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 Multi-layer 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
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

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

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

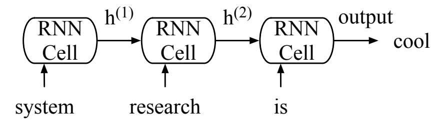


Figure 1: An unfolded chain-structured RNN. All RNN Cells in the chain share the same parameter weights.

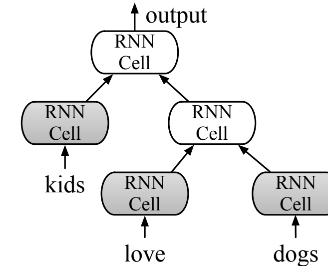


Figure 2: An unfolded tree-structured RNN. There are two types of RNN cells, leaf cell (grey) and internal cell (white). All RNN cells of the same type share the same parameter weights.

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

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

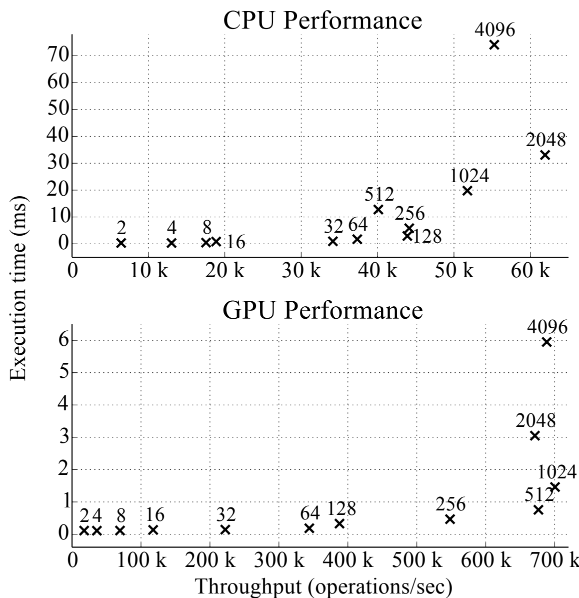


Figure 3: Latency vs. throughput for computing a single step of LSTM cell at different batch sizes for CPU and GPU. The value on the marker denotes the batch size.

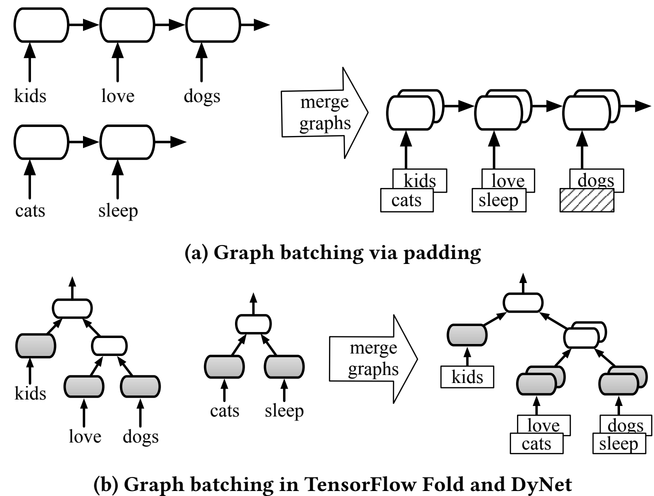
for classifying the sentiment of a sentence [33] and the semantic relatedness of two sentences [28].

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 $b = 2, 4, \ldots 2048$. 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 $2h \times 4h$.

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

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.

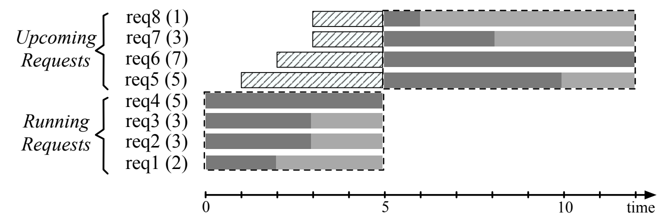
Unfortunately, graph batching is far from ideal for RNN inference and negatively affects both the latency and throughput. Graph batching incurs extra latency due to unnecessary synchronization because an input cannot start executing unless *all* requests in the current batch have finished. This is further 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

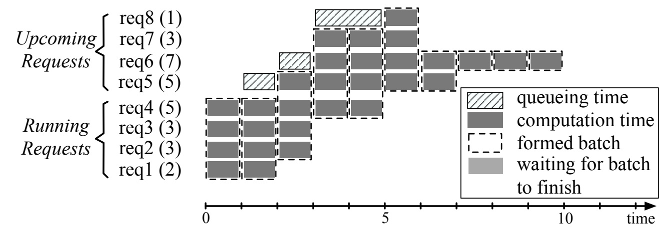
We propose cellular batching for RNN inference. RNN has the unique feature that it contains many identical computational units connected with each other. Cellular batching exploits this feature to 1) batch at the level of RNN cells instead of whole dataflow graphs, and 2) let new requests join the execution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



(a) Graph Batching



(b) Cellular Batching

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

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

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.

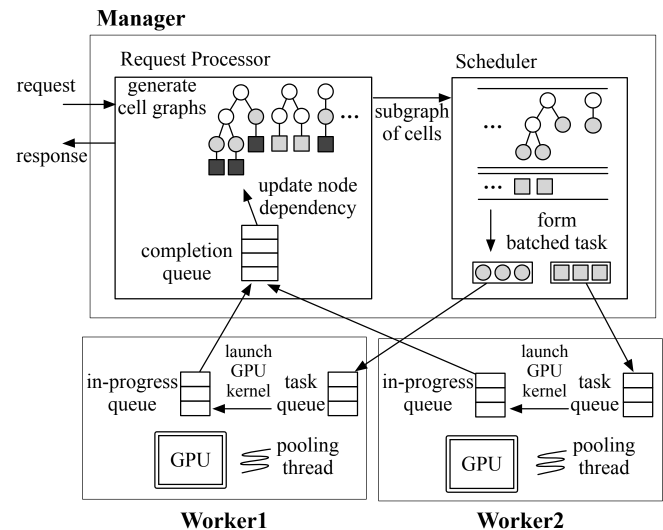


Figure 6: The system architecture of BatchMaker. In the cell graph, black means computed nodes, grey means nodes whose input is ready, and white means input dependency is not satisfied.

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

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase®
Smarter legal research.