Learning Deep Nearest Neighbor Representations Using Differentiable Boundary Trees

Explainable Machine Learning Report

PD Dr. Ullrich Köthe

Heidelberg University

Summer semester 2018

Author

Benedikt Kersjes

1 Introduction

To explain the decisions of a machine learning model for a specific instance, it can be useful to compare them to the results for similar instances. However, it is not trivial to compute similar instances as it is necessary to define a suitable representation of the instances and an appropriate distance metric. The Differentiable Boundary Tree algorithm addresses the first problem by *learning* a representation which results in simple class boundaries in the transformed feature space [2]. Therefore, the algorithm improves the performance of the original Boundary Tree algorithm [1].

For the seminar, I was assigned the paper by Zoran et al. in which the Differentiable Boundary Tree algorithm was introduced. As we had only limited time for our seminar presentation, it was not possible to cover all parts of the topic in detail, but it was necessary to concentrate on specific and hopefully interesting aspects. In my seminar presentation, I concentrated on the procedure of the original Boundary Tree algorithm and the derived Differentiable Boundary Tree algorithm which I explained in detail. In addition to that, I focused on the experiments Zoran et al. conducted on the derived algorithm. However, I just briefly explained the math behind the differentiable version of the algorithm and also did not talk much about the properties of the original algorithm.

To cover the remaining aspects of the topic, I decided to focus on them in this report. Nevertheless, I think it is necessary to start with an explanation of the original algorithm, as I already did in the presentation. Otherwise the report would not be useful without having heard the talk before. Therefore, in the second section, I will describe the Boundary Tree algorithm and some interesting properties, which I did not mention in the talk. In the third section, I will focus on the math behind the Differentiable Boundary Tree algorithm. Zoran et al. skip several intermediate steps in their paper which might be useful for some people to understand the algorithm in detail.

2 Boundary Tree Algorithm

The Differentiable Boundary Tree algorithm is build on the Boundary Tree algorithm, which did not use representation learning and is therefore not as powerful as the advanced Differentiable Boundary Tree algorithm. Nevertheless, it is important to understand the original algorithm first before focusing on the Differentiable Boundary Tree algorithm. The original algorithm was invented by Mathy et al. and published in 2015 [1]. The description in this report is based on the original paper [1], as well as on the description in [2].

The Boundary Tree algorithm is an online learning algorithm, that can be trained and queried fast. A Boundary Tree is constructed by iterating over the training set instance by instance. For each instance the current tree is traversed, starting at the root node. To select the next node, the distance of the current node and all its children are compared to the current instance. The next node is the node with the smallest distance to the current instance. If the current node does not change, i.e. the current node is closer to the current instance than all of its children, we stop at this node. If we reach a leaf node, the algorithm also stops. The node at which the algorithm stops is called the locally closest node, as it is closer to the query node than all its ancestors and its direct children. The label of the locally closest node is then compared to the label of the current instance. If the labels match, the current instance is discarded and will not be added to the tree. If the locally closest node and the current instance have different labels, the current instance is added as child to the locally closest node. From this algorithm definition directly follows the property, that each edge in the Boundary Tree crosses a boundary between two classes, thus the name Boundary Tree.

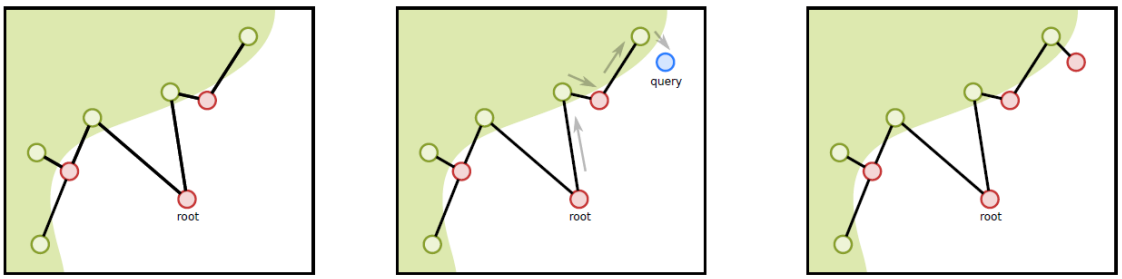

Figure 2.1: Visualisation of the original Boundary Tree algorithm. The current tree on the left, the traversion for the blue query point in the middle and the resulting tree on the right.

Figure 2.1 visualizes a simple example of a Boundary Tree. The tree is trained on a two class dataset (green and red). The boundary between the two classes is also shown in the image. On the left, the current tree is shown, before the blue query point is processed. In the center, the traversion through the tree for the blue query point is visualized. In this example, the locally closest is a leaf node and has a different class label than the query node. Therefore, the query node is added to the tree, which results in the image on the right.

Mathy et al. who invented the Boundary Tree algorithm used the algorithm in a ensemble setting, which they called Boundary Forest. The idea is to train multiple Boundary Trees on the same training set. When a query point needs to be evaluated, the output of all Boundary Trees is combined to determine the result. For instance, in a classification setting, the majority vote could determine the predicted label. For regression, one could take the average over all single tree outputs. Interestingly, it seems to be sufficient for decorrelation to start each tree with a different instance from the training set, i.e. to give each tree a different root node, to train each tree with the root nodes of all other trees and to start online learning then [1]. This means, after each tree has been trained with a small number of instances, all further instances are in the same order for all trees. However, this does not seem to reduce the algorithm's overall performance.

Mathy et al. state, that a Boundary Tree has an interesting property, which they call *immediate one-shot learning*. This means, that a Boundary Tree, which is trained on a specific training instance and directly after that queried for the same training instance, will definitely give the right answer for that instance. This is easy to prove, as the path for the query point will be the same as for training on the same instance before. If the labels of the locally closest node matched the training instance's label, the training instance would not have been added to the tree. Therefore, the tree would be exactly the same and the output would be again correct. If the labels did not match, the training instance would have been added as a child to the locally closest node. This would also output the correct result, as the distance of the query point to itself is of course smaller than the distance to the locally closest node. Obviously, this property does only hold if no other node was inserted in the meantime, but Mathy et al. found out, that the performance on the full training set after training is still below 1% for all data sets they considered.

3 Differentiable Boundary Trees

In the original version of the Boundary Tree algorithm, the instances were represented in the original feature space. This is not very useful, especially when dealing with pixel input. Of course it is possible to use another hand-crafted representation for the data, but the optimal representation will highly depend on the application and could therefore not be reused in most cases. Differentiable Boundary Trees solve this problem by applying a transformation function on the data, which can be learned by gradient descent on a cost function which is assigned to each Boundary Tree.

The idea of the cost function is to calculate, how probable a path through the tree is for a given query point. If this is done naively, the probability for the actual path through the tree for a given query point is 1 and the probability for all other paths is 0. For better generalization, the Differentiable Boundary Tree algorithm models the transitions between nodes by stochastic events, which results in smoother path probabilities.

The transition probabilities are defined for nodes in a transition neighbourhood. A transition neighbourhood consists of a parent node and all its children. The transition probability for a node gives the probability for the transition from the parent node to that specific node. The probability depends on the distance between the query point and the node. The higher the distance, the lower the transition probability.

$$p(x_i \rightarrow x_j|y) = \underset{j \in \{i, child(i)\}}{SoftMax}(-d(x_j, y)) \tag{3.1}$$

Where i is the index of the root node of the neighbourhood and $child(i)$ is a function that returns the indices of all direct children of x_i. There is also a probability to stay at the current node if $j = i$. $d(x_j, y)$ is the distance between node x_j and the query point. Although the representation of the data is learned in this algorithm, the distance function is still hand-crafted and can be defined arbitrarily. For their experiments, Zoran et al. simply used the Euclidean distance and achieved quite reasonable results [2].

Having defined this, the probability for a path follows directly by multiplying the single transition probabilities, since the transitions are conditionally independent of

each other. This is due to the fact, that the distance of a node in the training set to the query point is independent of the distance of any other node to the query point. The transition probability only depends on these distances, thus the transitions are conditionally independent of each other.

$$p(path|y) = \prod_{i \to j \in path} p(x_i \to x_j|y) \tag{3.2}$$

As we are interested in class probabilities, we calculate the probability for a class given a specific path as follows:

$$p(c|path, y) = (\prod_{i \to j \in path} p(x_i \to x_j|y))c(x_{final}) \tag{3.3}$$

Where $c(x_{final})$ is the indicator function for the class label c. The probability for a class label over all possible paths follows by taking the expectation over all paths.

$$p(c|y) = \mathbb{E}_{path|y}(p(c|path, y)) \tag{3.4}$$

However, this is computationally infeasible, since there is a infinite number of paths through the tree. According to [2], the probability can be approximated by considering only the greedy path, which is the exact same path the original Boundary Tree algorithm would take for a given query.

$$\mathbb{E}_{path|y}(p(c|path, y)) \approx p(c|path^*, y) \tag{3.5}$$

$$p(c|y) = (\prod_{i \to j \in path^*} p(x_i \to x_j|y))c(x_{final}) \tag{3.6}$$

Considering multiple paths however gives us softer class predictions, since we do not only receive a probability for the class of the final node in our path, but for all classes present in any of the final nodes of our paths. Therefore, we can also consider all siblings (and the parent) of our final node, i.e. all nodes in the final transition's neighbourhood. We define $path^+$ to be $path^*$ reduced by its last transition and x_l to be the final node of $path^+$.

$$p(c|y) = \prod_{i \to j \in path^+} p(x_i \to x_j|y)) * (\sum_{x_k \in \{x_l, sibling(x_{final}), x_{final}\}} p(x_l \to x_k|y) c(x_k)) \tag{3.7}$$

Multiplying this out would give us the sum of several paths, which brings us closer to the expectation over all paths and is therefore a better and smoother approximation of the class predictions. Note, that this probabilities will never sum up to 1, as the expectation over all paths and all classes sums up to 1 and we are considering only a small subset of all paths. Therefore, the probabilities are normalized to get a correct distribution. Applying the logarithm gives us the following log probabilities:

$$log\ p(c|y) = \sum_{i \to j \in path^+} log\ p(x_i \to x_j|y)\ + log\ (\sum_{x_k \in \{x_l, sibling(x_{final}), x_{final}\}} p(x_l \to x_k|y)\ c(x_k)) \tag{3.8}$$

If we have only two classes, at least one node of each class will be present in the final node's neighbourhood. This follows from the property, that each edge in the tree crosses a boundary between two classes. Therefore, all siblings of the final node are of the same class as the final node and their parent is of the other class. Thus, all probabilities will be non-zero. For a problem with more than two classes, this does not hold. In these cases we need to somehow handle possible zero probabilities when applying the logarithm. The authors of [2] do neither mention this problem nor provide code, so we cannot be sure, how they address this issue.

As we want to learn a better representation than the raw input representation, we can apply a transformation function to all inputs which are learned by a neural network.

$$\begin{aligned} log\ p(c|f_\theta(y)) = & \sum_{i \to j \in path^+} log\ p(f_\theta(x_i) \to f_\theta(x_j)|f_\theta(y)) \\ & + log\ (\sum_{x_k \in \{x_l, sibling(x_{final}), x_{final}\}} p(f_\theta(x_l) \to f_\theta(x_k)|f_\theta(y)) c(x_k)) \end{aligned} \tag{3.9}$$

Zoran et al. insert this last equation into the cross-entropy loss function to be able to perform gradient descent on the parameters of the transformation network. An important note is that all above equations require a fixed tree, which does not change during training. Otherwise, paths and probabilities would change along with the

transformation function which could lead to undesired results [2].

To train on a single instance, Zoran et al. automatically construct a network architecture, which consists of multiple parts that all share their weights. Each part transforms the nodes of a transition neighbourhood and outputs one part of the log probability of the path. Figure 3.1 illustrates an example network architecture.

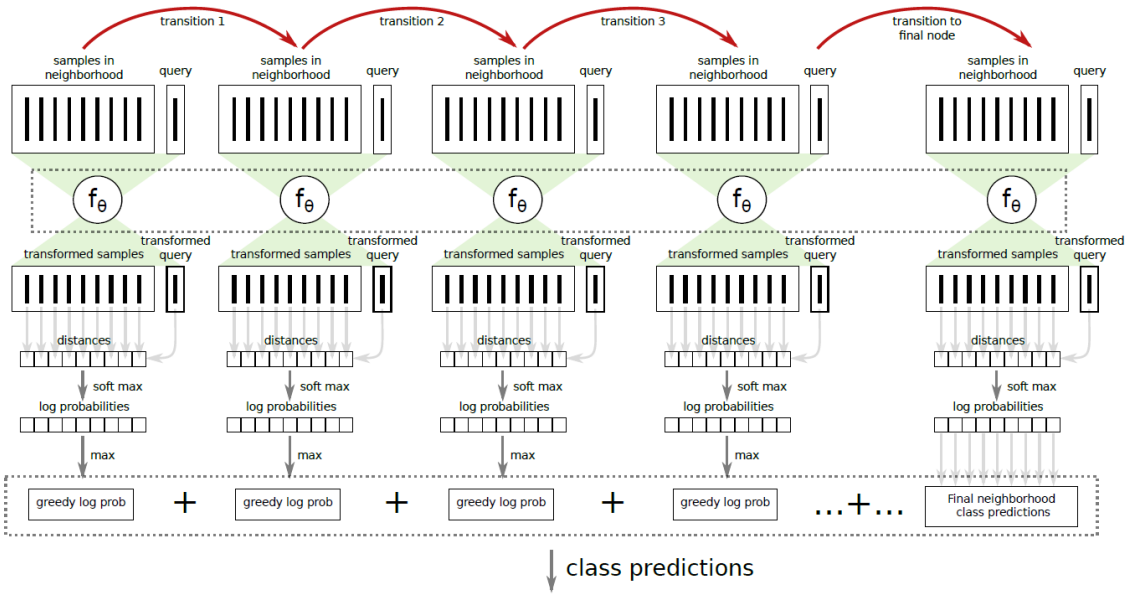

Figure 3.1: Visualisation of an example architecture for a path with 4 transitions.

4 Conclusion

The Differentiable Boundary Tree algorithm is a promising improvement of the original Boundary Tree algorithm. It has a simple structure, can be queried fast and provides a high accuracy for the experiments Zoran et al. conducted.

A disadvantage of the algorithm is, that the algorithm cannot be used in batched training yet, as the network architecture depends on the path through the network and is different for each training instance.

5 References

[1] Charles Mathy, Nate Derbinsky, José Bento, Jonathan Rosenthal, and Jonathan S Yedidia. The boundary forest algorithm for online supervised and unsupervised learning. In *AAAI*, pages 2864–2870, 2015.

[2] Daniel Zoran, Balaji Lakshminarayanan, and Charles Blundell. Learning deep nearest neighbor representations using differentiable boundary trees. *arXiv preprint arXiv:1702.08833*, 2017.