computation/21859902.

Fegade, P., Chen, T., Gibbons, P., and Mowry, T. Cortex: A compiler for recursive deep learning models. *Proceedings of Machine Learning and Systems*, 3:38–54, 2021.

Gao, P., Yu, L., Wu, Y., and Li, J. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*, pp. 1–15, 2018.

Huang, Z., Xu, W., and Yu, K. Bidirectional LSTM-CRF models for sequence tagging. *arXiv preprint arXiv:1508.01991*, 2015.

Kwon, W., Yu, G.-I., Jeong, E., and Chun, B.-G. Nimble: Lightweight and parallel GPU task scheduling for deep learning. *Advances in Neural Information Processing Systems*, 33:8343–8354, 2020.

Landau, G. M., Parida, L., and Weimann, O. Using PQ trees for comparative genomics. In *Annual Symposium on Combinatorial Pattern Matching*, pp. 128–143, 2005.

Looks, M., Herreshoff, M., Hutchins, D., and Norvig, P. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*, 2017.

Marcus, M., Kim, G., Marcinkiewicz, M. A., MacIntyre, R., Bies, A., Ferguson, M., Katz, K., and Schasberger, B. The Penn treebank: Annotating predicate argument structure. In *Human Language Technology: Proceedings of a Workshop held at Plainsboro, New Jersey*, 1994.

Meidanis, J., Porto, O., and Telles, G. P. On the consecutive ones property. *Discrete Applied Mathematics*, 88(1-3): 325–354, 1998.

Neubig, G., Dyer, C., Goldberg, Y., Matthews, A., Ammar, W., Anastasopoulos, A., Ballesteros, M., Chiang, D., Clothiaux, D., Cohn, T., et al. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017a.

Neubig, G., Goldberg, Y., and Dyer, C. On-the-fly operation batching in dynamic computation graphs. In *Advances in Neural Information Processing Systems 30*, 2017b.

Nothman, J., Ringland, N., Radford, W., Murphy, T., and Curran, J. R. Learning multilingual named entity recognition from wikipedia. *Artificial Intelligence*, 194:151–175, 2013.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pp. 8024–8035, 2019.

Räihä, K.-J. and Ukkonen, E. The shortest common supersequence problem over binary alphabet is NP-complete. *Theoretical Computer Science*, 16(2):187–198, 1981.

Socher, R., Huval, B., Manning, C. D., and Ng, A. Y. Semantic compositionality through recursive matrix-vector spaces. In *Joint conference on empirical methods in natural language processing and computational natural language learning*, pp. 1201–1211, 2012.

Su, J., Tan, Z., Xiong, D., Ji, R., Shi, X., and Liu, Y. Lattice-based recurrent neural network encoders for neural machine translation. In *AAAI Conference on Artificial Intelligence*, volume 31, 2017.

Tai, K. S., Socher, R., and Manning, C. D. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

Watkins, C. J. and Dayan, P. Q-learning. *Machine learning*, 8(3):279–292, 1992.

Xu, S., Zhang, H., Neubig, G., Dai, W., Kim, J. K., Deng, Z., Ho, Q., Yang, G., and Xing, E. P. Cavs: An efficient runtime system for dynamic neural networks. In *Usenix Annual Technical Conference*, pp. 937–950, 2018.

Zha, S., Jiang, Z., Lin, H., and Zhang, Z. Just-in-time dynamic-batching. *arXiv preprint arXiv:1904.07421*, 2019.

Zhang, Y. and Yang, J. Chinese NER using lattice LSTM. *arXiv preprint arXiv:1805.02023*, 2018.

Appendices

A. Dynamic Batching

A.1. Proof of NPC property

For a directed acyclic graph $G(V, E)$, each node has a type $t \in T$, we define *batch sequence* as a sequence of types $s \in T^*$, that can be used iteratively as the next type in Algorithm 1 to batch the whole dataflow graph. The *Batching* problem is to find a batch sequence with the smallest possible length, denoted as an *optimal batching sequence*.

Theorem A.1 (NP-hard for Batching). *Batching is NP-hard.*

Proof. We prove the NP-hardness by reducing from *Shortest Common Supersequence* (SCS). Given an alphabet A, a set of strings, $s_1, s_2, ..., s_n$ in A, the SCS problem finds the shortest common supersequence for these strings. Treating each letter in the string as a node, the string is a chain of nodes, which is a DAG. Therefore, $s_1, s_2, ..., s_n$ compose a DAG with many independent chains. Suppose the *optimal batching sequence* for this DAG is found, we claim that it is exactly the common supersequence for these strings. On one side, every string must appear as a substring in the *optimal batching sequence* to complete the batching. On the other side, if there is a common supersequence shorter than the *optimal batching sequence*, this common supersequence is also a legal batching sequence. This is because in Algorithm 1, we greedily batch nodes in the frontier once their type is equal to the one in the batching sequence. So it is sufficient for a string to appear as a subsequence to be fully batched. This yields the contradiction. So the *optimal batching sequence* is the common supersequence, indicating SCS can be solved by *Batching* with poly time encoding. So, *Batching* is NP-hard. □

To our knowledge, there is no constant guaranteed approximation algorithm for the SCS problem, and hence neither for *Batching*.

A.2. Proof of the sufficient condition on batching

Lemma A.2. *If* $\frac{|Frontier_{a_t}(G_t)|}{|Frontier(G_t^{a_t})|} = 1$*, then there exists a shortest batching sequence starting with* a_t.

Proof. Proof by contradiction. If this is not true, let S be the set of operations of the first batch whose operation type is a_t. Then, we must have $S \subset Frontier(G_t^{a_t})$. Because $\frac{|Frontier_{a_t}(G_t)|}{|Frontier(G_t^{a_t})|} = 1$, meaning that S is ready to execute for the first batch. Thus, by moving S to the first batch committed, we get one of the shortest batching sequences starting with a_t. Contradiction. □

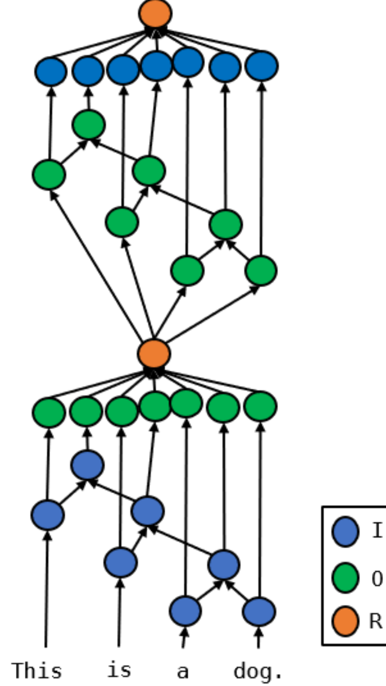

Figure 10. Example when the FSM approach fails to work well.

A.3. Lower Bound

For a dataflow graph G and type set T, the lower bound of kernel launches is given by

$$|Batching^*(G)| \geq \Sigma_{t \in T} Depth(G_t). \qquad (2)$$

The heuristic behind the formula is that it requires at least $Depth(G_t)$ steps to fully execute the G_t. Because of the dependency between G_ts, the execution requires at least $\Sigma_{t \in T} Depth(G_t)$ steps to finish.

A.4. Case the FSM does not cover

There are cases when the FSM cannot find a good policy. In the fake example in Fig. 10, we concatenate two tree networks, but the second has the type of Internal node and Output Node swapped. Here, the FSM in Fig. 2 does not work well because the first tree requires batching Input node S_2 while the second requires batching the Output node. This problem can be solved by introducing into the state encoding phase information such as the portion of nodes committed.

B. PQ tree

In this section, we illustrate the functions used in Algorithm 2 and give the proof of its time complexity.

Detailed Illustration

The supporting functions for BROADCASTCONSTRAINT

Algorithm 3 Algorithms for functions in BROADCASTCONSTRAINT

```
1: function GETSUBTREECONS(o)
2:   root = FindRoot(o)
3:   nodeToLeaves = getNodeToLeaves(root) {A function maps nodes to leaves in its subtree. Realized by a traversal of the tree on the recursive function nodeToLeaves(node) = node.isLeaf?{node} : {nodeToLeaves[child]|child ∈ node.children}}
4:   constraints = getList()
5:   for node in getNodesInSubTree(root) do
6:     if node is P-node then
7:       cons = ∪_{child∈node.children} nodeToLeaves(child)
8:       constaints.push(cons)
9:     else if node is Q-node then
10:      for child ∈ node.children do
11:        sib = child.nextSibling()
12:        cons = ∪{nodeToLeaves(child), nodeToLeaves(sib)}
13:        constraints.push(cons)
14:      end for
15:    end if
16:  end for
17:  return constraints
18: end function
19: function PARSECONSTRAINTS(constraints, batch)
20:   uniformConstraints = ∪_{o∈batch.operands}{o.index(x)|x ∈ getSubtreeCons(o)} {Parse operand-wise consecutive constraint.}
21:   constraints = getList()
22:   for o in batch.operands do
23:     {Transform constraint by alignment information.}
24:     for cons in uniformConstraint do
25:       constraints.append({o[x]|x ∈ cons})
26:     end for
27:   end for
28:   return constraints
29: end function
30: function APPLYCONSTRAINTS(constraints, tree, updatedOperands)
31:   for cons in constraints do
32:     suc = ReduceAndGetChanged(tree, cons, updatedOperands)
33:     if suc is False then
34:       return False
35:     end if
36:   end for
37:   return True
38: end function
```

are shown in Algorithm 3. The FINDROOT function is supported by the BUBBLE method in the vanilla PQ tree algorithm to search the root for the minimal subtree for a set of leaf roots. The REDUCEANDGETCHANGED method is supported as an extension to the REDUCE method in the vanilla PQ tree algorithm to add a consecutive constraint to the PQ tree and record the batches whose tree structure gets changed. It needs to maintain a mapping between the P/Q node with the batches and updates it when the tree structure gets updated in the REDUCE step.

Algorithm 4 ParseEquivNodeOrderPair

```
1: function PARSEEQUIVNODEORDERPAIR(tree, batch)
2:   Q = getQueue() {Queue on equivalent nodes.}
3:   for i in batch.operands.front().size() do
4:     Q.push((o[i]|o ∈ batch.operands))
5:   end for
6:   EquivNodeOrderPairs = getList()
7:   Find the root and calculate the leaf count for the subtree of the first operand.
8:   while True do
9:     {A leaf-to-root search performed parallel on operands in one batch.}
10:    nodes = Q.pop()
11:    node = nodes.front()
12:    if node is P node then
13:      EquivClass = {(node, node.referenceRrder)| node ∈ nodes}
14:      EquivNodeOrderPairs.add((P, EquivClass))
15:    else if node is Q node then
16:      EquivClass = {(node, getDirection(node, node.referenceOrder))|node ∈ nodes}
17:      EquivNodeOrderPairs.add((Q, EquivClass))
18:    end if
19:    node.parent.leafCnt = node.parent.leafCnt− node.leafCnt
20:    if node.parent.leafCnt is 0 then
21:      Q.push((node.parent|node ∈ nodes))
22:    end if
23:    for node in nodes do
24:      {Reference Order is used to decide the node order.}
25:      node.parent.referenceOrder.append(node)
26:    end for
27:    if node.isRoot then
28:      {Stop Condition: Root Found.}
29:      Break.
30:    end if
31:  end while
32:  return EquivNodeOrderPairs
33: end function
```

The PARSEEQUIVNODEORDERPAIR method is given in Algorithm 4 to parse the equivalent node order pairs on the isomorphic tree structures for operands in a batch. It is performed by simultaneous bottom-up traversal for operands in this batch.

The methods concerning the Union Find data structure are listed in Algorithm 5. In this problem, the UnionFindSet data structure is a set of nodes, and each node has two attributes: (i) $parent$, the pointer to the node's parent, or the decider of its order, and (ii) σ, the transformation that transforms the node's order (a permutation for P-node or reverse for Q-node) to its parent's. Given a node, the FIND method returns the root node of this node and the node's relative order with the root. In FIND method, the equivalence relationship between two node order pairs, i.e. ($node_1$, σ_1), ($node_2$, σ_2), is built. The constraint conveyed is that if $node_1$ has order σ then $node_2$ must have order $\sigma \circ \sigma_1^{-1}\sigma_2$, and this is encoded into the data structure by building a

Algorithm 5 Extended Union-Find set algs

```
1: function GETUNIONFINDSET(nodes)
2:   for node in nodes do
3:     node.parent = node
4:     node.σ = I {Identical transformation}
5:   end for
6: end function
7: function FIND(node)
8:   σ = I {order relative to the root.}
9:   while node.parent is not node do
10:    σ = σ ∘ node.σ
11:    node = node.parent
12:  end while
13:  return node, σ
14: end function
15: function UNION(node_1, σ_1, node_2, σ_2)
16:   p_1, σ_3 = Find(node_1)
17:   p_2, σ_4 = Find(node_2)
18:   if p_1 is not p_2 then
19:     p_1.parent = p_2
20:     p_1.σ = σ_3^{-1} σ_4 σ_2^{-1} σ_1
21:   else if σ_1^{-1} σ_2 is σ_3^{-1} σ_4 then
22:     {Compatible. Do nothing. Already equivalent.}
23:   else
24:     {Incompatible.}
25:     return False
26:   end if
27:   return True
28: end function
```

Algorithm 6 Get Memory Allocation Order

```
1: function GETLEAFORDER(tree, POrder, QOrder)
2:   root = tree.root
3:   order = getList()
4:   S = getStack()
5:   S.push(root)
6:   while S.notEmpty() do
7:     {Depth first traversal}
8:     node = S.pop()
9:     if node is P node then
10:      p, σ = POrder.find(node)
11:      for child in σ(node.children) do
12:        GetLeafOrder(child)
13:      end for
14:    else if node is Q node then
15:      p, direction = POrder.find(node)
16:      for child in node.children following direction do
17:        GetLeafOrder(child)
18:      end for
19:    else
20:      {Leaf Node here}
21:      order.append(order)
22:    end if
23:  end while
24:  return order
25: end function
```

relationship between their roots. If their roots are not the same, an edge connects them with the transformation satisfying the information (line 20). If they are the same, then $node_1, node_2$'s relative order to the root must satisfy the constraint (line 21). Otherwise, the equivalence relationship is not compatible and this relationship is dropped.

Finally, we obtain the memory allocation sequence by a depth-first traversal satisfying the constraint we found on the node order (Algorithm 6).

Complexity

Lemma B.1. *For the batching problem, PQ tree memory allocation algorithm's time complexity is* $O(\Sigma_{batch \in batches}|batch| \max^2_{batch \in batches} |batch|)$ *where* $|.|$ *counts the operation in a batch.*

Proof. The REDUCE step on a consecutive constraint S in the PQ tree is $O(|S|)$. Thus, the time complexity for PQ-tree construction is $O(\Sigma_{batch \in batches}|batch|)$. In the BROADCASTCONSTRAINT pass, the while-loop body (lines 10-20) can only perform O($\Sigma_{batch \in batches}|batch|$). This is because every update on the PQ-tree structure either transfers a P-node into a Q-node or introduces a new node, the total times for updates on the tree structure are bounded by the number of internal nodes for the PQ-tree and are further bounded by the number of leaf variables. Then, for the while-loop body, the GETSUBTREECONS method on an operand with k variables needs $O(k^2)$ to compute as the GETNODETOLEAVES method needs to assign each node with the set of its leaves and the number of nodes is bounded by k. It is not hard to see the rest of PARSECONSTRAINT has lower complexity. Thus, for a batch with m variables, it takes $O(mk^2)$ to compute the PARSECONSTRAINT. For the APPLYCONSTRAINTS step, the REDUCEANDGETCHANGED step can be implemented to have the same complexity as REDUCE, where a bi-direction map is used to store the relationship between the node and the batch, once a node is deleted or inserted, a callback is used to update this table in constant time. Then, APPLYCONSTRAINTS's complexity is bounded by the sum of variables in the constraints, which is also $O(mk^2)$. In all, for the BROADCASTCONSTRAINT pass, suppose there are n variables, the time complexity is $O(nmk^2)$. Under the batching setting, each node appears in the result operand of a batch once and only once, $\Sigma_{batch \in batches}|batch| = nm$, and $k \leq \max_{batch \in batches} |batch|$. Thus, the time comlexity is bounded by $O(\Sigma_{batch \in batches}|batch| \max^2_{batch \in batches} |batch|)$.

For the DECIDENODESORDER function, PARSEEQUIVNODEORDERPAIR on $batch$ requires $O(|batch|)$ time to traverse the graph. The $Union$ method is $O(\alpha(n))$, where $\alpha(n)$ is the extremely slow-growing inverse Ackermann function. Thus, the time complexity is $O(\Sigma_{|batch| \in batches}(|batch| + O(\alpha(n)))) \approx O(\Sigma_{|batch| \in batches}(|batch|))$.

In all, the time complexity of the PQ

Table 6. Input Throughput Comparison

Input/s	**Torch**	**TF**	**DyNet**	**Cavs**	**ED-Batch**
CPU	39	37	671	480	1163
GPU	84	26	2723	6243	6736

tree memory allocation algorithm is $O(\Sigma_{batch \in batches} |batch| \max^2_{batch \in batches} |batch|)$.

□

C. More on Our Comparison with Cortex

Cortex (Fegade et al., 2021) is highly specialized for optimizing the recursive neural network and it requires the user to express the tensor computation and specify the optimization through TVM's domain-specific language (Chen et al., 2018). This framework is fundamentally different from general frameworks like ED-Batch and DyNet in that it does not rely on vendor libraries and the user is given a full chance to optimize computation from the graph level to the operation level. This gives expert users the chance to squeeze the performance by specializing the application to the hardware but is burdensome for common users who basically want to prototype an application.

The experiment for the comparison between ED-Batch with Cortex performs on TreeLSTM and TreeGRU. Cortex does not support the LSTM-NMT model because it has a tensor-dependent control flow. We did not compare the rest of the models basically because of the lack of expertise in writing the schedules in TVM. The optimizations we used in Cortex include the automated *linearization* and *auto-batching*. For the user-given optimizations, we did not perform kernel fusion and the individual operators were optimized by loop transformations like loop tiling, loop reorder, and axis binding. In the end, for the TreeLSTM case it takes us 30 lines of python code to specify the computations and 105 lines of TVM schedules to optimize the kernel.

D. Comparison with Torch and TF

As discussed in Cavs (Xu et al., 2018)'s and DyNet (Neubig et al., 2017a)'s paper, their frameworks are one to two orders of magnitude faster than traditional frameworks like Torch (Paszke et al., 2019) and TensorFlow (Abadi et al., 2016) across hardware platforms and workloads. We validate this fact by a case study on TreeLSTM across CPU and GPU (batch size of 256 and model size of 512). As shown in Table 6, ED-Batch has 29x/37x higher throughput than Torch/TensorFlow on CPU, and 80x/280x higher throughput on GPU.