Simulating Dynamic Tensor Rematerialization

Altan T. Haan

A senior thesis submitted in partial fulfillment of
the requirements for the degree of

Bachelor of Science
with Departmental Honors

Paul G. Allen School of Computer Science & Engineering
University of Washington

June 2020

Supervised by Zachary Tatlock

Approved by: ______________________________
Zachary Tatlock
Assistant Professor

Presentation of work given on May 20th, 2020.

1 INTRODUCTION

In recent years, deep learning (DL) has exploded in popularity [1]. An increasingly obvious trend in DL research is the exponential growth of model sizes, as evidenced by models like BERT [2] and GPT-3 [3]. Recently, specialized DL hardware accelerators have also grown in popularity [4, 5, 6]. Consequently, new models often hit the limits of physical memory, either across all presently-available hardware or in specific memory-limited instances.

A popular approach to the memory problem is *gradient checkpointing*, which trades memory for additional compute by recomputing intermediate activations (a classic case of the space-time tradeoff). Checkpointing has the benefit of being *semantics-preserving*, unlike other approaches such as quantization [7]. Originating in the automatic differentiation community (AD), DL researchers have adapted the technique and produced increasingly optimal algorithms for training deep learning models under memory constraints. However, such approaches have assumed a static computation graph in order to perform offline planning. With DL models and applications also increasing in dynamism [8, 9, 10], these approaches cannot easily be applied.

Dynamic Tensor Rematerialization (DTR), a novel *dynamic* runtime technique for reducing memory usage during the training of DL models, addresses this problem [11]. DTR replaces the offline checkpointing problem with an online one, taking inspiration from caching, and operates as a thin runtime layer within a DL framework. Due to DTR's dynamic and embedded nature, implementations of it will invariably need to deal with the low-level implementation details of modern DL frameworks. In the case of PyTorch [12], this includes (but is not limited to) notions of aliasing, operator signatures, statefulness, and liveness. The complex interactions present in production DL systems can make debugging and designing implementations hard, which in turn hinders the efficient development of higher-level abstract algorithms.

In this paper, we present `simrd`: a simulator for Dynamic Tensor Rematerialization.[1] `simrd` bridges the gap between high-level algorithm design and low-level algorithm implementation. Through `simrd`, we evaluate DTR's theoretical performance by simulating the execution of a variety of DL models, both with and without dynamism. We additionally characterize the asymptotic computational overhead incurred by DTR, through the simulation of a classic toy model used in the literature, and analyze DTR's behavior by examining the resulting execution traces. `simrd` enables quick and efficient exploration of the algorithm design space, while capturing the core semantics of a complex DL framework.

The paper is structured as follows: in Sec. 2, a high level overview of DTR and its motivations is given; in Sec. 3, we motivate the need for a simulation; in Sec. 4, we give a comprehensive exposition of `simrd`'s design, along with precise formalizations of DTR's heuristics and operation; in Sec. 5, we use `simrd` to investigate the performance of DTR under various settings, across a variety of benchmarks; lastly, we give concluding remarks in Sec. 6, and reflect on the benefits of having a simulation during the research and development cycle.

[1] `simrd` is pronounced like "simmered."

2 BACKGROUND: DYNAMIC TENSOR REMATERIALIZATION

In order to understand the importance of `simrd`, we give brief motivations for DTR, along with a high-level summary of DTR's operation and core heuristics.

2.1 *Previous Work*

As mentioned in the introduction, *gradient checkpointing* is a popular approach to solving the memory problem in DL research. Originating in the AD community [13, 14, 15], *checkpointing* works by freeing certain intermediate computations and recomputing them as needed from "checkpointed" values during backpropagation (see [16] for backpropagation, although the idea itself is even older). In DL research, checkpointing exploits the fact that intermediate *activations* take up most of the used memory during training [17].

In Chen *et. al* [18], the authors develop a simple checkpointing scheme for linear DL models of n layers, which involves placing checkpoints evenly every $\sqrt{n}$ layers. This reduces the peak required memory to $2\sqrt{n}$ (assuming all layers have unit size), while only requiring n additional computations. While this technique only works for linear networks (which excludes models like ResNet [19]), the authors give a greedy algorithm which works on general computation graphs (although without any optimality guarantees).

The authors of [20] refine Chen's algorithm by explicitly considering size and computation time in their checkpointing scheme. In [21] and [22], the authors examine the problem from a more graph-theoretic perspective, and make use of techniques from graph theory to produce increasingly general and optimal solutions. Lastly, a recent paper introduces Checkmate [23], which transforms scheduling into a constraint solving problem, and uses an integer linear programming (ILP) solver to obtain optimal schedules. Note that [21] and [23] rephrase the checkpointing problem as *tensor rematerialization*, taking inspiration from register rematerialization [24].

2.2 *Dynamic Tensor Rematerialization*

While prior work on automatic gradient checkpointing (and subsequently, *tensor rematerialization*) has been steadily improving in optimality, they have essentially been limited to static models. In some cases, dynamic models such as RNNs can be "unrolled" [25] into large static models which can then be tackled using static methods. However, this is not generalizable; highly complex models like the Neuro-Symbolic Concept Learner [9] cannot be "unrolled" in any sensible way. Lazily executing dynamic models (in order to build a static computation graph) also fails in the case of control flow which depends on computed values.

A DYNAMIC APPROACH. DTR approaches the problem from a fundamentally dynamic perspective. In particular, DTR treats GPU memory as a large cache with a high miss penalty; tensors can be evicted and cached by freeing their memory and (re)computing them, respectively. When an operation needs more free memory than is available, DTR evicts tensors from memory until there is sufficient free space. On the other hand, when an evicted tensor t is needed by an operation, t is recomputed. Note that this recomputation may incur further recursive recomputations, if previous dependencies are also evicted.

In this way, DTR builds up a computation graph while eagerly executing model code, in the style of PyTorch [12]. Consequently, from the perspective of the algorithm, arbitrary dynamism is indistinguishable from static model architecture. Algorithms 1 and 2 show the high-level operation of DTR.

EVICTION HEURISTIC. At the core of DTR's operation is an *eviction heuristic* h. This heuristic assigns a value $h(t)$ to each resident tensor t, representing (in some approximate way) the cost to evict t. Whenever DTR needs to evict tensors, it repeatedly evicts the lowest cost tensor determined using h. Importantly, h can factor in arbitrary metadata such as tensor size or age, although these metadata must be maintained by the runtime system.

```
fn PerformOp($op$, $[x_1, \dots, x_n]$, $y$) :=
    Input: operator $op$, tensors $x_1, \dots, x_n$, destination tensor $y$
    Result: stores $y \leftarrow op(x_1, \dots, x_n)$
    foreach $x \in [x_1, \dots, x_n]$ do
        if $x$ is evicted then
            PerformOp($x.op$, $x.op.inputs$, $x$);
        end
    end
    while insufficient memory do
        Evict();
    end
    $y \leftarrow op(x_1, \dots, x_n)$;
```

Algorithm 1: `PerformOp`, which intercepts every operator call.

```
fn Evict() :=
    Result: the cheapest resident tensor is evicted
    let $t^* := \arg\min_{\text{resident tensor } t} h(t)$;
    deallocate $t^*$ and mark as evicted;
```

Algorithm 2: `Evict`, which frees memory as needed.

2.3 *The h_{DTR} heuristics.*

As part of DTR's design, the other authors and I proposed a class of related heuristics called h_{DTR}. At a high level, the h_{DTR} heuristics formalize the notion of the "cheapest to compute, least likely to be needed, and largest" tensor. To capture these three properties, h_{DTR} is defined parametrically as

$$h_{DTR}(s, m, c)(t) := \frac{c(t)}{m(t) \cdot s(t)},$$

where s, m, c are measures of tensor *staleness*, *size*, and *compute cost* respectively.

Each measure can be naturally defined in terms of per-tensor metadata such as last access time, memory buffer size, and operator compute time respectively. However, there is a far more precise notion of *compute cost* which is highly nonlocal. It follows from the following observation: for an evicted tensor t, the total compute cost incurred by rematerializing t is at least the *sum of the operator compute times for all the evicted tensors which* t *recursively depends on*, in the sense of the recursion in Algorithm 1. This global structural knowledge is absolutely critical in preventing long chains of evicted tensor dependencies, and is the core insight which makes simple static checkpointing schemes work (such as in Chen's $\sqrt{n}$ scheme [18]).

For this reason, we introduced the notion of the *evicted neighborhood* $e^*(t)$ of a given tensor t. Intuitively, this is the set of evicted tensors which t recursively depends on (as above), together with those evicted tensors which recursively depend on t. We also introduced an undirected relaxation $\tilde{e}^*$ of e^*, which incurs lower maintenance overhead. Both of these constructions will be formally defined in Sec. 4.2. With these, the observation above is effectively captured, which leads to three main variants of h_{DTR}:

$$\texttt{DTR-Full}(t) := \frac{c(t) + \sum_{u \in e^*(t)} c(u)}{m(t) \cdot s(t)}, \quad \texttt{DTR-EqClass}(t) := \frac{c(t) + \sum_{u \in \tilde{e}^*(t)} c(u)}{m(t) \cdot s(t)},$$

$$\texttt{DTR-Local}(t) := \frac{c(t)}{m(t) \cdot s(t)}.$$

`DTR-Local` can be thought of as the naive (but low-maintenance) baseline variant. In all cases, s, m, c are approximated using per-tensor metadata as above.

3 WHY SIMULATE?

While it may be tempting to immediately begin prototyping DTR within a DL framework, this approach has several pitfalls from both a research and engineering perspective.

3.1 *Specification Incompleteness*

The description of DTR given in Sec. 2.2 and algorithms 1 and 2 is, in some sense, complete. It gives all that is needed to *get started* on implementing DTR in a real DL framework. However, subtleties quickly appear when we think concretely about how to proceed. For example, what happens when one input to an operation gets evicted while we are rematerializing another? This is clearly a situation that shouldn't happen, so we need to introduce some notion of locking. Similarly, we need to consider what happens when no more resident tensors can be evicted.

Additionally, does the order in which we rematerialize arguments matter? Since evictions are performed on-demand, the specific sequence of queued recomputations directly affects the peak memory required *and* the incurred computational overhead. This is a hidden "hyperparameter" of the algorithm which may be easily missed during implementation. Other hyperparameters might only become evident deep into the implementation process, at which point their effects on the system as a whole may be hard to gauge.

3.2 *Implementation Complexity*

Digging into a DL framework exposes further problems that go beyond specification incompleteness. For example in PyTorch [12], operators are functions from tensors to tensors, not just a single output tensor. These operators cannot be decoupled into sub-operators for each output tensor, so rematerializing one output causes the rest to be rematerialized as well. On a more representational level, PyTorch has a notion of *aliasing* that goes beyond the language-level. For example, the transpose of a matrix tensor in PyTorch refers to the *same underlying memory* as the original tensor. It is unclear what "evicting the transpose" would (or should) mean in this case. Each tensor is also reference-counted and freed when all references are lost; this interaction must be carefully handled by DTR to prevent memory leaks and performance degradation. These problems represent a fundamental mismatch between DTR's (implicit) model and the reality of a DL system.

Further complexities include in-place operators (which mutate tensor data), non-rematerializable tensors such as inputs and weights, and specific state invariants which the codebase assumes (either explicitly or implicitly). Large-scale interactions within the PyTorch codebase can also be subtle, as evidenced by the intricate design of PyTorch's automatic differentiation subsystem [26], which had to deal with many of the above complexities.

3.3 *Theory and Practice*

From a research perspective, devoting significant amounts of time and energy into the low-level implementation of an algorithm, without having analyzed it beforehand, presents a non-negligible risk. In the best case, the algorithm performs well, which could motivate further theoretical analysis backed up by empirical results. However, in the worst (or perhaps even "average") case, empirical results of the algorithm may simply be bad. In this case, not only must a decision be made as to whether or not the research should be continued, there is also considerable uncertainty about *why* the algorithm failed to perform. If the research continues, even more time must be invested in profiling the algorithm *in situ* to identify the reasons for the failure. This requires a careful analysis of both the underlying DL framework and the behavior of the core algorithm. Alternatively, if the research is discontinued, the researchers may forever be plagued with the possibility that the idea was indeed good, but the execution subpar.

Regardless of the outcome, in the DL memory-efficiency literature, analytical bounds of algorithmic performance are extremely common (and even expected) [18, 21, 22, 20, 23, 27]. It stands to reason that an effective presentation of DTR should likewise include an analysis of its theoretical performance; indeed, this is simply good research practice and etiquette. Given this precondition, there is little benefit to diving into implementation first, before a theoretical analysis is performed. Obtaining good theoretical bounds on the algorithm allows for more time to be spent on specific implementation optimizations, while a negative result can help move research in more promising directions.

3.4 *Simulations in the Literature*

Simulations have also been used in prior DL memory savings research to investigate the performance of proposed algorithms. In [27], the authors design algorithms for memory-efficiently training RNNs using backpropagation through time (BPTT). To evaluate their algorithm designs, they simulate them under various conditions, and in some cases compare the simulations to real implementation results. These simulations were also used in the proof of an analytical bound, providing evidence by numerically checking large inputs.

In the case of DTR, a simulator can be further thought of as a *static instantiation* of the algorithm. By feeding the simulator a static computation graph extracted from a DL framework (with an associated cost model), we can use the resulting execution schedule as a statically planned checkpointing scheme. Note that both [20] and [23] obtain approximate costs of tensor operations by first running each operation using random inputs and recording the time; this can be directly compared to our PyTorch logging approach (see Sec. 4.6).

4 SIMULATOR DESIGN

To address the problems raised in Sec. 3, a simulator for DTR thus needs to:

1. sufficiently model the *core* semantics of the underlying DL system,
2. provide a complete specification of DTR, and
3. enable efficient exploration of the algorithm design space.

Since DTR's emphasis is on *dynamic* models, we chose to target the PyTorch DL framework [12]. PyTorch executes model code eagerly (*e.g.* relying on the Python interpreter), as opposed to building a static computation graph using an embedded DSL (which is popular in frameworks like TensorFlow [28]). As a consequence, PyTorch supports arbitrary (Python) control flow, which has enabled the design and implementation of models with highly complex dynamism such as the Neuro-Symbolic Concept Learner [18]. In this section, we show how `simrd`'s design satisfies these three key requirements.

4.1 *Fundamental Abstractions*

To match the core semantics of PyTorch, `simrd` needs to complicate the simple description of DTR given in Sec. 2.2. More specifically, `simrd` models PyTorch's *tensor-storage* distinction, in which each *tensor* object is a particular *view* of an underlying memory buffer called a *storage*. A view can be thought of as metadata which gives structure to raw memory, by encoding how it should be accessed. This allows PyTorch to replace certain (usually) costly "tensor" operations with metadata operations. For example, transposing a matrix in PyTorch does not require the memory to be copied and reordered, but is instead a cheap metadata operation. The following abstractions capture this behavior.

STORAGE. At its core, DTR is a runtime system for reducing memory usage. As such, *storages* (*i.e.*, buffers of memory) are the underlying unit which DTR operates on. Each storage S supports the following operations:

- $\mathit{size}(S) : \mathbb{N}$, the size of S in bytes.
- $\mathit{root}(S) : \textbf{Tensor}$, the tensor whose parent operation computes the contents of S (there is exactly 1 for each storage).
- $\mathit{tensors}(S) : \textbf{List}[\textbf{Tensor}]$, all tensors which view the S.
- $\mathit{resident}(S) : \textbf{bool}$, true iff S is in memory.
- $\mathit{locks}(S) : \mathbb{N}$, the number of locks on S held by DTR. S is locked once for each pending rematerialization that requires it, and released once for each completed rematerialization. This prevents the eviction problem mentioned in Sec. 3.
- $\mathit{refs}(S) : \mathbb{N}$: the number of external references to S, *i.e.*, those held by user code. This models the reference counting mechanic employed by PyTorch.

Note that a storage S is *evictable* if and only if $\mathit{resident}(S) \wedge \mathit{locks}(S) = 0$. When a storage S loses all external references (*i.e.*, when $\mathit{refs}(S) = 0$), PyTorch permanently frees it from memory since it is no longer accessible by the source program. We call this behavior *banishing* as opposed to eviction, and describe how it can be modeled by `simrd` in Sec. 4.4.

TENSOR. Each tensor t supports the following operations:

- $\mathit{storage}(t) : \textbf{Storage}$, the storage viewed by t. Note that t is an alias (denoted $\mathit{alias}(t)$) if and only if $t \neq \mathit{root}(\mathit{storage}(t))$.
- $\mathit{op}(t) : \textbf{Op}$, the parent operation which computes t (*i.e.*, the view metadata), along with its underlying storage if and only if $\neg \mathit{alias}(t)$.
- $\mathit{refs}(t) : \mathbb{N}$, the number of external references to t. Note that for each storage S we have $\mathit{refs}(S) = \sum_{t \in \mathit{tensors}(S)} \mathit{refs}(t)$.
- $\mathit{size}(t) : \mathbb{N}$, the size of t, defined as 0 if $\mathit{alias}(t)$ and $\mathit{size}(\mathit{storage}(t))$ otherwise.

- $\mathit{defined}(t)$: **bool**, true if and only if $\mathit{op}(t)$ has been performed *after* the last time $\mathit{storage}(t)$ was evicted. Note $\neg \mathit{alias}(t) \implies (\mathit{defined}(t) \iff \mathit{resident}(\mathit{storage}(t)))$.

A tensor t can be used in a computation if and only if $\mathit{defined}(t)$; this notion of definedness models the assumption that evicting a storage S destroys all tensor objects which view S. In principle, it may be possible in PyTorch to keep the tensor objects (since they're just metadata) while evicting storages, but this could heavily increase code complexity due to internal invariants which assume non-null storages. Regardless, this can be changed easily to better model a given DTR implementation.

OPERATOR. We assume operators have type **List**[**Tensor**] $\to$ **List**[**Tensor**], and that they are pure in their arguments (*i.e.*, do not depend on any other external state). Each operator *op* supports the following operations:

- $\mathit{cost}(\mathit{op}) : \mathbb{N}$, the compute cost of *op*.
- $\mathit{inputs}(\mathit{op})$: **List**[**Tensor**], the input tensors to *op*.
- $\mathit{outputs}(\mathit{op})$: **List**[**Tensor**], the output tensors to *op*.

`simrd` can, however, support mutating operations: see Sec. 4.6 for details.

4.2 *Formal Metadata Definitions*

While our abstract description of DTR in Sec. 2 is over tensors, `simrd` operates over storages rather than tensors. Thus, we must correctly and completely define the metadata our heuristics use over storages, providing notions of cost, staleness, and data dependencies for storages rather than for tensors.

COST. For a given storage S, we define the compute cost of S as

$$\mathit{cost}(S) := \sum_{t \in \mathit{tensors}(S)} \mathit{cost}(\mathit{op}(t)).$$

This is a worst-case estimation: it represents the compute cost which is incurred when every tensor view of S needs to be rematerialized. An alternative definition is simply $\mathit{cost}(\mathit{op}(\mathit{root}(S)))$, which may be acceptable as aliasing operations are typically much cheaper than non-aliasing.

STALENESS. We estimate the staleness of S by tracking the *last access* time of each $t \in \mathit{tensors}(S)$. The last access time $\mathit{last\_access}(t)$ is defined as the most recent time when t was referenced by a queued operation. Naturally, we define $\mathit{last\_access}(S) = \max_{t \in \mathit{tensors}(S)} \mathit{last\_access}(t)$. Staleness, given the current time $\mathcal{T}$, is then defined as $\mathit{stale}_{\mathcal{T}}(S) := \mathcal{T} - \mathit{last\_access}(S)$.

DATA DEPENDENCIES. The dependencies of S are the set of storages

$$\mathit{deps}(S) := \{\mathit{storage}(u) \mid \exists t.\ t \in \mathit{tensors}(S) \wedge u \in \mathit{inputs}(\mathit{op}(t))\} \setminus \{S\}.$$

Note that we exclude S since it is not a true dependency (each alias tensor in $\mathit{tensors}(S)$ technically "depends" on S). Another possible approximation of the above is to simply take the dependencies of $\mathit{root}(S)$; although this ignores potential dependencies of aliasing operations, it is precise if all aliasing operations only depend on S.

We now define the *dependents* of S as the set $\mathit{deps}^{\top}(S)$ consisting of all T with $S \in \mathit{deps}(T)$. With this definition, DTR can operate over the dependency graph (V, E) where V is the set of storages and $(S, T) \in E$ iff $S \in \mathit{deps}(T)$. Note that (V, E) is implicitly indexed by time $\mathcal{T}$, with V being the set of at-least-once computed storages at $\mathcal{T}$ and E being the dependency relations at $\mathcal{T}$. Note that V excludes all banished tensors.

EVICTED NEIGHBORHOOD. The *evicted neighborhood* e^*, as introduced in Sec. 2.3, works without modification over the storage dependency graph. We now give a formal definition. Let $\mathit{deps}_e(S)$ be the evicted subset of $\mathit{deps}(S)$, and likewise for $\mathit{deps}_e^{\top}(S)$. Now, let D_e and $D_e^{\top}$ be the transitive closures of the relations

$$\{(T, S) \mid T \in \mathit{deps}_e(S)\} \quad \text{and} \quad \{(S, T) \mid T \in \mathit{deps}_e^{\top}(S)\},$$

respectively. Then, $e^*(S) := \{T \mid (T, S) \in D_e\} \cup \{T \mid (S, T) \in D_e^{\top}\}$. Intuitively, $e^*(S)$ is the set of evicted storages that must be resident to compute all $t \in \mathit{tensors}(S)$, together with the set of evicted storages T that need S to be resident before all $t \in \mathit{tensors}(T)$ can be computed.

RELAXED (`eqclass`) EVICTED NEIGHBORHOOD. Actually tracking $e^*(S)$ can be computationally expensive due to the directed and changing nature of the graph. For each S, $e^*(S)$ depends on its specific ancestors and descendants; there does not appear to be a simple way of maintaining a single global data structure to track this information as tensors are evicted and rematerialized. A solution will likely involve a dynamic graph connectivity data structure, which would greatly increase the complexity of `simrd`'s implementation.

We approach this problem by relaxing the definition of the evicted neighborhood. At a high level, our solution works as follows: given a storage dependency graph $G = (V, E)$, we first forget edge directions to obtain the undirected dependency graph $\tilde{G}$. Now, let $\tilde{G}_e$ be the subgraph obtained by removing all resident storages (and any edges including them). Each connected component of $\tilde{G}_e$ is then an *evicted component*, with each evicted $T \in V$ belonging to exactly one component $\epsilon^*(T)$. Then, the (relaxed) evicted neighborhood for a resident storage S is defined as

$$\tilde{e}^*(S) := \left(\bigcup_{T \in \mathit{deps}_e(S)} \epsilon^*(T) \right) \cup \left(\bigcup_{T \in \mathit{deps}_e^{\top}(S)} \epsilon^*(T) \right).$$

Note the structural similarity in this definition with $e^*(T)$; they are indeed similar, but $\tilde{e}^*(S)$ overapproximates the neighborhood by ignoring edge directions. Each evicted component can be efficiently represented using a *Union-Find* (or *disjoint-set*) data structure with very good asymptotic complexity for merging and obtaining static set metadata. In the case of DTR, each component tracks the sum of the compute costs of its elements (with the union of two components having the sum of each constituent cost). This enables very cheap querying of compute costs over $\tilde{e}^*(S)$.

However, despite this optimization, *splitting* is not a supported operation on disjoint-sets.[2] Approaches to splitting would also need to recover the original compute costs of each set, which may require traversing the whole set if done naively. Unforunately, DTR regularly splits evicted components during rematerialization. In order to deal with this, we use the following overapproximation: when a (previously) evicted storage S belonging to $\epsilon^*(S)$ is rematerialized, we set $\epsilon^*(S).cost := \epsilon^*(S).cost - cost(S)$. While resident storages thus never count towards the compute cost of a component, "phantom connections" between evicted storages may accumulate over time (likely depending on the connectedness of the underlying dependency graph). Despite this limitation, this approximation worked well in practice, as shown in Sec. 5.3.

4.3 *Formal Heuristic Definitions*

Having defined the metadata above, we can now formally define the h_{DTR} variants mentioned in Sec. 2.3. (Recall that h_{DTR} heuristics compute a score using measures of size, computational cost, and staleness and evict the tensor with the smallest score, corresponding to the intuition that the tensor evicted should be large, unlikely to be rematerialized, and cheap to rematerialize if it does need to be rematerialized.)

$$\texttt{DTR-Full}(S) := \frac{cost(S) + \sum_{T \in e^*(S)} cost(T)}{size(S) \cdot stale_{\mathcal{T}}(S)},$$

$$\texttt{DTR-EqClass}(S) := \frac{cost(S) + \sum_{T \in \tilde{e}^*(S)} cost(T)}{size(S) \cdot stale_{\mathcal{T}}(S)} \approx \frac{cost(S) + cost^*_{\epsilon}(S)}{size(S) \cdot stale_{\mathcal{T}}(S)},$$

$$\texttt{DTR-Local}(S) := \frac{cost(S)}{size(S) \cdot stale_{\mathcal{T}}(S)}.$$

Note that `simrd`'s implementation of `DTR-EqClass` uses the splitting approximation described above, with $\tilde{e}^*(S)$ depending on the specific sequence of evictions and rematerializations. $cost^*_{\epsilon}(S)$ in the second expression represents this statefulness as a sum over the stateful neighboring ϵ^* costs.

BASELINE HEURISTICS. For completeness, we define a few "baseline" heuristics that are intuitive and have been used in prior work:

$$\texttt{LRU}(S) := \frac{1}{stale_{\mathcal{T}}(S)}, \quad \texttt{Largest}(S) := \frac{1}{size(S)}, \quad \texttt{Random}(S) := X \sim U(0,1).$$

The `LRU` (least recently used) heuristic is a common cache eviction policy [29], while `Largest` is similar to `GreedyRemat` from [21].

[2] This can be seen as a variant of the Union-Find-Split problem, which typically requires the use of more complex data structures such as link-cut trees.

4.4 *Implementation Details*

RUNTIME STATE. In what follows, we denote the collective runtime state of `simrd` as R, and use the dot notation to indicate stateful reads and writes of runtime values. Specifically, `simrd` tracks the following runtime state:

- $R.h : (\textbf{Storage}, \textbf{Metadata}) \to \mathbb{R}$, the eviction heuristic, interpreted as an eviction cost (the lowest-cost storage is evicted). We write $R.h(S)$ for convenience, when the choice of metadata can be inferred from the heuristic.
- $R.\texttt{budget} : \mathbb{N}$, the memory budget in bytes.
- $R.\texttt{memory} : \mathbb{N}$, the current memory usage in bytes.
- $R.\mathcal{T} : \mathbb{N}$, the current clock time in some unit of granularity, such as nanoseconds.
- $R.\texttt{pool} : \textbf{List}[\textbf{Storage}]$, list of all currently evictable storages.

EVICTION AND BANISHING. To evict a given storage S, we set all tensors in S to be undefined, remove S from the pool, and decrease $R.\texttt{memory}$ by $\mathit{size}(S)$. Cached metadata are also updated as necessary.

Banishing, which is *permanent* eviction, is slightly more subtle; in particular, it can only be done for S when $\mathit{deps}_e^\top(S) = \emptyset$. Banishing then proceeds by evicting S as above, but with the additional effect of removing S entirely from the dependency graph. Each $T \in \mathit{deps}^\top(S)$ is then locked (and effectively becomes an non-rematerializable constant). Storages locked in this way are said to be *pinned* (and have a special flag in `simrd`), to distinguish them from those locked during rematerialization, and we permit them to be banished in the future. Note that banishing can be performed on evicted S when the above condition is met, in which case the eviction is skipped.

(RE)MATERIALIZATION. When a tensor t is to be (re)materialized, its parents' storages are first locked by incrementing the lock count (so that they don't get evicted while they are still needed) and undefined parents are recursively rematerialized. We fix a rematerialization order based on tensor identifiers, which are unique and monotonically increasing.[3] We then increment $R.\texttt{memory}$ by $\sum_{u \in \mathit{outputs}(\mathit{op}(t))} \mathit{size}(u)$ (performing evictions as necessary), and move $R.\mathcal{T}$ forward by $\mathit{cost}(\mathit{op}(t))$. Multi-output operations must be handled carefully so as to not leak memory: we make sure to *decrease* $R.\texttt{memory}$ by $\mathit{size}(u')$ for each $u' \in \mathit{outputs}(\mathit{op}(t))$ that was defined *prior* to the rematerialization. This models the immediate freeing of doubly-computed ephemeral tensors in the PyTorch implementation. Lastly, locks on parent storages are freed and unlocked storages (including any newly rematerialized ones) are added back into $R.\texttt{pool}$.

CONSTANTS. `simrd` models non-rematerializable constants like weights and inputs by creating dummy "constant" tensors using nullary operators with 0 cost and pinning the resulting storage. This allows `simrd` to have a full picture of the computation

[3] Different orderings, such as by tensor size, are possible but left to future work.

graph. Furthermore, log-accurate banishing (which can free pinned memory) requires knowledge of constants, as PyTorch reference-counts constants.

4.5 *Additional Runtime Optimizations*

EAGER EVICTION. When the final external reference to a storage S is lost, we know that the underlying DL framework would have reclaimed the memory used by S. To utilize this information as opposed to doing nothing, we can either banish S or simply evict S normally. When banishing, must first check that S has no evicted dependents; if it does, then we retry banishing each time a dependent is rematerialized. Banishing has the ability to evict constants, but at the downside of pinning potentially exploding amounts of memory. The alternative (*eager eviction*) is easier to implement and simply involves evicting S normally (if possible). This prevents the problem of over-pinning memory, but with the downside that constants can never be evicted. In practice, eager evictions allowed us to save more memory (see Sec. 5.4 for details).

CACHING METADATA. To avoid costly recomputations of metadata during heuristic evaluations, we cache the local cost $\mathit{cost}(S)$ for each S (as it only changes when new aliases are made). Additionally, for the `DTR-Full` heuristic, we avoid recomputing $e^*(S)$ at each evaluation by caching and only recomputing after evictions or rematerializations that directly affect $e^*(S)$. Such recomputations are further optimized by tracking the evicted ancestors and descendants separately (allowing them to be recomputed independently, depending on the position of the affected storage).

4.6 *Log-Replaying Mechanism*

LOG FORMAT. We logged PyTorch operations as a sequence of abstract instructions corresponding to the semantics of the actions we were easily able to instrument in the framework. Every PyTorch tensor is given a unique identifier string upon creation, which is recorded and used in the log. In this section, each PyTorch tensor t corresponds to a simulator tensor $[\![t]\!]$.

The log contains the following instructions:

- `MEMORY`(t, size): logs that t uses *size* memory; treated as 0 if $\mathit{alias}([\![t]\!])$.
- `ALIAS`(t_o, t_i): logs that $[\![t_o]\!]$ is an alias of $[\![t_i]\!]$, *i.e.*, two different views of the same storage. t_i can either be a tensor identifier or $\perp$; if $t_i = \perp$, then t_o does not alias another tensor (t_o's parent operation created its storage).
- `CALL`$(\mathit{inputs}, \mathit{outputs}, \mathit{cost}, \mathit{op})$: logs the operator call $\mathit{outputs} = \mathit{op}(\mathit{inputs})$ with compute cost *cost*. This instruction is followed by $|\mathit{outputs}|$ `MEMORY` and `ALIAS` instructions to log information about each output. Each `CALL` corresponds to a simulator operator $[\![\mathit{op}]\!]$ with inputs $\{[\![i]\!] \mid i \in \mathit{inputs}\}$ and new simulator tensor outputs $\{[\![o]\!] \mid o \in \mathit{outputs}\}$.
- `MUTATE`$(\mathit{inputs}, \mathit{inputs}', \mathit{cost}, \mathit{op})$: logs the in-place (mutating) operator call $\mathit{op}(\mathit{inputs})$ with compute cost *cost*, which modifies $\mathit{inputs}' \subseteq \mathit{inputs}$.

- `CONSTANT`(t): logs that $[\![t]\!]$ is a constant, and is followed by a `MEMORY` instruction.
- `COPY`(t_o, t_i): logs a new identifier t_o with $[\![t_o]\!] = [\![t_i]\!]$. This increments $\mathit{refs}([\![t_i]\!])$. This happens when Python code like "`x = y`" is called where `y` is a PyTorch tensor and `x` is a fresh variable; this action neither creates a new storage nor a new view but only has `x` *point* to the same view as `y`.
- `COPYFROM`(t_o, t_i): logs the PyTorch code $t_o = t_i$ where each side is an existing tensor. This decrements $\mathit{refs}([\![t_o]\!])$, increments $\mathit{refs}([\![t_i]\!])$, and updates $[\![t_o]\!] \mapsto [\![t_i]\!]$. Intuitively, this corresponds to Python code like "`x = y`" where `y` is a PyTorch tensor and `x` was already assigned to a PyTorch tensor; in PyTorch, `x` is mutated to match `y`.
- `RELEASE`(t): logs the destructor of the PyTorch tensor t. This decrements $\mathit{refs}([\![t]\!])$.

SUPPORTING MUTATION. To support mutation from in-place operators, `simrd` adds a "reference layer" that mutates cloned tensors, allowing for a uniform interface for all operators. Given a mutation instruction `MUTATE`$(\mathit{inputs}, \mathit{inputs}', \mathit{cost}, \mathit{op})$, let i_{new} be a new unique identifier for each $i \in \mathit{inputs}'$, and let $\mathit{inputs}'_{\mathit{new}} = \{i_{\mathit{new}} \mid i \in \mathit{inputs}'\}$. We then proceed by treating op as a pure operator from inputs to $\mathit{inputs}'_{\mathit{new}}$, where each newly created simulated tensor $[\![i_{\mathit{new}}]\!]$ is non-aliasing and has size $\mathit{size}(\mathit{storage}([\![i]\!]))$. Lastly, we decrement $\mathit{refs}([\![i]\!])$ and update the mapping $[\![i]\!] \mapsto [\![i_{\mathit{new}}]\!]$. Intuitively, we are modeling the transformation

$$\mathit{op}(t) \rightsquigarrow \texttt{Tensor}\ t' = \mathit{copy}(t); \mathit{op}(t'); t = t'.$$

Note that in a naive PyTorch implementation of DTR, a mutation of i may produce incorrect results when $[\![i]\!]$ is an alias, since the mutation layer would create a clone but aliases would still point to the old storage. Potential solutions in real implementations would be to propagate the above transformation to all aliases of a storage (costly) or to mutate storage pointers (which may lead to significantly increased implementation complexity).

OUTPUT CONDITION. All live tensors at the end of a log (*i.e.*, all t with $\mathit{refs}(t) > 0$) are treated as outputs which the users want (*i.e.*, gradients, loss, prediction). They are thus rematerialized (if not defined) and locked to ensure they persist. This prevents `simrd` from incorrectly reporting better results by evicting computed weight gradients and never rematerializing them. This permits the user to perform the weight update step outside of DTR immediately after the backward pass ends.

5 EVALUATION

Now that we have a formal specification of DTR, along with several potential heuristics, we can evaluate its performance in various scenarios using various metrics. In this section, we characterize the performance of DTR in both in an asymptotic sense and in a practical, real-world context. We find that DTR can attain asymptotically optimal performance, while also producing schedules on real models with excellent computational overhead.

5.1 *Asymptotic Performance*

PROBLEM DEFINITION. To characterize the asymptotic performance of DTR under the h_{DTR} variants, we adopt a theoretical setup which models Chen *et. al*'s analysis [18]. Figure 1 shows the computation graph of the toy model we consider; each t_i

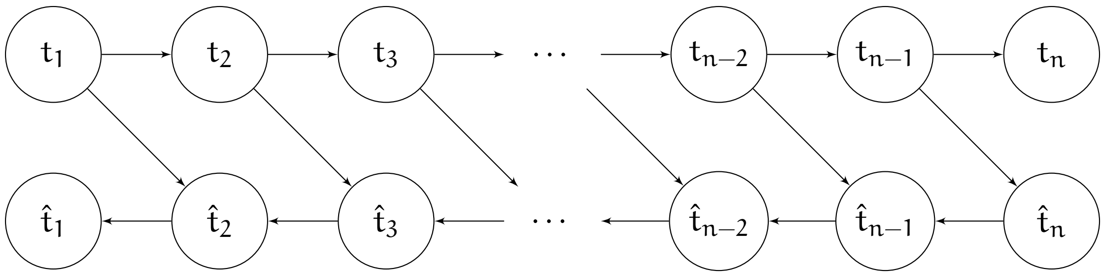

Figure 1: A simple linear, feed-forward network model and its associated gradient graph.

represents the ith layer in the network, and $\hat{t}_i$ the corresponding gradient. As in [18], we assume all tensors have unit size and compute cost, and that there is no aliasing. Note that we omit constants from the graph, since they do not affect asymptotic performance.

We then wish to answer the following: how many *additional computations* does DTR perform, for an n layer network given a budget B and an eviction heuristic h? Plotting the additional computations versus n thus gives a characterization of DTR's asymptotic performance.

EXPERIMENTAL SETUP. We examined the `DTR-Full` and `LRU` heuristics under $B = \lceil 2\sqrt{n} \rceil$ and $B = \lceil \log_2 n \rceil$, for an exponentially spaced set of n up to 2^{13}. To exploit liveness information (which Chen's analysis also incorporates), we utilized the banishing mechanic mentioned in Sec. 4.5, with reference counting and computation order extracted from the pseudocode program in Algorithm 3 (ignoring edge cases at the index boundaries).

```
let $ts$ := [];
forall $i \in [1, \dots, n]$ do
    $ts$.append(new $t_i$ using $t_{i-1}$);
end
let $g$ := $\hat{t}_n$ using $ts[n-1]$;
forall $j \in [1, \dots, n]$ do
    $ts$.pop();
    $g$ := $\hat{t}_{n-j}$ using $ts[n-j-1]$ and $g$;
end
```

Algorithm 3: The program whose reference counts is used for liveness information.

RESULTS. The results are shown in Figures 2 and 3. For the $\lceil 2\sqrt{n} \rceil$ budget, we plotted the theoretical performance of Chen's approach as a baseline comparison.

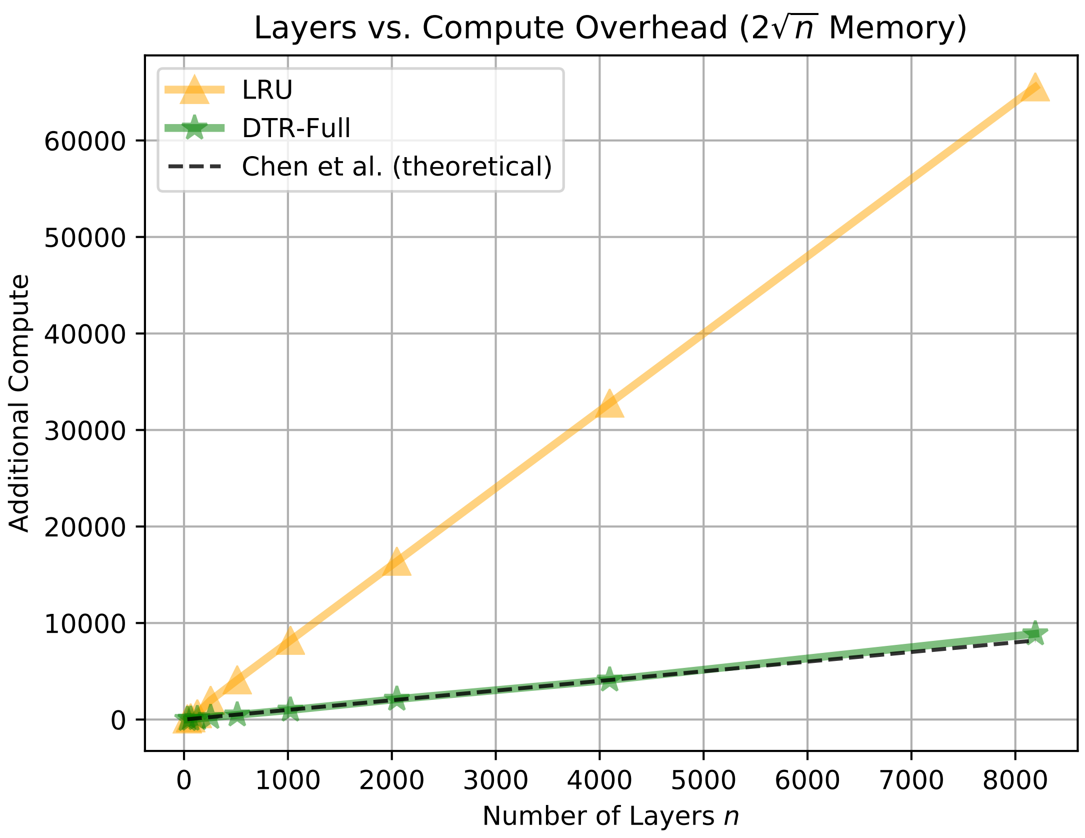

Figure 2: Additional computations vs. number of layers n, for $B = \lceil 2\sqrt{n} \rceil$.

Amazingly, the `DTR-Full` heuristic appears to match Chen's theoretical overhead almost exactly, at approximately n additional computations for a network of n layers. The simple `LRU` heuristic, while performing noticeably worse, also seems to be linear (which matches Chen's result, asymptotically). This result is highly encouraging, as it suggests that DTR can produce not only asymptotically optimal, but truly near optimal solutions without having any prior knowledge of the model.

The results in Figure 3 is also very encouraging: the computational overhead of DTR under both heuristics appears to grow like $n \log n$ under a $\Omega(\log n)$ budget, which matches asymptotically a secondary result in Chen *et. al*.

5.2 *Algorithmic Behavior*

EXPERIMENTAL SETUP. As shown in Sec. 5.1, DTR performs surprisingly well in terms of asymptotic overhead when using certain heuristics (ignoring runtime overhead). The specific behavior of DTR which enables this, however, cannot be determined from the performance graph. To address this opacity, we trace DTR's execution on this toy model by logging storage allocations and deallocations. From this data, we can reconstruct the state of memory at each timestep during the execution. Thanks to the simple linear forward-backward structure of the toy model, we can arrange each memory state as a column of cells which are either filled or empty (corresponding

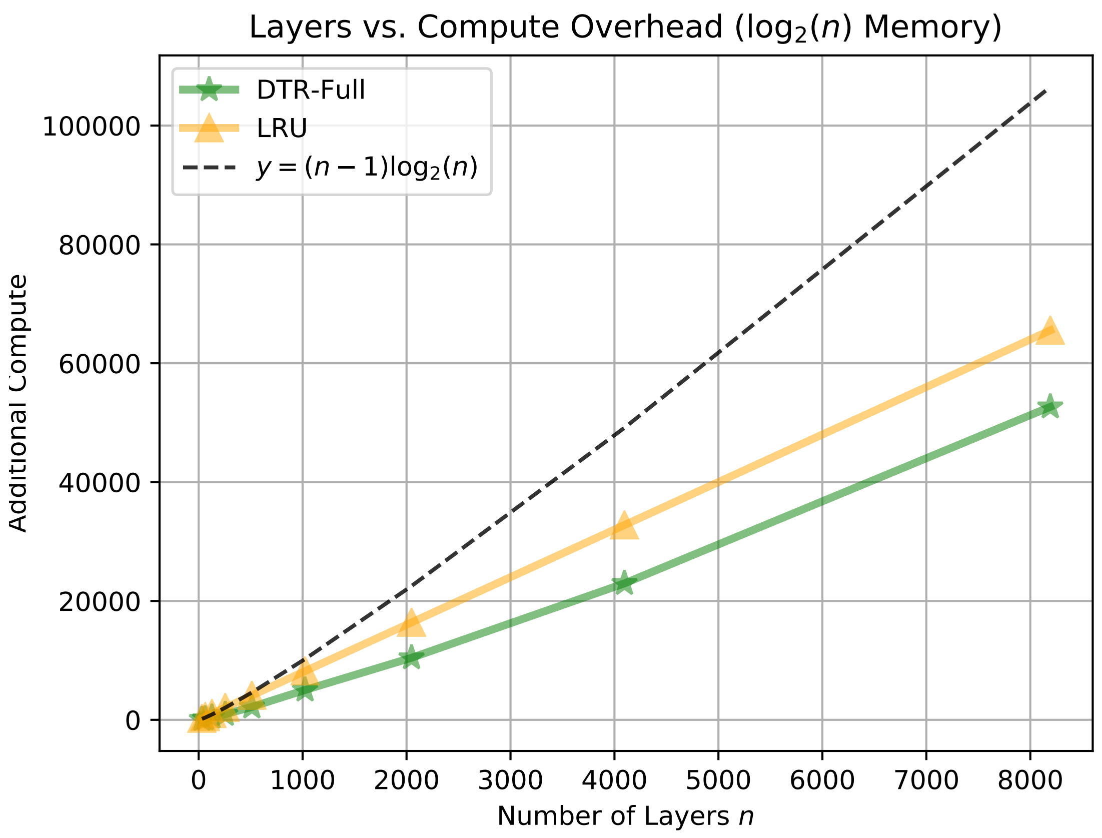

Figure 3: Additional computations vs. number of layers n, for $B = \lceil \log_2 n \rceil$.

to allocated and deallocated, respectively), with increasing y corresponding to later computations in the network. Then, placing the columns from left to right allows us to quickly visually inspect the execution. Each row then represents the memory state of a single tensor across time.

We perform this tracing for the toy model with $n = 101$ and the budget/heuristic combinations from Sec. 5.1. In each resulting trace, the brighter cells represent the corresponding gradient tensor of the forward tensor indicated by the row.

RESULTS. The memory traces are shown in Figures 7, 8, 9, 10. As the traces show, the `DTR-Full` heuristic has an intricate recursive substructure, while the `LRU` heuristic has more completely filled, "boxy" regions. This is a direct consequence of the global vs. local information available to the heuristics, respectively. The `LRU` heuristic does not "know" how to space the checkpoints as effectively as `DTR-Full`, which spaces the checkpoints (horizontal lines) more evenly, preventing long chains of rematerializations.

5.3 *PyTorch Log Performance*

EXPERIMENTAL SETUP. We ran `simrd` on logs generated from a variety of models. Specifically, we chose three static vision models [19, 30, 31] investigated in previous work [23] and three dynamic models [32, 8, 33] that exhibit different kinds of control flow. The vision models were run on batches of 32 with 3-channel images (size 32×32 for ResNet and DenseNet, 512×512 for UNet); LSTM ran on a sequence of length 32, with 10×100 entries; and TreeLSTM ran on a binary tree of depth 6, with 32×100 entries. The logs used for our simulated evaluation were produced by running each model 50 times on a single input on a machine with an NVidia Titan V GPU (CUDA 10.1, CuDNN 7.6.4) and a 16-core AMD Ryzen Threadripper 1950X on Ubuntu 18.04, using the final "warmed-up" log. The logged portion includes executing the forward pass, computing the loss, and performing the backward pass.

For all simulations, we considered a heuristic to be *thrashing* if it required at least $3\times$ the amount of computation before finishing, since we assume this to be due to long chains of unfavorable rematerializations and unviable in practice.

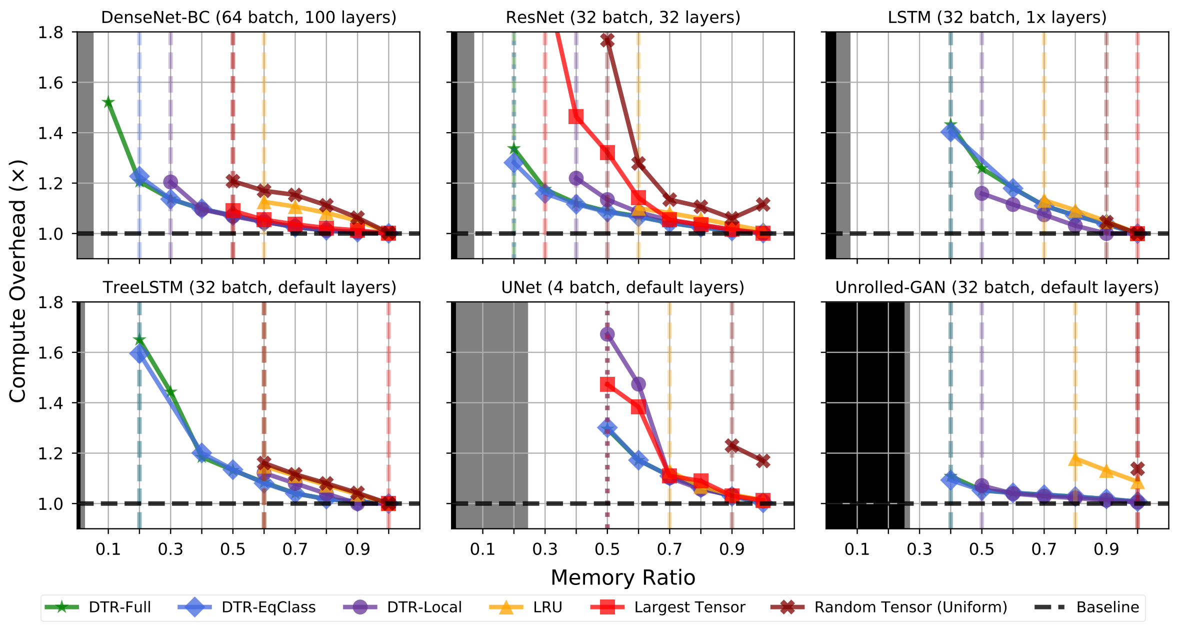

Figure 4: Simulated results comparing different heuristics on various models, comparing rate of computational slowdown for different budgets (fractions of the original peak memory usage). The black area in each graph corresponds to the memory required to store inputs and weights, while the gray area denotes the single operator requiring the most memory to be live at once. The dashed and dotted lines represent the last ratio before thrashing and out-of-memory errors, respectively.

RESULTS. For all the models in Figure 4, our simulations show significant savings at reasonable compute overheads. While we were unable to implement existing static checkpointing schemes as baselines due to the complexity of aliasing and mutation (especially in the dynamic models), we note that these results save similar amounts of memory compared to expert manual modifications to models. For example, a manually optimized DenseNet-BC model [34] achieved 56.1% memory consumption at a 25.5% slowdown, while our simulated trial using `DTR-EqClass` yields 20.0% memory consumption at 22.7% overhead. Although these numbers are not directly comparable since the simulation does not include the dynamic analysis overhead,[4] they illustrate that DTR (with suitable heuristics) can achieve memory-computation tradeoffs that otherwise justify intervention by an expert. Furthermore, unlike existing static approaches, DTR automatically saves memory on models with arbitrary dynamism, though it began to thrash at lower budgets for LSTM and TreeLSTM. In all cases, the results show that more complex heuristics achieve better memory savings with lower operator overhead, though these complex heuristics also introduce more *runtime* overhead, which must be considered in implementations of DTR. Notably, even the least sophisticated heuristics like `LRU` (requiring very little runtime overhead) achieved memory savings of up to 30% with very little overhead before thrashing, indicating that some memory savings from checkpointing can be readily obtained. See Sec. 5.5 for a more detailed analysis of runtime overhead.

LIMITATIONS. We encountered several illustrative failure modes for DTR in common public implementations of DL models. In PyTorch's official language model examples [35] and with a popular TreeLSTM implementation [36], a single bottleneck (which turned out to be an encoder or embedding) used over 50% of the baseline memory (not including inputs and weights). Obtaining memory savings in such cases is difficult, as DTR needs to compute the bottleneck while still maintaining an effective checkpointing structure. Notably, a manual PyTorch checkpointing implementation [37] was able to save memory on the language models by splitting the embedding to avoid this bottleneck. We implemented our own versions of these models (closely following the original papers) without such bottlenecks in order to capture the essence of the models' dynamic structures. These issues would arise in any automatic checkpointing scheme, illustrating that the design of a DL model is a very relevant factor in the applicability of checkpointing techniques.

h_{DTR} ABLATION STUDY. First, we will analyze the three sources of information (metadata) for the h_{DTR} heuristic. Recall that $h_{\mathrm{DTR}}(s, m, c)(t) = c(t)/[m(t) \cdot s(t)]$, where s is a measure of staleness, m is a measure of size, and c is a measure of compute cost. For this study, we take s and m to be the staleness and size functions defined in Sec. 4. For compute cost c, we compare the following alternatives (see Sec. 4 for definitions): the full e^*, the approximation $\tilde{e}^*$, and the local cost. We allow each measure to be entirely ablated (*e.g.*, $s(t) = 1$, which we denote $s = \mathrm{no}$). Note that as discussed previously, the heuristic has been lifted to operator over storages.

In figures 11, 12, 13, 14, we have $s, m \in \{\mathrm{yes}, \mathrm{no}\}$ and $c \in \{e^*, \mathrm{EqClass}, \mathrm{local}, \mathrm{no}\}$. Each figure fixes a choice of c, varying s and m. The general trend shown the figures is that higher metadata complexity (corresponding to more precise notions of the evicted neighborhood) enables more savings, while staleness and size are required for acceptable computational overhead. It is interesting to note that the importance of staleness and size is dependent on the specific model architecture. For example, cost and size alone does far better than cost and staleness for the static models (DenseNet, ResNet, UNet), whereas the opposite is true for the dynamic models. This may be due to model depth or the distribution of tensor sizes, or to the increasing impact

[4] In theory, we could extract the schedule from the simulator, and consider this an instance of static checkpointing.

of individual checkpoints at lower budgets; further research may shed more light on the influence of model-specific characteristics like these. Additionally, we may note that the $\tilde{e}^*$ approximate cost performs comparably to the e^* exact cost while requiring less information, validating our belief that the evicted components are a useful approximation (and that the stateful splitting optimization does not impact performance severely).

5.4 *Banishing and Eager Eviction*

EXPERIMENTAL SETUP. For this experiment, we compared the `DTR-Full` heuristic with banishing (permanent removal) against that with eager evictions, as described in Sec. 4.5. We only used the e^* cost because it performed much better than local cost, and because it would have been more complicated to update the definition of $\tilde{e}^*$ to account for banished neighbors. The same PyTorch logs as in Sec. 5.3 were used, with a thrashing limit of $5\times$ overhead.

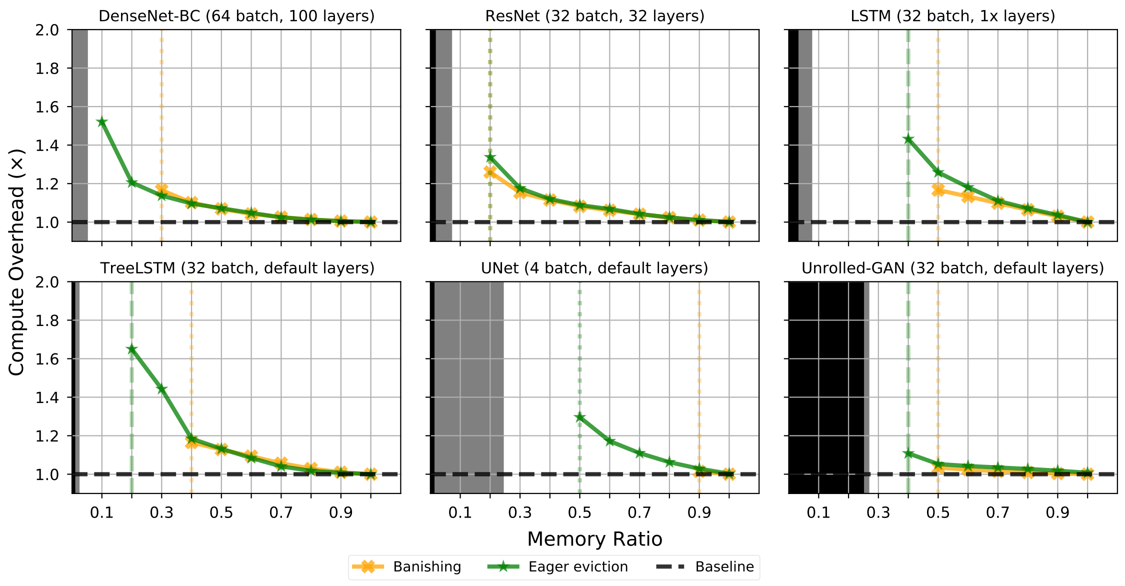

Figure 5: Results for the `DTR-Full` heuristic, comparing banishing and eager evictions.

RESULTS. As the curves in Figure 5 show, banishing is unable to achieve the same level of savings across most models tested as eager eviction. For UNet, the difference is large: banishing can only save 10% memory (and OOMs at 0.8 ratio), while eager eviction allows for 50% savings. However, banishing still achieves decent savings on most models, even obtaining better computational overhead under the same budget and savings for ResNet. Since banishing potentially allows for greatly lowered runtime overhead, implementations of DTR can consider conditionally enabling it in situations where the tradeoff is more desirable.

5.5 *Estimating Runtime Overhead*

The biggest weakness of the preceding experiments has been the lack of runtime overhead analysis. More precisely, while DTR may be able to achieve impressively low computational slowdown, such results are weakened if DTR's runtime overhead (maintaining tensor metadata, finding the cheapest tensor to evict, etc.) scale extremely poorly. In the case of static models, we can treat the simulator as an offline solver by extracting an execution schedule; however, this is clearly infeasible for dynamic models, for the same reason as existing checkpointing approaches. In this experiment, we estimate the runtime overhead of DTR using the robust metric of *storage accesses* during an execution, as will be described below.

EXPERIMENTAL SETUP. We used the same PyTorch logs as in Sec. 5.3, and compared the main three h_{DTR} variations: `DTR-Full`, `DTR-EqClass`, `DTR-Local`. Intuitively, we should expect these three variants to reside in different overhead complexity classes, as a consequence of their metadata complexity and maintenance costs.

For this experiment, we tracked the number of *storage accesses* made during evaluations of heuristics and maintenance of metadata. We chose this metric over wall-clock time, since our Python implementation of `simrd` is not heavily optimized and could potentially fail to reflect the real performance of the algorithm. Storage accesses, on the other hand, do reflect operations that would be performed by a real implementation. For the `DTR-Full` heuristic, this included each storage visited during the updating and rebuilding procedures for maintaining e^* for resident storages. For the `DTR-EqClass` heuristic, this included each storage visited whenever the Union-Find data structure was traversed for each evicted component (which occurs mainly during merging and when reading the compute cost). The `DTR-Local` heuristic does not need to maintain any non-local metadata. For all heuristics, each heuristic evaluation counted as one storage access.

RESULTS. As Figure 6 shows, the accesses made by each heuristic are generally separated by at least an order of magnitude. This confirms our intuitions about the runtime overhead of each heuristic, and supports the design of `DTR-EqClass` as a good middle ground (both in terms of runtime and computational overhead). However, these overhead figures could be improved with better-optimized implementations of the heuristics, as our implementation recomputes heuristics often, even when it may be possible to store the scores for tensors and maintain them in a sorted order. (Reformulating staleness to avoid having to use the current time might help.) Using persistent data structures that can be incrementally updated and maintain a sorted order will make these heuristics much more efficient, though this would also increase the complexity of the implementation.

6 CONCLUSION

In this paper, we introduced `simrd`: a simulator for Dynamic Tensor Rematerialization. `simrd` serves as an intermediate representation of DTR, at a level lower than DTR's

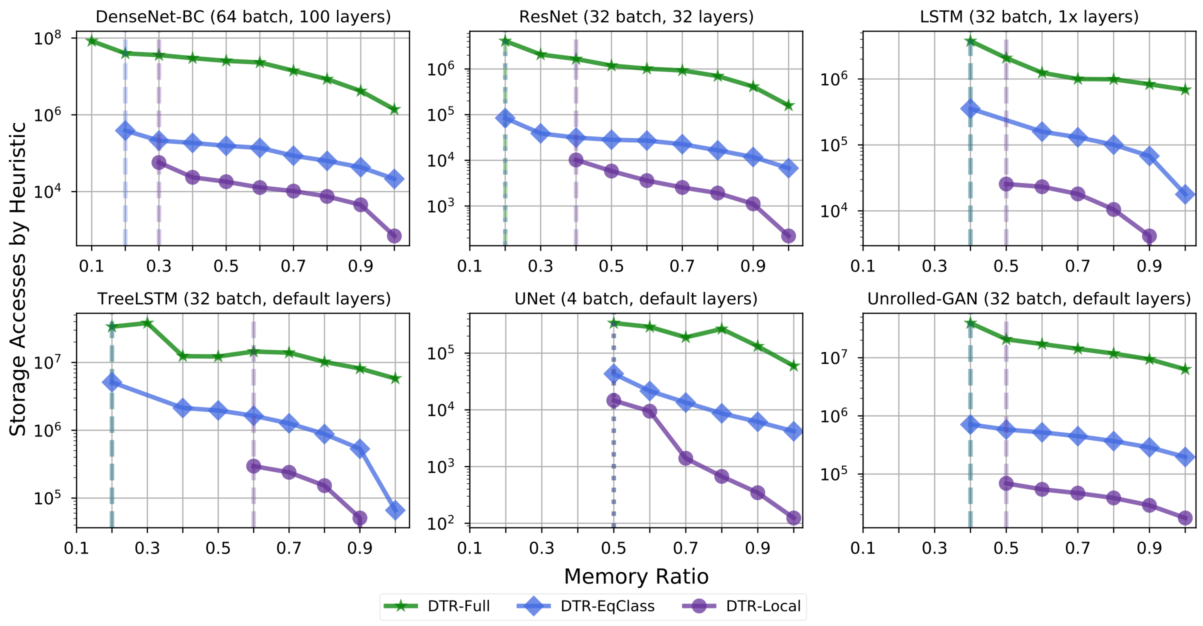

Figure 6: Total storages accesses incurred by heuristic evaluations and metadata maintenance, compared across different memory ratios, for the three main h_{DTR} variants.

core description but higher than a full implementation in a deep learning framework. Consequently, `simrd` enabled rapid exploration and benchmarking of the algorithm design space, while sufficiently modeling the real world constraints of a production DL framework. By simulating a toy model used in the literature, we characterized DTR's asymptotic performance favorably, giving confidence in the approach. Then, we simulated DTR's performance on a variety of recorded, real-world model logs. This enabled a comprehensive evaluation that cemented DTR as a valid approach, even if only used statically. Finally, the abstractions and definitions produced during the development of `simrd` helped guide the design of a functioning DTR prototype in PyTorch [11], by serving as a lower-level specification that could be translated more directly to implementation code.

7 ACKNOWLEDGEMENTS

Large portions of this thesis were prepared as part of (and adapted from) a larger group effort, currently under review, with my collaborators Marisa Kirisame, Steven Lyubomirsky, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock [11]. Special thanks to Marisa for prototyping the PyTorch log format. While `simrd` is largely my own work, the broader DTR project is absolutely the product of the whole DTR team, and the simulator could not have been possible without them.

This work was supported by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

REFERENCES

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[4] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 1–12, New York, NY, USA, 2017. Association for Computing Machinery.

[5] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[6] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, June 2018.

[7] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *International Conference on Machine Learning*, pages 2849–2858, 2016.

[8] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks, 2015.

[9] Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B. Tenenbaum, and Jiajun Wu. The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision, 2019.

[10] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks, 2017.

[11] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic tensor rematerialization. Manuscript submitted for publication, 2020.

[12] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[13] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *CoRR*, abs/1502.05767, 2015.

[14] Andreas Griewank and Andrea Walther. Treeverse: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. 03 1998.

[15] Jeffrey Mark Siskind and Barak A. Pearlmutter. Divide-and-conquer checkpointing for arbitrary programs with no user annotation. *Optimization Methods and Software*, 33(4-6):1288–1330, Sep 2018.

[16] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[17] Nimit Sharad Sohoni, Christopher Richard Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré. Low-memory neural network training: A technical report. *CoRR*, abs/1904.10631, 2019.

[18] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[20] Julien Herrmann, Olivier Beaumont, Lionel Eyraud-Dubois, Julien Hermann, Alexis Joly, and Alena Shilova. Optimal checkpointing for heterogeneous chains: how to train deep neural networks with limited memory, 2019.

[21] Ravi Kumar, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua Wang. Efficient rematerialization for deep networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 15172–15181. Curran Associates, Inc., 2019.

[22] Mitsuru Kusumoto, Takuya K Inoue, Gentaro Watanabe, Takuya Akiba, and Masanori Koyama. A graph theoretic framework of recomputation algorithms for memory-efficient backpropagation. In *NeurIPS*, 2019.

[23] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In *Proceedings of Machine Learning and Systems 2020*, pages 497–511, 2020.

[24] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, page 311–321, New York, NY, USA, 1992. Association for Computing Machinery.

[25] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, Mar 2020.

[26] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[27] Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-efficient backpropagation through time. *CoRR*, abs/1606.03401, 2016.

[28] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[29] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[30] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul 2017.

[31] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, page 234–241, 2015.

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

[33] Luke Metz, Ben Poole, David Pfau, and Jascha Sohl-Dickstein. Unrolled generative adversarial networks, 2016.

[34] Geoff Pleiss, Danlu Chen, Gao Huang, Tongcheng Li, Laurens van der Maaten, and Kilian Q. Weinberger. Memory-efficient implementation of densenets. https://github.com/gpleiss/efficient_densenet_pytorch, 2017.

[35] PyTorch Team. Word-level language modeling rnn. https://github.com/pytorch/examples/tree/master/word_language_model, 2020.

[36] Riddhiman Dasgupta. treelstm.pytorch. https://github.com/dasguptar/treelstm.pytorch, 2018.

[37] Trading compute for memory in pytorch models using checkpointing. https://github.com/prigoyal/pytorch_memonger/blob/master/tutorial/Checkpointing_for_PyTorch_models.ipynb, 2017.

A ADDITIONAL FIGURES

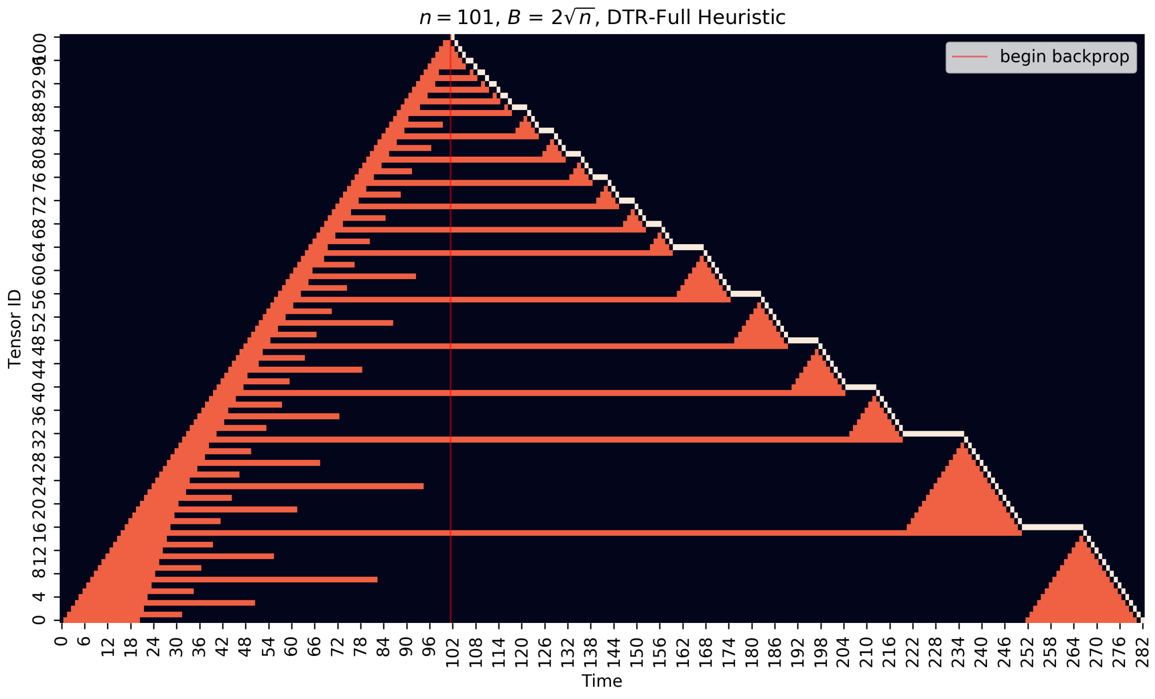

Figure 7: Memory trace for $B = \lceil 2\sqrt{n} \rceil$, $h =$ `DTR-Full`.

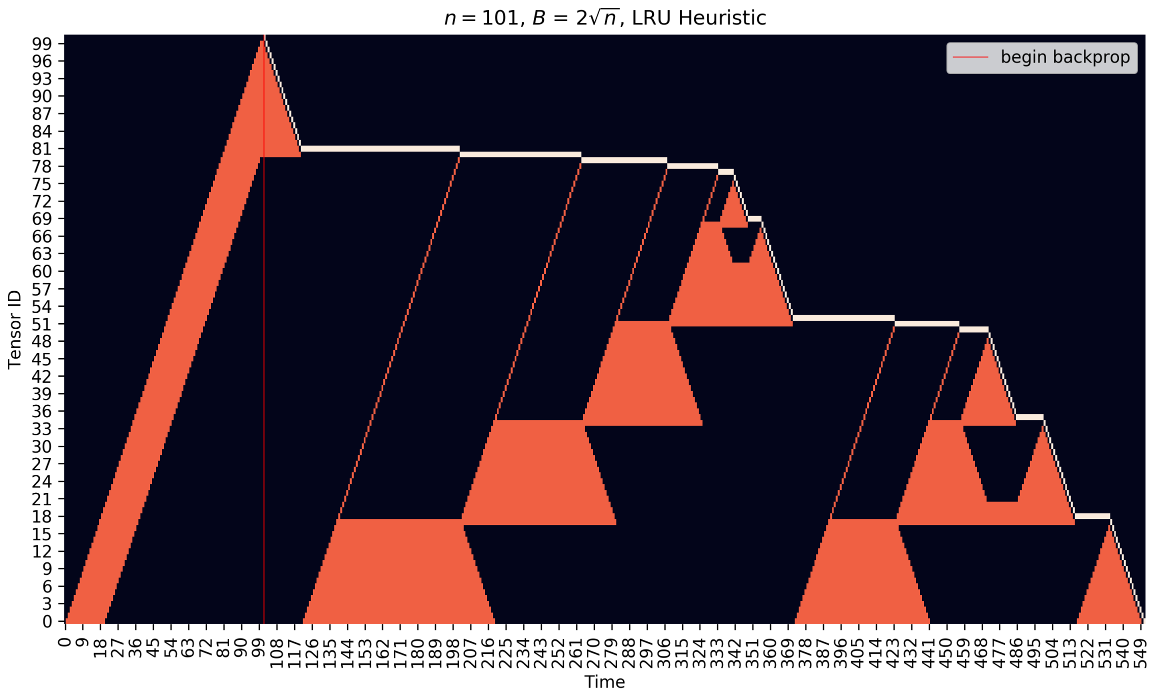

Figure 8: Memory trace for $B = \lceil 2\sqrt{n} \rceil$, $h =$ `LRU`.

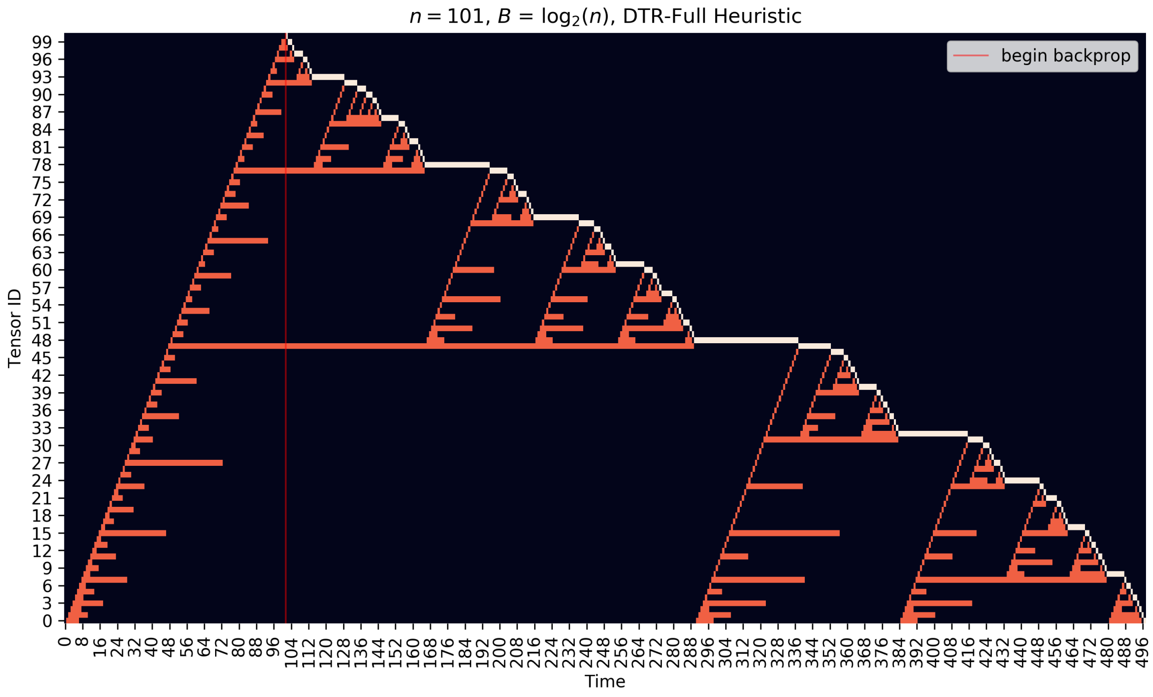

Figure 9: Memory trace for $B = \lceil \log_2 n \rceil$, $h =$ `DTR-Full`.

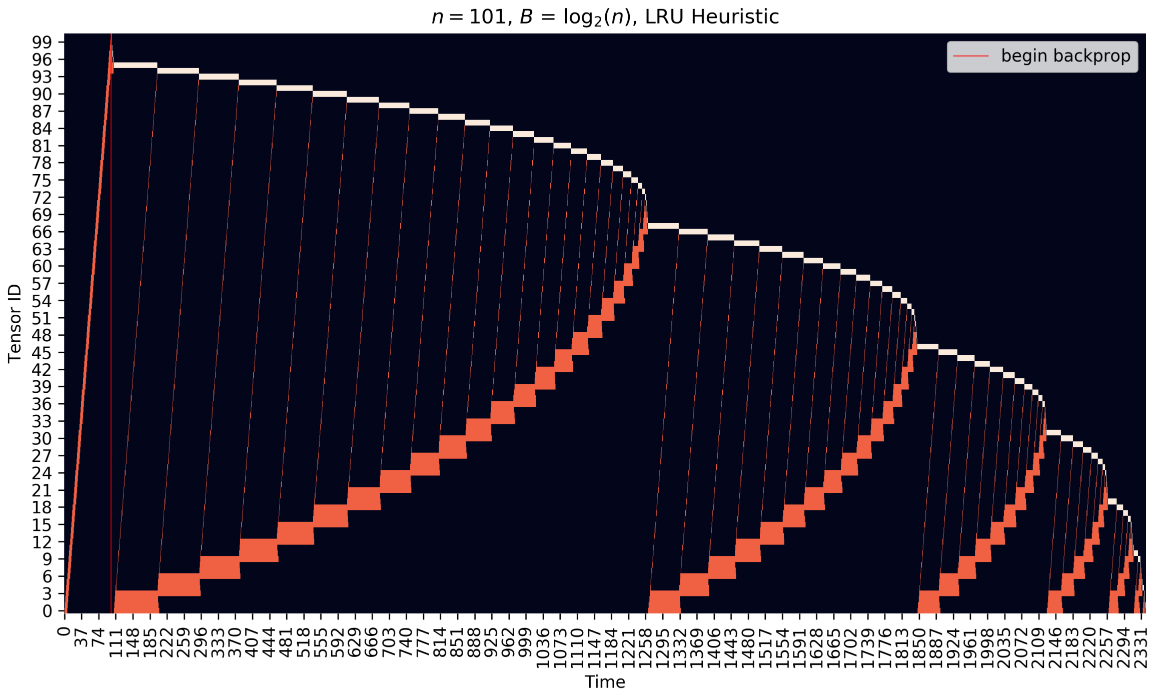

Figure 10: Memory trace for $B = \lceil \log_2 n \rceil$, $h =$ `LRU`.

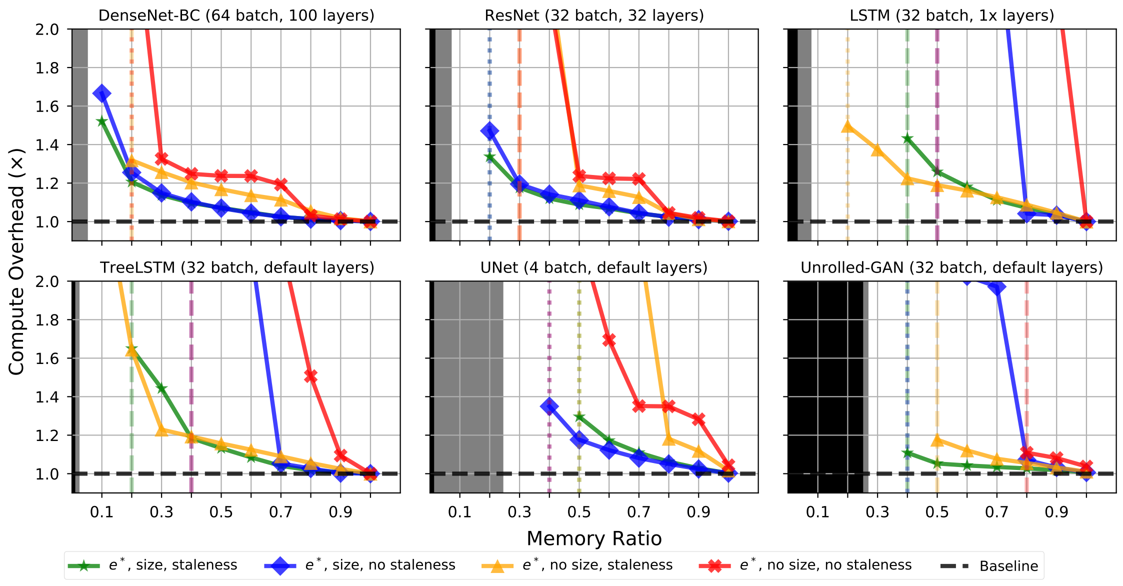

Figure 11: Results for fixed $c = e^*$, varying s and m.

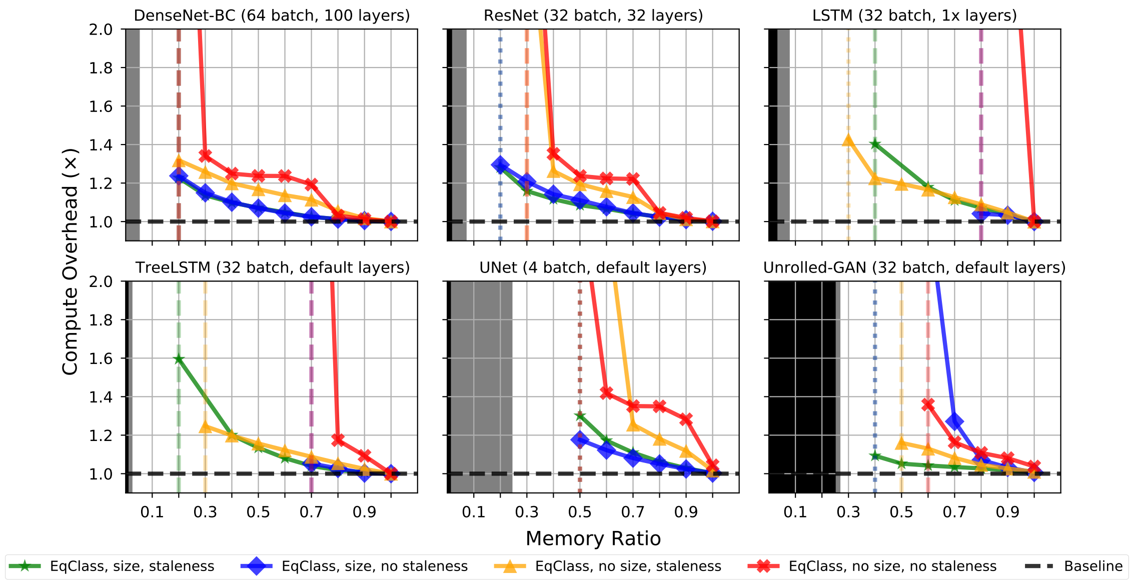

Figure 12: Results for fixed $c = \mathrm{EqClass}$, varying s and m.

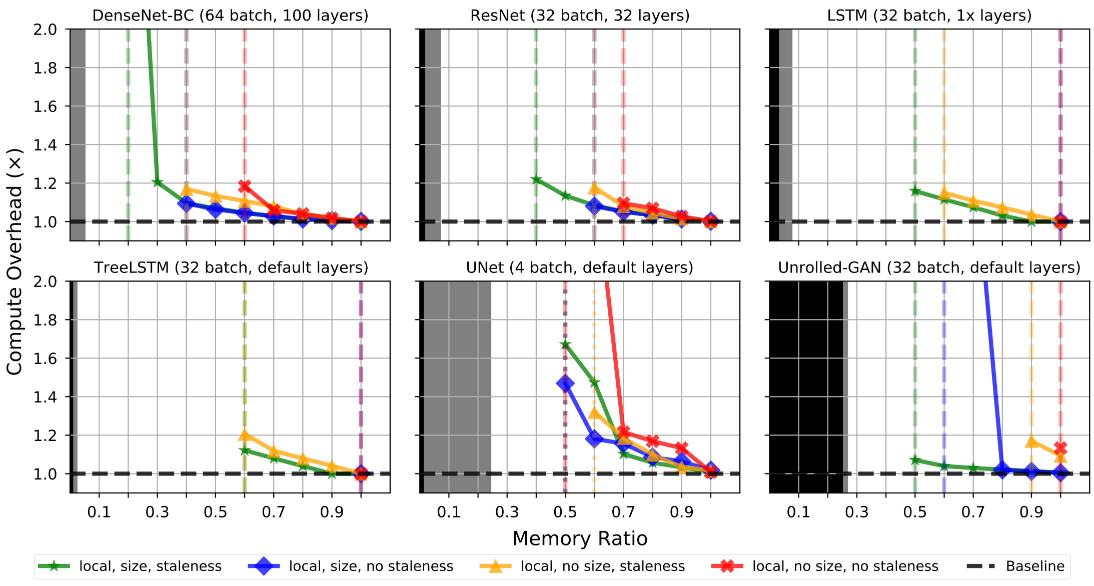

Figure 13: Results for fixed $c = \text{local}$, varying s and m.

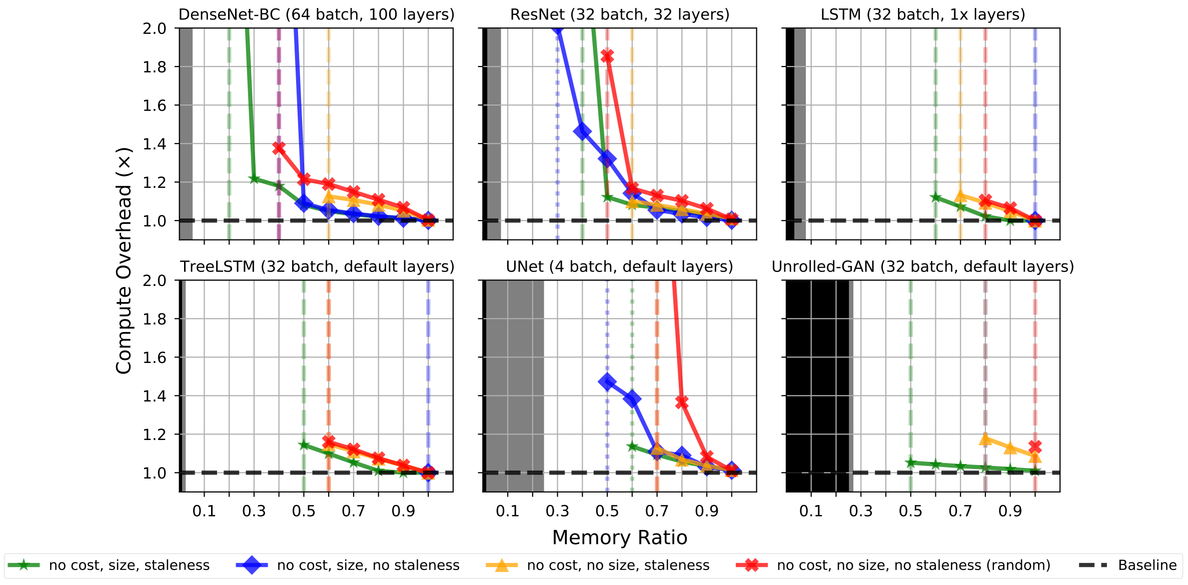

Figure 14: Results for fixed $c = \text{no}$, varying s and m.