DynPartition: Automatic Optimal Pipeline Parallelism of Dynamic Neural Networks over Heterogeneous GPU Systems for Inference Tasks

Fan Pu Zeng
Carnegie Mellon University
School of Computer Science

Vivswan Shah
University of Pittsburgh
Electrical and Computer Engineering

Yudong Liu
Carnegie Mellon University
School of Computer Science

Abstract

Dynamic neural networks are slowly gaining popularity due their ability to adapt their structures or parameters to different inputs, leading to notable advantages in terms of accuracy, computational efficiency, and adaptivity, in comparison to static models which have fixed computational graphs and parameters. We propose a novel reinforcement learning-based scheduler called DynPartition that performs dynamic partitioning of computation across multiple heterogeneous GPUs for dynamic neural network inference tasks . Our scheduler is trained through previous iterations to generate an optimal forward schedule across heterogeneous GPUs given the network input. Our experiments show that the RL-based scheduler can successfully converge towards optimal distribution of computation across devices during inference tasks.

1 Introduction

The need to scale up machine learning in the trend of a rapid growth of data both in volume and in variety raises the demand for distributing the computation of machine learning models architectures to different devices [8]. Although there has been a considerable amount of work studying the distributed machine learning for large complex neural network architectures, most of these work focuses on distributing static neural networks to multiple devices following simple and trivial strategies. However, as modeling complex relations drastically increases the demand for more adaptive neural networks, dynamic networks emerged as an alternative to static models. Since these models can adapt their structures or parameters to different inputs, they have notable advantages in terms of accuracy, computational efficiency, and adaptivity [1]. Therefore, performing distributed batching, training and inference on large dynamic model architectures has become a new challenge.

The trivial and widely adopted solution would be to distribute each stage of the dynamic neural networks into one of the GPUs and follow a fixed partition of computation across iterations. However, a non-dynamic strategy would inevitably lead to poor load-balancing schedules across devices as the stage complexity in each GPU change according to inputs, and is therefore incapable of optimally distributing workload across workers over iterations. Under such scenarios, faster workers or those with lighter workloads are under-utilized and will idle while waiting for the slower or more heavily-loaded workers for essential gradient or weight synchronization, which in turn results in inefficiency of the entire cluster. [9]

To address these challenges, we developed DynPartition, which is a reinforcement-learning based scheduler with a Deep Q Network architecture, which takes as input a fixed-size representation of the input, and outputs an allocation of nodes in the computation graph to the device (CPU or one of the CUDA devices) that it should run on.

We will show that it is able to successfully partition dynamic neural network computation across multiple GPUs, while also discussing the limitations of our experimental results due to the restrictions of the evaluation framework we developed.

2 Problem Statement

In order to develop a dynamic partitioning policy to ensure Heterogeneous GPUs accomplish their tasks almost synchronously in each iteration, there are three factors we need to be able to model in our approach:

1. The computational capability of each GPU,
2. The dynamic architectures and connections within the network,
3. The cost of the operations that we need to perform, including the time of running the forward pass in each module and the overhead of transferring data across devices.

All of these factors will be modeled either implicitly or explicitly in SCHEDULERENV.

Solving the open problem of distributed dynamic machine learning is not easy. We aim to start with an ideal scenario of performing inference on a dynamic neural network, where the neural network takes on a tree-like structure, in which the dynamism of the model lies with the connections between the different modules. A typical example of networks that satisfy our requirements are TreeLSTM networks [3].

Github: https://github.com/fanpu/DynPartition

3 Background and Related Work

We investigate the question of how we can efficiently perform inference on a large model distributed over multiple heterogeneous devices by optimizing for how the model is split between the different devices. We consider this under the context of model parallelism, where the model weights are too large to fit on a single GPU and are therefore split up among multiple GPUs. We clarify that we are not concerned about data parallelism, which is a separate context where the entire model is loaded on each GPU, and they perform inferences on the input in parallel.

We now provide some background on our setup and approach, and also provide an overview of existing work done in this space.

3.1 Motivation

Previous work from Mirhoseini et al. 2017 [4] developed reinforcement-learning based strategies for optimal partitioning that perform up to 20% faster than human expert's placements. As the size of neural network models grows, and the number and diversity of computational resources continue to increase, the problem of solving for optimal partitioning will become progressively more complex, and we predict that it will be increasingly the case that optimal policies found by algorithms will surpass those crafted by human experts by even larger margins.

This suggests that there can be a lot of gains in inference efficiency from our proposal to build a general framework for optimal partitioning of models to GPU resources, which directly translates to money saved from running these large models, and is good for both business and the environment.

The approach to model parallelism that we will use in this paper is pipeline parallelism. In pipelined parallelism, the layers of the network are split into stages, and groups of consecutive stages are assigned to different GPUs. The communication between GPUs between each group of stages represents an overhead that should be minimized. Model parallelism can also be thought of as splitting the model vertically, if we imagine the model diagram with layers going from left to right.

Note that we always want the stages on the same GPU to be consecutive in order to minimize the amount of communication overhead.

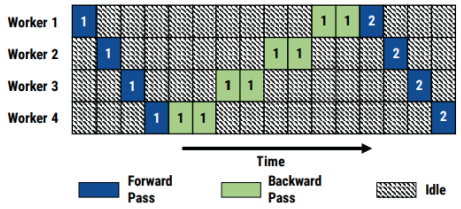


Figure 1: Workers waiting on backward passes to complete results in pipeline stalls and inefficient utilization of GPU resources. [6]

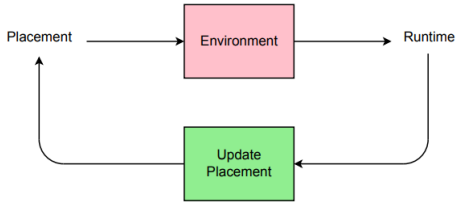


Figure 2: Overview diagram of RL based device placement model. [4].

A key design consideration in pipeline parallelism is to balance the workload between the GPUs evenly and to avoid pipeline stalls. Pipeline stalls pose a particularly bad problem for training tasks as illustrated by Figure 1, since the next batch has to wait for the backward pass to complete across all the stages of each node before it can proceed. This problem is addressed in depth by PipeDream [6], which solves it by adding inter-batch pipelining to intra-batch parallelism, and shows that the method results in training performance that is 5.3x faster than existing intra-batch parallelism techniques.

Our paper will instead focus on inference tasks instead of training tasks, which avoids the problem of pipeline stalls due to waiting for the backward pass.

3.2 Device Placement Optimization using Reinforcement Learning Methods

Mirhoseini et al. 2017 [4] addresses a similar problem as us in the specific setting of optimizing the placement of operations in a TensorFlow computational graph among heterogeneous devices. This is achieved using reinforcement learning to learn a placement policy that optimizes for the execution time, and is illustrated in Figure 2. The execution time is a summary statistic the two variables that we care about: computational time and communication time. The authors observed empirically that using

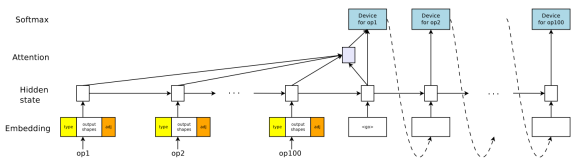


Figure 3: Device placement architecture diagram. [4].

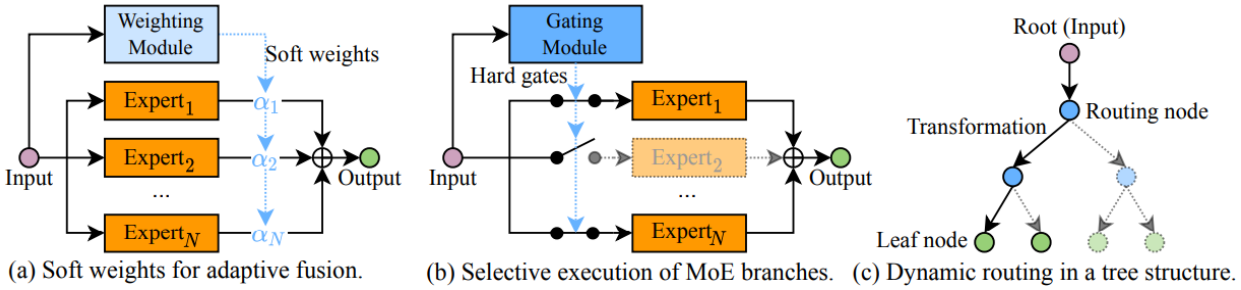


Figure 4: The three different type of dynamic neural network with routing are: (a) MoE with soft weighting, (b) MoE with hard gating, and (c) Tree structure (in this figure each node acts as a routing node as well, but in reality routing can be separated from the compute node.) [1]

the square root of the running time as the reward signal results in a more robust learning process. The architecture of their prediction model is a sequence-to-sequence model with LSTM and a content-based attention mechanism, and is sketched out in Figure 3. They evaluated their approach on neural network architectures including Inception-V3, Recurrent Neural Language Model, and Neural Machine Translation.

The primary challenge to their approach of attempting to optimize over all the operations in the TensorFlow computational graph is that there are usually thousands of operations, making training very unstable due to vanishing/exploding gradients. As such, they employed many heuristics to merge operations into what they call co-location groups, which are always assigned together to the same device, therefore decreasing the granularity of their optimization problem.

Our approach will take the opposite extreme by instead having the user specify natural high-level partitions of the model, which is faster to optimize over and to perform placement predictions on. While this comes at a cost of not being able to recover the global best partitioning strategy, our hypothesis is that the natural abstraction boundary between different network components form a sufficiently good partition to optimize over, as intra-component nodes tend to have high communication which makes splitting them them up very costly.

We also note that Mirhoseini et al. 2017 [4] considers optimization in the context of static neural networks based on the compiled computation graph in TensorFlow. On the other hand, our approach will generalize to dynamic neural networks.

3.3 Scaling Giant Models With Automatic Sharding

Google introduced the GShard library [2] that allows tensors to be easily sharded across nodes. This is achieved by annotating tensors as either `replicate`, `split`, or `shard`, which indicates whether the tensor should be replicated, partitioned along a specified dimension, or sharded across multiple dimensions respectively. GShard can then automatically perform sharding for the tensors and adapt to the underlying hardware, with the user to also specify some manual partitioning policies as needed. [2]

GShard does not require the user to annotate every tensor, and the compiler is capable of using heuristics to infer which sharding attribute to apply to un-annotated tensors. Annotations are usually only performed manually on expensive operators like Einsums, where the cost of getting it wrong is much higher.

We adopt a similar annotation-based approach from GShard by having the user manually specify a vertical network partition.

3.4 Dynamic Neural Networks

To introduce additional novelty in our work, our approach will be considered in the context of dynamic neural networks. Dynamic neural networks are neural networks that can change their structure or behavior during runtime, as opposed to static neural networks, which have a fixed architecture and do not change during runtime.

Examples of dynamic neural networks include tree long short-term memory (TreeLSTM) networks and MoE transformer models.

3.5 Distributed Dynamic Neural Networks

As illustrated in Figure 4, this paper focus on the investigation of dynamic neural networks in which dynamism of the model lies with the connections between the different modules, that is as the dynamism of the model originates from routing steps taken after each output. Although the distribution of MoE based dynamic neural networks over multiple nodes has been accomplished by GShard using data parallelism approach [2], hence is not the focus of our paper. Instead, we center our attention on Dynamic Routing Neural Networks with a Tree structure. The self-contained nature of each stage within this structure enables us to distribute the network through pipeline parallelism without modifying the modules themselves. This approach, coupled with real-time learning of DynPartition, allows us to optimize node distribution for the inference task based on previous inference examples. DynPartition node distribution is optimized to minimize overall computation time during inference and reduces

data transfer between GPUs.

4 Methodology

We re-iterate the context of the problem that we are solving, and address their unique challenges and simplifications that it affords us.

Our context is to optimize for partitioning via **pipeline parallelism**, where the models are **dynamic neural networks**, evaluated on **inference tasks**, with underlying **heterogeneous hardware**.

1. **Pipeline parallelism**: It is easier to estimate communication costs between different stages if we only consider pipeline parallelism, even in the presence of dynamism. The underlying optimization problem also becomes much easier to solve. In contrast, it is much harder to optimize for tensor parallelism where data must frequently be exchanged between different GPUs. Since cross-GPU talk is expensive as it requires traversing through the interconnect, doing so is only beneficial in the most extreme cases when the benefits are very significant. Therefore, in the spirit of simplifying the problem without degrading the optimal partitioning strategy too much, we will only consider pipeline parallelism.

2. **Dynamic neural networks**: Many state-of-the-art large neural networks increasingly rely on dynamism, which allows for greater expressiveness and can also result in greater efficiency. We target dynamic neural networks in order to support this ongoing trend.

   However, dynamic neural networks also introduces challenges. The cost of running input through a stage now depends not only on the stage and underlying hardware, but also depends on the input due to dynamism in the network. It can be challenging to learn a estimate of what different inputs will cost. Many dynamic neural networks, such as Transformers models, can also be challenging to partition for pipeline parallelism.

3. **Inference tasks**: The level of dynamism exhibited by dynamic neural networks tend to be lower during inference as opposed to training time. This helps to simplify the problem of estimating the cost functions. For instance, dropout in neural networks is only applied during training, but not during inference. However, for models like LSTMs which handle variable-length input there is dynamism during both inference and training.

   Furthermore, considering this only in the context of inference avoids the issue of gradients faced during training, which could result in significant dynamic cross-talk between GPUs that can be very hard to model. Hence this is another form of simplification. In addition, most of the computational resources is spent on model inference after it is trained, making it an ongoing cost and hence there is significant incentive to optimize for inference.

4. **Heterogeneous hardware**: The rapid pace of advancements in GPU hardware means that computational resources available are increasingly heterogeneous. Heterogeneous computing also makes sense from a cost perspective, where spare capacity from cloud providers like AWS Spot instances can be used at a fraction of the price compared to on-demand pricing, with the caveat that one has to flexibly adapt between whichever hardware that is available. We thus tailor our approach to address this ongoing trend.

4.1 Input Processing Pipeline

To represent the structure of tree datasets in a tensor, a multi-step approach was followed. First, the tree was converted into a list by in-order traversal. This ensures that the information from each node was captured in a sequential and deterministic manner.

Next, the resulting list was converted into an array that contained each node information, including its position within the tree, its depth, and its parent node. That is, mapping the nodes in the list to their respective positions in the array, thus creating a structured representation of the tree.

Finally, to represent this array in a tensor, a sinusoidal positional embedding technique was employed. This method involves representing each position in the array with a unique vector that captures its position in the sequence. By doing so, the tensor could effectively represent the structured information of the variable-length tree as a fixed-size matrix.

Overall, this approach involved a series of steps that effectively captured the structure of tree datasets in a fixed-size tensor representation, which can then be used for the RL pipeline for training DynPartition.

4.2 Our Approach

We introduce a new framework called DynPartition that is built on top of PyTorch. DynPartition is currently specialized for dynamic neural networks that has a tree-like computation structure, such as TreeLSTMs and our MathFunc framework (Section 5). Extending this to generic dynamic neural networks is given as future work (Section 10).

From a high-level point of view, the way DynPartition works is summarized in Figure 5.

SCHEDULERENV is a custom-developed Gymnasium environment that provides a uniform interface for RL agents (see Section 6.1). It provides as states the encoded states of randomly sampled input from our training dataset, and takes in actions as input. It then executes the

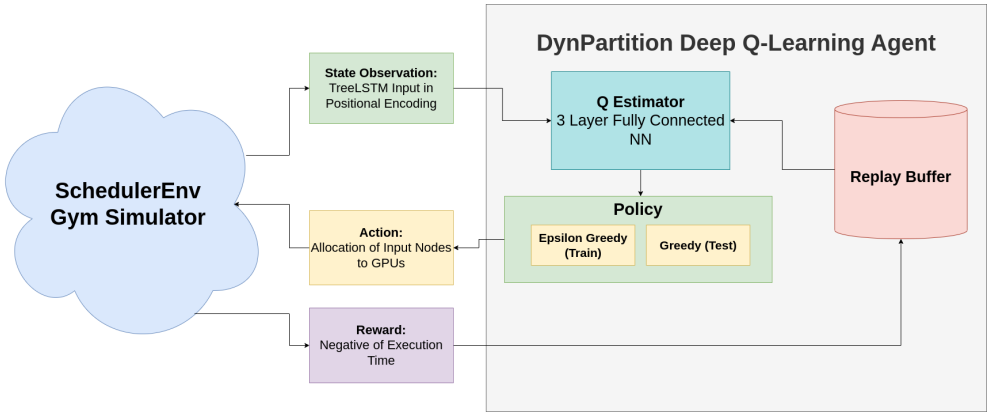
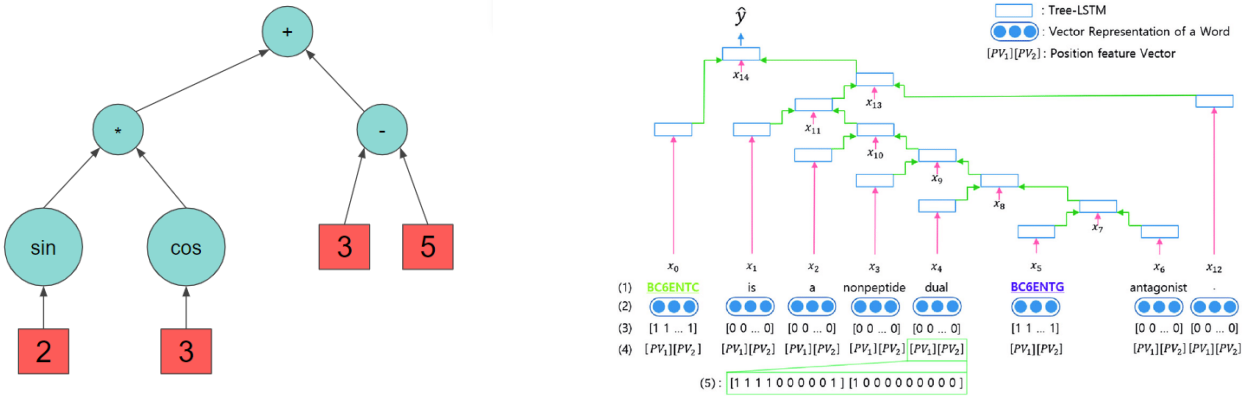


Figure 5: High-level architecture overview of DynPartition. The two main components that interact in a feedback learning loop are SCHEDULERENV and the DynPartition Deep Q-Learning Agent



(a) MathFunc architecture.

(b) TreeLSTM architecture. [3].

Figure 6: MathFunc and TreeLSTM Models

action, and outputs a reward, which is the negative of the execution time (in ms).

DynPartition is a Deep Q-Learning agent that tries to learn an optimal policy by interacting with SCHEDULERENV, with the goal of maximizing its expected rewards. Having covered an overview of our approach, we now dive into specifics.

5 TreeLSTM and MathFunc

DynPartition will initially be evaluated on MathFunc (a mathematical model) to speed up the creation and debugging significantly since this can be run on any GPU without any specific requirements (explained in following sub-section). The testing will be performed on pre-trained TreeLSTM [3] to make sure DynPartition is applicable in the real life.

5.1 MathFunc

MathFunc is a mathematical computational model that can compute math equation using tree structure, as depicted in Figure 6a. The architecture of MathFunc comprises a hierarchy of mathematical functions organized in a tree structure, with the root node gives the output of the math equation and the leaves nodes representing the input features. This tree structure allows for efficient computation by enabling the sharing of computation between different branches of the model. By utilizing MathFunc, we were able to significantly reduce the run-time of our DynPartition creation and diagnostic processes. This has been particularly beneficial as it has allowed us to perform these tasks on standard computing devices with any GPU. Ultimately, this has led to a more efficient and cost-effective approach to creating and diagnosing DynPartition.

5.2 TreeLSTM

Tree-LSTM (Long Short-Term Memory) is a neural network that is designed for modeling tree structures to represent the connection of different words in a sentence. It is an extension of the traditional LSTM network, which is commonly used for sequential data, such as natural

language sentences.

Tree-LSTM is consist of nodes similar to how tree is consist of nodes. Each node in the tree has an associated LSTM cell. The input to each cell includes the current node's content as well as the output from the LSTM cells of the node's children. This allows the model to capture long-term dependencies and context from the tree structure. The structure of TreeLSTM can be seen in Figure 6b.

As described thorough out this paper, DynPartition is designed to optimize the computation of Tree-Based Dynamic Routing Neural Network, hence the main evaluation of the paper will be on TreeLSTM [3].

5.3 Dataset

The MathFunc dataset was generated dynamically through the on-the-fly combination of 8-10 simple mathematical functions, such as addition, subtraction, sine, cosine, and others. This approach allows for the creation and testing of DynPartition more efficiently and quickly, due to the simple and lightweight nature of these basic math functions. By combining them in a tree model randomly, we were able to create dataset which can be used in MathFunc, a dynamic tree-based neural network.

For pre-training and inference task of TreeLSTM, Stanford Sentiment Treebank [7] was used. The Stanford Sentiment Treebank is a corpus with fully labeled parse trees that allows for a complete analysis of the compositional effects of sentiment in language. [7]

6 Deep-Q Reinforcement Learning

6.1 SchedulerEnv

We created a Gymnasium environment called SchedulerEnv in order to have a standard interface for our RL learning algorithm. This also makes it easier to evaluate and test other RL algorithms in the future. With reference to the right side of Figure 5, SchedulerEnv is built as follows:

- **State**: Upon each reset of the environment, a new input is randomly sampled and fixed for the duration of the episode.

  In addition, each episode lasts only for a single time step. In other words, after an action has been supplied, a reward is returned and the episode ends.

- **Action**: The action that the user supplies is an allocation of nodes to devices. Since the number of nodes can be arbitrary, in practice we set this to 128 nodes, noting that almost all the inputs in our SST dataset have fewer than 128 nodes, and hence we removed all inputs with over 128 nodes. For inputs that have fewer than 128 nodes, we mask out the actions of the nodes that do not exist.

- **Reward**: the reward is the negative execution time of the forward pass of the network based on the allocation in milliseconds, repeated and averaged over 10 runs in order to reduce noise. We elaborate further in Section 9.2 why noise reduction is necessary in order for DynPartition to train well.

6.2 Deep-Q Network Agent

Our DynPartition RL framework is based off Deep-Q Networks (DQN), also known as Deep Q-Learning, which has been shown to achieve superhuman results on tasks such as playing Atari games [5].

In DQN, we use a neural network to try to learn the Q function, which is a mapping from state and action tuples, to the value of the state. Since our SchedulerEnv environment only has a single time step per episode, this is equivalent to a mapping from the state encoding and device allocation to the expectation of the negative of the running time.

In addition, we make use of a replay buffer (as illustrated in Figure 5). While the primary motivation for using a replay buffer in [5] was to minimize correlations between the state-action-reward tuples, in our case because each episode only has a single time step, there are no correlations between the tuples. Instead, the goal is to avoid catastrophic forgetting and recency bias, especially because there is large variability in the execution times of different inputs that could give misleading reward signals. This is further elaborated in Section 9.3.

During training time, DynPartition uses an ε-greedy for selecting actions based on the rewards predicted by the Q-network. This means that for each node, with $1-\varepsilon$ probability, it chooses the best device predicted by the network, and with ε probability, it chooses a device at random. This hence provides a trade-off between exploration and exploitation, which is important for learning.

During test time, DynPartition uses a greedy policy to maximize its expected rewards.

7 Computational Infrastructure

Creation and testing of DynPartition requires compute nodes with GPUs. These compute resources are provided by the Pitt Center for Research Computing.

The Pitt Center for Research Computing consist of 4 homogeneous GPUs per node. The models of GPUs available to use are: GeForce GTX1080 Ti, V100, and A100. However, since Pitt CRC imposes a cost for usage of these resources, we only used them for final evaluation and results.

8 Results

The following experiments are conducted on a machine in the Pitt Center for Research Computing with access to Nvidia GeForce GTX 1080Ti GPU and NVIDIA A100 Tensor Core GPUs.

8.1 Benchmarks

We performed benchmarks on the data transfer speeds, speed of synchronous vs asynchronous pipeline executions, and computation and communication latencies on CPU and GPU. These benchmark results allows us to better interpret the results from the SCHEDULERENV learning curves.

8.1.1 Data Transfer Speeds

A floating point tensor of 1000 by 1000 size was used to measure the data transfer speeds between devices as shown in Figure 7. It is clear from the figure that the data transfer speed between CUDA (GPU) devices is approximately 2.2x times the data transfer speed between CUDA and CPU. Hence, for large transfers it is more efficient to bypass the CPU if possible.

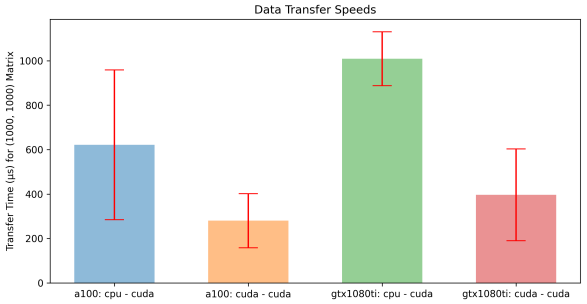


Figure 7: Data Transfer Latency on Different Devices.

8.1.2 Synchronous vs Asynchronous Pipelines

DynPartition assigns a separate thread to each node of the tree, enabling computation of a node as soon as the necessary data is available. Despite the use of individual threads for each node, the threads must still wait for the completion of threads associated with the nodes' children. Unfortunately, the notification of the child threads' completion is not instantaneous due to the operating system and processor's functioning, resulting in additional overhead. Figure 8 demonstrates the observed overhead, which cannot be avoided as the utilization of separate threads is necessary for independent device operation. The TreeLSTM's small size and limited computation requirements further highlight this overhead, as discussed in the section 9.1.

8.1.3 Execution Time against Number of GPUs

The results presented in Figure 9 illustrate the latency and overhead cause by impact of parallel processing TreeLSTM and MathFunc on different number of devices. As anticipated, the latency increases with an increase in the number of devices, due to the fact that the data transfers between devices increases as the number of device increase hence increasing the overall latency and overhead. Notably, this trend is more prominent in MathFunc than in TreeLSTM, as MathFunc involves less computational load, hence manifying the effects of latency in

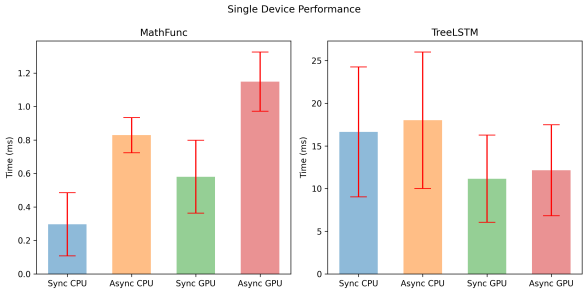


Figure 8: Execution Time for Synchronized vs Async Pipelines.

comparison to TreeLSTM.

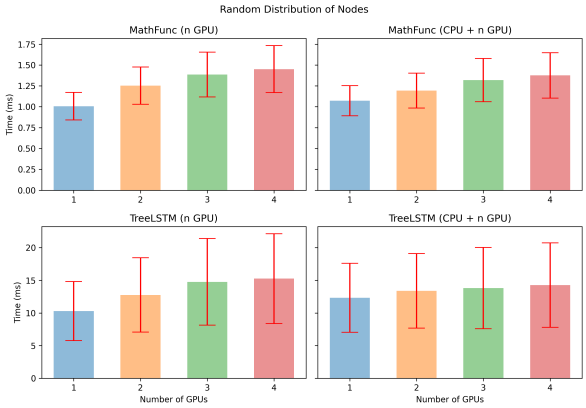


Figure 9: Execution time against GPUs used.

8.1.4 Left-Right Split and State-Output Split

Figure 10 shows the latency associated with when all the nodes left of the root node in the tree are computed in device 0 and all the node right to the root are computed in device 1. While Figure 11 shows the latency associated with if all the state calculations of each node were performed on device 0 and all the output calculation of each node were performed on device 1.

From these results, we can see that the latency in CPU-GPU calculation is faster than that in GPU-GPU calculation, that is CPU is performing faster than GPU, which is contrary to what we expected. This is due to the fact that MathFunc and TreeLSTM are small models hence can be easily computed by CPU as discussed in the section 9.1 with more details.

8.2 SCHEDULERENV Learning Curves

The following experiments were conducted on a machine with access to 4 NVIDIA A100 Tensor Core GPU. There are therefore five output devices in total: one CPU device, and 4 CUDA devices. The plots are aggregated in Figure 12. Each of the experiments are repeated 10 times with different random seeds to account for the stochasticity in the reinforcement learning process, with the shaded regions showing the min and max of the rewards at each

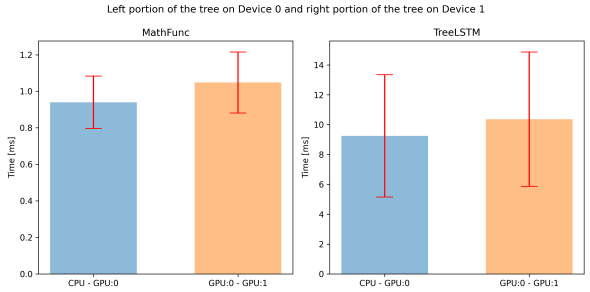


Figure 10: Left portion of the tree is on Device 0 and Right portion of the tree is on Device 1

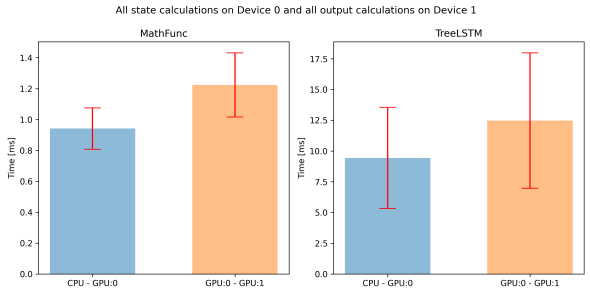


Figure 11: State calculation on Device 1 and Output calculation on Device 2

episode.

8.2.1 Static Split (GPU)

In the static split GPU baseline policy allocates, all nodes in the TreeLSTM input are allocated onto the GPU. The learning curve is given in Figure 13a. It has an average reward of -11.63, where higher rewards are better.

8.2.2 Static Split (CPU)

Like the previous baseline policy, the static split CPU baseline policy allocates all nodes in the TreeLSTM graph to the CPU instead. The learning curve is given in Figure 13b. It received an average reward of -6.34.

8.2.3 Random Split

In the random split baseline, nodes are randomly allocated with equal probability to either the CPU or the GPU. The learning curve is given in Figure 13c. It received an average reward of -10.11.

8.2.4 DynPartition RL Model

The learning curve is given in Figure 13d, with an average reward of -6.40 at the end. Its reward began increasing after an initial short plateau, showing that DynPartition was learning how to perform better allocations, before tapering off at around 4000 episodes.

9 Discussion

Overall, our experiment shows that DynPartition was capable of learning a good device allocation policy given sufficient training. Our experiments also yielded several interesting observations with implications on the generality of our results, which we now discuss.

9.1 Static CPU Baseline Performs Better than Static GPU Baseline

Comparing between Figure 13a and Figure 13b, it was surprising that allocating everything to the CPU was actually faster than allocating everything to the GPU, which contradicts conventional wisdom.

This is likely due to the fact that the size of the dynamic neural networks generated by the inputs are not large enough for the asymptotic performance advantages of the GPU to kick in and to overcome the overhead of GPU communication. Indeed, most of the input had relatively small trees of under 50 nodes.

However, it is difficult to scale our inputs without also negatively impacting the difficulty of training our RL model. This is because the action space of SCHEDULERENV is proportionate to the maximum number of nodes in the TreeLSTM input, as it must output an allocation for each node.

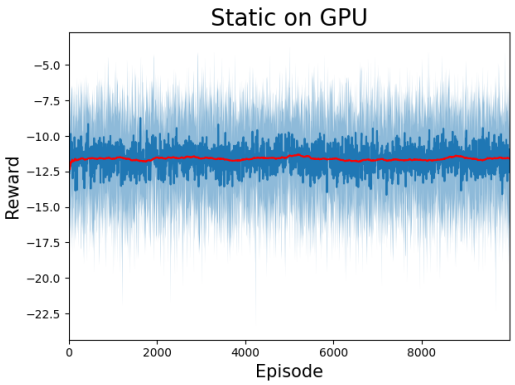
The fact that the naive CPU-only allocation is optimal for our TreeLSTM framework presents challenges to our evaluation and the generality of our results. Even though Figure 13d shows that DynPartition was able to learn the best partitioning strategy by putting everything on the CPU, this is prima facie a relatively simple policy similar to our static CPU baseline, and does not require it to learn any complicated representations or policies. It is therefore unclear whether DynPartition would perform as well when a more complicated policy is optimal.

As such, in Section 10 we discuss how we could design an alternative framework for future work that can adapt to arbitrary dynamic neural network architectures instead of TreeLSTM, where allocating nodes on the GPU would perform better than just putting it on CPU, and where learning a more complicated policy of splitting it between multiple GPUs is necessary.

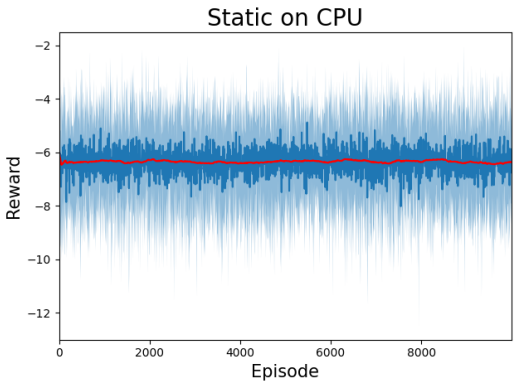
9.2 Noisy Training Signal

We initially received very pessimistic results during development where the rewards of DynPartition was not improving over time, even over a large number of episodes and random seeds. This was because on our development environment (which was one of our personal machines and not a dedicated instance), there were a lot of other background processes going on. For instance, Xorg [1], the X Window display server uses GPU acceleration to render the user interface, which likely results in contention with our experiments. In addition, we observed many instances of crashes in rewards in the noisy environment (Figure 13), although such crashes do not occur on our dedicated test environment in the Pitt Center for Research Computing (Figure 12). This highlights the importance of running SCHEDULERENV in an environment with as

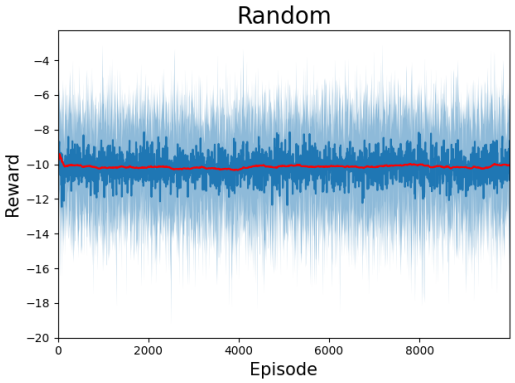
[1] https://www.x.org/wiki/



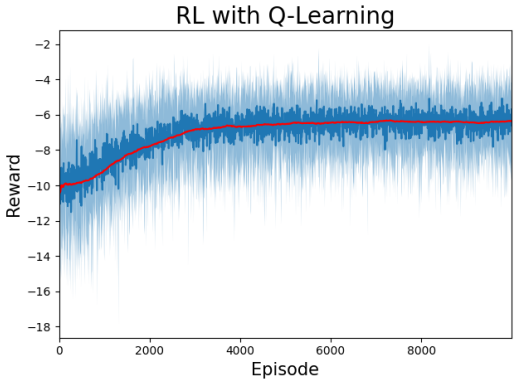
(a) All nodes are on GPU



(b) All nodes are on CPU



(c) Each node is randomly chosen to be on GPU or CPU



(d) Each node device is chosen by DynPartition

Figure 12: Learning Curves on a dedicated test environment with Nvidia A100 over 10,000 episodes on the SCHEDULERENV environment on baseline and DynPartition. Repeated over 10 different random seeds, with the max and min shaded, and the line representing the mean. ($Reward = -time\ [ms]$)

little noise as possible for the best results.

9.3 Variability in Input Sizes

Figure 12 shows significant noise in the learning curves. We investigated this by selecting 3 random inputs, and measuring the rewards over 100 iterations where the allocations were homogeneous on only either the CPU or GPU, with the mean and standard deviation of the rewards plotted in Figure 14.

We see that in addition to the non-negligible noise in the measured execution times for each individual input, there was also a large variance in the mean of the execution times over the different inputs, due to differences in the sizes of the inputs. This means that even when an allocation was actually favorable, a larger input tree would still incur a longer execution time and incorrectly signal to the RL learning agent that the allocation was bad, resulting in longer training times and also training instability.

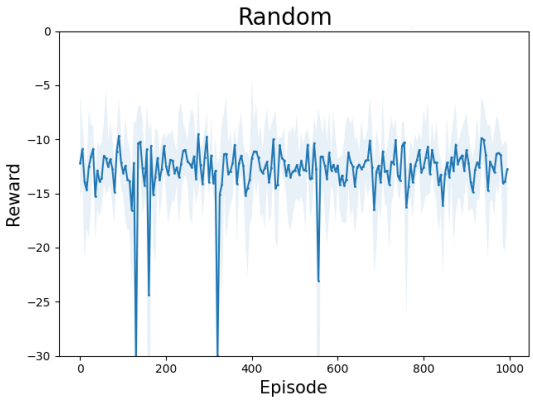
Unfortunately, having run-time complexity depend on the input is in fact a feature of dynamic neural networks, which means that this will be a fundamental challenge to any learning algorithm. One approach to alleviate this issue would be to try to normalize the execution time, with one possible approach being dividing the measured execution time against the execution time on a homogeneous static CPU allocation, which we leave to future work.

9.4 Input Positional Embedding Trade offs

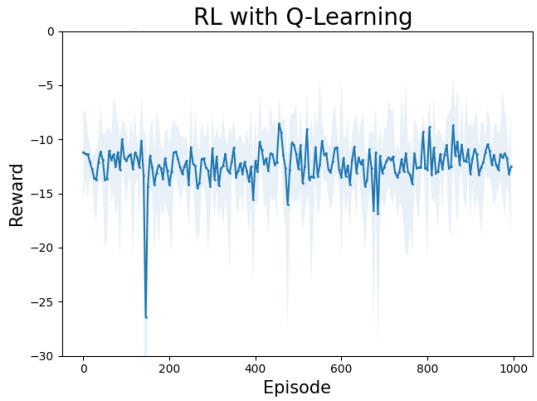
We were initially concerned that the positional embedding representation of the TreeLSTM inputs could pose difficulties to DynPartition in learning a good policy, since it has to learn to reason about an embedding of the input instead of the raw input (which has variable length). However, we were unable to test this hypothesis, since the learnt optimal policy was very simple (see Section 9.1).

(a) All nodes are on GPU

(b) All nodes are on CPU



(c) Each node randomly chosen to be on GPU or CPU



(d) Each node device is chosen by DynPartition

Figure 13: Learning curves on a noisy development environment with a GTX 1650 Super and a single CPU over 1000 episodes on the SCHEDULERENV environment on baseline and DynPartition. Repeated over 5 different random seeds, with the max and min shaded, and the line representing the mean. The environment was sufficiently noisy that the test reward for DynPartition was not able to improve over time.

10 Future Work

In this section, we discuss future work that follows immediately from our learnings and experimental results.

10.1 A General Dynamic Neural Network Evaluation Framework

In our current experiment on TreeLSTM and MathFunc, the GPU-only based scheduling failed to outperform CPU-based scheduling. We think this issue is likely caused by the scale of our network. The network architectures we used are too small to offer GPU an advantage over CPU through parallel computation. Instead, the transfer of data between GPU and CPU memory likely caused additional overhead that outweighs the advantages of using a GPU. For future work, we propose conducting further experiments on larger and more generic dynamic neural network architectures. This will require a library capable of encoding arbitrary dynamic network architectures into tensors so that they can be feed into our RL-scheduler. One way to do this is to implement a dynamic version of MODULE class in PyTorch.

10.2 Investigating Hysteresis from Allocations

When a computer scientist performs an allocation from nodes in the TreeLSTM graph to devices manually, a natural approach will be favor locality in adjacent nodes, so that cross-device talk is minimized as much as possible. Therefore, we might similarly expect the DynPartition Q-Learning agent to also learn locality in allocations on more complicated dynamic neural network structures.

However, this could result in hysteresis in the node allocation path, which could be a possible failure mode that results in slower or unstable training. With respect to Figure 15, consider an input graph in the TreeLSTM context where the agent learns to prefer locality in assignments (with respect to the in-order traversal of the tree, as this was one of the available information encoded in our positional embedding). However, a naive application

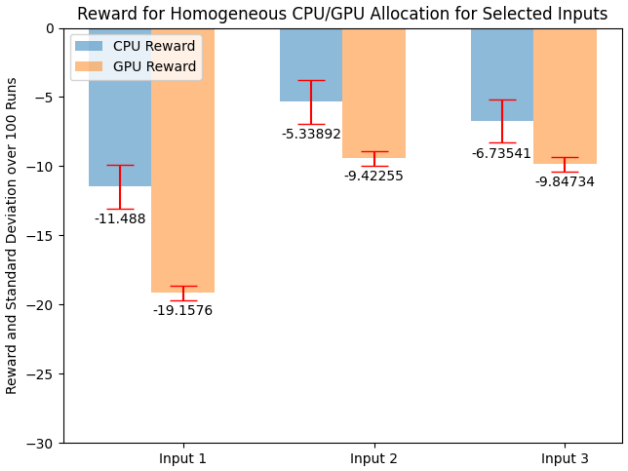


Figure 14: Mean and standard deviation of reward of 3 selected inputs, where allocations are either homogeneously on either the CPU or GPU. Run on a GTX 1650 Super GPU.

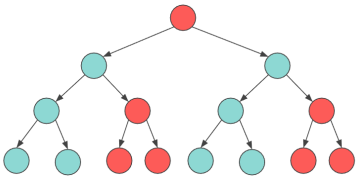


Figure 15: A 2-device setup, where red nodes represent allocation onto the CPU, and red nodes represent allocations onto the GPU. Learning to prefer locality could result in hysteresis in allocation.

of such a strategy could result in an allocation like in Figure 15, where there are some degenerate execution paths where data is repeatedly moved from the CPU to GPU and back which bottlenecks the progress of the entire system, and results in a poor training reward even though other parts of the allocation may be improving.

It would be interesting to explore to what extent this occurs, and how this can be mitigated to improve training times.

11 Conclusion

We showed that DynPartition can successfully learn how to split the computation of inference tasks across heterogeneous GPUs according to the network architecture and input data. The use of a reinforcement-learning approach allows for end-to-end modeling of the underlying factors contributing to the cost of computation in forward passes.

Although we were unable to conduct further experiments on more complex or generic dynamic network architectures due to time limitations, the evidence of learning based on the rewards that DynPartition receives still serves as a successful example of using RL-based methods for modeling system-level overheads to guide dynamic allocation of computation in deep learning tasks. We believe our approach carries potential for greatly improving the computational time-efficiency of dynamic neural networks, which would be possible with a generic encoding function for different architectures.

References

[1] Han, Y., Huang, G., Song, S., Yang, L., Wang, H., and Wang, Y. Dynamic neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence 44*, 11 (2021), 7436–7456.

[2] Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., and Chen, Z. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668* (2020).

[3] Maillard, J., Clark, S., and Yogatama, D. Jointly learning sentence embeddings and syntax with unsupervised tree-lstms. *Natural Language Engineering 25*, 4 (2019), 433–449.

[4] Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., and Dean, J. Device placement optimization with reinforcement learning. *CoRR abs/1706.04972* (2017).

[5] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning, 2013.

[6] Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2019), SOSP '19, Association for Computing Machinery, p. 1–15.

[7] Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A. Y., and Potts, C. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing* (2013), pp. 1631–1642.

[8] Wang, H., Niu, D., and Li, B. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications* (2019), pp. 1288–1296.

[9] Ye, Q., Zhou, Y., Shi, M., Sun, Y., and Lv, J. Dlb: A dynamic load balance strategy for distributed training of deep neural networks. *IEEE Transactions on Emerging Topics in Computational Intelligence* (2022), 1–11.