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.git.

1. Introduction

Batching accelerates the training and inference for deep neural networks (DNN) because it (1) launches fewer kernels resulting in lower kernel launch and scheduling overhead on the CPU, and (2) 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., 2015b), 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: (1) to enable operation-level batching at runtime (Looks et al., 2017; Neubig et al., 2017a; Zha et al., 2019), i.e. *dynamic batching*, and (2) to extract static parts (Xu et al., 2018), i.e. the static subgraphs (e.g. the LSTM cell), from the dataflow graph and optimize 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 applications.

Further, 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 emit gather/scatter operations before and after each batch (Xu et al., 2018; Neubig et al., 2017a) or 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

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

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

FSMs represents a sweet-spot between 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 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 achieved on average 1.15x, 1.39x, and 2.45x speedup 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.git.

2. FSM-based Algorithm for Dynamic Batching

In this section, we identify the shortcomings of current batching techniques, propose a new FSM-based dynamic batching algorithm, and the mechanism to learn it by RL.

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. Therefore, the batching algorithm cannot have high complexity to avoid severe runtime overhead. 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. As a result, the heuristics used for dynamic batching in the current systems often find a suboptimal policy. As shown later (Fig.9), the number of batches executed by current frameworks can be cut by up to 3.27 times.

Algorithm 1 FSM-based Dynamic Batching

```
**Input:** Dataflow Graph $G$, State Encoding Function $E$, Policy $\pi$;
**while** G.notEmpty() **do**
    nextType = $\pi(E(G))$
    batch = [v for v in Frontier(G) if v.type is nextType]
    Execute batch.
    Update the Frontier.
**end while**
```

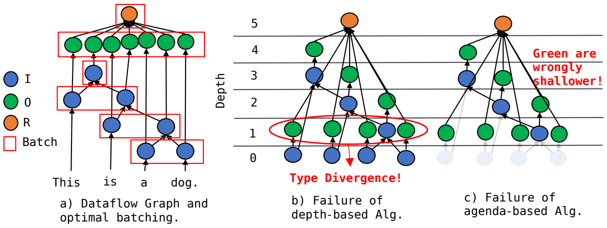
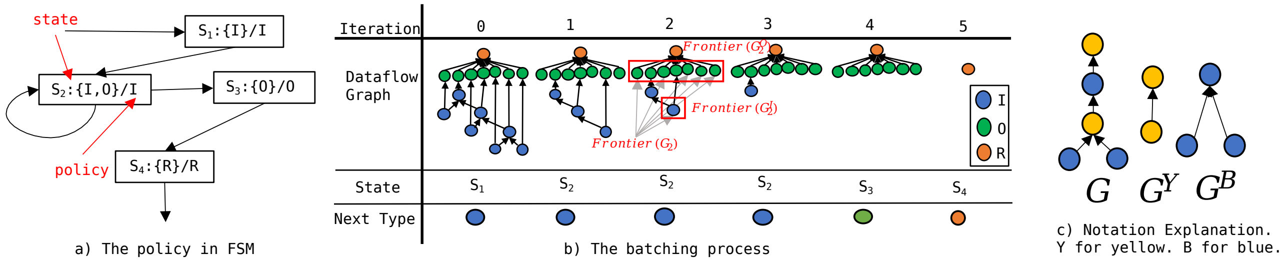

I
O
R
Batch
This
is
a
dog.
a) Dataflow Graph and optimal batching.
Depth
5
4
3
2
1
0
Type Divergence!
b) Failure of depth-based Alg.
Green are wrongly shallower!
c) Failure of agenda-based Alg.

Figure 1. Example on current dynamic batching algorithms.

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 result 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 executes all O nodes in one batch. However, the depth-based algorithm in Fig. 1(b) executes the O nodes in four batches because they have different topological depths. For the agenda-based algorithm, when it is deciding the next batch after batching the I nodes as the first batch (Fig. 1(c)), because the O nodes have a lower average depth ($\overline{Depth} = (1+1+1+1+2+3+4)/7 = 1.85$) than the I nodes ($\overline{Depth} = (1+2+3)/3 = 2$), the algorithm will pick the O nodes for the next batch. As the result, the agenda-based algorithm launches one extra batch compared to the optimal batching (7 vs. 6 batches) because it does two O batches (its 2nd batch and 6th batch), whereas the optimal (Fig. 1(a)) does only one O batch (its 5th batch).

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



a) The policy in FSM

b) The batching process

c) Notation Explanation. Y for yellow. B for blue.

Figure 2. Dynamic Batching Policy by FSM

2.2. FSM-based Dynamic Batching

To fully overcome the limitation of specific graph statistics, we found that an FSM-based approach (1) offers the opportunity to specialize for network structure under a rich design space of potential batching policies and (2) can generalize to any number of input instances, as long as they share the same regularity in topology.

Shown in Alg.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 $S = E(G)$, 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. 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 E. 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 Alg. 1. Now we elaborate on the state encoding, reward design, and training respectively.

For better understanding, we first explain some 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 (See example in Fig.2(c)), $Frontier(G)$ refers to the set of ready-to-execute operations, $Frontier_a(G)$ refers to the subset of $Frontier(G)$ with type a.

State Encoding: The design of state encoding should carry enough information to capture the network's regularity and be as simple as possible to avoid heavy runtime overhead. In practice, we experimented with three ways of encoding: (1) $E_{base}(G) = \{v.type | v \in Frontier(G)\}$ is the set of operation types on the frontier, (2) $E_{max}(G) = (E_{base}(G), argmax_{t \in T} |Frontier_t(G)|)$ is $E_{base}(G)$ plus the most common type on the frontier and (3) $E_{sort}(G) = sort(\{v.type \in T | v \in Frontier(G)\}, t : |Frontier_t(G)|)$ 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, thus increasing the parallelism exploited. The reward function is defined as

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

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

Lemma 2.1 (Sufficient Condition for Optimal Batching)**.** [2] *If* $\frac{|Frontier(G_t^{a_t})|}{|Frontier_{a_t}(G_t)|} = 1$*, then there exists a shortest batching sequence starting with* a_t*.*

The second term is inspired by a sufficient condition for optimal batching (Lemma 2.1) to prioritize the type 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. 9), this sufficient condition continues to hold.

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. And 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.

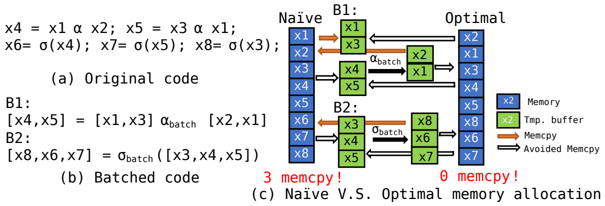
The key observation for training is that the policy (i.e. FSM) is independent of the input instance and the batch size, and is only dependent on the DNN's type. This observation enables us to train on a single batch and generalize to larger batch sizes of different input instances. During training, the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



```
x4 = x1 α x2; x5 = x3 α x1;
x6= σ(x4); x7= σ(x5); x8= σ(x3);
```

(a) Original code

```
B1:
[x4,x5] = [x1,x3] α_batch [x2,x1]
B2:
[x8,x6,x7] = σ_batch([x3,x4,x5])
```

(b) Batched code

(c) Naïve V.S. Optimal memory allocation

Figure 3. Example on memory allocation. 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 an 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 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 $B1$, $\{x_6, x_7, x_8\}, \{x_4, x_3, x_5\}$ 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

[2] Proof in the Appendix A.2

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 PQ tree, representing potential permutations of S that elements in each subset are consecutive. Fig.4(a) shows the PQ tree for the example code. The tree has three kinds of nodes: P-node, Q-node, and leaf 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-nodes 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.

Two passes obtain the order annotation to the PQ tree (Fig.4, Alg.2). 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.

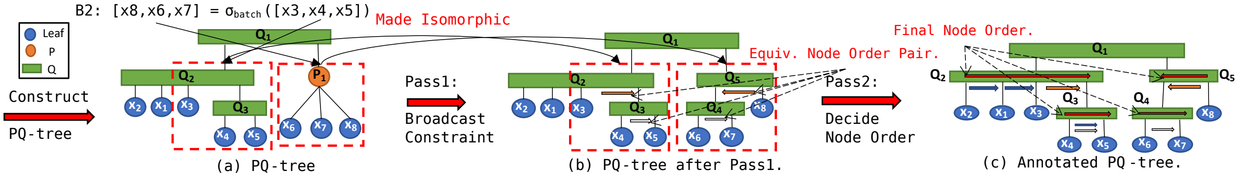
We walk through the algorithm in the example. At first, the PQ tree is constructed by the standard algorithm to satisfy the adjacency constraint (Fig.4(a)). After that, BROADCASTCONSTRAINT pass makes the tree-structure of operands in a batch isomorphic by repeatedly parsing adjacency constraints (line 12) and broadcasting the 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 destination operand, and $\{x_6, x_7\}$ is applied to the PQ tree as an adjacency constraint[3], resulting in the P_1 node replaced 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.

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
```

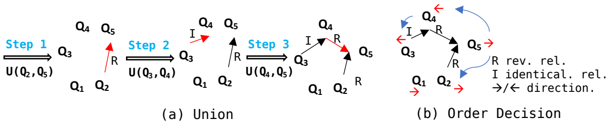
In DECIDENODESORDER (line 24), we assign directions for the Q-node and the permutation for the P-node. We start by parsing the equivalence relationship (line 28) among Q-node-direction pairs or the P-node-permutation pairs from the isomorphic tree structures after the first pass, e.g. $< Q_2, \leftarrow> \iff < Q_3, \leftarrow>$ for $B1$, $< Q_3, \leftarrow> \iff < Q_4, \leftarrow>$ and $< Q_2, \leftarrow> \iff < Q_5, \rightarrow>$ for $B2$. After that, we spread the equivalence relationship across batches with the support of *union–find data structure*. In the algorithm, a graph carrying the equivalence relationship is constructed by iteratively UNION equivalent relationship

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



(a) PQ-tree (b) PQ-tree after Pass1. (c) Annotated PQ-tree.

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



(a) Union (b) Order Decision

Figure 5. Illustration for order decision in Pass2.

(line 29-35) among the node-order pairs. The nodes are composed of all P/Q-nodes in the tree, and an directed edge $< n_s, n_t, f >$ between nodes n_s and n_t with transformation function f indicates the n_s's order after transformation f should be the same with n_t's order.

Fig.5, shows a graph construction process for the example in Fig.4. When processing $< Q_2, \leftarrow> \iff < Q_5, \rightarrow>$ in step 1, a $< Q_2, Q_5, R >$ edge is added to the graph, indicating Q_2's direction is determined by the reverse of Q_5's direction. When processing the $< Q_2, \leftarrow> \iff < Q_3, \leftarrow>$ 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 an $< Q_4, Q_5, R >$ edge. In this way, Q_2 and Q_3 always have the same order. Finally, the deciders in the graph are assigned arbitrary directions, which spread across the graph following the relationship on the edge (Fig.5(b)).

The PQ tree memory allocation algorithm's time complexity is given in lemma 3.1, showing that under certain constraint, the PQ tree algorithm scales linearly with the size of the dataflow graph and the batch size. Empirically, the algoirthm runs in tens of milliseconds for common subgraphs (Table 4).

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 operation 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 problem.

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.

Detailed explanation of the algorithm is in Appendix. B.

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 of 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 faster than traditional frameworks like Pytorch (Paszke et al., 2019) and TensorFlow (Abadi et al., 2016) (see Appendix. D). As stated 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 subgraph by PQ Tree. On the other side, the agenda-based algorithm and the depth-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.

Table 1. Models and datasets used in 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., 2015a)	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[4]
A lattice-based GRU network for neural machine translation (Su et al., 2017)	LatticeGRU	

They follow an increase in dynamism, from chains to trees and graphs. Except for lattice-based networks, all workloads appeared as the benchmark for past works.

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 evaluation.

5.2. Overall Performance

We compare 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 bath 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 chain-based models, the BiLSTM-tagger and LSTM-NMT, ED-Batch achieved on average 1.20x, 1.11x speedup on CPU and 1.20x, 1.12x on GPU. Because the network structure is basically made up of chains, shown in Fig.9, both the agenda-based algorithm and the FSM-based batching algorithm find the optimal batching policy. On the other hand, the LSTMCell is 1.54x faster with the PQ-tree optimization compared to the one with the DyNet's memory allocation, which explains the speedup.

For the tree-based model, compared to agenda/depth-based batching heuristic 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 speedup on CPU and 1.23x, 1.29x speedup 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, the 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 the better dynamic batching and static subgraph optimization. For the lattice-based models' network structure in Fig.7, the FSM-based algorithm prioritizes the execution of 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.9). For the static subgraph, the used LSTMCell and GRUCell's latency is cut by 34% and 35%, which adds to the speedup.

5.3. Analysis

Where does ED-Batch's speedup come from? Shown in Fig.8, we decompose the inference pass into the construction time, scheduling time, 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 left time of the forward pass, mainly composed of the execution of operations. While having similar construction/scheduling time, ED-Batch speeds up Cavs DyNet in the great cutdown in execution time benefited from better batching and fewer kernels for data movement.

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

	Latency (ms)		Mem Kernels/Subgraph		Memcpy Amount (kB)	
Subgraph	value	ratio	value	ratio	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

Table 3. 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

Table 4. 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

Does the algorithm find a good enough batching policy? Shown in Fig.9, compared to the agenda/depth-based batching algorithm, ED-Batch's FSM-based batching algo-

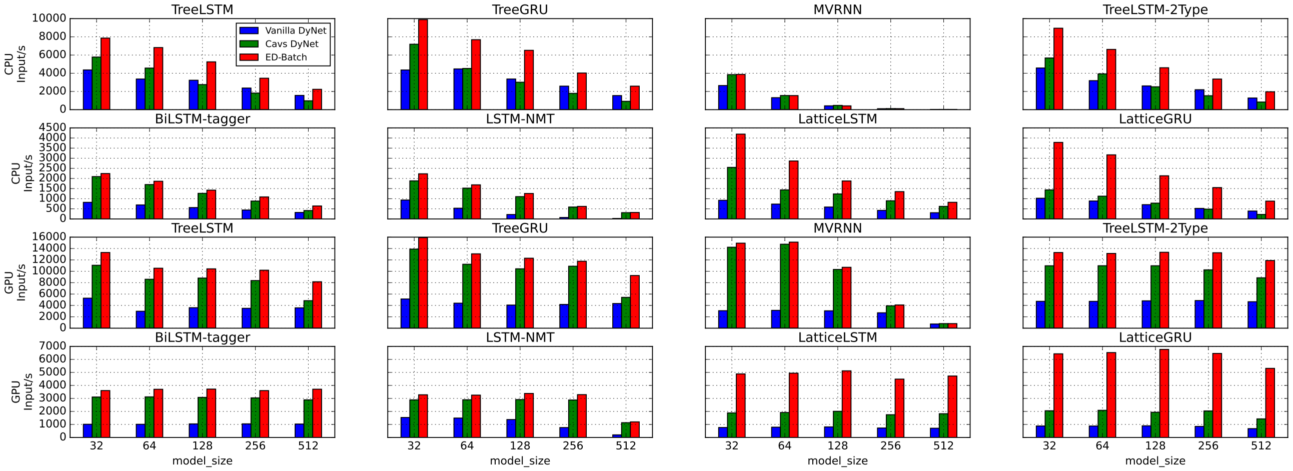


Figure 6. ED-Batch v.s. Vanilla/Cavs DyNet: Inference Throughput

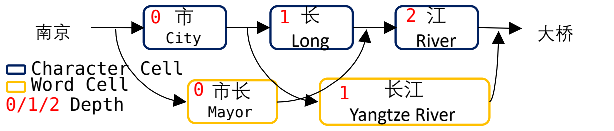


Figure 7. Lattice Network for Chinese NER. The topology for one input sentence is a chain of character cells with jump links of word cells. Agenda/Depth-based batching algorithm fails to batch the word cells together.

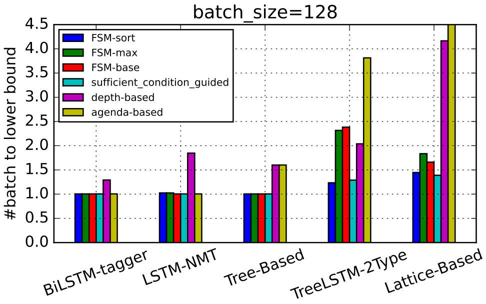


Figure 9. The number of batches for different batching algorithms. FSM-base/sort/max refers to the FSM based algorithm with different state encodings.

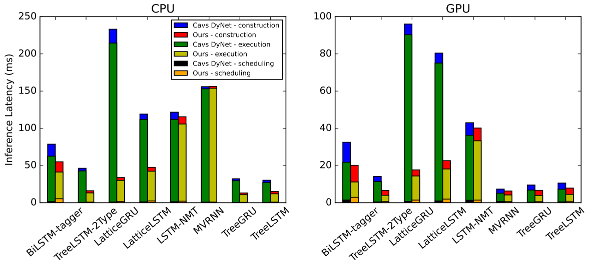


Figure 8. Cavs DyNet v.s. ED-Batch: Time Decomposition when model size is 128 and batch size is 64.

Table 5. ED-Batch v.s. 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**

rithm uniformly executes fewer batches. 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, Lattice-Based models.

To demonstrate the efficiency of the reward function, we measure the number of batches executed by a sufficient-condition-guided heuristic, which selects the type for the next batch that maximizes the second term in Eq.1. Shown in Fig. 9, this heuristic executes batches paramount to the best FSM-based algorithm. However, this heuristic has higher time complexity and adds to unacceptable 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* (Remained data transfer is caused by broadcast that 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. Cortex (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 the user's 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 5, ED-Batch can speed up Cortex by up to 3.98x.

The compilation overhead. The training of RL model is empirically efficient. We trained the RL for up to 1000 trials and stopped early if the number of batches reaches the lower bound (check every 50 iterations). The most onerous job (see in Table 3), which is to train for the lattice-based network, takes 22 seconds for 1000 iterations to train on a dataflow graph of 11626 nodes and results in 1040 states in the Q-table. On the static subgraph, batching policy is obtained by the grid search and the PQ tree optimization is applied afterward. Shown in Table 4, It takes tens of milliseconds to optimize the static subgraph.

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), inference (Fegade et al., 2021; Fegade, 2023; Zha et al., 2019), and serving (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 expressiveness of the state encoding enables FSM-based method to accomodate to 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 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 for dynamic DNNs. Also, we mitigated the memory transfer introduced by batching through the memory layout optimization based on the PQ-tree algorithm. The experimental results showed that our approach achieved significant speedup compared to current frameworks by the reduction in the number of batches and data movement.

Acknowledgements

We would like to thank all reviewers for very constructive comments that help us improve the paper. We thank Peking University's supercomputing team for providing the hardware to run the experiments and thank Kevin Huang of CMU for his help in early exploration of this research space.

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 2016, Savannah, GA, USA, November 2-4, 2016*, pp. 265–283, 2016. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

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., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., et al. Tvm: An automated end-to-end optimizing compiler for deep learning. *arXiv preprint arXiv:1802.04799*, 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, Beijing, China, July 2015. Association for Computational Linguistics. doi: 10.3115/v1/P15-1168. URL https://aclanthology.org/P15-1168.

Fegade, P. Auto-batching Techniques for Dynamic Deep Learning Computation, PhD thesis. 1 2023. doi: 10.1184/R1/21859902.v1. 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. Springer, 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, March 8-11, 1994*, 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. *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: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, 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 *Proceedings of the 2012 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 *Proceedings of the 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*, 2015a.

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*, 2015b.

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 *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pp. 937–950, USA, 2018. USENIX Association. ISBN 9781931971447.

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 Alg.1 to batch the whole dataflow graph. The *Batching* problem is to find a batch sequence with the smallest possible length, denoted as *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 that in the Alg.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. □

Till now, there is no constant guaranteed approximation algorithm for the SCS problem, and so does *Batching*.

A.2. Proof of the sufficient condition on batching

Lemma A.2. *If* $\frac{|Frontier(G_t^{a_t})|}{|Frontier_{a_t}(G_t)|} = 1$*, 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 $\frac{|Frontier(G_t^{a_t})|}{|Frontier_{a_t}(G_t)|} = 1$, 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. □

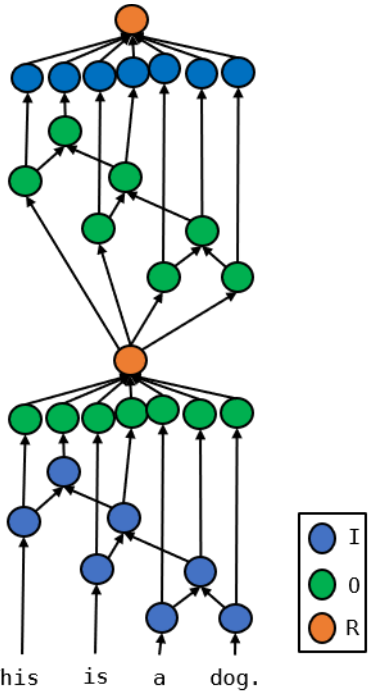


Figure 10. Example for when the FSM doesn't work.

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. And 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 the first tree requires batching Input node S_2 while the second requires batching the Output node. This problem can be solved by introducing the phase information like the portion of nodes committed into the state encoding.

B. PQ-tree

In this section, we illustrate the functions uncovered in Alg.2 and give the proof on its time complexity.

Detailed Illustration

The supporting functions for BROADCASTCONSTRAINT

Algorithm 3 PQ tree Memory Allocation

```
function BroadcastConstraint {$tree$, $\mathcal{B}$}
  visited = getSet()
  for $batch$ in $\mathcal{B}$ do
    if batch in visited then
      continue
    end if
    Q = Queue()
    Q.push($batch$)
    while not Q.isEmpty() do
      $b$ = Q.pop();
      visited.insert($b$)
      $cons$ = ParseConstraints ($b$)
      $suc$, $updatedBatches$ =ApplyConstraints ($cons$,$tree$)
      if $suc$ is False then
        $\mathcal{B}$.erase(b)
      else
        for $b$ in $updatedBatches$ do
          Q.push($b$)
        end for
      end if
    end while
  end for
end function
function DecideNodesOrder {$tree$, $\mathcal{B}$}
  $POrder$ = getUnionFindSet(tree.PNodes) {A union-find set to decide QN-node's direction.}
  $QOrder$ = getUnionFindSet(tree.QNodes) {A union-find set to decide PN-node's permutation.}
  for $batch$ in $\mathcal{B}$ do
    $EquivPairs$ = ParseEquivNodeOrderPair ($tree$, $batch$)
    for $EquivPair$ in $EquivPairs$ do
      if $EquivPair$ is a P node pair then
        POrder.Union($EquivPair$)
      else if $EquivPair$ is a Q node pair then
        QOrder.Union($EquivPair$)
      end if
    end for
  end for
  return $QOrder$, $POrder$
end function
function main {$X$, $\mathcal{B}$}
  {$X$ the variable set, $\mathcal{B}$ the batches }
  $tree$ = ConstructPQTree ($X$, $\mathcal{B}$)
  BroadcastConstraint ($tree$, $\mathcal{B}$)
  $QOrder$, $POrder$ = DecideNodesOrder ($tree$, $\mathcal{B}$)
  return GetLeafOrder ($tree$, $QOrder$, $POrder$)
end function
```

Algorithm 4 Algorithms for functions in BroadcastConstraint

```
function getSubtreeCons {$o$}
  $root$ = FindRoot ($o$)
  $nodeToLeaves$ = getNodeToLeaves (root) {A function maps nodes to leaves in its subtree. Realized by a traversal of the tree on the recursive function $nodeToLeaves(node) = node.isLeaf?\{node\} : \{nodeToLeaves[child]|child \in node.children\}$}
  $constraints = getList()$
  for $node$ in getNodesInSubTree (root) do
    if $node$ is P-node then
      $cons = \cup_{child \in node.children} nodeToLeaves(child)$
      $constaints.push(cons)$
    else if $node$ is Q-node then
      for $child \in node.children$ do
        $sib = child.nextSibling()$
        $cons = \cup\{nodeToLeaves(child), nodeToLeaves(sib)\}$
        $constraints.push(cons)$
      end for
    end if
  end for
  return $constraints$
end function
function ParseConstraints {$constraints$, $batch$}
  $uniformConstraints = \cup_{o \in batch.operands}\{o.index(x)|x \in getSubtreeCons(o)\}$ {Parse operand-wise consecutive constraint.}
  $constraints = getList()$
  for $o$ in $batch.operands$ do
    {Transform constraint by alignment information.}
    for $cons$ in $uniformConstraint$ do
      $constraints.append(\{o[x]|x \in cons\})$
    end for
  end for
  return constraints
end function
function ApplyConstraints {$constraints$, $tree$,$updatedOperands$}
  for $cons$ in $constraints$ do
    $suc$ = ReduceAndGetChanged ($tree$, $cons$,$updatedOperands$)
    if $suc$ is False then
      return False
    end if
  end for
  return True
end function
```

are shown in Alg.4. 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.

The ParseEquivNodeOrderPair method is given in Fig.5 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 Alg.6. In this problem, the UnionFindSet data structure is a set of nodes, and each node has two attributes:1. $parent$, the pointer to the node's parent, or the decider of its order; 2. 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 relationship between their roots. If their roots are not the same, an edge connects them with the transformation satisfying the information (line 6). If they are the same, then $node_1, node_2$'s relative order to the root must satisfy the constraint (line 6). 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 (Alg.7).

Algorithm 5 ParseEquivNodeOrderPair

```
function ParseEquivNodeOrderPair $\{tree, batch\}$
    Q = getQueue() {Queue on equivalent nodes.}
    for $i$ in $batch.operands.front().size()$ do
        Q.push(($o[i]|o \in batch.operands$))
    end for
    EquivNodeOrderPairs = getList()
    Find the root and calculate the leaf count for the subtree of the first operand.
    while True do
        {A leaf-to-root search performed parallel on operands in one batch.}
        $nodes$ = Q.pop()
        $node = nodes.front()$
        if $node$ is P node then
            $EquivClass = \{(node, node.referenceRrder)|node \in$
          $nodes\}$
            $EquivNodeOrderPairs.add((P, EquivClass))$
        else if $node$ is Q node then
            $EquivClass =$
            $\{(node, getDirection(node, node.referenceOrder))|node \in$
          $nodes\}$
            $EquivNodeOrderPairs.add((Q, EquivClass))$
        end if
        $node.parent.leafCnt =$
        $node.parent.leafCnt - node.leafCnt$
        if $node.parent.leafCnt$ is 0 then
            Q.push(($node.parant|node \in nodes$))
        end if
        for node in nodes do
            {Reference Order is used to decide the node order.}
            node.parent.referenceOrder.append(node)
        end for
        if $node.isRoot$ then
            {Stop Condition: Root Found.}
            Break.
        end if
    end while
    return EquivNodeOrderPairs
end function
```

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, time complexity for PQ-tree construction is $O(\Sigma_{batch \in \mathcal{B}}|batch|)$. In the BroadcastConstraint pass, the while-loop body (line 3) can only perform O($\Sigma_{batch \in \mathcal{B}}|batch|$). 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

Algorithm 6 Extended Union-Find set algs

```
function getUnionFindSet $\{nodes\}$
    for node in nodes do
        node.parent = node
        node.$\sigma$ = I {Identical transformation}
    end for
end function
function Find $\{node\}$
    $\sigma = I$ {The order relative to the root.}
    while $node.parent$ is not $node$ do
        $\sigma = \sigma \circ node.\sigma$
        $node = node.parent$
    end while
    return node, $\sigma$
end function
function Union $\{node_1, \sigma_1, node_2, \sigma_2\}$
    $p_1, \sigma_3$ = Find $(node_1)$
    $p_2, \sigma_4$ = Find $(node_2)$
    if $p_1$ is not $p_2$ then
        $p_1.parent = p_2$
        $p_1.\sigma = \sigma_3^{-1}\sigma_4\sigma_2^{-1}\sigma_1$
    else if $\sigma_1^{-1}\sigma_2$ is $\sigma_3^{-1}\sigma_4$ then
        {Compatibale}
        Do nothing. Already equivalent.
    else
        {Incompatible.}
        return False
    end if
    return True
end function
```

store the relationship between the node and the batch, once it 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 tree memory allocation algorithm is $O(\Sigma_{batch \in batches}|batch| \max^2_{batch \in batches}|batch|)$.

□

C. More On 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 optimiza-

Algorithm 7 Get Memory Allocation Order

```
function GetLeafOrder $\{tree, POrder, QOrder\}$
  root = tree.root
  order = getList()
  S = getStack()
  S.push(root)
  while S.notEmpty() do
    {Depth first traversal}
    node = S.pop()
    if node is P node then
      p, $\sigma$ = POrder.find(node)
      for child in $\sigma(node.children)$ do
        GetLeafOrder (child)
      end for
    else if node is Q node then
      p, $direction$ = POrder.find(node)
      for child in node.children following $direction$ do
        GetLeafOrder (child)
      end for
    else
      {Leaf Node here}
      order.append(order)
    end if
  end while
  return order
end function
```

tion 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 doesn't 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 doesn't 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 optimization we used in cortex includes 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/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 faster than traditional framework 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). Shown in Table 6, Torch/TensorFlow are 29/37x slower than ED-Batch on CPU, and 80/280x slower on GPU.

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