Low Latency RNN Inference with Cellular Batching

Pin Gao*†‡
Tsinghua University

Lingfan Yu*
New York University

Yongwei Wu†
Tsinghua University

Jinyang Li
New York University

ABSTRACT

Performing inference on pre-trained neural network models must meet the requirement of low-latency, which is often at odds with achieving high throughput. Existing deep learning systems use batching to improve throughput, which do not perform well when serving Recurrent Neural Networks with dynamic dataflow graphs. We propose the technique of cellular batching, which improves both the latency and throughput of RNN inference. Unlike existing systems that batch a fixed set of dataflow graphs, cellular batching makes batching decisions at the granularity of an RNN "cell" (a subgraph with shared weights) and dynamically assembles a batched cell for execution as requests join and leave the system. We implemented our approach in a system called BatchMaker. Experiments show that BatchMaker achieves much lower latency and also higher throughput than existing systems.

CCS CONCEPTS

• **Computer systems organization** → *Data flow architectures*;

KEYWORDS

Recurrent Neural Network, Batching, Inference, Dataflow Graph

ACM Reference Format:
Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. 2018. Low Latency RNN Inference with Cellular Batching. In *EuroSys '18: Thirteenth EuroSys Conference 2018, April 23–26, 2018, Porto, Portugal.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3190508.3190541

1 INTRODUCTION

In recent years, deep learning methods have rapidly matured from experimental research to real world deployments. The typical lifecycle of a deep neural network (DNN) deployment consists of two phases. In the *training* phase, a specific DNN model is chosen after many design iterations and its parameter weights are computed based on a training dataset. In the *inference* phase, the pre-trained model is used to process live application requests using the computed weights. As a DNN model matures, it is the inference phase that consumes the most computing resource and provides the most bang-for-the-buck for performance optimization.

Unlike training, DNN inference places much emphasis on low latency in addition to good throughput. As applications often desire real time response, inference latency has a big impact on the user experience. Among existing DNN architectures, the one facing the biggest performance challenge is the Recurrent Neural Network (RNN). RNN is designed to model variable length inputs, and is a workhorse for tasks that require processing language data. Example uses of RNNs include speech recognition [3, 22], machine translation [4, 46], image captioning [44], question answering [40, 47] and video to text [20].

RNN differs from other popular DNN architectures such as Multilayer Perceptrons (MLPs) and Convolution Neural Networks (CNNs) in that it represents *recursive* instead of fixed computation. Therefore, when expressing RNN computation in a dataflow-based deep learning system, the resulting "unfolded" dataflow graph is not fixed, but varies depending on each input. The dynamic nature of RNN computation puts it at odds with biggest performance booster—*batching*. Batched execution of many inputs is straightforward when their underlying computation is identical, as is the case with MLPs and CNNs. By contrast, as inputs affect the depth of recursion, batching RNN computation is challenging.

Existing systems have focused on improving training throughput. As such, they batch RNN computation at the granularity of unfolded dataflow graphs, which we refer to as *graph batching*. Graph batching collects a batch of inputs, combines their dataflow graphs into a single graph whose operators represent batched execution of corresponding operators in the original graphs, and submits the combined graph to the backend for execution. The most common form of graph batching is to pad inputs to the same length so that the resulting graphs become identical and can be easily combined. This is done in TensorFlow [1], MXNet [7] and PyTorch [34]. Another form of graph batching is to dynamically analyze a set of input-dependent dataflow graphs and fuse equivalent operators to generate a conglomerate graph. This form of batching is done in TensorFlow Fold [26] and DyNet [30].

Graph batching harms both the latency and throughput of model inference. First, unlike training, the inputs for inference arrive at different times. With graph batching, a newly arrived request must wait for an ongoing batch of requests to finish their execution completely, which imposes significant latency penalty. Second, when inputs have varying sizes, not all operators in the combined graph

*P. Gao and L. Yu equally contributed to this work.
†Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNLIST), Tsinghua University, Beijing 100084, China; Research Institute of Tsinghua University in Shenzhen, Guangdong 518057, China.
‡Work done while at New York University.

EuroSys '18, April 23–26, 2018, Porto, Portugal
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5584-1/18/04…$15.00
https://doi.org/10.1145/3190508.3190541

can be batched fully after merging the dataflow graphs for different inputs. Insufficient amount of batching reduces throughput under high load.

This paper proposes a new mechanism, called *cellular batching*, that can significantly improve the latency and throughput of RNN inference. Our key insight is to realize that a recursive RNN computation is made up of varying numbers of similar computation units connected together, much like an organism is composed of many cells. As such, we propose to perform batching and execution at the granularity of cells (aka common subgraphs in the dataflow graph) instead of the entire organism (aka the whole dataflow graph), as is done in existing systems.

We build the BatchMaker RNN inference system based on cellular batching. As each input arrives, BatchMaker breaks its computation graph into a graph of cells and dynamically decides the set of common cells that should be batched together for the execution. Cellular batching is highly flexible, as the set of batched cells may come from requests arriving at different times or even from the same request. As a result, a newly arrived request can immediately join the ongoing execution of existing requests, without needing to waiting for them to finish. Long requests also do not decrease the amount of batching when they are batched together with short ones: each request can return to the user as soon as its last cell finishes and a long request effectively hitches a ride with multiple short requests over its execution lifetime.

When batching and executing at the granularity of cells, BatchMaker also faces several technical challenges. What cells should be grouped together to form a batched task? Given multiple batched tasks, which one should be scheduled for execution next? When multiple GPU devices are used, how should BatchMaker balance the loads of different GPUs while preserving the locality of execution within a request? How can BatchMaker minimize the overhead of GPU kernel launches when a request's execution is broken up into multiple pieces?

We address these challenges and develop a prototype implementation of BatchMaker based on the codebase of MXNet. We have evaluated BatchMaker using several well-known RNN models (LSTM [24], Seq2Seq [38] and TreeLSTM [39]) on different datasets. We also compare the performance of BatchMaker with existing systems including MXNet, TensorFlow, TensorFlow Fold and DyNet. Experiments show that BatchMaker reduces the latency by 17.5-90.5% and improves the throughput by 25-60% for LSTM and Seq2Seq compared to TensorFlow and MXNet. The inference throughput of BatchMaker for TreeLSTM is 4× and 1.8× that of TensorFlow Fold and DyNet, respectively, and the latency reductions are 87% and 28%.

2 BACKGROUND

In this section, we explain the unique characteristics of RNNs, the difference between model training and inference, the importance of batching and how it is done in existing deep learning systems.

2.1 A primer on recurrent neural networks

Recurrent Neural Network (RNN) is a family of neural networks designed to process sequential data of variable length. RNN is particularly suited for language processing, with applications ranging

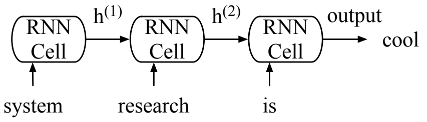


Figure 1: An unfolded chain-structured RNN. All RNN Cells in the chain share the same parameter weights.

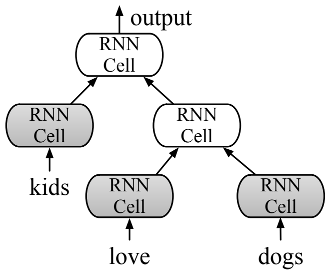


Figure 2: An unfolded tree-structured RNN. There are two types of RNN cells, leaf cell (grey) and internal cell (white). All RNN cells of the same type share the same parameter weights.

from speech recognition [3], machine translation [4, 46], to question answering [40, 47].

In its simplest form, we can view RNNs as operating on an input sequence, $X = [x^{(1)}, x^{(2)}, ..., x^{(\tau)}]$, where $x^{(i)}$ represents the input at the i-th position (or timestep). For language processing, the input X would be a sentence, and $x^{(i)}$ would be the vector embedding of the i-th word in the sentence. RNN's key advantage comes from parameter sharing when processing different positions. Specifically, let f_θ be a function parameterized with θ, RNNs represent the recursive computation $h^{(t)} = f_\theta(h^{(t-1)}, x^{(t)})$, where $h^{(t)}$ is viewed as the value of the hidden unit after processing the input sequence up to the t-th position. The function f_θ is commonly referred to as an RNN cell. An RNN cell can be as simple as a fully connected layer with an activation function, or the more sophisticated Long Short-Term Memory (LSTM) cell. The LSTM cell [24] contains internal cell state that store information and uses several gates to control what goes in or out of those cell state and whether to erase the stored information.

RNNs can be used to model a natural language, solving tasks such as predicting the most likely word following an input sentence. For example, we can use an RNN to process the input sentence "system research is" and to derive the most likely next word from the RNN's output. Figure 1 shows the unfolded dataflow graph for this input. At each time step, one input position is consumed and the calculated value of the hidden unit is then passed to the successor cell in the next time step. After unfolding three steps, the output will have the context of the entire input sentence and can be used to predict the next word. It is important to note that each RNN cell in the unfolded graph is just a copy, meaning that all unfolded cells share the same model parameter θ.

Although sequential data are common, RNNs are not limited to chain-like structures. For example, TreeLSTM [39] is a tree-structured RNN. It takes as input a tree structure (usually, the parse tree of a sentence [36]) and unfolds the computation graph to that structure, as shown in Figure 2. TreeLSTM has been used

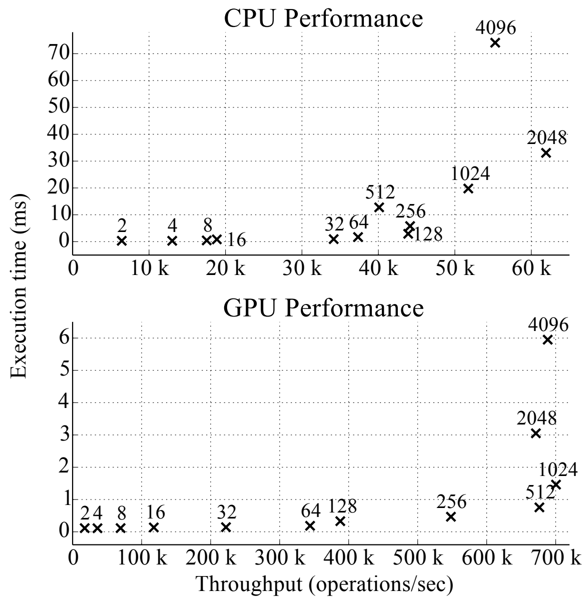


Figure 3: Latency vs. throughput for computing a single step of LSTM cell at different batch sizes for CPU and GPU. The value on the marker denotes the batch size.

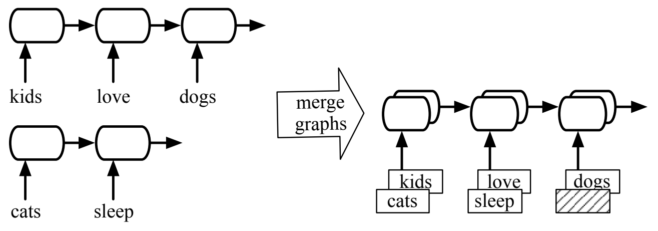
for classifying the sentiment of a sentence [33] and the semantic relatedness of two sentences [28].

2.2 Training vs. inference, and the importance of batching

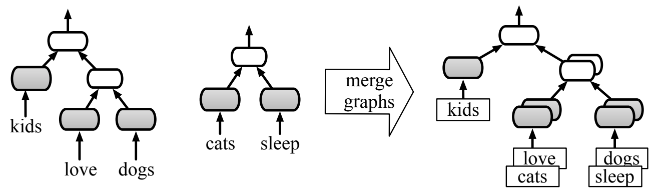
Deploying a DNN is two-phase process. During the offline *training* phase, a model is selected and its parameter weights are computed using a training dataset. Subsequently, during the online *inference* phase, the pre-trained model is used to process application requests.

At a high level, DNN training is an optimization problem to compute parameter weights that minimize some loss function. The optimization algorithm is minibatch-based Stochastic Gradient Descent (SGD), which calculates the gradients of the model parameters using a mini-batch of a few hundred training examples, and updates the parameter weights along computed gradients for the subsequent iteration. The gradient computation involves forward-propagation (computing the DNN outputs for those training samples) and backward-propagation (propagating the errors between the outputs and true labels backward to determine parameter gradients). Training cares about throughput: the higher the throughput, the faster one can scan the entire training dataset many times to arrive at good parameter weights. Luckily, the minibatch-based SGD algorithm naturally results in batched gradient computation, which is crucial for achieving high throughput.

DNN inference uses pre-trained parameter weights to process application requests as they arrive. Compared to training, there's no backward-propagation and no parameter updates. However, as applications desire real time response, inference must strive for low latency as well as high throughput, which are at odds with each other. Unlike training, there is no algorithmic need for batching during inference[1]. Nevertheless, batching is still required by inference for achieving good throughput.



(a) Graph batching via padding



(b) Graph batching in TensorFlow Fold and DyNet

Figure 4: Existing systems perform graph batching

To see the importance of batching for performance, we conduct a micro-benchmark that performs a single LSTM computation step using varying batch sizes (b)[2]. The GPU experiment uses NVIDIA Tesla V100 GPU and NVIDIA CUDA Toolkit 9.0. Figure 3 (bottom) shows the execution time of a batch vs. the overall throughput, for batch sizes $b = 2, 4, \dots 2048$. We can see that the execution time of a batch remains almost unchanged first and then increases sublinearly with b. When $b > 512$, the execution time approximately doubles as b doubles. Thus, setting $b = 512$ results in the best throughput. We also ran CPU experiments on Intel Xeon Processor E5-2698 v4 with 32 virtual cores. The LSTM cell is implemented using Intel's Math Kernel Library (2018.1.163). As Figure 3 (top) shows, batching is equally important for the CPU. On both the GPU and CPU, batching improves throughput because increasing the amount of computation helps saturate available computing cores and masks the overhead of off-chip memory access. As the CPU performance lags far behind that of the GPU, we focus our system development on the GPU.

2.3 Existing solutions for batching RNNs

Batching is straightforward when all inputs have the same computation graph. This is the case for certain DNNs such as Multi-layer Perceptron (MLP) and Convolution Neural Networks (CNNs). However, for RNNs, each input has a potentially different recursion depth and results in an unfolded graph of different sizes. This input-dependent structure makes batching for RNNs challenging.

Existing systems fall into two camps in terms of how they batch for RNNs:

(1) **TensorFlow/MXNet/PyTorch/Theano:** These systems pad a batch of input sequences to the same length. As a result,

[1]The SGD algorithm used in training is best done in mini-batches. This is because the gradient averaged across many inputs in a batch results in a better estimate of the true gradient than that computed using a single input.

[2]We configure the LSTM hidden unit size $h = 1024$. The LSTM implementation involves several element-wise operations and one matrix multiplication operation with input tensor shapes $b \times 2h$ and $2h \times 4h$.

each input has the same computation graph and the execution can be batched easily. An example of batching via padding is shown in Figure 4a. However, padding is not a general solution and can only be applied to RNNs that handle sequential data using a chain-like structure. For non-chain RNNs such as TreeLSTMs, padding does not work.

(2) **TensorFlow-Fold/DyNet:** In these two recent work, the system first collects a batch of input samples and generates the dataflow graph for each input. The system then merges all these dataflow graphs together into one graph where some operator might correspond to the batched execution of operations in the original graphs. An example is shown in Figure 4b.

Both above existing strategies try to collect a set of inputs to form a batch and find a dataflow graph that's compatible with all inputs in the batch. As such, we refer to both strategies as *graph batching*. Existing systems use graph batching for both training and inference. We note that graph batching is ideal for RNN training. First, since all training inputs are present before training starts, there is no delay in collecting a batch. Second, it does not matter if a short input is merged with a long one because mini-batch (synchronous) SGD must wait for the entire batch to finish in order to compute the parameter gradient anyway.

Unfortunately, graph batching is far from ideal for RNN inference and negatively affects both the latency and throughput. Graph batching incurs extra latency due to unnecessary synchronization because an input cannot start executing unless *all* requests in the current batch have finished. This is further exacerbated in practice when inputs have varying lengths, causing some long input to delay the completion of the entire batch. Graph batching can also result in suboptimal throughput, either due to performing useless computation for padding or failing to batch at the optimal level for all operators in the merged dataflow graph.

3 OUR APPROACH: CELLULAR BATCHING

We propose cellular batching for RNN inference. RNN has the unique feature that it contains many identical computational units connected with each other. Cellular batching exploits this feature to 1) batch at the level of RNN cells instead of whole dataflow graphs, and 2) let new requests join the execution of current requests and let requests return to the user as soon as they finish.

3.1 Batching at the granularity of cells

Graph batching is not efficient for inference because it performs batching at a coarse granularity–a dataflow graph. The recursive nature of RNN enables batching at a finer granularity–an RNN cell. Since all unfolded RNN cells share the same parameter weights, there is ample opportunity for batching at the cell-level: each unfolded cell of a request X can be batched with any other unfolded cell from request Y. In this way, RNN cells resemble biological cells which constitute all kinds of organisms. Although organisms have numerous types and shapes, the number of cell types they have is much more limited. Moreover, regardless of the location of a cell, cells of the same type perform the same functionality (and can be batched together). This characteristic makes it more efficient to batch at cell level instead of the organism (dataflow graph) level.

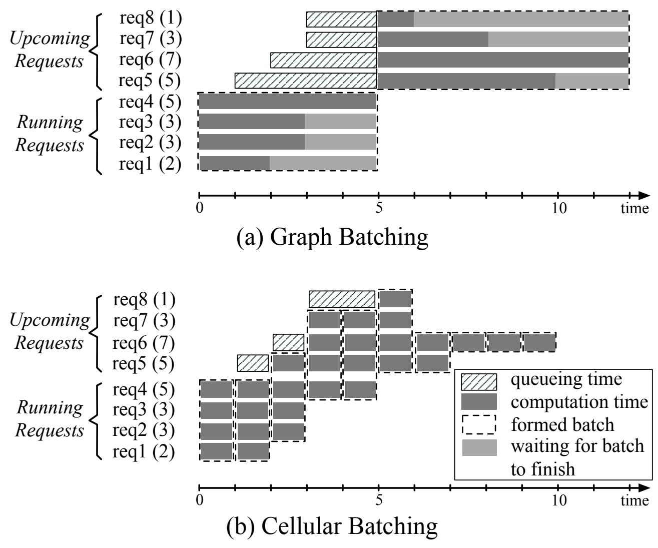


Figure 5: The timeline of graph batching and Cellular Batching when processing 8 requests from req1 to req8. The number shown in parenthesis is the request's sequence length, e.g. req1(2) means req1 has a sequence length of 2. Each row marks the lifetime of a request starting from its arrival time. Req1-4 are *Running Requests* as they arrive at time 0 and have started execution. Req5-8 are *Upcoming Requests* that arrive after the Req1-4.

More generally, we allow programmers to define a cell as a (sub-)dataflow graph and to use it as a basic computation unit for expressing the recurrent structure of an RNN. A simple cell contains a few tensor operators (e.g. matrix-matrix multiplication followed by an element-wise operation); a complex cell such as LSTM not only contains many operators but also its own internal recursion. Grouping operators into cell allows us to make the unfolded dataflow graph coarse-grained, where each node represents a cell and each edge depicts the direction in which data flows from one cell to another. We refer to this coarse-grained dataflow graph as cell graph.

There may be more than one type of cells in the dataflow graph. Two cells are of the same type if they have identical sub-graphs, share the same parameter weights, and expect the same number of identically-shaped input tensors. Cells with the same type can be batched together if there is no data dependency between them.

3.2 Joining and leaving the ongoing execution

In graph batching, the system collects a batch of requests, finishes executing all of them and then moves on to the next batch. By contrast, in cellular batching, there is no notion of a fixed batch of requests. Rather, new requests continuously join the ongoing execution of existing requests without waiting for them to finish. This is possible because a new request's cells at an earlier recursion depth can be batched together with existing requests' cells at later recursion depths.

Existing deep learning systems such as TensorFlow, MXNet and DyNet schedule an entire dataflow graph for execution. To support continuous join, we need a different system implementation that can dynamically batch and schedule individual cells. More concretely, our system unfolds each incoming request's execution into a graph of cells, and continuously forms batched tasks by grouping cells of the same type together. When a task has batched sufficiently many cells, it is submitted to a GPU device for execution. Therefore, as long as an ongoing request still has remaining cells that have not been executed, they will be batched together with any incoming requests. Furthermore, our system also returns a request to the user as soon as its last cell finishes. As a result, a short request is not penalized with increased latency when it's batched with longer requests.

Figure 5 illustrates the different batching behavior of Cellular Batching and graph batching when processing the same 8 requests. We assume a chain-structured RNN model and that each RNN cell in the chain takes one unit of time to execute. Each request corresponds to an input sequence whose length is shown in the parentheses. In the Figure, each row shows the lifetime of one request, starting from its arrival time. The example uses a batch size of 4.

In the beginning of time (t=0), the first 4 requests (req1-4) arrive. Under graph batching, these 4 requests form a batch and their corresponding dataflow graphs are fused together and submitted to the backend for execution. The system does not finish executing the fused graph until time t=5, as the longest request in the batch (req4) has a length of 5. In the meanwhile, newly arrived requests (req5-8) are being queued up and form the next batch. The system starts executing the next batch at t=5 and finishes at t=12. Under cellular batching, among the first 4 requests, the system forms two fully batched tasks, each performing the execution of a single (4-way batched) RNN cell. At t=2, the second task finishes, causing req1 to complete and leave the system. Since a new request (req5) has already arrived, the system forms its third fully batched task containing req2-5 at t=2. After finishing this task, another two existing requests (req2,req3) depart and two new ones are added (req6, req7) to form the fourth task. As shown in this example, cellular batching not only reduces the latency of each request (due to less queuing), but also increases the overall system throughput (due to tighter batching).

4 SYSTEM DESIGN

We build an inference system, called BatchMaker, based on cellular batching. This section describes the basic system design.

4.1 User Interface

In order to use BatchMaker, users must provide two pieces of information: the definition of each cell (i.e. the cell's dataflow graph) and a user-defined function that unfolds each request/input into its corresponding cell graph. We expect users to obtain a cell's definition from their training programs for MXNet or TensorFlow. Specifically, users define each RNN cell using MXNet/TensorFlow's Python interface and save the cell's dataflow graph in a JSON file using existing MXNet/TensorFlow facilities. The saved file is given to BatchMaker as the cell definition. In our current implementation, the user-defined unfolding logic is expressed as a C++ function which uses our given library functions to create a dataflow graph of cells.

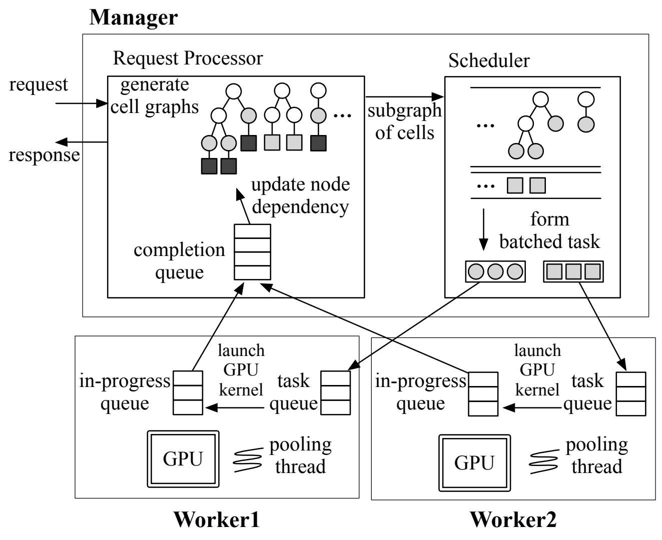


Figure 6: The system architecture of BatchMaker. In the cell graph, black means computed nodes, grey means nodes whose input is ready, and white means input dependency is not satisfied.

4.2 Software Architecture

BatchMaker runs on a single machine with potentially many GPU devices. Its overall system architecture is depicted in Figure 6. BatchMaker has two main components: *Manager* and *Worker*. The manager processes arriving requests and submits batched computation tasks to workers for execution. Depending on the number of GPU devices equipped, there may be multiple workers, each of which is associated with one GPU device. Workers execute tasks on GPUs and notify the manager when its tasks complete.

System initialization. Upon startup, BatchMaker loads each cell's definition and its pre-trained weights from files. BatchMaker "embeds" the weights into cells so that weights are part of the internal state as opposed to the inputs to a cell. For a cell to be considered batchable, the first dimension of each of its input tensors should be the batch dimension. BatchMaker identifies the type of each cell by its definition, weights, and input tensor shapes.

The workflow of a request. The manager consists of two submodules, *request processor* and *scheduler*. The request processor tracks the progress of execution for each request and the scheduler determines which cells from different requests would form a batched task, and selects a worker to execute the task.

When a new request arrives, the request processor runs the user-code to unfold the recursion and generates the corresponding cell graph for the request. In this cell graph, each node represents a cell and is labeled with a unique node id as well as its cell type. Request processor will track and update the dependencies of each node. When a node's dependencies have been satisfied (aka its inputs are ready), the node is ready to be scheduled for execution (§4.3). The scheduler forms batched tasks among ready nodes of the same cell type. Each type of cell has a desired maximum batch size, which is determined through offline benchmarking. Once a task has reached a desired batch size, it is pushed into the task queue of one of the workers.

Workers execute tasks on GPUs. Since the GPU kernel execution is asynchronous, the worker moves a task from the task queue to the in-progress queue once the task's corresponding GPU kernel has been issued. The worker uses a pooling mechanism to see whether some task has finished. It pops the finished task from the in-progress queue and pushes it into the completion queue in the request processor. The request processor updates node dependencies based on the completed task, and checks whether a request is finished. If so, its result is immediately returned.

We give a more detailed example of the request workflow in §4.4.

4.3 Batching and Scheduling

The scheduler needs to make decisions on what nodes should be batched together to form a task and what tasks to be pushed to which workers. The design must take into account three factors, locality, priority, and utilization of multiple GPUs, which are often in conflict with each other.

Locality refers to the preference that 1) the same set of requests should be batched together if they are to execute the same sequence of nodes, and 2) the execution of that sequence of nodes should stick to the same GPU. The reason for both 1) and 2) is to avoid memory copy. Prior to execution, the batched inputs of a cell must be laid out in contiguous GPU memory. Since the batched outputs of the execution are also stored in contiguous memory, there is no need for memory copy before each individual cell execution when executing the same set of requests on the same GPU. Conversely, if the batch of requests changes between two successive cell execution, one must do memory copy, called "gather", to ensure contiguous inputs. Furthermore, if the execution of successive cells switch from one GPU to another, one must copy data from one GPU to another.

Priority refers to the ability to prefer the execution of one type of cell over another. Many practical RNN models have multiple types of cells. For example, as shown in Figure 2, TreeLSTM has leaf cells and internal cells. The popular RNN-based Seq2Seq model has encoder cells and decoder cells. For these models, one can achieve better latency by preferentially executing DNN types that occur later in the computation graph. Therefore, in TreeLSTMs, internal nodes should be given preference over leaf nodes. In Seq2Seq models, decoder nodes should have priority over encoder nodes.

We design a simple scheduling policy to make the trade-off between locality, priority and good utilization of multiple GPUs. We support the locality preference by constructing and scheduling a batched task containing multiple node invocations instead of a single one. To enable this, the request processor analyzes the cell graph of a request to find a *subgraph* to pass to the scheduler. A *subgraph* contains a single node or a number of connected nodes with the property that all external dependencies to other parts of the graph have been satisfied. Furthermore, all nodes of a subgraph must be of the same cell type. For example, in the case of Seq2Seq, a sequence of encoders cells forms one subgraph and the sequence of decoder cells forms another subgraph.

Algorithm 1: Scheduling and Batching Algorithm

```
1  Bsizes: a set of supported batch sizes.
2  CellTypes: a set of cell types, each associated with a priority.
3  MaxTasksToSubmit: the maximum number of tasks that can
     be submitted to a worker.
4  def Schedule(worker):
5      $S \leftarrow \{ct \in$ CellTypes $\mid ct$.NumReadyNodes() $\geq$
         Bsizes.Max()$\}$;
6      if $S$.Size() = 0 :
7          $S \leftarrow \{ct \in$ CellTypes $\mid ct$.NumRunningTasks() = 0 and
             $ct$.NumReadyNodes() > 0$\}$;
8      if $S$.Size() = 0 :
9          $S \leftarrow \{ct \in$ CellTypes $\mid ct$.NumReadyNodes() > 0$\}$;
10     $ct \leftarrow$ GetCellTypeWithMaxPriority($S$);
11     Batch($ct$, $worker$);
12 def Batch(ct, worker):
13     $num\_tasks \leftarrow 0$;
14     while $num\_tasks$ < MaxTasksToSubmit :
15         $batch \leftarrow$ FormBatchedTask($ct$, $worker$);
16         if $batch$.Size() >= Bsizes.Min() or $num\_tasks = 0$ :
17             SubmitBatchedTask($batch$, $worker$);
18             UpdateNodesDependency($batch$);
19             $num\_tasks$++;
20             for $subgraph \in batch$ :
21                 $subgraph.pinned \leftarrow worker.id$;
                   # pinned is unset once subgraph has no
                     task running
22         else:
23             break;
24 def FormBatchedTask(ct, worker):
25     $batch \leftarrow \{\}$;
26     for $subgraph \in ct.subgraphs$ :
27         if $subgraph.pinned \in \{None, worker.id\}$ :
28             for $node \in$ subgraph's ready nodes :
29                 $batch \leftarrow batch \bigcup \{node\}$;
30                 if $batch$.Size() = Bsizes.Max() :
31                     return $batch$;
32     return $batch$;
```

Scheduling subgraphs. The scheduling algorithm is shown in Algorithm 1. For each type of cell, scheduler maintains a queue of subgraphs (the type of a cell is the same as the type of nodes in the subgraphs). The scheduler's "*Schedule*"' function (line 4) is invoked whenever some worker becomes idle, and it picks a cell type *ct* for execution in the following order: (a) *ct* whose queue contains more ready nodes (meaning nodes whose data dependency is satisfied) than the maximum batch size (line 5); (b) *ct* whose queue contains some ready nodes but that has no running tasks (line 6-7); (c) *ct* whose queue contains some ready nodes (line 8-9). If there are multiple choices from the same criterion, the scheduler chooses the one with the highest cell priority (line 10). Once a cell type is chosen, scheduler invokes "*Batch*" to form batched tasks (line 11).

Batching subgraphs. Given a cell type, "*Batch*" function (line 12) selects nodes from subgraphs in the cell's queue to form a batched task (line 15) and to submit to the device for execution (line

17). "*FormBatchedTask*" function (line 24) scans the queue to select nodes whose dependencies are satisfied (line 28) for a batched task. Each invocation of "*FormBatchedTask*" forms at most one batched task. For better locality, the scheduler submits several tasks to the same worker for execution. The number of tasks submitted is limited by the configurable parameter "MaxTasksToSubmit" (line 14). Setting the limit to a small number (default is 5) avoids forming too many tasks belonging to one cell type, which gives other types of cell a chance to be scheduled, allows new requests to join execution, and avoids the decrease in effective batch size if one cell type does not have enough ready nodes.

Once a batched task is submitted to the GPU worker, the scheduler updates the dependencies of those nodes in the batch (line 18) so a new set of cell nodes can be scheduled in subsequent batched tasks. Additionally, the scheduler pins those subgraphs to the same worker (line 20-21) to avoid scheduling the subsequent nodes in those subgraphs to a different worker (line 27). This is crucial to the correctness of the scheduling algorithm because tasks involving the same subgraph have data dependency. Tasks submitted to the same device will execute in the order of submission, and thus their dependency is fulfilled. By contrast, there is no such guarantee for tasks submitted to different workers. Besides, this also improves locality because all nodes in the same subgraph are preferentially scheduled to the same worker. The scheduler maintains a counter for each subgraph to count how many batched tasks contain nodes from this subgraph. When this counter is decreased to zero, the scheduler unpins the subgraph from a worker so that this subgraph may be scheduled on other workers in the future.

4.4 An example of TreeLSTM scheduling

To give a concrete example, we show the detailed workflow of a TreeLSTM request. When a TreeLSTM request x arrives at our system, the request processor applies the user-defined unfolding function to generate the cell graph for request x. Then the request processor analyzes the cell graph and breaks it up into subgraphs based on cell types. A TreeLSTM model has two types of cell: leaf cell and internal cell. Suppose request x is a complete binary tree with 16 leaf nodes. Then its cell graph will be partitioned into 17 subgraphs: one subgraph contains 31 internal tree nodes; each of the other 16 subgraphs contains a single leaf node. The set of leaf subgraphs are immediately passed to the scheduler because their dependencies are satisfied, whereas the subgraph of internal nodes remains at the request processor.

The scheduler maintains two queues, one for the internal cell type, the other for the leaf cell type. When the leaf cell type is scheduled, leaves of x will be put into potentially several batched tasks with leaf nodes from other requests. Batched tasks are pushed to the worker and executed. The request processor is notified when subgraphs finish. When all the leaf subgraphs of request x finish, the subgraph containing x's internal nodes has its dependencies satisfied and is then passed to the scheduler. When the internal cell type is scheduled, the scheduler puts the cells of x at successive levels of the tree in successive batched tasks to ensure that their dependencies are obeyed. The number of nodes from request x decreases at higher tree levels. But the scheduler will batch nodes from request x with nodes from other requests to keep the batch size close to the maximum allowed. Once all internal nodes of request x finish execution, the request processor gets notified. When there is no more subgraph to execute, request x departs immediately.

5 GPU OPTIMIZATION

The manager and workers threads in BatchMaker run on the CPU and RNN cells are scheduled to execute on the GPUs. The synchronization between CPU and GPU is non-trivial and has a big impact on the utilization of GPUs. In this section, we explain two optimizations in BatchMaker that are crucial for achieving good performance on GPUs.

Keeping the GPU busy. One should not schedule a GPU kernel for execution one at a time. Doing so is terrible for performance as the GPU sits idle waiting for the next kernel to be scheduled and launched. In BatchMaker, the worker asynchronously pushes all GPU kernels for a given task to the GPU's driver queue without waiting for any to finish. To ensure that the dependencies of each kernel are satisfied, the worker performs a topological sort of all operators within the cell and pushes kernels according to the sort order to the same GPU stream. This works because the GPU driver guarantees that kernels in the same stream are executed in the FIFO order. A worker may receive up to *MaxTaskToSubmit* number of tasks from the scheduler. The order in which the workers receive these tasks already correctly reflect the dependencies between cells. Thus, the worker also launches the kernels for multiple tasks based on their order in the task queue. By launching as many kernels as possible while obeying dependencies, we effectively reduce the kernel launch gap between operators and tasks.

Asynchronous Completion Notification. The worker cannot synchronously wait for a task to finish execution on the GPU. Nevertheless, the worker must learn quickly when a task has finished so it can inform the manager who will issue the next set of nodes to the scheduler. Existing solutions supporting asynchronous notification use the callback mechanism provided by GPU device drivers. However, these callback mechanisms have performance limitations. For example, the NVIDIA CUDA driver's callback mechanism blocks all kernel execution until the callback function finishes[15].

In order to let the worker learn of task completion asynchronously, we add a *signaling kernel* to the end of each task. The signaling GPU kernel changes a signal variable, which is an unsigned integer in our implementation. The signal variable is allocated in pinned host memory which can be accessed by GPU using zero copy. Whenever a task finishes execution, the signaling kernel will execute next and increase the signal variable by one, which means the GPU has finished the execution of one more task. On the worker side, it pushes the tasks that have been issued to GPU in a FIFO queue called *in-progress queue*. The worker uses a thread to continuously poll the status of the signal variable. Once the signal variable changes, the worker learns that the task at the top of the in-progress queue has been finished. It then pushes the completed task to the manager's completion queue.

6 IMPLEMENTATION

We implemented our system using the codebase of MXNet (version 0.10.0). In our current prototype, users need to provide the definition of each cell in a JSON file exported by MXNet's Python API. Users also need to provide a user-defined C++ function to generate the unfolded cell graph for each request.

During initialization, BatchMaker aims to materialize all the cells for each supported batch size on every available GPU device. Such materialization requires knowledge of the type and shape information of each operator's input/output tensors in order to allocate GPU memory and perform other compiler-level optimizations such as those done by NNVM [10], TensorFlow XLA [11].

We reuse the MXNet parsing mechanism to perform type and shape inference for each type of cell. However, in order to do that, BatchMaker needs to know how cells can be connected, and how the outputs of one cell may be used by other cells. Ideally, BatchMaker should learn this knowledge on the fly when real requests come in. To simplify implementation, our current prototype requires the user to provide an example request so that BatchMaker can apply the user-defined unfolding function to generate an example cell graph. This allows BatchMaker to perform type and shape inference and materialize cells during initialization.

During inference, BatchMaker re-use materialized cells over and over. For example, to execute an LSTM chain with length 5, our worker will execute the same materialized LSTM cell for 5 times.

7 EVALUATION

We evaluate BatchMaker on microbenchmarks and several popular RNN applications with real-world datasets. Our evaluation shows that BatchMaker provides significant performance advantages over existing systems (including MXNet, TensorFlow, TensorFlow Fold and DyNet).

The highlights of our results are:

- BatchMaker achieves much lower latency than existing systems. Under moderate load (meaning that the load is less than half of baseline system's peak), we reduce the 90-percentile latency by 37.5%-90.5% (for LSTM) and 17.5%-82.6% (for Seq2Seq) compared to TensorFlow and MXNet. For TreeLSTM, we reduce 90-percentile latency by 28% and 87% compared to DyNet and TensorFlow Fold respectively.
- BatchMaker also provides good throughput improvements. The throughput improvement over MXNet and TensorFlow is 25% (for LSTM) and 60% (for Seq2Seq). For TreeLSTM, the throughput of BatchMaker is 1.8× that of DyNet and 4× that of TensorFlow Fold.
- Our latency improvement mainly comes from reducing the queuing time of new requests. The performance advantage of BatchMaker is increased when the variance in the sequence length is large.

7.1 Experimental Setup

The Testbed. We run our tests on a Linux server with 4 NVIDIA TESLA V100 GPU cards connected by NVLink; each GPU has 16GB memory. The operating system is Ubuntu 16.04.1 LTS with Linux kernel version 4.13.0. NVIDIA CUDA Toolkit version is 9.0.

Applications, datasets, and workloads. We choose three popular RNN applications, LSTM, Seq2Seq, and TreeLSTM. All RNN cells used in these applications use hidden state size 1024. LSTM and Seq2Seq are both chain-structured RNNs. We use WMT-15 [42] Europarl German-English translation as our dataset. For LSTM, we randomly sample 100k English sentences. For Seq2Seq, we randomly sample 100k German-English sentence pairs. The maximum sentence length is 330 and the average length is 24. For TreeLSTM, We use Stanford's TreeBank [37] dataset with 10K parse trees of English sentences.

When evaluating an application, we sample a request from the dataset and issue it to the system with Poisson inter-arrival times. We adjust the average inter-arrival time to test the system's performance under varying load.

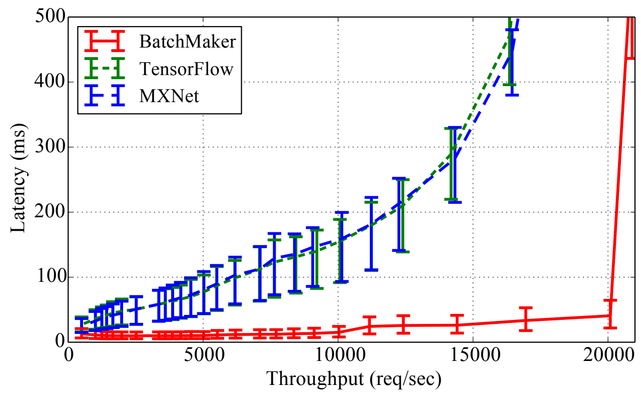
Comparison systems. We compare against MXNet (v0.12.0), TensorFlow (v1.4), TensorFlow Fold (v0.0.1) and DyNet (v2.0). MXNet and TensorFlow are representative systems that rely on padding to achieve batching. TensorFlow Fold and DyNet are two existing systems that perform graph batching by dynamically merging a set of dataflow graphs. For chain-structured RNNs, MXNet and TensorFlow achieve much better performance than TensorFlow Fold and DyNet and thus we focus on comparing BatchMaker to these two systems for LSTM and Seq2Seq benchmarks. As padding does not work with tree-structured RNNs, we focus on comparing BatchMaker to TensorFlow Fold and DyNet for the TreeLSTM benchmarks.

Bucketing optimization for MXNet and TensorFlow. Since padding wastes computation, we reduce the amount of padding in MXNet and TensorFlow by only batching requests of similar lengths. We refer to this as the "bucketing" strategy. Specifically, we assign each incoming request to a bucket based on its length. The width of a bucket refers to the maximum difference in lengths among requests in a bucket. We use the bucket width of 10 by default, which gives the best performance for our applications (§7.2). Since the WMT-15 dataset has a maximum sentence length of 330, using a width of 10 results in 33 buckets in total. The i-th bucket handles requests of length in the range (i*10,(i+1)*10]. We perform round-robin scheduling across buckets. To reduce latency when running in MXNet and TensorFlow, we materialize a dataflow graph for each bucket during initialization. This is because the cost of materializing a dataflow graph is substantial, owing to compiler optimization and GPU memory allocation.

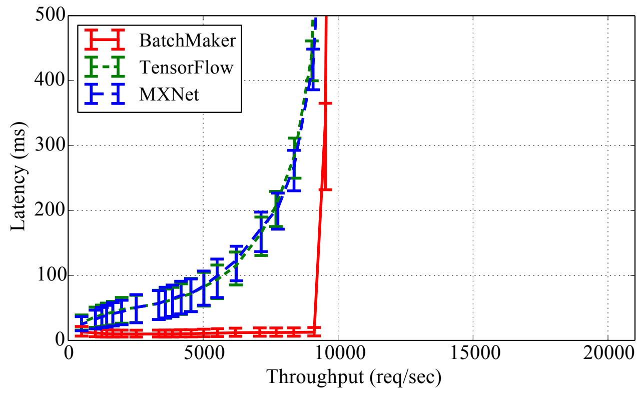
Batching configuration and optimization. Unless otherwise mentioned, we set the maximum batch size to be $b_{max} = 512$, which optimizes for throughput based on Figure 3 (bottom). In our evaluation of MXNet and TensorFlow, we do not use explicit timeouts when accumulating requests to form a batch; rather, even if it's not full, a batch can start execution (as a smaller batch) as long as some GPU device is idle and it is the batch's turn to execute according to the round-robin policy. As a result, when the request rate is low, the actual batch size executed in a system could be lower than the configured maximum. As we will demonstrate in (§7.2), this enables a large b_{max} to achieve the same low latency as a small b_{max}. Additionally, we found that this strategy achieves lower latency than any configuration of the timeout-based strategy.

7.2 Application Performance: LSTM

We evaluate LSTM inference performance on the WMT-15 Europarl dataset and compare BatchMaker with MXNet and TensorFlow.



(a) LSTM with maximum batch size 512



(b) LSTM with maximum batch size 64

Figure 7: LSTM performance on the WMT-15 Europarl dataset using 1 GPU. The figures plot the 90-percentile latency with error bars measuring the 99p and 50p latency. MXNet and TensorFlow use the bucket width of 10.

Figure 7 shows the throughput vs. latency results of all systems as the load increases (by reducing the request inter-arrival time). We set the bucket width of 10 by default for MXNet and TensorFlow.

Latency. Figure 7 (a) shows the 90-percentile latency vs. throughput (b_{max} = 512). The error bars represent the 50-percentile and the 99-percentile latency. As can be seen in the figure, BatchMaker achieved significantly lower latency than MXNet and TensorFlow. The 90p-latency of BatchMaker stays unchanged (12ms) when the throughput is less than 8K req/sec, and goes slightly up afterwards until peak throughput (20K req/sec). This is because when the load is moderate(< 8K req/sec), BatchMaker executes most requests in batch sizes no larger than 64. And as the throughput goes up, the batch sizes increase (up to b_{max} = 512), leading to the gradual increase in latency. By comparison, the smallest 90p-latency of MXNet and TensorFlow is 25ms and the latency increases quickly as the request rate increases. BatchMaker reduces latency because it allows incoming requests to join the currently executing batch, resulting in less queuing time. By comparison, in MXNet and TensorFlow, a new request may need to wait for multiple batches from different buckets to finish execution. Thus, the latency of MXNet and TensorFlow is much higher and increases quickly with increasing load. The latency variance for BatchMaker is also much smaller and is

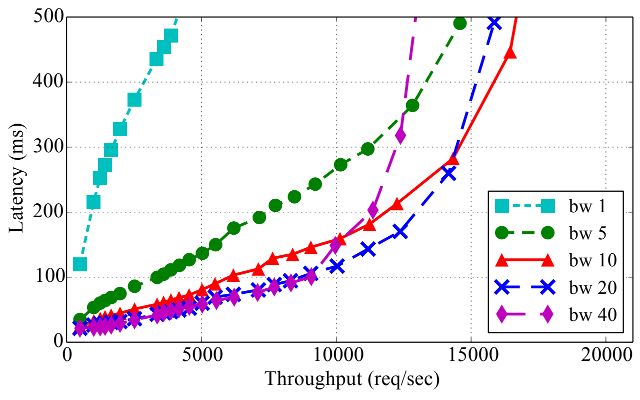


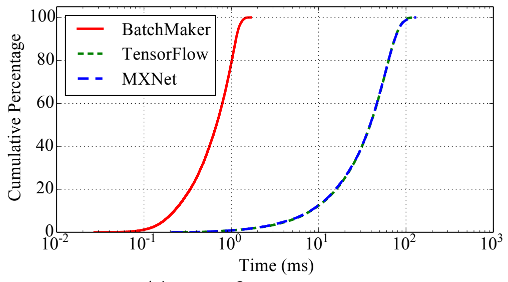
Figure 8: LSTM performance using MXNet with different bucket widths (bw-*). The maximum batch size is 512. For readability, we omit error bars and only show the 90-percentile latency.

caused by varying request sequence lengths. In § 7.3, we conduct additional experiments to confirm the reasons for BatchMaker's latency improvements.

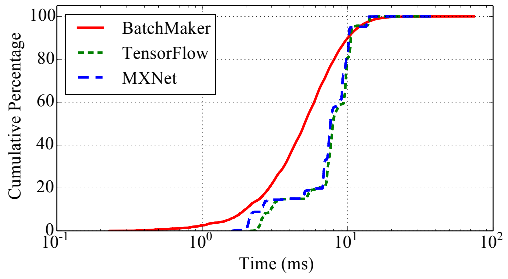
Throughput. In Figure 7 (a), the peak throughput of BatchMaker is 20K req/sec (b_{max} = 512), higher than those of MXNet and TensorFlow. As the load increases in MXNet and TensorFlow, a request must wait for more buckets to finish execution and each bucket also executes a larger batch of requests at a time. This causes the overall latency to shoot up beyond 500ms as the load increases to 16K req/sec. By contrast, BatchMaker is able to maintain a low queuing delay while packing more requests into a batch as the input load increases, resulting in much a higher peak throughput.

Effect of different maximum batch sizes. Figure 7 (b) shows the latency vs. throughput results using a smaller maximum batch size, b_{max} = 64. 64 and 512 are interesting batch size choices, because any batch size b < 64 has similar latency (for executing one step of LSTM) but lower throughput (than that of b = 64), and any batch size b > 512 has similar throughput but higher latency (than that of b = 512), as shown micro-benchmark in Figure 3 (bottom). Comparing Figure 7 (a) with (b), we see that b_{max} = 512 achieves similar latency as b_{max} = 64 (at low to moderate load) but much higher throughput. This is because at low load, all systems execute with effective batches sizes much smaller than the configured maximum. Therefore, the optimal configuration for all systems is to set the maximum batch size that optimizes for the throughput.

The bucket width trade-off in MXNet and TensorFlow. The granularity of buckets creates a trade-off between throughput and latency. Fine-grained bucketing reduces padding and wasteful computation. However, fine-grained bucketing uses a large number of buckets. This causes a batch for any given bucket to wait longer for batches from other buckets to finish execution before it catches its turn under the round-robin policy. By contrast, coarse-grained bucketing uses fewer buckets which results in shorter waiting time but increases the amount of padding and wasteful computation.



(a) CDF of queuing time



(b) CDF of computation time

Figure 9: Request queuing and computation time for LSTM on WMT-15 Europarl dataset under low load (5K req/sec).

Figure 8 shows the latency vs. throughput for MXNet under varying bucket widths. As the figure shows, coarse-grained bucketing (width 40) achieves better latency under low load (due to less waiting) but its peak throughput is also lower (due to more padding). On the contrary, using the smallest bucket width of 1 has the best peak throughput, but at the cost of higher latency for low to moderate load. Using the bucket width of 10 achieves a good tradeoff of low latency and high peak throughput.

7.3 Reasons for the performance gain

We investigate in more details the reasons why BatchMaker achieves better latency and throughput over baseline systems.

Reasons for the latency improvement. BatchMaker reduces the latency of a request in two aspects: 1) it reduces the queuing time by allowing a newly arrived request to join the execution of existing requests. 2) it reduces computation time by allowing shorter requests to be returned immediately upon completion without waiting for the longer ones. Figure 9 (a) and (b) shows the CDF of queuing time and computation time for LSTM on the WMT-15 Europarl dataset. Queuing time is measured from a request's arrival to its start of execution. Computation time is measured from a request's start of execution to the return of the execution result by the system. The lines in Figure 9 correspond to the points in Figure 7 (a) where the throughputs of all three systems are $\tilde{5}$K req/sec. The x-axis is shown in log scale.

In Figure 9(a), the 99-percentile queuing time for BatchMaker is 1.38 milliseconds, compared to > 100 milliseconds for MXNet and TensorFlow. In BatchMaker under low load, a newly arrived request waits for the current set of tasks to finish before joining the execution of existing batch of requests. With the input load 5K req/sec, we find that BatchMaker executes LSTM cells with batch size 64 most of the time, which takes takes about 185 microseconds

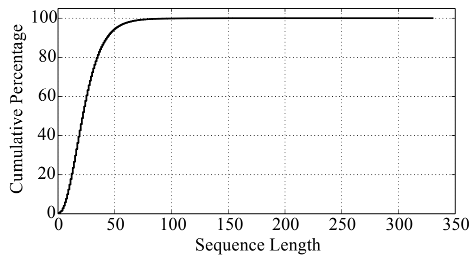


Figure 10: CDF of the sequence length in the WMT-15 Europarl dataset.

in microbenchmarks (Figure 3(bottom)). Due to scheduling and gathering overhead, BatchMaker needs about 250 microseconds to execute an LSTM step. Since BatchMaker submits at most 5 steps of LSTM cell to GPU (by setting the "MaxTasksToSubmit" in Algorithm 1 line 3), the incoming request can wait up to 0.25*5 = 1.25 milliseconds, which roughly matches BatchMaker's 99-percentile queuing time (1.38 ms). The queuing time of MXNet and TensorFlow is much larger; not only an incoming request has to wait for many buckets (out of 33 total buckets) to complete execution, but also each bucket's execution takes as many LSTM steps as the longest sequence in the batch.

As Figure 9 (b) shows, the computation time of BatchMaker is also less than that of MXNet and TensorFlow. Bucketing results in CDF lines with "jumps", as sequences with different lengths within the range of a bucket will be padded to the identical length to form a batch and complete their execution at the same time. When the bucket width is set to 10 for MXNet and TensorFlow, a request of length 21 will be padded to length 30, resulting in almost 50% padding overhead and latency increase. By contrast, BatchMaker allows any request that has completed its execution to be returned immediately, with the tradeoff of having to incur scheduling and gathering overhead in the middle of a request execution.

Comparing Figure 9(a) and (b), we can see that reduced queuing time is the more dominant factor for BatchMaker's latency improvement.

The impact of variable-length sequences. We examine how BatchMaker and baseline systems perform on artificial datasets with different variance of sequence lengths. Figure 10 shows the sequence length CDF of the WMT-15 Europarl dataset. We can see that about 99 percent of sequences have length less than 100. And the longest sequence length is 330. We generate an artificial dataset with fixed sequence length 24, which is the average sequence length of the WMT-15 dataset. Additionally, we sample two different datasets with different variance in sequence length from the WMT-15 dataset by clipping the maximum sequence length to be no longer than 50, and 100 respectively.

Figure 11 shows the performance of different systems under fixed-length inputs, and inputs with maximum length of 50, or 100. We can see that increasing the variance of input length causes the latency and throughput of baseline systems to get much worse. The increase in latency is due to requests waiting for more buckets as inputs with higher variance in length use more buckets. The decrease in throughput is due to baseline systems executing with smaller effective batch sizes when inputs with higher variance in length are spread across more buckets. By contrast, BatchMaker

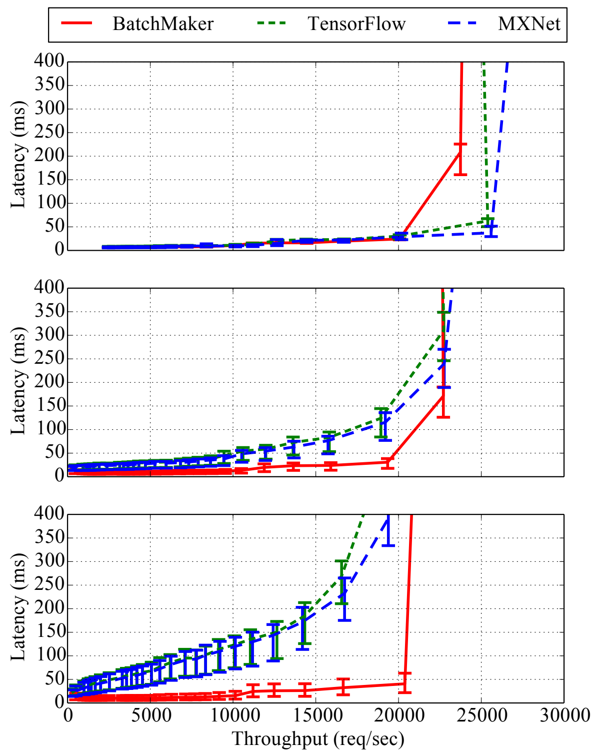


Figure 11: Performance under different sequence length variations. From top to bottom, the experiments use an artificial dataset of identical sequence length (24), a sampled WMT-15 dataset with maximum sequence length 50, and another one with maximum sequence length 100.

can maintain the same latency under low to moderate load despite increased input length variance.

Using the fixed-length artificial dataset, the baseline systems achieve better throughput than BatchMaker. Under the high load, baseline systems can form a full batch of 512 fixed-length inputs. As the execution time of one LSTM cell is approximately 784 microseconds for the batch size 512, one can execute at most $\frac{1}{784*10^{-6}*24} = 53$ batches/sec for inputs with length 24. Thus, the maximum system throughput is about 27136 req/sec, which is closely matched by those of the baseline systems. By contrast, the throughput of BatchMaker is about 87% of the maximum throughput, due to the overhead of scheduling and gathering. Although it is hard to see from the figure, BatchMaker still achieves better latency than baseline systems under low load by allowing new requests to join the execution of currently executing ones.

7.4 Application Performance: Sequence to Sequence

Background on Seq2Seq. Sequence to Sequence (Seq2Seq as an abbreviation) is a widely used RNN model in machine translation. A basic Seq2Seq model contains two types of RNN cells: encoder and decoder, as depicted in Figure 12. The encoder takes in a sequence of words as input. In each step, the encoder and decoder will convert a word to a vector by doing an embedding lookup, then feed this vector to an RNN Cell. The first decoder cell takes in the output

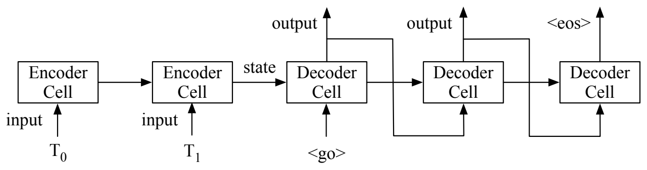


Figure 12: Seq2Seq with "feed previous" decoder

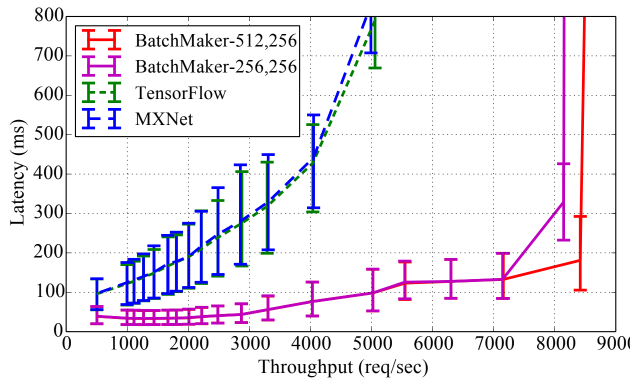
state of encoder and a $< go >$ symbol as input, and computes states, which are passed to the succeeding decoder cell. In addition to the state, the decoder cell outputs a word as well, which is obtained by applying a linear transformation and an argmax[3] [12]. The output word is also fed to the next step as the input. When the decoder outputs the $< eos >$ symbol, it means the decoder has finished, and there will be not any more decoder steps.

In our evaluation, we use LSTM as the RNN cell. Encoder and decoder cells do not share weights. We use the sampled WMT-15 Europarl German to English translation dataset with vocabulary size 30K. When doing inference using Seq2Seq, the decoded sequence length is not known a priori. Deployed systems typically configure the maximum decoding length to be the input sequence length plus a fixed threshold of extra steps. Decoding terminates when either the $< eos >$ symbol is generated or when the maximum decoding length is reached. For simplicity, in our Seq2Seq experiments, for a given input German sentence, we decode for a number of steps equal to the corresponding English sequence length. We do not use the knowledge of decoding length in any of our batching or scheduling decisions.

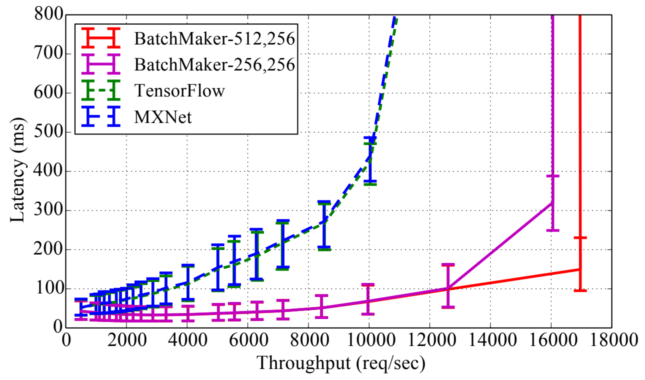
Batching and bucketing configuration. Seq2Seq model is different from LSTM in that it has two phases and the computation is very unbalanced. The decoding phase constitutes about 75% of the entire computation due to performing the output projection from the hidden dimension to the vocabulary dimension, which contains a large matrix multiplication. Our microbenchmarks show that batch size 256 is the best for decoder cells while 512 remains the best for encoder cells. Since graph batching requires that all operators in the dataflow graph use the same batch size, we use $b_{max} = 256$ for MXNet and TensorFlow to optimize for decoder performance. As BatchMaker supports different batch sizes for different cells, we evaluate two configurations; one using $b_{max} = 256$ for both encoders and decoders, and the other using $b_{max} = 512$ for encoders and $b_{max} = 256$ for decoders. We have also evaluated different bucketing choices for the baseline systems, and found that using the bucket width of 10 produces the best performance for baseline systems.

Multi-GPU performance. In the presence of more than one type of cells, BatchMaker can make more interesting scheduling choices when there are multiple GPU devices. Figure 13 shows the performance of various systems using 2 or 4 GPUs. Compared to baseline systems, the peak throughput of BatchMaker is much higher at around 8.5K req/sec for 2 GPUs and 17K req/sec for 4 GPUs. The latency of BatchMaker is also much lower; it is mostly flat at

[3] Argmax operator is not optimized in MXNet and TensorFlow, resulting in unacceptably slow performance. We implemented an optimized argmax CUDA kernel for all systems.



(a) Seq2Seq on 2 GPUs



(b) Seq2Seq on 4 GPUs

Figure 13: Performance of Seq2Seq on the WMT-15 Europarl German-to-English dataset using 2 and 4 GPUs. BatchMaker-*x*,*y* denotes configuring our system to use maximum batch size *x* for the encoder and *y* for the decoder. TensorFlow and MXNet use maximum batch size 256 and bucket width 10.

the beginning and goes up slowly until the throughput reaches the peak. By comparison, the latency of other systems goes up quickly as the load increases. We don't repeat the detailed performance breakdown analysis here as in section 7.3. One interesting feature of BatchMaker worth pointing out is that a request can leave the encoding phase sooner and also commence the execution of decoding earlier than baseline systems, thereby magnifying BatchMaker's performance improvement.

Configuring different b_{max} for encoding and decoding cells results in a small throughput improvement for BatchMaker (3.5 – 6%). Although using $b_{max} = 512$ improves the throughput of LSTM encoding cell execution by 20% (Figure 3), the overall throughput improvement is much less because the encoding phase constitutes only 25% of the overall computation.

7.5 Application Performance: TreeLSTM

As padding does not support batching for non-sequential inputs, we compare against TensorFlow Fold and DyNet for the TreeLSTM experiments.

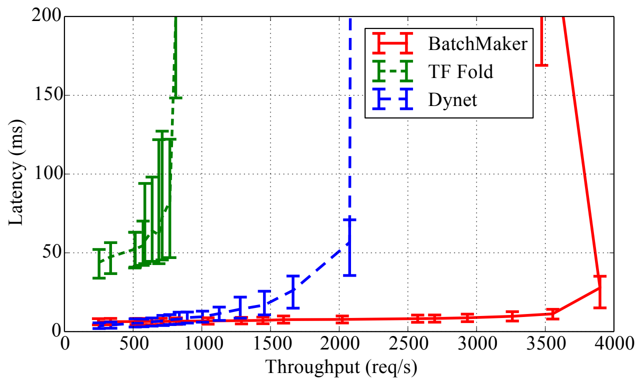


Figure 14: TreeLSTM performance on the TreeBank dataset with maximum batch size 64.

We use the popular Stanford TreeBank dataset [37]. Although all systems under evaluation can support arbitrary tree structure, the TreeBank dataset [37] contains only binary tree samples.

Baseline system configuration and optimization. We perform microbenchmarks using various input batch sizes and find that batching at most 64 input trees achieves the best performance for TensorFlow Fold and DyNet. We note that the batch size configured for DyNet and TensorFlow Fold bounds the maximum number of input trees rather than the number of operators merged into a single batched operator. For instance, even if a batch only contains one request which is a complete binary tree with 16 leaf nodes, TensorFlow Fold can concatenate all 16 leaf nodes together and execute the leaf TreeLSTM cell at batch size 16. It can execute the level above the leaf layer with batch size 8, and so on with the root level executed with batch size 1. As this example shows, the amount of batching decreases at higher levels of the trees. To be fair to baseline systems, BatchMaker is also configured to limit the number of batched cells in a task to 64.

TensorFlow Fold and DyNet perform batching by first generating the dataflow graph for each request and then merging the dataflow graphs together. For TensorFlow Fold, this step takes much longer than performing the actual computation. We optimize TensorFlow Fold by overlapping its graph construction/merging with actual execution[4]. We did not implement similar optimization for DyNet because of its code complexity. We note that DyNet's graph construction/merging overhead is much smaller than that of TensorFlow Fold,

TensorFlow Fold does not work with the latest TensorFlow version (v1.4) and is only compatible with v1.0[5]. Hence, we evaluate TensorFlow Fold using TensorFlow v1.0 and CUDA 8.0[6]. To see the performance disadvantage of using older versions, we conduct microbenchmarks on a single LSTM step using both versions, and find that using the older versions (TensorFlow v1.0 and CUDA 8.0) has a slow down of about 20%.

Performance on the TreeBank dataset. Figure 14 shows the latency vs. throughput for TreeLSTM on the TreeBank dataset. Due

[4]The overlapping is not perfect due to Python's poor multi-threading support.
[5]https://github.com/tensorflow/fold/issues/57
[6]TensorFlow v1.0 does not support CUDA 9

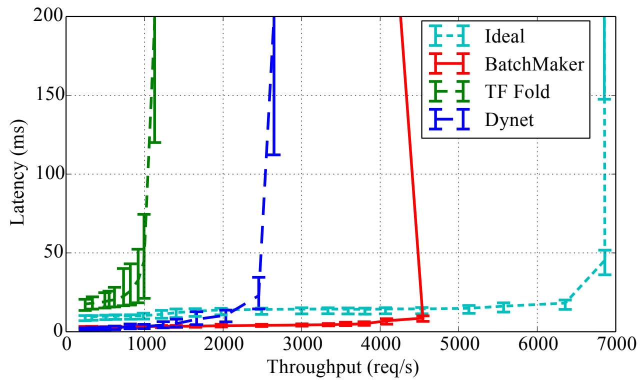


Figure 15: TreeLSTM performance on a synthetic dataset where each sample has the identical binary tree structure. The *ideal* line represents the ideal performance of executing these identical samples in maximum batch size of 64.

to its slow graph construction and merging, the throughput and latency of TensorFlow Fold are much worse than DyNet. Under moderate load (at 1K req/sec), the 90-percentile latency of BatchMaker is 6.8ms, compared to 9.5ms for DyNet. BatchMaker achieves much better peak throughput than DyNet (3.1K req/sec vs. 2.1K req/sec). The throughput difference is to due to DyNet's overhead in performing runtime dataflow graph merging and insufficient amount of batching at the higher levels of the trees.

Performance on a synthetic dataset of identical input trees. How does the variance in input tree structures contribute to BatchMaker's performance improvement? To understand this, we conduct experiments using a fake dataset whose input requests have the identical tree structure (a complete binary tree of 16 leaf nodes). We implement an ideal baseline system by hardcoding in TensorFlow a dataflow graph matching the fixed binary tree structure. Each node in this dataflow graph can execute up to 64 corresponding operations, one for each input in a batch size of 64. We evaluate the performance of all systems including the ideal baseline using the fixed tree dataset. The results are shown in Figure 15. As the figure shows, the peak throughput of BatchMaker is approximately 30% less than that of the ideal baseline. Note that the ideal baseline's latency is higher than that of BatchMaker and DyNet. This is because the ideal baseline executes a series of 31 TreeLSTM cells for a batch of inputs. By comparison, DyNet can batch cells within a request together if they are at the same tree depth; BatchMaker can additionally batch cells from different requests together if they arrive at different times.

8 RELATED WORK

Batching via padding. Theano [5], Caffe [25], TensorFlow [1], MXNet [7], Torch [8], PyTorch [34] and CNTK [13] are widely-used deep learning frameworks. Theano, TensorFlow, MXNet, and CNTK require users to build a static dataflow graph before training or inference. PyTorch is more imperative and allows the computation graph to be built dynamically as execution happens [41]. Gluon [9] is a recent package for MXNet supporting dynamic computation graph. When handling variable-sized inputs, all of these systems support batching via padding. CNTK [18] additionally introduce an optimization on padding that tries to fill up padded space with shorter requests. Doing so can improve system throughput by reducing the amount of wasted computation due to padding. As we mentioned earlier, padding does not work for non-chain-structured RNNs such as the TreeLSTM. Therefore, these systems do not natively support batching for the TreeLSTM.

Batching by merging dataflow graphs dynamically. As non-chain-structured RNNs such as TreeLSTM become popular, TensorFlow Fold [26] and DyNet [30] are developed to support batching for TreeLSTMs. Both systems use a similar approach to batch TreeLSTMs (called Dynamic Batching and on-the-fly batching [31] respectively). They first generate a dataflow graph for each data sample and then attempt to merge all dataflow graphs into one graph by combining nodes corresponding to the same operation while maintaining the data dependency. Graph batching allows both systems to support batched execution of variable computation graphs without padding, including TreeLSTM. The difference between them is that TensorFlow Fold, like our system, batches at the granularity of a cell whereas DyNet batches at the granularity of a single dataflow operator. Both TensorFlow Fold and DyNet try to batch a fixed set of dataflow graphs at a given time. By contrast, BatchMaker batches at the cell level and allows a request to dynamically join and leave the ongoing requests.

Systems specialized for inference. There are several frameworks that address the challenges during inference. LASER [2] and Velox [16] are systems that focus on optimizing the training and serving pipeline for traditional machine learning models such as logistic regression and matrix factorization. LASER and Velox address issues such as how to re-train models quickly upon observing additional data during deployment, how to balance exploration vs. exploitation to gain useful feedback while maintaining good user experience. These issues are orthogonal to the problem addressed by our system, namely, how to reduce the latency of batched execution for RNN inference. Clipper [17] is a general-purpose serving system that supports a variety of machine learning system backends, such as TensorFlow, Spark MLlib [29], and Caffe [25]. It uses existing batching techniques to achieve good throughput and additionally performs dynamic batch size adjustment to match requests' latency objective.

TensorFlow-Serving [32] is a recent system developed for serving TensorFlow models. TensorFlow-serving introduces "Batch" and "Unbatch" operators to TensorFlow's dataflow graphs. It is claimed that these operators "bear similarities to the batching approach of DyNet" [32]. The current implementation for these operators is not yet ready for deployment and thus we have not compared BatchMaker with TensorFlow-Serving.

Optimization for deep learning computation. TensorRT [14] is a deep learning inference optimizer and runtime for deep learning applications. For a given neural network, its optimizer and runtime will generate fused kernel to reduce the kernel launch overhead and memory footprint. TensorFlow XLA [11] is a domain-specific compiler that optimizes TensorFlow computation. It also applies optimization like kernel fusion to reduce the kernel launch overhead. These optimizations are orthogonal to our work, and can be applied to optimize the execution of each cell in our system. Persistent RNN [19] exploits the weight-sharing feature of RNN to accelerate the computation. In particular, persistent RNN store weights in the on-chip memory of GPU (e.g. register files) and reuse them across many timesteps. Doing so avoids loading weights from the global memory (which is much more expensive than on-chip memory) at each time step. Weight persistence reduces but does not eliminate the need for batching. Thus, it is complementary to cellular batching; both can be used together for additional performance improvements.

Batching and pipelined execution in other systems. In database systems, one query may contain many operators like *scan, join, sort, aggregate,* etc. Instead of executing independent queries separately, there are many systems[6, 21, 23, 27, 35, 43] that batch the execution of certain operators from different queries. Since the results of these operators can be reused for different queries, the database throughput can be improved. However, multi-query batching in database systems does not always improve performance, e.g. when the load is light and there is little or no overlap in the data processed by different queries. By contrast, the recursive nature of RNNs result in completed overlapped computation across different inputs, and we leverage this feature to guarantee improved latency and throughput in all scenarios. Additionally, a key feature of cellular batching is to enable a new request to join the execution of existing requests, which is not done in multi-query batching in the database.

Compared with traditional pipelining (in the context of hardware pipeline and software pipelining, e.g. SEDA[45]), our system is different in two aspects: 1) Traditional pipelining involves multiple processing elements each of which is sequential and operates independently of each other. By contrast, each GPU device represents a single processing element that is massively parallel and thus is best utilized using a kernel that performs batched execution. Cellular batching improves latency by allowing new requests to dynamically join existing requests in a series of batched kernel execution. 2) Different processing elements in a hardware pipeline have different functionalities. In our setting, different GPU devices have the same functionalities and can be used interchangeably. Therefore, instead of dictating a fixed pipelined path of execution across different GPU devices, it is better for performance and load balancing to use a general task scheduler to assign kernels to different GPUs, as is done in BatchMaker.

9 CONCLUSION

In this paper, we present a novel approach, called cellular batching, to achieve low-latency inference on Recurrent Neural Network models. cellular batching batches the execution of an inference request at the granularity of an RNN cell. Doing so allows a new request to join the execution of a batch of existing requests and to be returned as soon as its computation finishes without waiting for others in the batch to complete. We have built an RNN serving system called BatchMaker using cellular batching. Experiments on three popular RNN applications using real world dataset show that BatchMaker reduces latency by 17.5-90.5% and improves throughput by 25-80% compared with state-of-the-art systems including TensorFlow, MXNet, TensorFlow Fold, and DyNet.

We note that cellular batching is only beneficial for RNN inference. It does not improve the performance of training because, unlike inference, all training inputs are ready at the same time and the weight update algorithm typically requires waiting for all inputs within a batch to finish. Furthermore, our evaluation shows that BatchMaker benefits workloads whose inputs vary in length or structure (e.g. natural language sentences, parse trees etc.) Thus, we hypothesize that cellular batching would not improve inference for DNNs with fixed inputs such as CNNs and MLPs.

ACKNOWLEDGMENTS

This paper was greatly improved by the advice from our extraordinary fellow colleagues Minjie Wang, Chien-Chin Huang, Cheng Tan. And we thank the anonymous reviewers and our shepherd Gustavo Alonso for their valuable comments and helpful suggestions. We thank Shizhen Xu for providing the knowledge of GPU. The NVIDIA DGX-1 used for this research was donated by the NVIDIA Corporation. This work is supported by NVIDIA AI Lab (NVAIL) and GPU Center of Excellence, National Key Research & Development Program of China (2016YFB1000504), Natural Science Foundation of China (61433008, 61373145, 61572280, 61133004, 61502019, U1435216), National Basic Research (973) Program of China (2014CB340402). Pin Gao's work is also supported by the China Scholarship Council.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *OSDI*, Vol. 16. 265–283.
[2] Deepak Agarwal, Bo Long, Jonathan Traupman, Doris Xin, and Liang Zhang. 2014. Laser: A scalable response prediction platform for online advertising. In *Proceedings of the 7th ACM international conference on Web search and data mining*. ACM, 173–182.
[3] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. 2016. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International Conference on Machine Learning*. 173–182.
[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).
[5] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: A CPU and GPU math compiler in Python. In *Proc. 9th Python in Science Conf.* 1–7.
[6] George Candea, Neoklis Polyzotis, and Radek Vingralek. 2009. A scalable, predictable join operator for highly concurrent data warehouses. *Proceedings of the VLDB Endowment* 2, 1 (2009), 277–288.
[7] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).
[8] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. 2011. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop.*
[9] Distributed (Deep) Machine Learning Community. 2017. The Gluon Package. http://gluon.mxnet.io. (2017).
[10] Distributed (Deep) Machine Learning Community. 2017. NNVM: Open Compiler for AI Frameworks. https://github.com/dmlc/nnvm/. (2017).
[11] Tensorflow Community. 2017. Tensorflow XLA. https://www.tensorflow.org/performance/xla/. (2017).
[12] The Wikipedia Community. 2017. Argmax. https://en.wikipedia.org/wiki/Arg_max. (2017).
[13] Microsoft Corporation. 2015. Microsoft Cognitive Toolkit (CNTK). https://github.com/Microsoft/CNTK. (2015).
[14] Nvidia Corporation. 2017. TensorRT. https://developer.nvidia.com/tensorrt. (2017).
[15] Nvidia Corporation. 2018. CUDA Runtime API. http://docs.nvidia.com/cuda/cuda-runtime-api/index.html. (2018).

[16] Daniel Crankshaw, Peter Bailis, Joseph E Gonzalez, Haoyuan Li, Zhao Zhang, Michael J Franklin, Ali Ghodsi, and Michael I Jordan. 2014. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. *arXiv preprint arXiv:1409.3809* (2014).

[17] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System.. In *NSDI*. 613–627.

[18] William Darling. 2016. Recurrent Neural Networks with CNTK and Applications to the World of Ranking. https://www.microsoft.com/en-us/cognitive-toolkit/blog/2016/08/recurrent-neural-networks-with-cntk-and-applications-to-the-world-of-ranking/. (2016).

[19] Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh. 2016. Persistent rnns: Stashing recurrent weights on-chip. In *International Conference on Machine Learning*. 2024–2033.

[20] Jeffrey Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. 2015. Long-term recurrent convolutional networks for visual recognition and description. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2625–2634.

[21] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2012. SharedDB: killing one thousand queries with one stone. *Proceedings of the VLDB Endowment* 5, 6 (2012), 526–537.

[22] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. 2014. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567* (2014).

[23] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. 2005. QPipe: a simultaneously pipelined relational query engine. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, 383–394.

[24] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[25] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 675–678.

[26] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. 2017. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181* (2017).

[27] Darko Makreshanski, Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2017. Many-query join: efficient shared execution of relational joins on modern hardware. *The VLDB Journal* (2017), 1–24.

[28] Marco Marelli, Luisa Bentivogli, Marco Baroni, Raffaella Bernardi, Stefano Menini, and Roberto Zamparelli. 2014. Semeval-2014 task 1: Evaluation of compositional distributional semantic models on full sentences through semantic relatedness and textual entailment. In *Proceedings of the 8th international workshop on semantic evaluation (SemEval 2014)*. 1–8.

[29] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.

[30] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, et al. 2017. DyNet: The Dynamic Neural Network Toolkit. *arXiv preprint arXiv:1701.03980* (2017).

[31] Graham Neubig, Yoav Goldberg, and Chris Dyer. 2017. On-the-fly Operation Batching in Dynamic Computation Graphs. In *Advances in Neural Information Processing Systems*. 3974–3984.

[32] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. *arXiv preprint arXiv:1712.06139* (2017).

[33] Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan. 2002. Thumbs up?: sentiment classification using machine learning techniques. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*. Association for Computational Linguistics, 79–86.

[34] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. 2017. PyTorch. http://pytorch.org/. (2017).

[35] Lin Qiao, Vijayshankar Raman, Frederick Reiss, Peter J Haas, and Guy M Lohman. 2008. Main-memory scan sharing for multi-core CPUs. *Proceedings of the VLDB Endowment* 1, 1 (2008), 610–621.

[36] Richard Socher, Cliff C Lin, Chris Manning, and Andrew Y Ng. 2011. Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*. 129–136.

[37] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*. 1631–1642.

[38] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*. 3104–3112.

[39] Kai Sheng Tai, Richard Socher, and Christopher D Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075* (2015).

[40] Ming Tan, Cicero dos Santos, Bing Xiang, and Bowen Zhou. 2015. LSTM-based deep learning models for non-factoid answer selection. *arXiv preprint arXiv:1511.04108* (2015).

[41] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. 2015. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, Vol. 5.

[42] TUG 2015. WMT15 Machine Translation Task. http://www.statmt.org/wmt15/translation-task.html. (2015).

[43] Philipp Unterbrunner, Georgios Giannikis, Gustavo Alonso, Dietmar Fauser, and Donald Kossmann. 2009. Predictable performance for unpredictable workloads. *Proceedings of the VLDB Endowment* 2, 1 (2009), 706–717.

[44] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. 2015. Show and tell: A neural image caption generator. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3156–3164.

[45] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: an architecture for well-conditioned, scalable internet services. In *ACM SIGOPS Operating Systems Review*, Vol. 35. ACM, 230–243.

[46] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[47] Jun Yin, Xin Jiang, Zhengdong Lu, Lifeng Shang, Hang Li, and Xiaoming Li. 2015. Neural generative question answering. *arXiv preprint arXiv:1512.01337* (2015).