Contextual Memory Trees

Wen Sun 1  Alina Beygelzimer 2  Hal Daumé III 3  John Langford 3  Paul Mineiro 4

Abstract

We design and study a Contextual Memory Tree (CMT), a learning memory controller that inserts new memories into an experience store of unbounded size. It is designed to efficiently query for memories from that store, supporting logarithmic time insertion and retrieval operations. Hence CMT can be integrated into existing statistical learning algorithms as an augmented memory unit without substantially increasing training and inference computation. Furthermore CMT operates as a reduction to classification, allowing it to benefit from advances in representation or architecture. We demonstrate the efficacy of CMT by augmenting existing multi-class and multi-label classification algorithms with CMT and observe statistical improvement. We also test CMT learning on several image-captioning tasks to demonstrate that it performs computationally better than a simple nearest neighbors memory system while benefiting from reward learning.

1. Introduction

When a human makes a decision or answers a question, they are able to do so while very quickly drawing upon a lifetime of remembered experiences. This ability to retrieve relevant experiences efficiently from a memory store is currently lacking in most machine learning systems (§1.1). We consider the problem of learning an efficient online data structure for use as an external memory in a reward-driven environment. The key functionality of the Contextual Memory Tree (CMT) data structure defined here is the ability to insert new memories into a learned key-value store, and to be able to query those memories in the future. The storage and query functionality in CMT is driven by an optional, user-specified, external reward signal; it organizes memories in a tree structure, guaranteeing logarithmic time (in the number of memories) operations throughout (§3). Because CMT operates as a reduction to classification, it does not prescribe a representation for the keys and can leverage future advances in classification techniques.

More formally, we define the data structure CMT (§2), which converts the problem of mapping queries (keys) to memories (key-value pairs) into a collection of classification problems. Experimentally (§3), we show this is useful in three different settings. (a) Few-shot learning in extreme multiclass classification problems, where CMT is used directly as a classifier (the queries are examples and the values are class labels). Figure 1 shows that unsupervised CMT can statistically outperform other supervised logarithmic-time baselines including LOMTree (Choromanska & Langford, 2015) and Recall Tree (RT) (Daumé et al., 2017) with supervision providing further improvement. (b) Extreme multi-label classification problems where CMT is used to augment a One-Against-All (OAA) style inference algorithm. (c) Retrieval of images based on captions, where CMT is used similarly to a nearest-neighbor retrieval system (the queries are captions and the values are the corresponding images). External memories that persist across examples are also potentially useful as inputs to downstream applications; for instance, in natural language dialog tasks (Bartl & Spanakis, 2017) and in machine translation (Gu et al., 2018), it can be useful to retrieve similar past contexts (dialogs or documents) and augment the input to the downstream system with these retrieved examples. Memory-based systems can also be useful as a component of learned reasoning systems (Weston et al., 2014; Graves et al., 2016).

A memory $z = (x, \omega)$ is a pair of query $x$ and value $\omega$. CMT operates in the following generic online manner, repeated over time:

1. Given a query $x$, retrieve $k$ associated memories $(u, \langle z_1, z_2, \ldots, z_k \rangle) = \text{QUERY}(x)$ together with an identifier $u$.
2. If a reward $r_i$ for $z_i$ is observed, update the system via \text{UPDATE}($(x, z_i, r_i), u$).
3. If a value $\omega$ associated with $x$ is available, \text{INSERT} a new memory $z = (x, \omega)$ into the system.
A natural goal in such a system is a notion of self-consistency. If the system inserts \( z = (x, \omega) \) into CMT, then in subsequent rounds, one should expect that \( (x, \omega) \) is retrieved when \( \text{QUERY}(x) \) is issued again for the same \( x \). (For simplicity, we assume that all \( x \) are unique.) In order to achieve such self-consistency in a data structure that changes over time, we augment CMT with a “Reroute” operation, in which the data structure gradually reorganizes itself by removing old memories and re-inserting them on an amortized basis. We find that this Reroute operation is essential to good empirical performance (§4.5).

### 1.1. Existing Approaches

The most standard associative memory system is a map data structure (e.g., hashmap, binary tree, relational database); unfortunately, these do not generalize across inputs—either an input is found exactly or it is not. We are interested in memories that can generalize beyond exact lookups, and can learn to do so based on past successes and failures in an incremental, online manner. Because we wish to scale, the computation time for all operations must be at most logarithmic in the number of memories, with constant space overhead per key-value pair. Finally, as mentioned above, such a system should be self-consistent.

There are many existing approaches beyond hashmaps, all of which miss one of our desiderata (Figure 1). A basic approach for text documents is an inverted index (Knuth, 1997; Broder et al., 2003), which indexes a document by the words that appear therein. On the other end of the spectrum, supervised learning can be viewed as remembering (compiling) a large amount of experience into a predictor which may offer very fast evaluation, but generally cannot explicitly query for past memories (aka examples).

There has been substantial recent interest in coupling neural networks with nearest neighbor variants. Classical approaches are inadequate: a) Exact nearest neighbor algorithms (including memory systems that use them (Kaiser et al., 2017)) are computationally inefficient except in special cases (Dasgupta & Sinha, 2015; Beygelzimer et al., 2006) and do not learn. b) Approximate Nearest Neighbors via Locality-Sensitive Hashing (Datar et al., 2004) and MIPS (Shrivastava & Li, 2015) address the problem of computational time, but not learning. c) Nearest Neighbors with Learned Metrics (Weinberger et al., 2005) can learn, but are non-incremental.

More recent results combine neural architectures with forms of approximate nearest neighbor search to address these shortcomings. For example, (Rae et al., 2016) uses a representation learned for a task with either randomized kd-trees or locality sensitive hashing on a the Euclidean distance metric, both of which are periodically recomputed. The CMT instead learns at individual nodes and works for any representation, therefore, avoiding presupposing that a Euclidean metric is appropriate and could potentially productively replace the approximate nearest neighbor subsystem here.

Similarly, (Chandar et al., 2016) experiments with a variety of K-MIPS (Maximum Inner Product Search) data structures which the memory tree could potentially replace to create a higher ceiling on performance in situations where MIPS is not the right notion of similarity.

In (Andrychowicz & Kurach, 2016) the authors learn a hierarchical data structure over a pre-partitioned set of memories with a parameterized JOIN operator shared across nodes.
The use of pre-partition makes the data structure particularly sensitive to the (unspecified) order of that prepartition as discussed in appendix 6 of the LOMTree (Choromanska & Langford, 2015). Furthermore, tying the parameters of which sub-tree contains the best memory corresponding to At a high level, a CMT (Figure 2) is a near-balanced binary tree with a large number of labels for some classes is allowed well at prediction time, but all current approaches are non-incremental and require substantial training-time overhead. Finally, differentiable memory systems (Weston et al., 2014; Graves et al., 2016) are able to refine memories over time, but rely on gradient-descent-based techniques which incur a computational overhead that is inherently linear in the number of memories.

There are works on leveraging memory systems to perform few-shot learning tasks (Snell et al., 2017; Strubell et al., 2017; Santoro et al., 2016)). However they are not loga-

tic-based models (Salakhutdinov & Hinton, 2009; Rastegari et al., 2012), which learn a hash function that works well at prediction time, but all current approaches are non-
icapable of effectively operating at the large scales. Also note that they often address an easier version of the few-shot learning problem where training with a large number of labels for some classes is allowed as an initializer before the few-shot labels are observed. In contrast, we have no initialization phase.

2. The Contextual Memory Tree

At a high level, a CMT (Figure 2) is a near-balanced binary tree whose size dynamically increases as more memories are inserted. All memories are stored in leaf nodes with each leaf containing at most \(c \log n\) memories, where \(n\) is the total number of memories and \(c\) is a constant independent of the number of memories.

Learning happens at every node of CMT. Each internal node contains a learning router. Given a query, CMT routes from the root to a leaf based on left-or-right decisions made by the routers along the way. Each internal node optimizes a metric, which ensures both its router’s ability to predict which sub-tree contains the best memory corresponding to the query, and the balance between its left and right subtrees. CMT also contains a global learning scorer that predicts the reward of a memory for a query. The scorer is used at a leaf to decide which memories to return, with updates based on an external reward signal of memory quality.

2.1. Data Structures

A memory consists of a query (key) \(x \in \mathcal{X}\) and its associated value \(\omega \in \Omega\). We use \(z\) to denote the memory pair \((x, \omega)\) and define \(\mathcal{Z} = \mathcal{X} \times \Omega\) as the set of \(z\). Given a memory \(z\), we use \(z.x\) and \(z.\omega\) to represent the query and the value of \(z\) respectively. For instance, for multiclass classification, \(x\) is a feature vector and \(\omega\) is a label. Our memory store is organized into a binary tree. A leaf node in (left top) consists of a parent and a set of memories. Leaf nodes are connected by internal nodes as in (left, bottom). An internal node has a parent and two children, which may be either leaf or internal nodes, a count \(n\) of the number of memories beneath the node, and a learning router \(g : \mathcal{X} \rightarrow \{-1, 1\}\) which both routes via \(g(x)\) and updates via \(g.update(x, y)\) for \(y \in \{-1, 1\}\), or \(g.update(x, y, i)\) where \(i \in \mathbb{R}^+\) is an importance weight of \((x, y)\). If \(g(x) \leq 0\), we route \(x\) left, and otherwise right.

The contextual memory tree data structure in (right) has a root node, a parameter \(\alpha \in [0, 1]\) which controls how balanced the tree is, a multiplier \(e\) on the maximum number of memories stored in any single leaf node, and a learning scorer \(f : \mathcal{X} \times \mathcal{Z} \rightarrow \mathbb{R}\). Given a query \(x\) and memory \(z\), the learning scorer predicts the reward one would receive if \(z\) is returned as the retrieved memory for query \(x\) via \(f(x, z)\). Once a reward \(r \in [0, 1]\) is received for a pair of memory \(z\) and query \(x\), the learning scorer updates via \(f.update(x, z, r)\) to improve its ability to predict reward. Finally, the map \(M\) maps examples to the leaf that contains them, making removal easy.

Given any internal node \(v\) and query \(x\), we define a data structure path representing the path taken from \(v\) to a leaf: path = \(\{(v_i, a_i, p_i)\}_{i=1}^{N}\), where \(v_1 = v, a_i \in \{left, right\}\) is the left or right decision made at \(v_i, p_i \in [0, 1]\) is the probability with which \(a_i\) was chosen. As we show later, path communicates to the update rule the information needed to create an unbiased update of routers.

2.2. Algorithms

All algorithms work given a contextual memory tree \(T\). For brevity, we drop \(T\) when referencing its fields. We use \(\in_U P\) to chose uniformly at random from a set \(P\).

Algorithm (PATH) routes a query \(x\) from any node \(v\) to a
Algorithm 1 \( \text{PATH}(\text{query } x, \text{ node } v) \)

1: path \( \leftarrow \emptyset \)
2: while \( v \) is not a leaf do
3: \( a \leftarrow \text{if } v.g(x) > 0 \text{ then right else left} \)
4: Add \((v, a, 1)\) to path
5: \( v \leftarrow v, a \)
6: end while
7: path.leaf \( \leftarrow v \)
8: \( \text{return } \text{path} \)

Algorithm 2 \( \text{QUERY}(x, \text{ items to return } k, \text{ exploration probability } \epsilon) \)

1: \( \text{path} \leftarrow \text{PATH}(x, \text{root}), \text{path} = ((v_i, a_i, p_i))_{i=1}^n \)
2: \( q \in \mathbb{U} \{0, 1\} \)
3: if \( q \geq \epsilon \) then
4: \( \text{return } (\emptyset, \text{top}_k(\text{path.leaf}, x)) \)
5: else
6: \( \text{Pick } i \in \mathbb{U} \{1, \ldots, N + 1\} \)
7: if \( i \leq N \) then
8: \( \text{Pick } a' \in \mathbb{U} \{\text{right}, \text{left}\} \)
9: \( l = \text{PATH}(x, v_i, a').\text{leaf} \)
10: \( \text{return } ((v_i, a', 1/2), \text{top}_k(l, x)) \)
11: else
12: \( \text{return } ((\text{path.leaf}, 1, 1), \text{rand}_k(\text{path.leaf}, x)) \)
13: end if
14: end if

leaf, returning the path traversed. Algorithm 2 (QUERY) takes a query \( x \) as input and returns at most \( k \) memories. The parameter \( \epsilon \in [0, 1] \) determines the probability of exploration during training. Algorithm 2 first deterministically routes the query \( x \) to a leaf and records the path traversed, path. With probability \( 1 - \epsilon \), we simply return the best memories stored in path.leaf. For a query \( x \) and leaf \( l \), we use \( \text{top}_k(l, x) \) as a shorthand for the set of min\( \{k, |l.\text{mem}|\} \) memories \( z \) in \( l.\text{mem} \) with the largest \( f(x, z) \), breaking ties randomly. We also use \( \text{rand}_k(l, x) \) for a random subset of min\( \{k, |l.\text{mem}|\} \) memories in \( l.\text{mem} \).

Algorithm 3 \( \text{UPDATE}((x, z, r), (v, a, p)) \)

1: if \( v \) is a leaf then
2: \( f.\text{update}(x, z, r) \)
3: else
4: \( \text{return } \) \( \frac{r}{p}(1(a = \text{right}) - 1(a = \text{left})) \)
5: \( y \leftarrow (1 - \alpha)\hat{y} + \alpha(\log v.\text{left}.n - \log v.\text{right}.n) \)
6: \( v.g.\text{update}(x, \text{sign}(y), |y|) \)
7: end if
8: Run \text{REROUTE} \( d \) times

With the remaining probability \( \epsilon \), we uniformly sample a node along path including path.leaf. If we sampled an internal node \( v \), we choose a random action \( a' \) and call path(\( x, v.a' \)) to route \( x \) to a leaf. This exploration gives us a chance to discover potentially better memories stored in the other subtrees beneath \( v \), which allows us to improve the quality of the router at node \( v \). We do uniform exploration at a uniformly chosen node but other schemes are possible. If we sampled path.leaf, we return a random set of memories stored in the leaf, in order to update and improve the learning scorer \( f \). The shorter the path, the higher the probability that exploration happens at the leaf.

Algorithm 4 \( \text{INSERT}(v, \text{ memory } z, \text{ Reroute } d) \)

1: while \( v \) is not a leaf do
2: \( B = \log v.\text{left}.n - \log v.\text{right}.n \)
3: \( y \leftarrow \text{sign}\left((1 - \alpha)v.g(z, x) + \alpha B\right) \)
4: \( v.g.\text{update}(z, x, y) \)
5: \( v.n \leftarrow v.n + 1 \)
6: if \( v.g(z, x) > 0 \) then \( v.\text{right else v.left} \)
7: end while
8: \( \text{INSERTLEAF}(v, z) \)
9: Run \text{REROUTE} \( d \) times

After a query for \( x \), we may receive a reward \( r \) for a returned memory \( z \). In this case, Algorithm 3 (UPDATE) uses the first triple returned by QUERY to update the router making a randomized decision. More precisely, Algorithm 3

---

**Contextual Memory Tree**

- **Algorithm 1** \( \text{PATH}(\text{query } x, \text{ node } v) \)
- **Algorithm 2** \( \text{QUERY}(x, \text{ items to return } k, \text{ exploration probability } \epsilon) \)
- **Algorithm 3** \( \text{UPDATE}((x, z, r), (v, a, p)) \)
- **Algorithm 4** \( \text{INSERT}(v, \text{ memory } z, \text{ Reroute } d) \)

---

**Figure 3:** Data structures for internal and leaf nodes (left) and the full contextual memory tree (right).
computes an unbiased estimate of the reward difference of the left/right decision which is then mixed with a balance-inducing term on line 5. When randomization occurred at the leaf, the scorer \( f \) is updated instead.

The INSERT operation is given in Algorithm 5. It routes the memory \( z \) to be inserted according to the query \( z.x \) from the root to a leaf using internal learning routers, updating them on descent. Once reaching a leaf node, \( z \) is added into that leaf via INSERTLEAF. The label definition on line 3 in INSERT is the same as was used in [Beygelzimer et al., 2009]. That use, however, was for a different problem (conditional label estimation) and is applied differently (controlling the routing of examples rather than just advising a learning algorithm). As a consequence, the proofs of correctness given in section 3.3 differ.

When the number of memories stored in any leaf exceeds the log of the total number of memories, a leaf is split according to Algorithm 5 (INSERTLEAF). The leaf node \( v \) is promoted to an internal node with two leaf children and a binary classifier \( g \) with all memories inserted at \( v \).

Because updates are online, they may result in a lack of self-consistency for previous insertions. This is fixed by REROUTE (Algorithm 7) on an amortized basis. Specifically, after every INSERT operation we call REROUTE, which randomly samples an example from all the examples, extracts the sampled example from the tree, and then re-`inserts it. This relies on the REMOVE (Algorithm 6) operation, which finds the location of a memory using the hashmap then ascends to the parent cleaning up accounting. When a leaf node has zero memories, it is removed.

### 3. Properties

There are five properties that we want CMT to satisfy simultaneously (see Figure 1 (left) for the five properties). Storage (in appendix \( A.1 \)) and Incrementality (in appendix \( A.2 \)) are easy observations.

Appendix \( A.6 \) shows that in the limit of many REROUTES, self-consistency (defined below) is achieved.

**Definition 3.1** A CMT is self-consistent if for all \( z \) with a unique \( z.x \), \( z = \text{QUERY}(z.x, 1, 0) \).

Appendix \( A.7 \) shows a learning property: Every internal router asymptotically optimizes to a local maxima of an objective function that mirrors line 5 of UPDATE.

This leaves only logarithmic computational time, which we address next.

### 3.1. Computational Time

The computational time analysis naturally breaks into two parts, partition quality at the nodes and the time complexity given good partitions. To connect the two, we first define partition quality.

**Definition 3.2** A \( K \)-balanced partition of any set has each element of the partition containing at least a \( 1/K \) fraction of the original set.

When partitioning into two sets, \( K \geq 2 \) is required. Smaller \( K \) result in smaller computational complexities at the cost of worse predictive performance in practice.

Define the progressive training error of a learning router \( g \) after seeing \( T \) examples \( x_1, \ldots, x_T \) as \( p = \frac{1}{T} \sum_{t=1}^T \mathbb{1}[g(x_t) \neq y_t] \), where \( y_t \) is the label assigned in line 3 of INSERT, and \( g(x_t) \) is evaluated immediately after calling \( g.\text{update}(x_t, y_t) \) so a mistake occurs when \( g(x_t) \) disagrees with \( y_t \) after the update. The next theorem proves a bound on the partition balance dependent on the progressive training error of a node’s router and \( \alpha \).

**Theorem 3.3** (Partition bound) At any point, a router with a progressive training error of \( p \) creates a...
We implemented CMT as a reduction to Vowpal Wabbit’s default learning algorithm. The proof is in appendix A.5. This theorem establishes logarithmic time computation given that if all stated operations are atomic.

Theorem 3.4 (Computational Time) If every router $g$ in a CMT with $T$ previous calls to INSERT creates a $K$-partition, the worst case computation is $O((K + c) \log T)$ for INSERT, $O((K + c) \log T)$ for QUERY, and $O(1)$ for UPDATE if all stated operations are atomic.

The proof is in appendix A.5. This theorem establishes logarithmic time computation given that $K$-partitions are created. These two theorems imply that the computation is logarithmic time for all learning algorithms achieving a training error significantly better than 1.

4. Experiments

CMT is a subsystem for other learning tasks, so it assists other inference and learning algorithms. We test the application of CMT to three systems, for multiclass classification, multilabel classification, and image retrieval. Separately, we also ablate various elements of CMT to discover its strengths and weaknesses.

We implemented CMT as a reduction to Vowpal Wabbit’s default learning algorithm. The routers $(g)$ and the learning scorer $(f)$ are all linear functions and are incrementally updated by an Adagrad [Duchi et al., 2011] gradient method in VW. Similarly, most baselines are implemented in the same system with a similar or higher level of optimization.

4.1. Application: Online Extreme Multi-Class Classification

Since CMT operates online, we can evaluate its online performance using progressive validation [Blum et al., 1999] (i.e., testing each example ahead of training). Used online, we QUERY for an example, evaluate its loss, then apply UPDATE with the observed loss followed by INSERT of the data point. In a multiclass classification setting, a memory $z$ is a feature vector $x$ and label $\omega$. Given a query $x$, CMT returns a memory $z$ and receives a reward signal $1[z, \omega = \omega]$ for update. Finally, CMT inserts $(x, \omega)$.

We test the online learning ability of CMT on two multiclass classification datasets, ALOI (1000 labels with 100 examples per label) and WikiPara 3-shot (10000 labels with 3 examples per label), against two other logarithmic-time online multiclass classification algorithms, LOMTree [Choromanska & Langford, 2013] and Recall Tree [Daume et al., 2011]. We also compare against a linear-time online multiclass classification algorithm, One-Against-All (OAA).

Figure 4 summarizes the results in terms of progressive performance. On both datasets, we report entropy reduction from the constant predictor (the higher the better). The entropy reduction of a predictor $A$ from another predictor $B$ is defined as $\log_2(p_A) - \log_2(p_B)$, where $p_A$ and $p_B$ are prediction accuracies of $A$ and $B$.

Conclusion: CMT greatly outperforms the baselines in the small number of examples per label regime. This appears to be primarily due to the value of explicit memories over learned parameters in this regime.

4.2. Application: Batch Few-shot Multi-Class Classification

We can also use CMT in an offline testing mode as well by using CMT with multiple passes over the training dataset and testing it on a separate test set. We again use CMT on few-shot multi-class classification, comparing it to LOMTree and Recall Tree.

Starting first with the ALOI dataset, we tested both the unsupervised version (i.e., using only INSERT) and the supervised version (i.e., using INSERT for the first pass, and using UPDATE for subsequent passes). We used three passes for all algorithms. The supervised version of CMT achieved 26.3\% test prediction error, outperforming LOMTree (66.7\%) and
Recall Tree (28.8%). The supervised version of CMT also significantly outperforms the unsupervised one (75.8% error rate), showing the benefit of the update procedure. Since ALOI has 1000 classes, a constant predictor has prediction error larger than 99%.

We then test CMT on more challenging few-shot multi-class classification datasets, WikiPara S-shot (S = 1, 2, 3) and ImageNet S-shot (S = 1, 2, 3, 5) with only S examples per label. Figure 1 summarizes the statistical performance (entropy reduction compared to a constant predictor) of supervised CMT, unsupervised CMT (denoted as CMT (u)), and the two logarithmic-time baselines. For one-shot experiments (WP 1-s and IN 1-s on Figure 1), CMT outperforms all baselines. The edge of CMT degrades gradually over baselines as S increases (IN 5-s with S > 1 in Figure 1). All details are included in Table 6 in Appendix B.3.

**Conclusion:** The high performance of CMT with a small number of examples per label persists in batch training. The remarkable performance of unsupervised CMT over supervised baselines suggests self-consistency can provide nearest-neighbor performance without explicit reward.

### 4.3. Application: Multi-Label Classification with an External Inference Algorithm

In this set of experiments, instead of using CMT as an inference algorithm, we integrate CMT with an external inference procedure based on One-Against-All. CMT is not aware of the external multi-label classification task, so this is an example of how an external inference algorithm can leverage the returned memories as an extra source of information to improve performance. Here each memory z consists of a feature vector x and label vector \( \omega \in \{0, 1\}^M \), where M is the number of unique labels. Given a query x, its ground-truth label vector \( \omega \), and a memory z, we choose the F1-score between \( \omega \) and \( z, \omega \) as the reward signal. We set k to \( c \log(N) \) (i.e., CMT returns all memories in the leaf we reach). Given a query x, with the returned memories \{z_1, ..., z_k\}, the external inference procedure extracts the unique labels from the returned memories and performs a One-Against-Some (OAS) inference (Daumé et al., 2017) using the extracted labels. The external system then calls UPDATE for the returned memories. Since CMT returns logarithmically many memories, we guarantee that the number of unique labels from the returned memories is also logarithmic. Hence augmenting OAS with CMT enables logarithmic inference and training time.

We compare CMT-augmented OAS with multi-label OAA under the Hamming loss. We compare CMT-augmented OAS to OAA on three multi-label datasets, RCV1-1K (Prabhu & Varma, 2013), AmazonCat-13K (McAuley & Leskovec, 2013), and Wiki10-31K (Zubiaga et al., 2012). Bhatta et al., 2015). (The datasets are described in Table 4 in Appendix B.1.) Table 1 summarizes the performance of CMT and OAA. (LOMTree and Recall Tree are excluded because they do not operate in multi-label settings.)

**Conclusion:** CMT-augmented OAS achieves similar statistical performance to OAA, even mildly outperforming OAA on Wiki10-31K, while gaining significant computational speed up over a vector optimized OAA in training and inference on datasets with a large number of labels (e.g., AmazonCat-13K and Wiki10-31K). Note that the VW implementation of OAA operates at a higher level optimization and involves vectorized computations that increase throughput by a factor of 10 to 20. Hence we observe for RCV1-1K with 1K labels, OAA can actually be more computationally efficient then CMT. This set of experiments shows that CMT-augmented OAS can win over OAA both statistically and computationally for challenging few-shot multi-label datasets with a large number of labels.

### 4.4. Application: Image Retrieval

We test CMT on an image retrieval task where the goal is to find an image given a caption. We used three benchmark datasets, (1) UIUC Pascal Dataset (Rashtchian et al., 2010), (2) Flickr8k dataset (Hodosh et al., 2013), and (3) MS COCO (Lin et al., 2014), with feature representations described in B.1. Here, a memory z consists of (features of) a caption x and an image \( \omega \). Given a query, CMT returns a memory z = (x, \omega). Our reward function is the cosine similarity between the returned memory’s image z,\( \omega \), and the ground-truth image \( \omega \) associated with the query x.

To show the benefit of learning in CMT, we compare it to Nearest Neighbor Search (NNS) and a KD-Tree as an Approximate NN data structure on this task, using the Euclidean distance \( \|x - z, x\|_2 \) in the feature space of captions as the NNS metric. Both CMT and NNS are tested on a separate test set, with the average reward of the retrieved

---

Table 1: Hamming Loss, test time per example (ms), and training time (hr) for multi-label tasks.

<table>
<thead>
<tr>
<th>Approach</th>
<th>RCV1-1K</th>
<th>AmazonCat-13K</th>
<th>Wiki10-31K</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>loss</td>
<td>Test time</td>
<td>Train time</td>
</tr>
<tr>
<td>CMT</td>
<td>2.5</td>
<td>1.4ms</td>
<td>1.9hr</td>
</tr>
<tr>
<td>OAA</td>
<td>2.6</td>
<td>0.5ms</td>
<td>1.3hr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2 summarizes the speedup over NNS (implemented using a linear scan) and KD-Tree (KD tree implementation from scikit-learn [Pedregosa et al., 2011]). Note that in our datasets, the feature of a query is high dimensional (2^{20}) but extremely sparse. Since KD-Tree cannot take advantage of sparsity, both the construction and inference procedure is extremely slow (even slower than a NNS). We also emphasize here that a KD-Tree does not operate in an online manner. Hence in our experiments, we have to feed all queries from the entire training dataset to KD-Tree to initialize its construction, which makes it impossible to initialize the run of KD-Tree on MSCOCO.

**Conclusion:** The difference in reward is negligible (on the order of 10^{-3}) and statistically insignificant. (See Appendix Table 7 for details.) However, CMT is significantly faster.

<table>
<thead>
<tr>
<th></th>
<th>CMT unsup</th>
<th>CMT sup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pascal</td>
<td>5.7 / 9400</td>
<td>1.3 / 2100</td>
</tr>
<tr>
<td>Flickr8k</td>
<td>26.0 / 33000</td>
<td>6.0 / 7700</td>
</tr>
<tr>
<td>MSCOCO</td>
<td>21.0 / ~</td>
<td>6.5 / ~</td>
</tr>
</tbody>
</table>

Table 2: Speedups over linear NNS (left) and KD-Tree (right), in (unsup)ervised and (sup)ervised mode.

### 4.5. Ablation Analysis of CMT

We conduct experiments to perform an ablation study of CMT in the context of multi-class classification, where it operates directly as an inference algorithm.

We test the self-consistency property on WikiPara with only one training example per class (see Figure 5a). We ran CMT in an unsupervised fashion, by only calling `INSERT` and using $\|x - z\|$ as $f(x, z)$ to select memories at leaves. We report the self-consistency error with respect to the number of `REROUTE` calls per insertion (parameter $d$) after four passes over the dataset (tuned using a holdout set). As $d$ increases, the self-consistency error rapidly drops.

To show that `UPDATE` is beneficial, we use multiple passes to drive the training error down to nearly zero. Figure 5b shows the training error versus the number of passes on the WikiPara one-shot dataset (on the x-axis, we plot the number of additional passes over the dataset, with zero corresponding to a single pass). Note that the training error is essentially equal to the self-consistency error in WikiPara One-shot, hence `UPDATE` further enhances self-consistency due to the extra `REROUTE` operations in `UPDATE`.

To test the effect of the multiplier $c$ (the leaf memories multiplier), we switch to the ALOI dataset [Geusebroek et al., 2005], which has 100 examples per class enabling good generalization. Figure 5c shows that statistical performance improves with inference time and the value of $c$.

In Appendix §B.2, we include plots showing statistical and inference time performance vs $c$ in Figure 6, with inference time scaling linearly in $c$ as expected.

Last, we test CMT on a series of progressively more difficult datasets generated from ALOI via randomly sampling $S$ training examples per label, for $S$ in 1 to 100. ALOI has 1000 unique labels so the number of memories CMT stores scales as $S \times 1000$, for $S$-shot ALOI. We fix $c = 4$. Figure 5d shows the statistical performance vs inference time as $S$ varies. The prediction error drops quickly as we increase $S$. Appendix §B.2 includes detailed plots. Inference time increases logarithmically with $S$ (Figure 7b), matching CMT’s logarithmic time operation theory.

### 5. Conclusion

CMT provides a new tool for learning algorithm designers by enabling learning algorithms to work with an unsupervised or reinforced logarithmic time memory store. Empirically, we find that CMT provides remarkable unsupervised performance, sometimes beating previous supervised algorithms while reinforcement provides steady improvements.
Contextual Memory Tree

References


