Learning Deep Nearest Neighbor Representations Using Differentiable Boundary Trees

Daniel Zoran, Balaji Lakshminarayanan, Charles Blundell

arXiv:1702.08833v1 [cs.LG] 28 Feb 2017

정 태 성

모두연 자연어처리

Abstract

Introduce a new method called **differentiable boundary tree** which allows for learning **deep kNN representations** built on **boundary tree algorithm**

1. Introduction

k-NN : Challenges

- Finding good representations and distance measures for the success of a given k-NN method

- Representation is chosen ad-hoc or engineered by hand.

- Computational and memory requirements
  (As much as hardware has advanced, this still remains a big problem)

- Typically k-NN methods scale quite badly with data size, often with challenging trade-offs – faster retrieval usually requires more memory

Boundary Tree

- The Boundary Tree (or forest) algorithm has some very appealing properties in tackling the computational and memory requirements of k-NN methods.

- Refer to (Mathy et al., 2015)

- The word “boundary” in the name relates to its use in classification, where most of the nodes in a BT will be near the boundary between different classes.

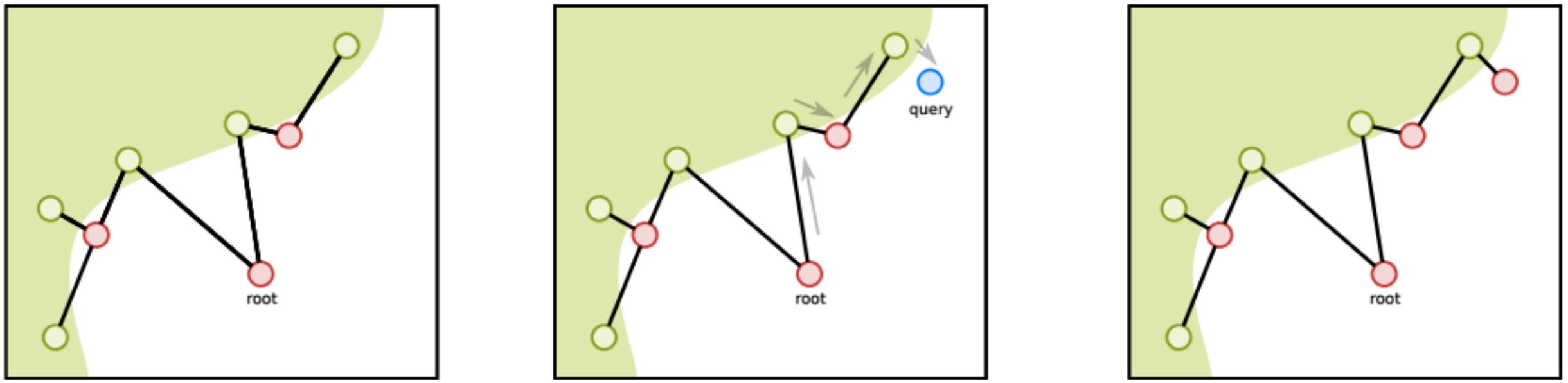


Figure 1. Boundary trees are built in an online manner, sample by sample. From left to right: Given the current tree (depicted in the left image) we start with the root node. For each query we recursively traverse the tree, choosing the locally closest node to the query node at each step. Once traversal stops we use the final node's class to make the prediction (middle image). If the prediction is wrong (as it is in this example) we add the query node as a child to the final node, resulting in a new tree (depicted in the right image). Otherwise the query node is discarded. Edges in the tree cross class boundaries by definition and samples will tend to reside close to the boundaries, hence the name "Boundary Tree".

BT’s interesting property

- Edges in the tree cross class boundaries
- Samples will tend to reside close to the boundaries

——> ”Boundary Tree”.

- **Boundary tree -> hierarchical data structure** that enables efficient k-NN queries.

- Boundary trees allow for **fast k-NN classification**, **regression** and **retrieval** and their **memory requirements grow very slowly** with the amount of data presented, all within a simple and elegant formulation.

2. Model

Differentiable boundary trees

Good representation for boundary trees?

- Such a representation should be one that **transforms the samples** in a way which **transforms the boundaries** between different classes to be **as simple as possible.**

-> This would result in **simpler tree structures**, **faster queries** and **less memory requirements**.

How to?

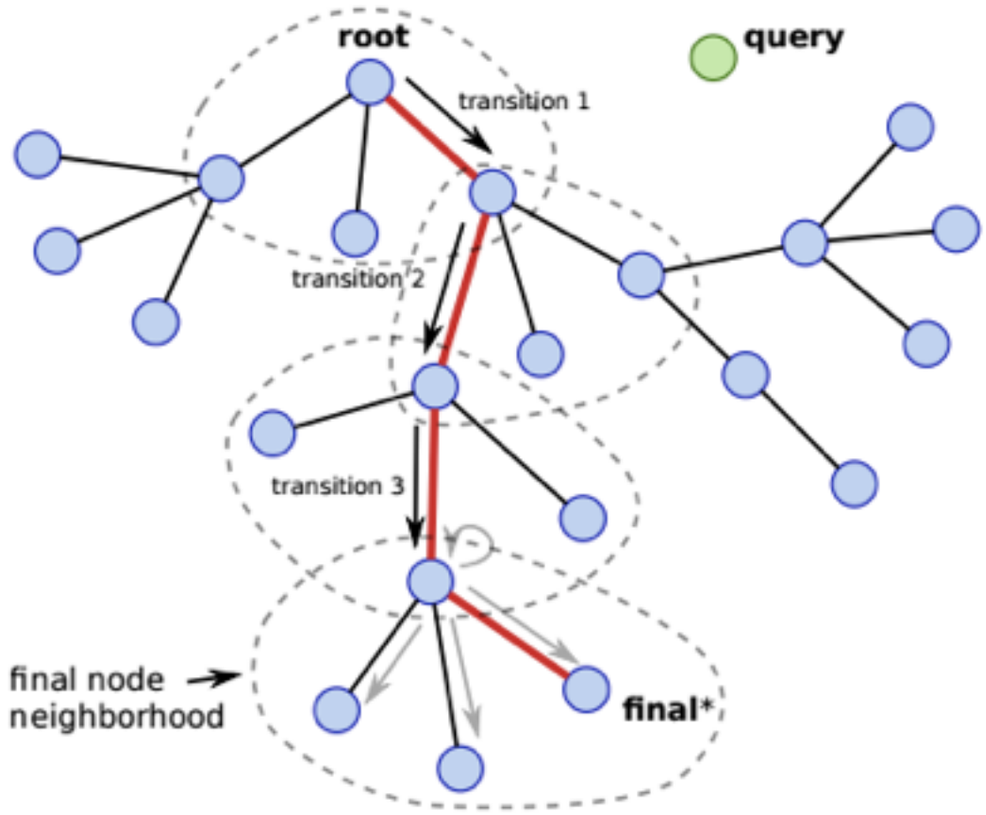
- Associate a differentiable cost function with a boundary tree and optimize it directly.

- Paper proposed to **model transitions** in the tree **as stochastic events** where the respective probabilities are a **function of the distance between the query node and the nodes in the tree**

$$p(\mathbf{x}_i \to \mathbf{x}_j|\mathbf{y}) = \underset{i,j\in\text{child}(i)}{\text{SoftMax}}(-d(\mathbf{x}_j,\mathbf{y}))$$

$$= \frac{\exp(-d(\mathbf{x}_j,\mathbf{y}))}{\sum_{j'\in\{i,j\in\text{child}(i)\}}\exp(-d(\mathbf{x}_{j'},\mathbf{y}))} \qquad (1)$$

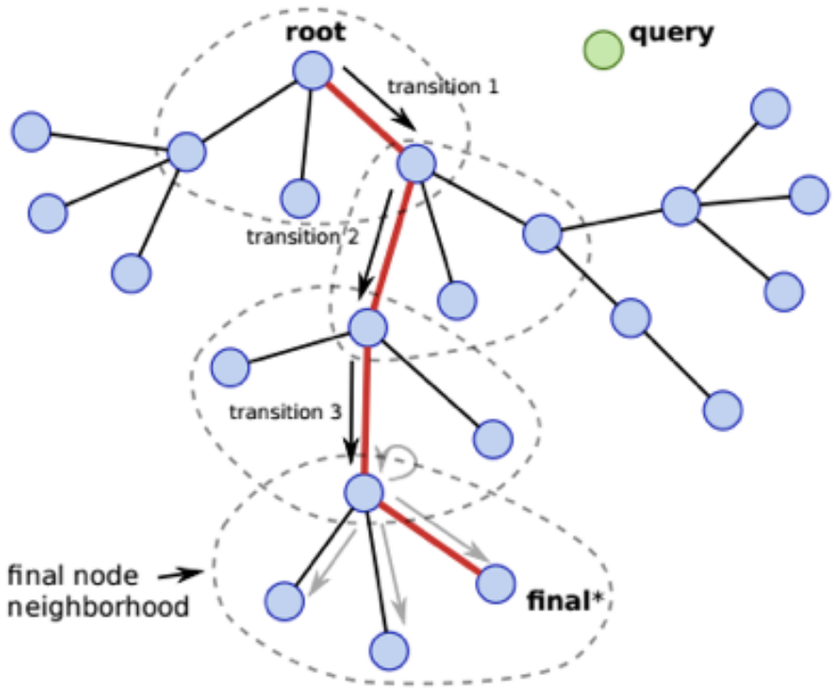
- x : the features of a data point
- c : class label (one-hot)
- xi : given the current node
- y : query node
- xj : xj ∈ {child(xi), xi}, that is one of the children of node xi or staying at this node, as the following
- d(x, y) : distance between x and y (L2)



$$d(\mathbf{x},\mathbf{y}) = \sqrt{\sum_k (x_k - y_k)^2} \qquad (2)$$

x can be an embedding (as we show below, after obtaining a differentiable cost function we can augm
with a neural network to produce these embeddings) and not necessarily raw data.

$$p(\text{path}|\mathbf{y}) = \prod_{i \to j \in \text{path}} p(\mathbf{x}_i \to \mathbf{x}_j|\mathbf{y}) \quad (3)$$



$$p(c|\mathbf{y}) = \mathbb{E}_{\text{path}|\mathbf{y}}(p(c|\text{path}, \mathbf{y})) \approx p(c|\text{path}^*, \mathbf{y}) \quad (4)$$

$$\log p(c|\mathbf{y}) = \sum_{\mathbf{x}_i \to \mathbf{x}_j \in \text{path}^\dagger|\mathbf{y}} \log p(\mathbf{x}_i \to \mathbf{x}_j|\mathbf{y}) + \log \sum_{\mathbf{x}_k \in \text{sibling}(\mathbf{x}_{\text{final}^*})} p(\text{parent}(\mathbf{x}_k) \to \mathbf{x}_k|\mathbf{y})c(\mathbf{x}_k) \quad (5)$$

$$\log p(c|f_\theta(\mathbf{y})) = \sum_{\mathbf{x}_i \to \mathbf{x}_j \in \text{path}^\dagger|f_\theta(\mathbf{y})} \log p\left(f_\theta(\mathbf{x}_i) \to f_\theta(\mathbf{x}_j)|f_\theta(\mathbf{y})\right) + \log \sum_{\mathbf{x}_k \in \text{sibling}(\mathbf{x}_{\text{final}^*})} p(\text{parent}(f_\theta(\mathbf{x}_k)) \to f_\theta(\mathbf{x}_k)|f_\theta(\mathbf{y}))c(\mathbf{x}_k) \quad (6)$$

- **Each transition** is **conditioned** upon a query node, **y**
- **Each transition** is **conditionally independent** of the **p**

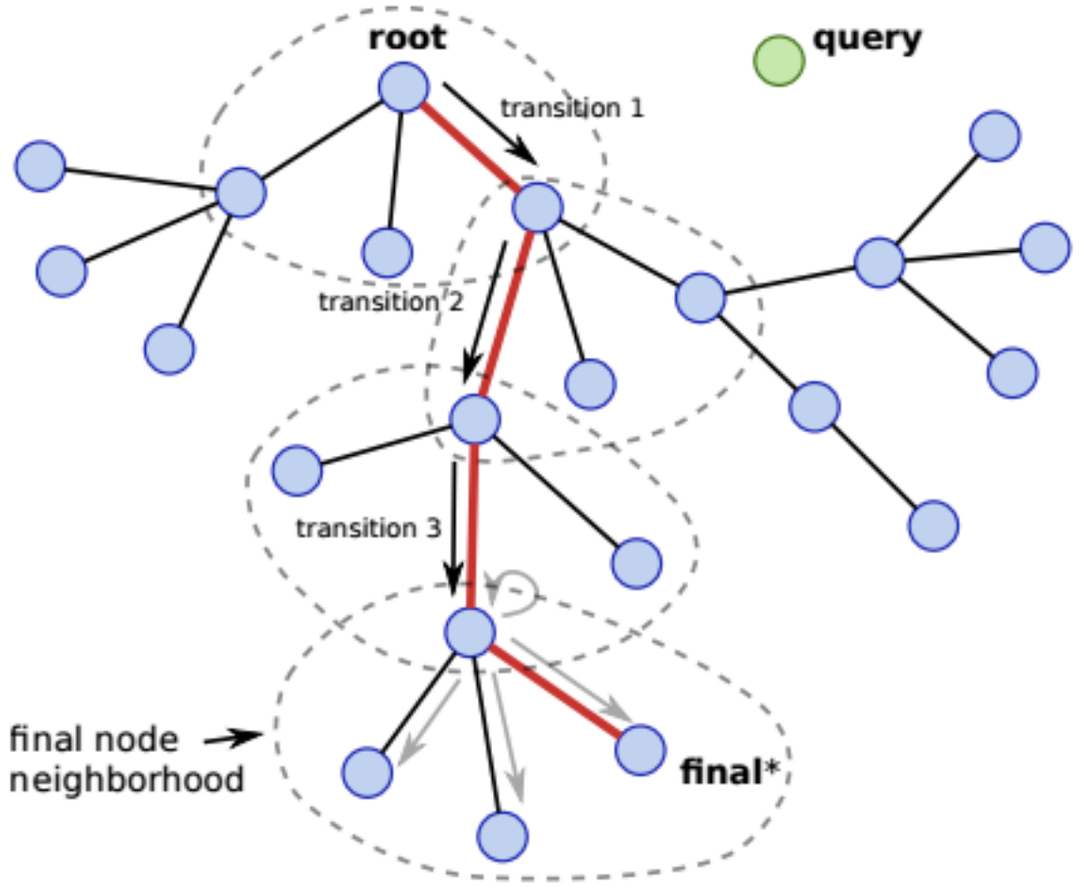


Figure 2. Visualization of the different neighborhoods and transitions involved in the construction of the cost function in Equation 6. The tree is presented in an arbitrary 2D space here (for visualization). Given the query node we greedily traverse the tree down path* (marked in red) after transforming all samples through f_θ. The probability of each transition is calculated up until the final node's neighborhood. Here we aggregate the nodes' class labels, weighted by their respective transition probability, to build the class prediction output. See Figure 3 for a visualization of the associated neural net which is used to compute the cost function.

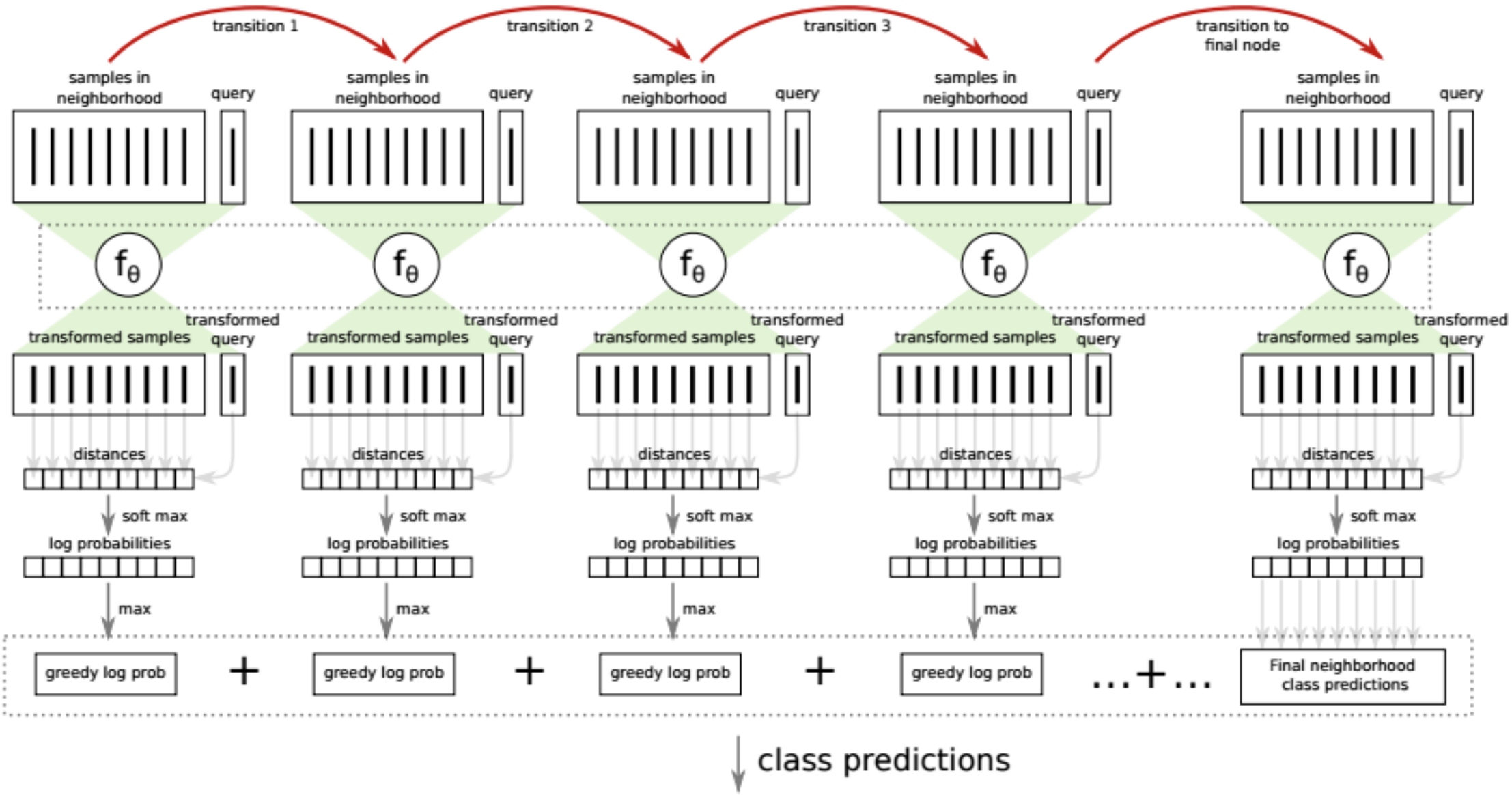


Figure 3. The neural net needs to be dynamically constructed for each query point. For each transition through the tree modules outputting the transformed samples are shared. Each module takes the transformed samples, calculates distances between them and the transformed query point, then converts to log probabilities. Transitions are based on the transformed samples. These are combined with the final node's predictions to produce the class predictions, and the loss is propagated through the built network. See Figure 2 for a visualization of the corresponding tree structure and path.

Algorithm 1 Learning representation using differentiable boundary trees

Initialize f_θ to random weights
while not converged **do** ▷ *Convergence when change is below a threshold*
    Discard current tree T and initialize new tree
    Train new tree T using samples transformed with current f_θ
    for $t = 1, \mathsf{NumSamplesForRepresentations}$ **do**
        Get next training sample $\mathbf{y}$
        Calculate loss using Equation 6 and current tree T
        Take gradient of loss w.r.t parameters θ and perform gradient step
    end for
end while

4. Experiments and analysis

4.1. Half-moon dataset

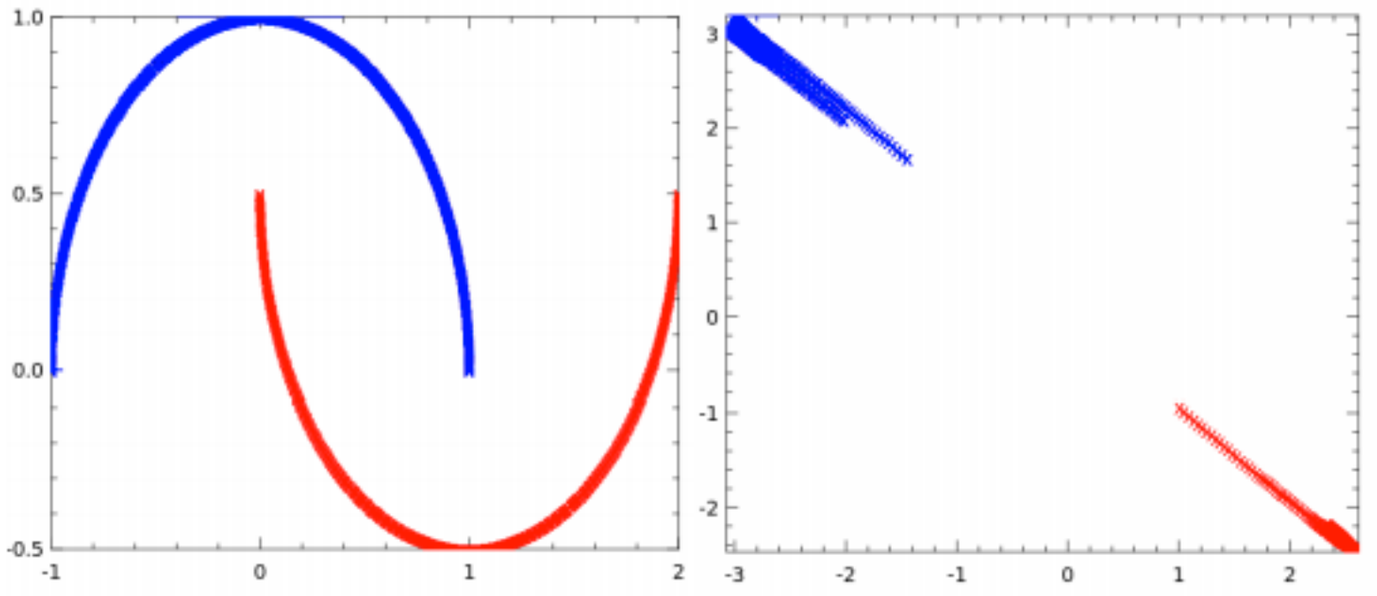


Figure 4. Half moon dataset and the learned transformation. On the left is the raw data, on the right is the transformed data using a learned representation. We train a simple 3 hidden layer fully connected net with our method. As can be seen, the network learns to separate the two classes completely. Using this representation, a trained boundary tree has only 2 nodes and achieves 0% error on both train and test sets.

- 1000 samples (20 samples for a boundary tree, 10 gradient steps)
- For the transform **fθ,** trained a 3 layer, fully connected net (2 → **100** → **100** → **30** → 2 units)
- 결과: When using this representation, training a boundary tree results in a **2 node tree** which yields **0% training** and **0% test errors**.

4.2. MNIST classification

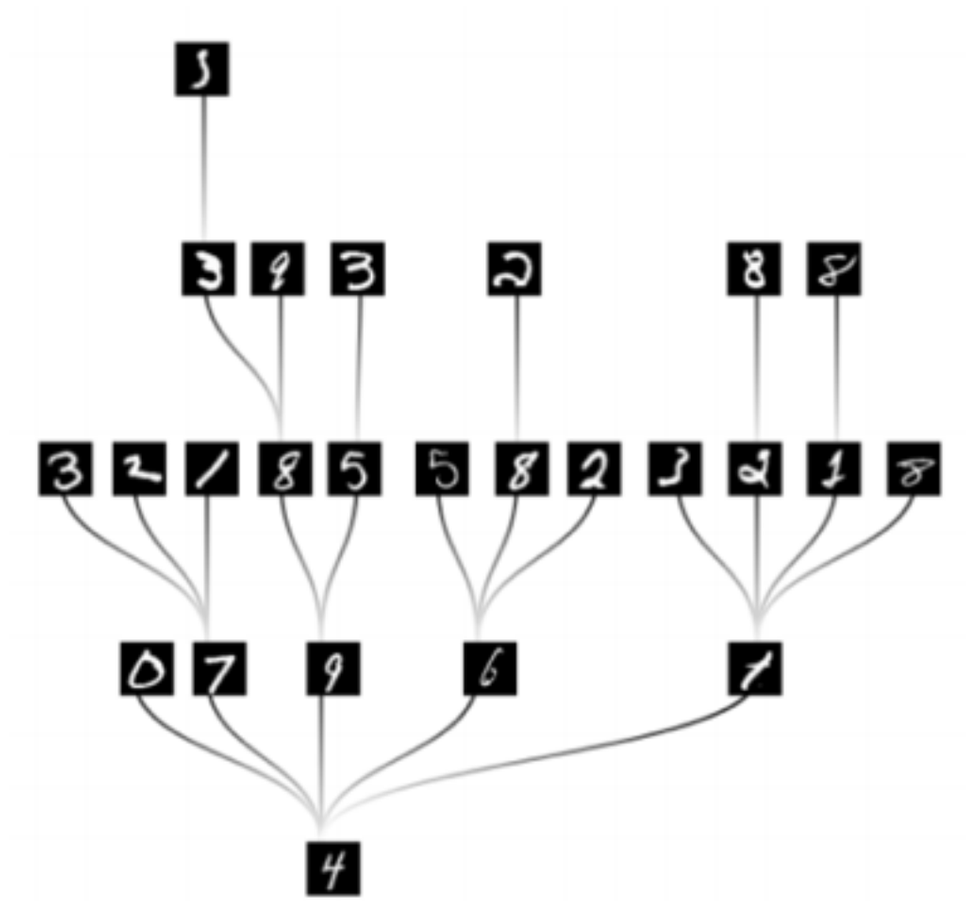


Figure 5. A tree built using the trained representation over MNIST digits using 1000 samples. Samples are presented here in the original pixel space, but the learned representation is used to construct the tree. Note the simple, interpretable structure — nodes are either prototypical examples or boundary cases. Remarkably, this tree still achieves less than 2% error on the test set. See text for details.

1. 1000 samples for BT

2. 1000 gradient steps

(위 1, 2 과정 반복 until convergence)

784 → 400 → 400 → 20 units respectively).

Adam Optimizer

2% error on the test set

Tree has a very **interpretable** structure

How can the tree store such a small number of samples and still achieve this level of performance?

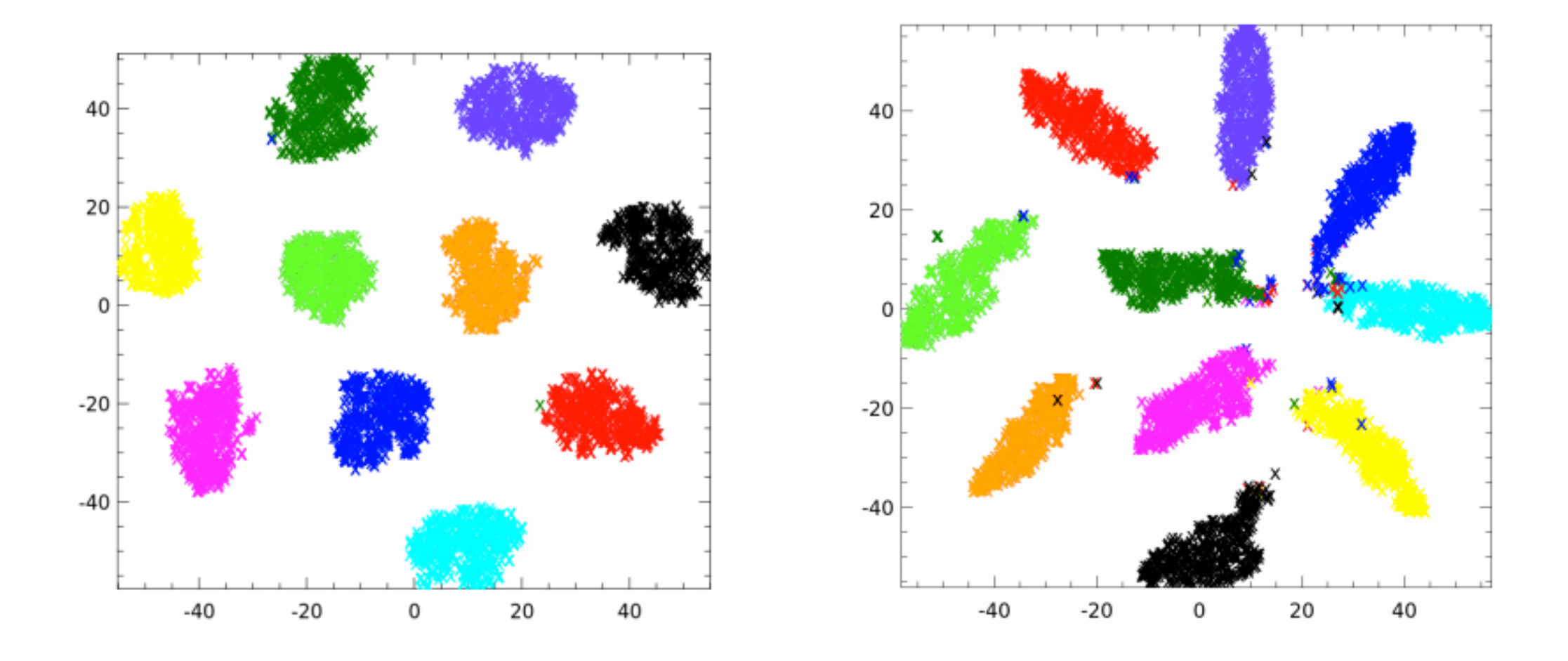


Figure 7. t-SNE visualisation of representations. On the left are samples projected using our trained representation (from the training set). As can be seen, the transform separates the classes nicely, allowing the tree to have a very small number of nodes in order to achieve its task. On the right is the t-SNE plot made with neural net classifier outputs. Though classes are separated nicely, it's not as clean as our result.

4.3. Limitations and scaling

- Due to the discrete nature of the path we take through the tree we need to build a different computation graph for each query node. This makes batching very inefficient and thus prevents us from running experiments on larger scale data such as ImageNet images requiring the use of GPUs.

- 트리 생성(a discrete operation)과 representation 업데이트(continuous)라는 다른 프레임웍 사이를 오가는 알고리즘의 반복성 - 같은 프레임웍이었다면 좀 더 세련되고 효율적..

- 초기에 prediction의 높은 에로율로 인해 트리가 커질 수 있음. 적은 샘플로 트리를 생성함으로써 이 문제를 좀 완화시킴.

- Though we cannot learn the representation directly on larger datasets we can certainly try to visualize what a boundary tree trained on pre-trained representation would yield.

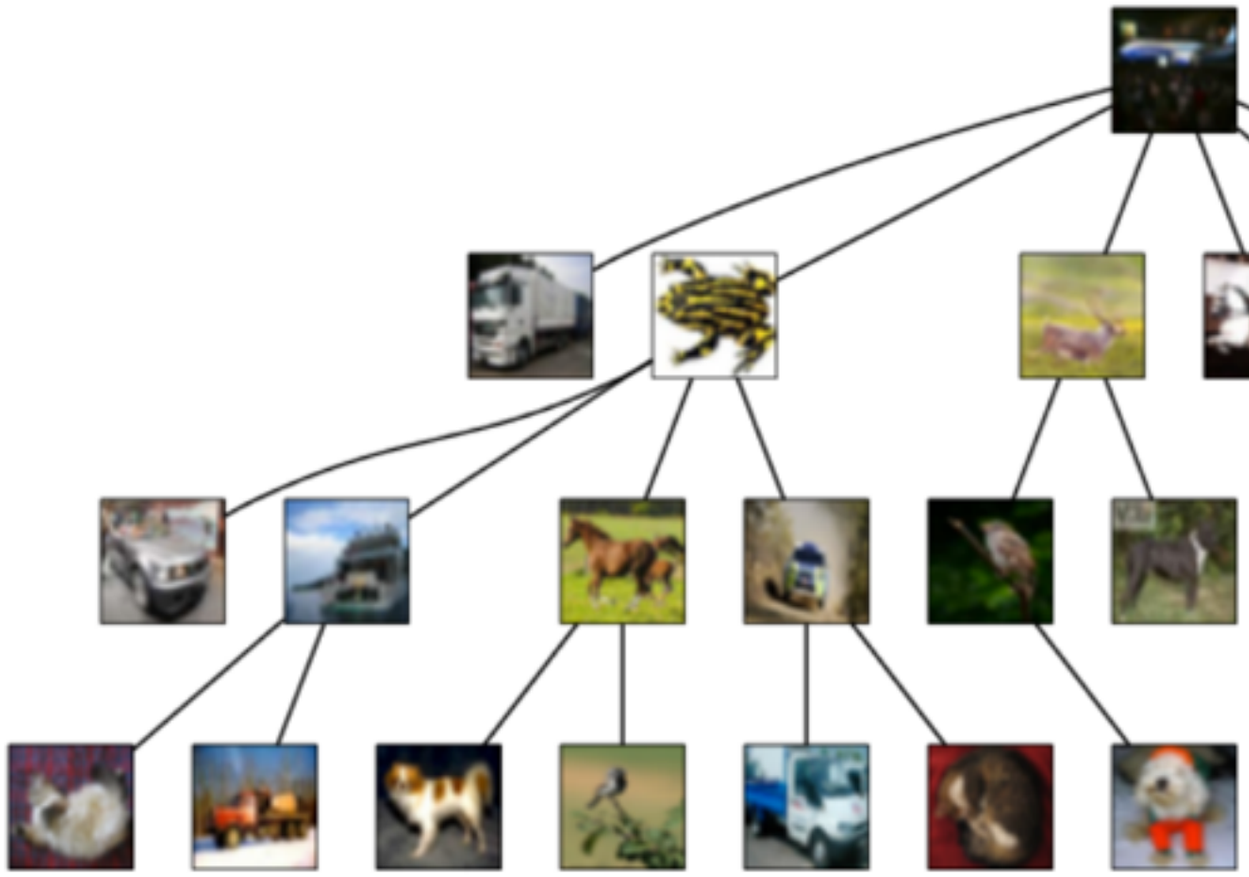


Figure 8. A tree built using the pre-trained representation over CIFAR10 images using 100 samples. Samples are presented here in the original pixel space, but the learned representation is used to construct the tree. Note the simple, interpretable structure — nodes are either prototypical examples or boundary cases. This tree achieves 13.06% error on the test set using just 22 samples. See text for details.

Figure 8 shows an example of this: we used a pre-trained VGG style network which was optimized for predicting classes on the CIFAR10 dataset using the cross entropy loss. We then use the outputs from the layer before the softmax, as embeddings for the inputs. As can be seen, the tree learned from 100 samples is quite compact with 22 nodes. The classification error for the tree is 13.06% vs. 9% for the pre-trained net.

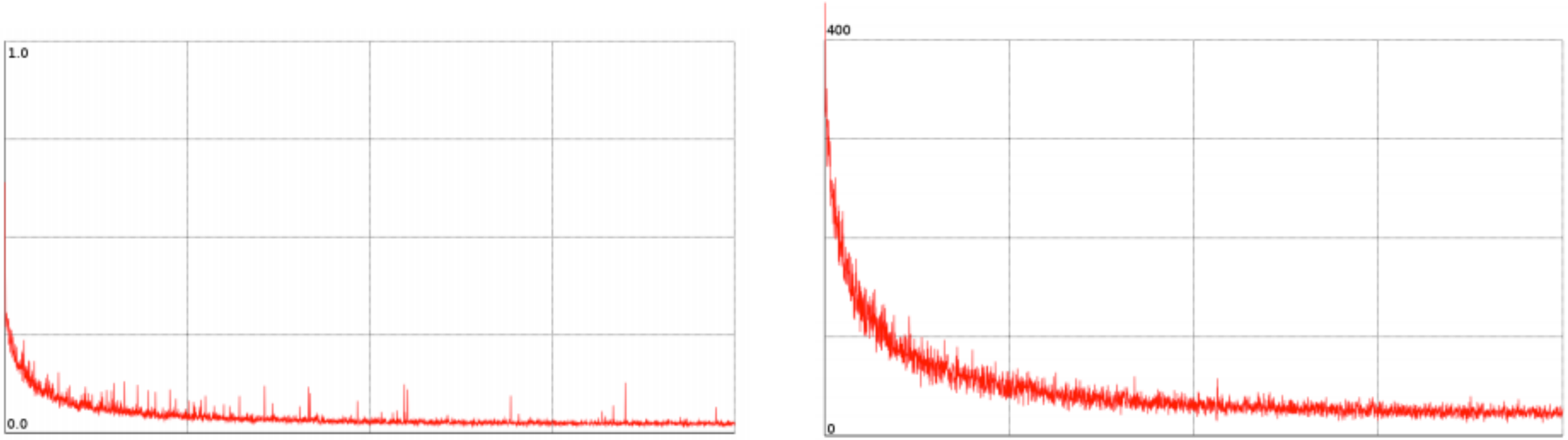


Figure 6. Test error (left) and number of nodes in the tree (right) as a function of iterations. The tree was built using 1000 samples, and then, keeping the tree fixed the representation was updated over another 1000 gradient steps, repeating this. Note that as learning progresses the tree needs to keep less and less nodes. By the end of learning only about 25 nodes are needed to achieve less than 2% test error.

Method	Test Error Rate	Number of nodes
Boundary tree (raw pixels)	11.01%	8536
Boundary tree (pre-trained net)	5.5%	2906
1-NN (Raw pixels)	5.0%	60,000
Neural net (directly as classfier)	2.4%	-
Boundary tree (our learned representation)	**1.85%**	**202**

Table 1. Results on MNIST classification. The trees were built over the entire 60,000 sample training set. All results are reported for a single tree (not with forests). Note that with our learned representation only a small handful of samples is needed. The neural net used is a fully connected net as described in the text.

Full results on the test set are presented in Table 1. As can be seen, using the representation yields better results than using raw-pixels (as in (Mathy et al., 2015)) with a single tree. We also find that training a network with a comparable architecture directly on the classification task yields a representation which is less suited for a k-NN method such as the boundary tree (second row in the table). Figure 6 shows the test error and number of nodes in the tree as a function of iterations. As can be clearly seen, as the representation improves, the tree needs to keep less and less nodes — by the end of learning, the tree has just about 25 nodes, still achieving below 2% error on the test set. How can the tree store such a small number of samples and still achieve this level of performance? In order to understand what the representation is doing, we plotted a t-SNE (Van der Maaten and Hinton, 2008) visualisation for the learned 20 dimensional representation, projected down to 2D. Figure 7 shows the results, together with the result of a pre-trained MNIST network of similar structure (trained directly on classification). As can be seen, the representation we learn clearly separates the classes from one another, much more than the neural net directly trained to classify. This enables the simple tree structure which is learned after transformation.

5. Discussion

- We have presented a method to learn useful representations for k-NN methods. Using boundary trees we are able to derive a differentiable cost function which allows for learning of such representations.

- The resulting representation allows for simple, interpretable tree structures with good performance in query time and accuracy.

- While the method has some limitations we feel this is an important research direction and are looking forward to further explore this directions using newly available dynamic batching tools such as TensorFlow Fold.

References

A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *Proceedings of the 23rd international conference on Machine learning*, pages 97–104. ACM, 2006.

L. Breiman. Random forests. *Machine learning*, 45(1): 5–32, 2001.

M. Craven and J. W. Shavlik. Extracting tree-structured representations of trained networks. In *Advances in Neural Information Processing Systems*, pages 24–30, 1996.

J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3 (3):209–226, 1977.

J. Goldberger, G. E. Hinton, S. T. Roweis, and R. Salakhutdinov. Neighbourhood components analysis. In *Advances in neural information processing systems*, pages 513–520, 2004.

E. Hoffer and N. Ailon. Deep metric learning using triplet network. In *Similarity-Based Pattern Recognition*, pages 84–92. Springer, 2015.

M. I. Jordan and R. A. Jacobs. Hierarchical mixtures of experts and the EM algorithm. *Neural computation*, 6(2): 181–214, 1994.

D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

G. Koch, R. Zemel, and R. Salakhutdinov. Siamese neural networks for one-shot image recognition. In *International Conference for Machine Learning*, 2015.

P. Kontschieder, M. Fiterau, A. Criminisi, and S. Rota Bulo. Deep neural decision forests. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1467–1475, 2015.

T. Liu, A. W. Moore, and A. Gray. New algorithms for efficient high-dimensional nonparametric classification. *The Journal of Machine Learning Research*, 7:1135–1158, 2006.

C. Mathy, N. Derbinsky, J. Bento, J. Rosenthal, and J. Yedidia. The boundary forest algorithm for online supervised and unsupervised learning. In *AAAI*, 2015.

M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)*, 2:331–340, 2009.

L. Van der Maaten and G. Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9(2579-2605):85, 2008.

K. Q. Weinberger, J. Blitzer, and L. K. Saul. Distance metric learning for large margin nearest neighbor classification. In *Advances in neural information processing systems*, pages 1473–1480, 2005.