# dreaml: A library for dynamic reactive machine learning

**Eric Wong**
Machine Learning Department
Carnegie Mellon University
Pittsburgh, PA 15213
ericwong@cs.cmu.edu

**Terrence Wong**
Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213
tw@andrew.cmu.edu

**J. Zico Kolter**
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
zkolter@cs.cmu.edu

## Abstract

We present the dreaml (dynamic reactive machine learning) library, a machine learning library aimed at letting users iteratively and incrementally build machine learning models. The library is based upon two main constructs: a hierarchical dynamic data frame data structure, and a computational graph that stores arbitrary functional transforms (both synchronous and asynchronous) between portions of the stored data. We describe the system and illustrate its use on a simple MNIST example with random features. As a whole, dreaml aims to significantly reduce the "non-interactive" waiting time that is present in much machine learning modeling, letting users constantly adjust and evaluate models as they are trained.

## 1 Introduction

A common pattern that occurs in the practice of machine learning is what we call the "train-wait-adjust" loop. Practitioners will set up a machine learning problem, begin training a model on the data, wait for the model to be fit, and then adjust the problem (e.g. adding/changing features, choosing a different model, or adjusting hyperparameters) to improve performance. Naturally, one would assume that nothing but the most productive tasks occurs during the "wait" period (writing papers, applying for funding, cleaning the office coffee machine, etc.), but evidence shockingly suggests that this may not always be the case.

In this paper, we present dreaml (dynamic reactive machine learning), a new open-source machine learning framework and library aimed at mitigated this cycle. The framework is based upon two key principles: 1) all data (including feature transformations of the underlying data), parameters, and predictions are stored and organized in a unified block data structure, the hierarchical dynamic data frame (HDDF); 2) all machine learning operations, from simple feature transformations to full runs of complex algorithms, are represented as transformations upon elements in the HDDF. Following the principles of functional reactive programming, changes to data in the HDDF automatically propagate to their dependent elements in a continual manner. This framework encourages users to interactively build models in an online manner, incrementally adding new features, adding new data, adjusting hyperparameters, and combining models together, all while continually monitoring the resulting performance of the model. While many of the fundamental algorithms underlying this process are well established (i.e., warm-starting optimization problems, online learning algorithms, etc), the dreaml library combines these methods with suitable incremental data structures and reactive programming principles, to make a streamlined system for this process. The library is available at `http://www.dreaml.io`, with all source hosted at `http://github.com/locuslab/dreaml`.

## 2 Background and related work

The dreaml library builds upon several themes in machine learning, distributed systems, and reactive programming. The work on functional reactive programming (FRP) [1] underlies much of the computational model used by dreaml. Like the FRP model, all operations in dreaml are stored as functional transforms of data. When the underlying data changes (or more importantly, changes size), the changes to computations immediately start to propagate through to dependent data blocks.

The process described above also closely resembles the functional data transformations developed in recent years such as those that define Spark RDDs [2], and (even more closely), the data graph and shared data table of the GraphLab framework [3]. However, the goals and implementation of these formalisms under dreaml are substantially different from these previous frameworks. In dreaml, the main goal of the computational graph and data transformations is to enable incremental user-defined data transformations in a *shared-memory* architecture, where the focus is very much on making "medium data" problems fast and responsive to user control. While we believe this could eventually be brought together with these distributed systems, this is currently not an aim of the library.

Finally, the data structures in dreaml build most directly upon the data frame abstraction introduced in the S language [4], now well-known in the R data.frame class [5] and the Python Pandas [6] library. The main additions of the dreaml library, which we will discuss shortly, are the inclusion of hierarchical indices, and an alternative data store based upon a block sparse dictionary of keys format that allows for efficient insertion and removal of large blocks in the data frame; this essentially organizes *all* data and transformations for a machine learning problem within a single data frame.

## 3 The dreaml library

Fundamentally, the dreaml library provides an easy interface for users to incrementally modify and evaluate a machine learning model and optimization process as the model is being trained. The two building blocks for the library are the hierarchical dynamic data frame (a data frame data structure intended for rapid and efficient incremental modification), and the notion of reactive synchronous and asynchronous data transformations.

### 3.1 Example

The elements of the dreaml library are best described with reference to a concrete example. Consider the following code, which trains a stochastic gradient descent algorithm on the MNIST digit recognition problem. Many of the operations here are ultimately bundled into a single library call, but we include them all here for exposition of the underlying design.

```
import dreaml as dm
df = dm.DataFrame()
df["data/train/", "input/raw/"] = dm.DataFrame.from_matrix(X_train)
df["data/train/", "input/label/"] = dm.DataFrame.from_matrix(y_train)
df["data/test/", "input/raw/"] = dm.DataFrame.from_matrix(X_test)
df["data/test/", "input/label/"] = dm.DataFrame.from_matrix(y_test)

df["data/", "features/pca/"] = dm.PCA(df["data/train/", "input/raw/"],
                                      df["data/", "input/raw/"],
                                      num_bases=50)
df["data/", "features/ks1/"] =  dm.KitchenSinks(df["data/", "features/pca/"],
                                                num_features = 1000)
df["weights/", "features/"] = dm.SGD(dm.softmax,
                                     df["data/train/","features/"],
                                     df["data/train/", "input/label/"],
                                     learning_rate = 1e-4,
                                     batch_size = 50,
                                     lam = 1e-3)
df["data/", "metrics/"] = dm.Metric([dm.softmax, dm.multiclass_error],
                                    df["weights/", "features/"],
                                    df["data/", "features/"],
                                    df["data/", "input/label/"])
# after some time
df["data/train/", "features/ks2/"] =  dm.KitchenSinks(df["data/","features/pca/"],
                                                      num_features = 1000)
```

We will describe the internal elements in more detail shortly, but from a user standpoint, this code:

1. Loads the training and testing features and labels into the data frame.
2. Runs PCA and generates 1000 random kitchen sink features.
3. Starts running stochastic gradient descent (continually) to minimize the softmax loss with a specified learning rate and minibatch size.
4. Generates predictions and computes metrics (continually) on the training and test sets.
5. After waiting some time, the user generates an additional set of 1000 random kitchen sink features; the stochastic gradient descent optimizer and metrics automatically respond to the new features in the data.

### 3.2 HDDFs

The hierarchical dynamic data frame (hereafter referred to as an HDDF) is the data structure used for storing and manipulating all data within the dreaml framework. This data frame is akin to the standard R data.frames or Pandas DataFrame object but with two main additions: 1) the indices for both rows and columns are indexed by a hierarchical structure reminiscent of a file system; and 2) we use a multiple block sparse structure to allow for fast insertion and removal of data blocks within the HDDF.

**Hierarchical indices** The hierarchical index system of an HDDF allows data to be naturally organized within a single data frame according to a simple file-system-like hierarchy, where rows/columns are akin to "files" in the system, and directories group together similar rows/columns. Internally, the hierarchical index is represented as an ordered dictionary, where each key is either a "file" that points to a actual row or column in the data frame, or a "directory" that points to another ordered dictionary. For example, in the above code, "data/train/" is a (row) directory containing a collection of 50,000 "files" corresponding to each training example in the MNIST data set. The columns have a similar directory structure that indicates the raw input, labels, generated features, etc. Standard matrix methods of integer, slice, and Boolean indexing are also allowed, as are wildcard characters in the directory structure.

**Block sparse (dictionary of keys) storage format** Internally, the HDDF stores data as a "dictionary of keys" *block sparse matrix*. This structure stores all the elements of the HDDF as a dictionary, where keys are pairs of row partition / column partition pairs (a partition is an index at the *block* level, to differentiate from the hierarchical index) and values are matrices (which can themselves either be full or sparse in any format, for example NumPy arrays or SciPy sparse matrices). A set of "files" in the hierarchical index maps to row or column partitions, plus the particular integer offsets into these partitions. While these blocks often map directly to the directory structure of the hierarchical index, this is not a constraint: a single block can span multiple directories, or a single directory could contain multiple blocks. The key advantage of this storage structure is that incrementally modifying the data frame to add an additional block is a cheap operation, and requires no copying/re-arranging of existing data.

**Caching** At any point the user can request the underlying matrix for an HDDF or subset thereof. However, the matrix can possibly span multiple blocks in the HDDF or be a subset of a block in the HDDF, and piecing together the requested matrix is a costly operation. To prevent significant overheard, we cache requested matrices (in any type desired by the user, e.g. dense, sparse, etc) for fast, repeated retrieval. Cache entries need to be evicted and written back to the HDDF when multiple cache entries overlap. Users can also request cached entries as read-only or read-write, which enables different optimizations for the cache eviction logic.

### 3.3 Data transformations and the computational graph

All operations in the dreaml library are considered to be transformations between sections of an HDDF. The example above illustrates this, with `df["data/", "features/pca/"]` being generated by a call that uses as input the data in `df["data/train/", "input/raw/"]`. These dependencies between sections of the HDDF, along with the transformations, are represented

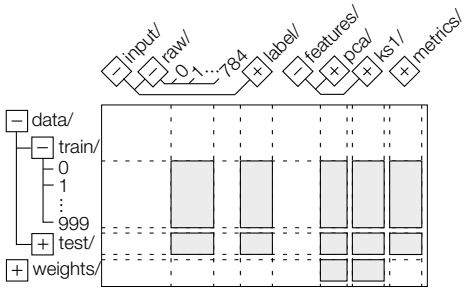


Figure 1: Underlying block sparse matrix representation of the data frame for the MNIST example. Gray boxes represent initialized matrices within the data frame, with dashed lines linking the blocks to their corresponding rows/columns.

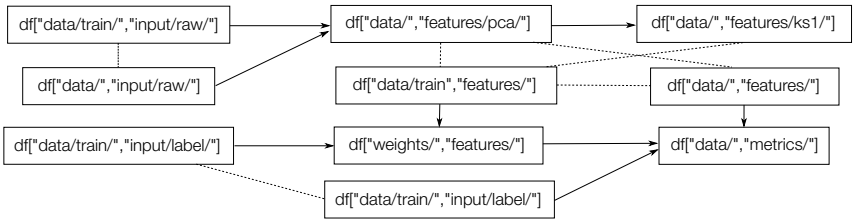


Figure 2: Example of the computational graph generated up to the call to `dm.Metrics` in the MNIST example. Arrows represent the explicit dependencies from data transformations, and dashed lines represent implicit hierarchical dependencies.

as edges and nodes in a computational graph. Maintaining this graph enables us to rebuild the transformation automatically when the underlying data changes.

**Synchronous and asynchronous apply operations** All transformations are represented via an "apply" operation on the data frame, which takes an arbitrary user-provided function and arguments, and saves the result at the target location in the data frame. This effectively lets the user define new blocks in the HDDF as a transformation of other blocks. Information about each transformation and the arguments it depends on is saved within the computational graph. When a block within the HDDF is changed, we traverse the computational graph and rerun all dependent transformations within the graph to reactively update the HDDF entries. Additionally, there is an implicit graph induced by the hierarchical index structure: directories and their subdirectories are dependent on each other, so updates are also communicated to implicitly dependent nodes in the graph. Data transformations can be either one-shot (i.e., synchronous) operations that block dependent computations until they are finished, or continual (i.e., asynchronous) operations that are applied continuously to the relevant portions of the data frame.

For example, in the above code, the last line generates more random features while stochastic gradient descent is running and stores them in `df["data/train/", "features/ks2/"]`. This is implicitly dependent on `df["data/train/", "features/"]`, which is used by the stochastic gradient descent process. Via the computational graph, the stochastic gradient descent and metrics transformations are both automatically restarted to operate on the updated, full feature matrix.

**Example: PCA** For example, the `dm.PCA` function above is implemented as follows:

```
class PCA(BatchTransform):
    def func(self,target_df,X_pca_df, X_full_df, num_bases=50):
        X_full = X_full_df.get_matrix()
        X_pca = X_pca_df.get_matrix()
        X_mean = np.mean(X_pca,axis=0)
        X_std = np.std(X_pca,axis=0)
        _,s,v_T = la.svd((X_pca - X_mean) / X_std)
        target_df.set_matrix(((X_full - X_mean) / X_std).dot(v_T.T[:,:numbases]))
```

### 3.4 Frontend & Visualization

To facilitate the interaction between the user and the model, a web frontend displays visualizations of the HDDF data structure, the dependencies formed in the computational graph, and real time plots of various metrics. The plots, as seen in the figure below, are updated in real time and provide insight into the model while asynchronous apply operations run in the background. More importantly, they provide online feedback to the user when making adjustments to the model, allowing the user to continually monitor the performance in response to changes in the model. For example, a user can keep making adjustments such as adding new features or adjusting hyperparameters, all while evaluating the changes based on immediate feedback from the plots.

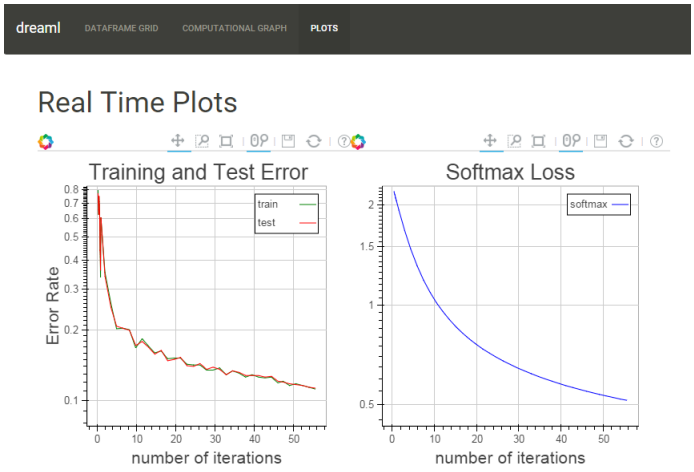


Figure 3: Example of the web interface to view various real time plots of metrics generated while running continuous transformations, such as training error, testing error, and the objective value of an optimization process.

## References


[1] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM Computing Surveys (CSUR)*, 45(4):52, 2013.

[2] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

[3] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[4] John M Chambers and Trevor J Hastie. *Statistical models in S*. CRC Press, Inc., 1991.

[5] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[6] Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.