# Boundary Trees & Differential Boundary Trees

Jay Ricco, David Van Chu

Due: August 8, 2017

## 1 Introduction

The overall goal of this project was to understand and further study the function, effectiveness and efficiency of the Boundary Tree and Differentiable Boundary Tree algorithms.

## 2 Background Knowledge

### 2.1 The Boundary Tree Algorithm

Boundary Trees were introduced in the paper, *The Boundary Forest Algorithm for Online Supervised and Unsupervised Learning* (Mathy, Derbinsky, et.al. 2015), as an instance-based learning algorithm which has the properties of being (i) fast to train, (ii) fast to query, and (iii) able to incrementally learn complex data distributions in an on-line setting.

## 3 Boundary Trees

### 3.1 The Model

To query the Boundary Tree, we take an example as an input and find the closest node using Euclidean distance as the distance function. We return the closest node, and from that, we can get the class associated with the node. This is the class that the Boundary Tree thinks the example is.
To train the Boundary Tree, we query the existing tree with the example, and if the node returned does not have the same class as the example's target class, we add a new node to the tree with the example and corresponding target class.

### 3.2 Testing

Testing both the MNIST and CIFAR10 datasets with Boundary Trees was fairly quick, taking only a few minutes to complete a cycle of training and testing a tree.

#### 3.2.1 MNIST

The MNIST dataset is a large dataset of handwritten digits from 0-10. It contains 60,000 training examples and a test set containing 10,000 examples.

#### 3.2.2 CIFAR10

The CIFAR10 dataset is a large dataset of images containing 50,000 training examples and 10,000 testing examples, split into 10 different classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

### 3.3 Observations (MNIST)

Initial observations indicated that:

1. The maximum branching factor value (k) is crucial with respect to the performance of the algorithm on a given data set.
2. While the total testing time and accuracy plateau, with $\uparrow$ examples, training time $\epsilon O(n)$.
3. BT's perform far better with MNIST than CIFAR; this shows inadequacy of the Euclidean distance metric for high feature complexity use-cases.

Figure 1: For each test the branching factor is changed, and each data point was acquired by taking an average over 10 runs.

| k | Training Time (s) | Testing Time (s) | Accuracy (%) |
|---|---|---|---|
| 2 | 75.96443768 | 14.89228232 | 79.248 |
| 3 | 76.26015964 | 14.11042054 | 82.253 |
| 5 | 72.54322867 | 14.19308667 | 84.988 |
| 10 | 93.82171679 | 18.15971713 | 87.397 |
| 20 | 128.2968537 | 25.59608793 | 88.211 |
| 50 | 165.1073234 | 34.94613609 | 88.036 |
| 100 | 164.9357249 | 35.71075509 | 87.952 |
| $\infty$ | 167.4871073 | 36.10234139 | 87.952 |

After running the tests, we noticed that setting the branching factor to 5-10 seemed to be the best if both

time and accuracy is a concern, as anything less than 5 will result in lower accuracy while taking about the same amount of time to train and test. With branching factors larger than 10, training and testing time grew quite a bit with only a small increase in accuracy.
Another interesting point is that with a larger branching factor, the training time itself varies by quite a bit, ranging from 140.47 seconds to 223.79 seconds, whereas tests ran with a lower branching factor resulted in consistent training times.

Figure 2: MNIST dataset, varying the number of examples, averaged over 10 runs.

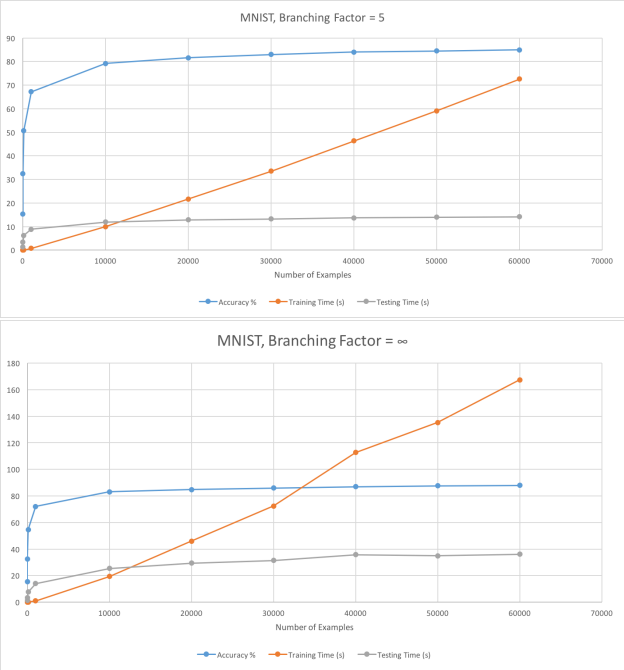



Figure 4: CIFAR10 dataset, varying the number of examples, averaged over 10 runs.

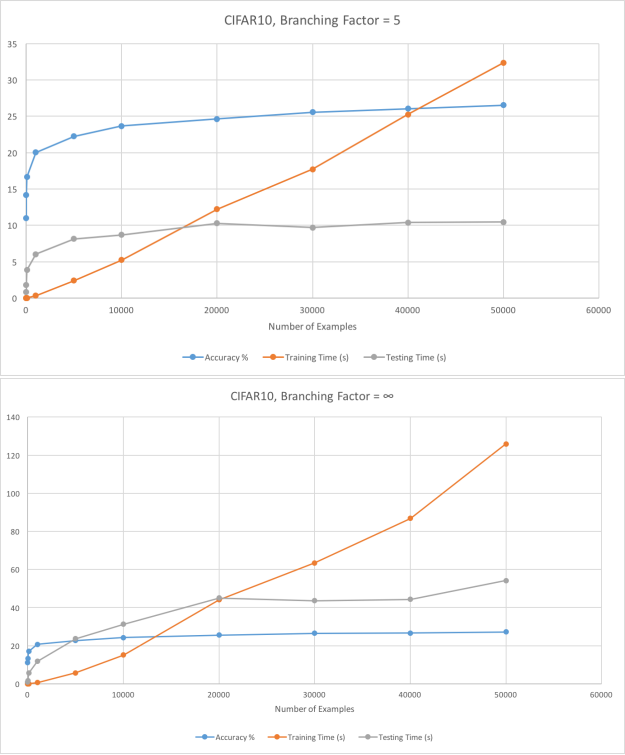



Figure 3: For each test the branching factor is changed, and each data point was acquired by taking an average over 10 runs.

| k | Training Time (s) | Testing Time (s) | Accuracy (%) |
|---|---|---|---|
| 2 | 31.6216013 | 10.17820182 | 23.928 |
| 3 | 28.85782518 | 9.416226149 | 25.422 |
| 5 | 32.36283786 | 10.46060543 | 26.509 |
| 10 | 36.72829039 | 12.98402145 | 27.169 |
| 20 | 51.99976389 | 18.622286033 | 27.708 |
| 50 | 79.45825586 | 29.97079742 | 27.453 |
| 100 | 98.72666419 | 38.09738462 | 27.629 |
| ∞ | 125.9124599 | 54.22032864 | 27.283 |

### 3.4 Observations (CIFAR10)

We see similar results when using Boundary Trees on the CIFAR10 dataset, although it never achieves over 27.7% accuracy. Using a smaller branching factor quickens the training time significantly compared to the MNIST dataset results.

## 4 Differentiable Boundary Trees

### 4.1 Introduction

The Differentiable Boundary Tree algorithm seeks to solve a simple problem: reducing the inadequacies imparted by a poor choice in distance function; coupled with the derived issues of improper features and high feature dimensionality. Most modern machine

learning methods have the ability to use easily obtainable, but technically poor, raw features and then learn to transform and distort them in such a way as to produce an output that is useful. Classical Boundary Trees lack this property, or in other words - will require massive numbers of stored nodes in order to classify examples from distributions with complex, non-linear decision boundaries.

## 4.2 The Model

DBT's build on Boundary Trees in that they apply essentially the same algorithm, except now there is something of a transformational encoder prior to the input, and transitions in the tree are modeled probabilistically (and thus, continuously), instead of movements between discrete objects. This allows the tree to be reduced, essentially, to a probability distribution whose density function represents the expected class returned, with respect to some query example's encoded feature vector.
To do this, at every neighborhood until we reach the closest node in the tree, a transition distribution is formed by applying the softmax function to the negative distances between every member of the local neighborhood and the query's feature vector. This produces higher probabilities if two distinct nodes are closer in the transformed space, and lower probabilities if they are farther apart. The maximum of that distribution is the next node we transition to, and those probabilities are summed over all but the last transition. When the final neighborhood is reached, the class distribution is determined by the expected class of that neighborhood, conditioned on the path probability that has been greedily generated during the algorithm's descent.
At this point, applying that expectation to a loss function, we can optimize the encoder's parameters with the goal of producing class probabilities that are as-close-as possible to the example's one-hot encoded class (which is really just a perfect target probability distribution). This has the effect of not only teaching the transform to maximize the distances between classes, but it also minimizes the tree's depth by requiring fewer examples to bridge incorrectly classified gaps in the learned feature space's decision boundary.

## 4.3 Testing

For all tests, the number of nodes, training time, back-propagation time, and loss were recorded for every batch iteration until convergence (or manual stop).

### 4.3.1 Half Moon

The first test performed on the DBT algorithm was using the Half Moon dataset.
The data is randomly generated and takes the form of two dimensional class manifolds which interlock in a non-linear fashion. The goal of this test is to prove that non-linear learning is occurring - if the manifolds can be manipulated in such a way as to make them linearly separable, then the algorithm works.

### 4.3.2 digits

The digits dataset is a smaller, less complex 'version' of MNIST. This dataset was used to produce the graphic showing the clustering of the transformed input as training continues. This dataset caused no issues, converging in only a few minutes after 700 iterations. In the future, more testing needs to be done regarding profiling - there just wasn't enough time with regard to this project.

### 4.3.3 MNIST

The algorithm did not handle MNIST well. The full dataset took 30 hours to reach convergence with a minimum storage requirement of 16 nodes out of 1000 training examples. This is good with respect to the result, but the training took far too long to be reasonable and often, depending on the order that examples are shown to the tree, convergence may not be reached all together.

## 4.4 Issues & Challenges

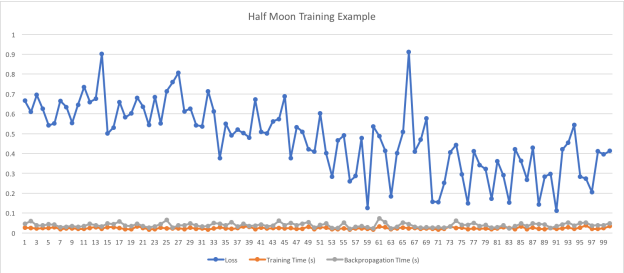
### 4.4.1 Dynamic Graph Computation

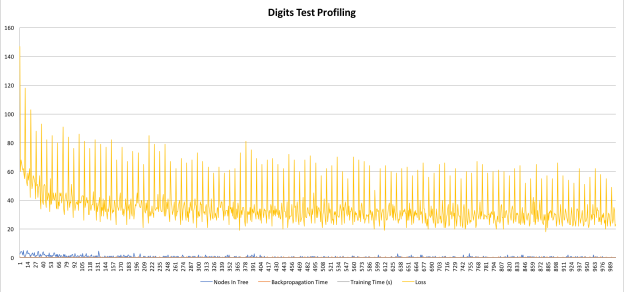
The first significant set-back originated from the computation libraries back-end. An initial implementation was designed and built in TensorFlow; however this library only supports computation on static graphs. In order to be able to dynamically generate the computation graph required for calculating probabilities down the tree, PyTorch was the library required for implementation. Switching libraries added about 3 weeks of work (learning curve and all), which was a costly and frustrating mistake.

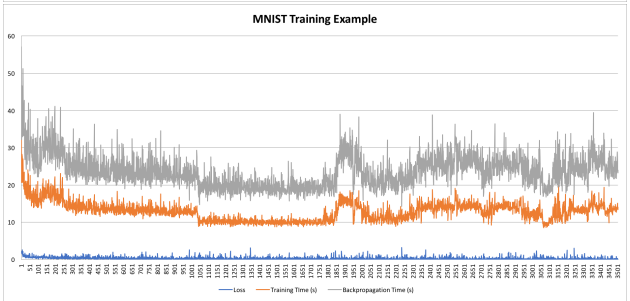
## 4.5 Testing Time & Convergence

With no code samples and little direct implementation information in the original paper, much of the implementation's structure was guessed and assumed; this is an obvious source for error between data given by the author's and data collected/ observations made by further experimentation. Still,

despite that - the algorithm's convergence was very dependent on the structure of the transformation, the activation function, the loss function, the optimizer, and the range of the input data. As well, training on large data sets (like MNIST) bore extremely long waiting times which slowed down progress tremendously. It's hard, if not impossible to clearly and faithfully visualize the learning's progress for any dimensionality greater than 3, and the time it takes for the loss to show noticeable signs of convergence is not short enough.




# A Source Code Listings

The Github for this project, which holds a track record of all the work done can be found at: https://github.com/jayricco/AIFinalProject

| FileName | Description |
|---|---|
| Report_BT.py | Main MNIST Testing Implementation (BT) |
| ReportBTCifar.py | Main CIFAR10 Testing Implementation (BT) |
| DifferentiableBoundaryTrees.py | Main Testing Implementation (DBT) |
| MNIST.py | MNIST Dataset handler (DBT) |
| main.py | Main Driver (DBT) |
| DifferentiableBoundaryTrees_VDigits.py | Gif Producing Implementation (DBT) |
| DifferentiableBoundaryTrees_VHalfMoon.py | Gif Producing Implementation (DBT) |

# B Data Set Listings

| Data Set | Source |
|---|---|
| MNIST | `torchvision.datasets` |
| CIFAR-10 | `torchvision.datasets` |
| digits | `sklearn` |
| Half Moon | `sklearn` |
| Ground Truth | `Generated` |