JMLR: Workshop and Conference Proceedings 45:378–390, 2015 ACML 2015

Autoencoder Trees

Ozan İrsoy OIRSOY@CS.CORNELL.EDU
Department of Computer Science
Cornell University
Ithaca, NY 14853

Ethem Alpaydın ALPAYDIN@BOUN.EDU.TR
Department of Computer Engineering
Boğaziçi University
Bebek, İstanbul 34342 Turkey

Editor: Geoffrey Holmes and Tie-Yan Liu

Abstract

We discuss an autoencoder model in which the encoding and decoding functions are implemented by decision trees. We use the soft decision tree where internal nodes realize soft multivariate splits given by a gating function and the overall output is the average of all leaves weighted by the gating values on their path. The encoder tree takes the input and generates a lower dimensional representation in the leaves and the decoder tree takes this and reconstructs the original input. Exploiting the continuity of the trees, autoencoder trees are trained with stochastic gradient-descent. On handwritten digit and news data, we see that the autoencoder trees yield good reconstruction error compared to traditional autoencoder perceptrons. We also see that the autoencoder tree captures hierarchical representations at different granularities of the data on its different levels and the leaves capture the localities in the input space.

Keywords: Autoencoders, decision trees, neural networks, deep learning.

1. Introduction

To find the hidden structure in data, one unsupervised learning method is the autoencoder which is composed of an encoder and a decoder put back to back. The encoder maps the original input to a generally lower dimensional or sparse hidden representation, and the decoder takes it and reconstructs the original input. The idea is that if the decoder can reconstruct the original input faithfully, the hidden representation should be a meaningful and useful one. Conventional autoencoders use use a single layer, perceptron-type neural network for the encoding and decoding functions, which implements an affine map followed by a nonlinearity for the encoder (Cottrell et al., 1987). In deep learning, the idea is to stack multiple such autoencoders as a multilayer perceptron thereby learning more abstract hidden representations in higher layers (Bengio, 2009).

In this work, we explore an autoencoder model where decision trees are used for the encoding and decoding functions, instead of a single- or multi-layer perceptrons. We use the soft decision tree model where the internal decision nodes use a soft multivariate split defined by a gating function and the overall output is the average of all leaves weighted by the gating values on their paths (section 2). Since the output of such a soft tree is continuous, we

can use stochastic gradient-descent to update the parameters of encoder and decoder trees simultaneously to minimize reconstruction error, as in conventional autoencoder approaches. With the chain rule, the error terms of the hidden representation of the decoding layer are passed back to the encoding layer as its external error (section 3). We will show in our experimental results (section 4) that such autoencoder trees can learn as well as autoencoder perceptrons while learning in an unsupervised manner a hierarchical decomposition of the data into subspaces which respect localities in the data.

2. Soft Decision Tree

A decision tree is a hierarchical structure with internal decision nodes and terminal leaves. An internal decision node redirects the given instance to one of its children. In supervised learning, leaves include a prediction label, such as a class label for classification or a numeric response for regression.

As opposed to the hard decision node which implements a hard split, a soft decision node redirects instances to all its children but with different probabilities, as given by a *gating function* $g_m(\boldsymbol{x})$—the hard decision tree is a special case where $g(\boldsymbol{x}) \in \{0, 1\}$. This architecture is equivalent to that of the hierarchical mixture of experts (Jordan and Jacobs, 1994). Let us consider a *soft binary tree* where each internal node has two children, named left and right. The response y_m at a node m is recursively calculated as the weighted average of the responses of its left child ml and right child mr:

$$y_m(\boldsymbol{x}) = \begin{cases} \rho_m & \text{if } m \text{ is leaf} \\ g_m(\boldsymbol{x}) y_{ml}(\boldsymbol{x}) + (1 - g_m(\boldsymbol{x})) y_{mr}(\boldsymbol{x}) & \text{otherwise} \end{cases} \tag{1}$$

To choose among the two outcomes, we define the gating function, $g_m(\boldsymbol{x}) \in [0, 1]$, as the *sigmoid function* over a linear split:

$$g_m(\boldsymbol{x}) = \frac{1}{1 + \exp[-(\boldsymbol{w}_m^T \boldsymbol{x})]} \tag{2}$$

Note that our decision nodes are *multivariate*, that is, they use all of the input features, as opposed to the univariate trees that use a single feature in each split. Geometrically speaking, though univariate splits are orthogonal to one of the axes, multivariate splits are *oblique* and can take any orientation, which makes them more generally applicable.

Separating the regions of responsibility of the left and right children can be seen as two-class classification problem and from that perspective, the gating model implements a discriminative (logistic linear) model estimating the posterior probability of the left child: $P(\text{left}|\boldsymbol{x}) \equiv g_m(\boldsymbol{x})$ and $P(\text{right}|\boldsymbol{x}) \equiv 1 - g_m(\boldsymbol{x})$.

In the case of supervised learning, ρ stored at a leaf corresponds to the predicted value. In regression, ρ is a scalar. In classification, we can apply a sigmoid nonlinearity at the root node to convert the output into a probability value. Note that $\boldsymbol{\rho}$ can be a vector response as well: For example with $K \geq 2$ classes, $\boldsymbol{\rho}$ is a K-dimensional vector and softmax nonlinearity is used at the root node to convert the K outputs to posterior probabilities. In our case of unsupervised learning, the dimensionality of $\boldsymbol{\rho}$ will be set to the dimensionality of the hidden representation we want to learn.

Note that the soft decision tree defines a continuous response function of the parameter space, conditioned on the structure of the tree. This means that given a tree structure, the parameters, $\{\boldsymbol{\rho}_m, \boldsymbol{w}_m\}_m$(response values at the leaves and splitting hyperplanes of the internal nodes) can be learned by minimizing an objective function over the tree response with a continuous optimization method, e.g., stochastic gradient-descent. For supervised tasks such as classification or regression, conventional objective functions such as cross-entropy or squared error can be employed.

In order to learn the parameters $\boldsymbol{\rho}$ and $\boldsymbol{w}$ with a gradient-based method, we can use backpropagation to efficiently compute the gradients. Let E denote the overall error to be minimized and E^i denote the error over a single data instance $\boldsymbol{x}^i$. Let us define $\boldsymbol{\delta}_m = \partial E^i / \partial \boldsymbol{y}_m(\boldsymbol{x}^i)$, which is the *responsibility* of the node m. Backpropagating the error from the root towards the leaves, we have

$$\frac{\partial E^i}{\partial \boldsymbol{w}_m} = g_m(\boldsymbol{x}^i)(1 - g_m(\boldsymbol{x}^i))\big(\boldsymbol{\delta}_m^{iT}(\boldsymbol{y}_{ml}(\boldsymbol{x}^i) - \boldsymbol{y}_{mr}(\boldsymbol{x}^i))\big)\boldsymbol{x}^i \tag{3}$$

$$\frac{\partial E^i}{\partial \boldsymbol{\rho}_m} = \boldsymbol{\delta}_m^i \tag{4}$$

with

$$\boldsymbol{\delta}_m^i = \begin{cases} \boldsymbol{y}_r(\boldsymbol{x}) - \boldsymbol{r}^i & \text{if } m \text{ is root} \\ \boldsymbol{\delta}_{pa(m)}^i g_m(\boldsymbol{x}^i) & \text{if } m \text{ is a left child} \\ \boldsymbol{\delta}_{pa(m)}^i (1 - g_m(\boldsymbol{x}^i)) & \text{if } m \text{ is a right child} \end{cases} \tag{5}$$

where $pa(m)$ is the parent of node m.

3. Autoencoder Trees

The observation that the soft decision tree output is a a continuous function of the parameter space (for a given tree structure) leads to the following conclusion: A soft decision tree can be trained not just with supervised error signal but also with an unsupervised signal, as well as an error term backpropagated from a further layer of information processing that uses the tree output as an input. This follows from a simple application of the chain rule when computing the gradients.

In this work, based on this observation, we define an autoencoder approach by stacking two soft trees back to back where the encoding and decoding functions are implemented by two soft decision trees.

Let $\boldsymbol{t}(\boldsymbol{x})$ denote a soft decision tree as defined in Equations 1 and 2. Let us define an autoencoder tree pair $\boldsymbol{t}_e(\boldsymbol{x})$ and $\boldsymbol{t}_d(\boldsymbol{x})$ with the following interpretation: The encoder tree encodes the d dimensional input $\boldsymbol{x}$ into a k dimensional intermediate (or hidden) representation $\boldsymbol{h} = \boldsymbol{t}_e(\boldsymbol{x})$ (where $k < d$), and the decoder tree decodes the initial input from the hidden representation, $\hat{\boldsymbol{x}} = \boldsymbol{t}_d(\boldsymbol{h})$.

We see in Figure 1 an example on MNIST dataset in which an autoencoder tree is used to reduce the dimensionality to two. The first encoder tree takes the 784-dimensional digit image as its input $\boldsymbol{x}$; it then uses Equation 1 recursively and gets to all of the leaves—all the gating weight vectors of this encoder tree has 784+1 dimensions. Because we want

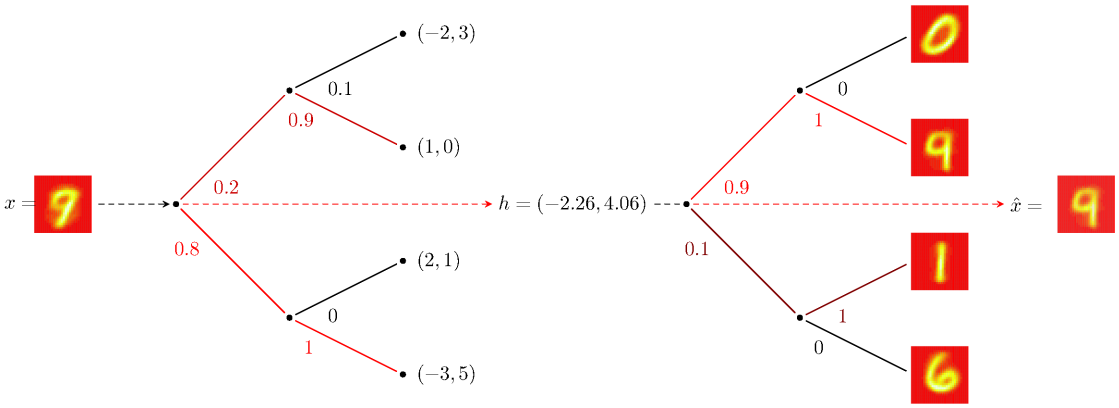

Figure 1: Operation of an autoencoder tree. Given a 784d input image x, the encoder tree assigns it a 2d hidden representation $h = (-2.26, 4.06)$ by making a soft selection among its leaf responses (left). Similarly, decoder tree reconstructs the input when given this h (right).

to decrease dimensionality to two, all the leaves store a two-dimensional value. We then calculate the average of all leaves weighted by the gating values of all the leaves and this gives us the two-dimensional final representation for the digit. We denote this output by $\boldsymbol{t}_e(\boldsymbol{x})$.

The second, decoder tree takes this two-dimensions as its input and again using Equation 1 recursively (but this time with 2+1 dimensional gating weights) calculates its leaves; because we want to reconstruct the original input, the leaves of the decoder tree are 784 dimensional. Again by calculating a weighted sum of all the leaves we get the final reconstructed image. We denote this by $\boldsymbol{t}_d(\boldsymbol{t}_e(\boldsymbol{x}))$.

We want the decoded response to be as close as possible to the initial input ($\boldsymbol{x} \approx \hat{\boldsymbol{x}}$) which can be implemented by minimizing the reconstruction error on a training set $\{\boldsymbol{x}^i\}_{i=1}^N$:

$$E = \frac{1}{2}\sum_i \|\boldsymbol{x}^i - \hat{\boldsymbol{x}}^i\|^2 = \frac{1}{2}\sum_i \|\boldsymbol{x}^i - \boldsymbol{t}_d(\boldsymbol{t}_e(\boldsymbol{x}^i))\|^2 \tag{6}$$

Let $\theta_e \in \{\boldsymbol{\rho}_m, \boldsymbol{w}_m\}_{m\in e}$ and $\theta_d \in \{\boldsymbol{\rho}_m, \boldsymbol{w}_m\}_{m\in d}$ be a parameter of the encoder and decoder trees respectively. Then, we can update parameters in both trees by gradient descent:

$$\frac{\partial E^i}{\partial \theta_d} = \frac{\partial E^i}{\partial \boldsymbol{t}_d(\boldsymbol{h}^i)}\frac{\partial \boldsymbol{t}_d(\boldsymbol{h}^i)}{\partial \theta_d} = (\boldsymbol{x}^i - \hat{\boldsymbol{x}}^i)^T\frac{\partial \boldsymbol{t}_d(\boldsymbol{h}^i)}{\partial \theta_d} \tag{7}$$

$$\frac{\partial E^i}{\partial \theta_e} = \frac{\partial E^i}{\partial \boldsymbol{h}^i}\frac{\partial \boldsymbol{h}^i}{\partial \theta_e} = (\boldsymbol{x}^i - \hat{\boldsymbol{x}}^i)^T\frac{\partial \boldsymbol{t}_e(\boldsymbol{h}^i)}{\partial \boldsymbol{h}^i}\frac{\partial \boldsymbol{t}_e(\boldsymbol{x}^i)}{\partial \theta_e} \tag{8}$$

where $\frac{\partial \boldsymbol{t}_\cdot(\boldsymbol{x})}{\partial \theta_\cdot}$ can be computed as before. Additionally, the computation of $\boldsymbol{\delta h} = \frac{\partial E}{\partial \boldsymbol{h}}$ requires the derivative of a tree response with respect to its input (for the decoder tree):

$$\frac{\partial \boldsymbol{t}(\boldsymbol{x}^i)}{\partial \boldsymbol{x}^i} = \sum_m g_m(\boldsymbol{x}^i)(1 - g_m(\boldsymbol{x}^i))\big(\delta_m^{iT}(y_{ml}(\boldsymbol{x}^i) - y_{mr}(\boldsymbol{x}^i))\big)w \tag{9}$$

That is, δh is the error responsibility of the hidden representation backpropagated from the decoder tree (top layer) to the encoder tree (bottom tree).

Again on Figure 1, we can see how back propagation is done: Because we know the input, we know the desired output for the decoder tree, and we can calculate the mean square error (reconstruction error here). Then we use backpropagation, that is, the chain rule, to update the parameters of the second decoder tree (Equation 7) and then back propagating even further, to update the parameters of the first encoder tree (Equation 8).

When the encoder and decoder trees have multiple levels, backpropagating may be too slow and we use a layer-by-layer training, as in conventional autoencoder training. For both the encoder and the decoder trees, we start with a tree of depth two. After iterating for some number of epochs, we split every leaf into a tree and hence get trees of depth three, and we continue doing so until we get to the final required depth. During those depth increments, all tree parameters are updated and not just the most recently introduced ones, which allows for finetuning. When splitting, the new children inherit the response $\boldsymbol{\rho}$ values of their parents with an additive small random noise.

For a single instance $\boldsymbol{x}$, the main bottleneck in the perceptron autoencoder is the two matrix-vector products (encoder and decoder), which yield $O(d \cdot k)$ time where d and k denote respectively the number of dimensions of the input and the encoding. The memory storage consists of those matrices, which yield the same complexity for space.

For encoder trees, gating computation is $O(d)$ per node, which yield $O(n \cdot d)$ time complexity where n denotes the number of nonterminal nodes in the tree. Combining the results of two children is simply $O(k)$ per node, which result in $O(n \cdot k)$ complexity. The two parts give a total of $O(n(d + k))$ which can be considered as $O(n \cdot d)$ since $d > k$ will hold in the case of dimensionality reduction. A similar analysis will yield the same time complexity for the decoder tree (by swapping d and k). Storage will also be similar since in the encoder (decoder), we will have vectors of size d (k) in each internal node and vectors of size k (d) in each leaf node.

Therefore, if the number of nodes in the autoencoder tree and the number of hidden units in the perceptron autoencoder are comparable, they have similar time and space complexities. Note that by batching, the overall time complexity might be improved, which is also true for the autoencoder trees.

4. Experiments

4.1. Experimental Setting

Data. We evaluate our models on two data sets: MNIST handwritten digit database (LeCun and Cortes, 1998) and the 20 Newsgroups data set (twn). MNIST contains 60,000 training and 10,000 test examples of handwritten digit images which are 28 by 28 pixels (784 dimensional). Output labels are the ten digits. 20 Newsgroups data (20News) contains

18,846 instances of newsgroup documents (partitioned into training and test sets with 60%-40% ratio), with output labels denoting the subject matter (category) out of 20 classes. With the bag-of-words representation, the dimensionality is about 60,000, but we sort the words in terms of their frequencies, discard the top 100 (non informative stop words) and use the next 2,000.

Baselines. We use two autoencoder perceptrons: One has a single layer, that is, a linear map and nonlinearity for the encoder, and linear map for the decoder. The second uses the stacked two-layer perceptron autoencoder where we first reduce the dimensionality to 50 using the conventional autoencoder, and using the 50 dimensional representation, we once again reduce to the final dimensionality. In both cases, nonlinearity is the hyperbolic tangent.

Tree and network training. Both autoencoder perceptrons and autoencoder trees are trained with stochastic gradient-descent, in the online setting (i.e. minibatch size is 1). For both, we employ a diagonal variant of AdaGrad (Duchi et al., 2011), which yields smooth and fast convergence. We train for a total of 240 epochs. Autoencoder trees (both the encoder and the decoder trees) start from a depth of two, and the depth is incremented at every 40th epoch, until they reach their final depth (five or six). We employ a simple L2 regularization on connection weights for autoencoder perceptrons and hyperplane split parameters ($\boldsymbol{w}_m$) and the leaf responses ($\boldsymbol{\rho}_m$) for autoencoder trees.

4.2. Results

We report the reconstruction errors per each epoch of stochastic gradient-descent on the two datasets in Figure 2. For MNIST, we report the error in the scale of a single pixel; for 20News, error is in the scale of a single word in the bag-of-words representation, relative to the maximum number of occurrence of each word (All inputs are normalized between 0 and 1).

We see that on MNIST, autoencoder trees can attain a better reconstruction error than perceptron when reducing to two dimensions, however autoencoder perceptron is better than trees when reducing to ten dimensions. For the autoencoder trees, we observe that the dimensionality of the hidden representation does not have a strong effect on the performance, as seen by very close reconstruction error rates. On the other hand, we observe improved performance as we increase the depth of trees. This suggests that the topology itself might be more important than the dimensionality of the hidden representation for autoencoder trees.

For 20News, convergence is less smooth for all architectures. For both reduction to two and ten dimensions, autoencoder trees yield a better reconstruction error than autoencoder perceptrons. There is a gain by reducing to ten instead of two for autoencoder trees. However, again, this gain is relatively small compared to the gain resulting from an extra level of depth in the tree, as seen by the difference between five- and six-deep trees (which is especially large in the case of reducing to ten dimensions).

For mapping of MNIST digits to two dimensions, we show resulting representations in Figure 3. For the autoencoder perceptron, we observe a strong tendency to saturate the non-linearity and not utilize the softness of the sigmoid threshold and hence we see all instances

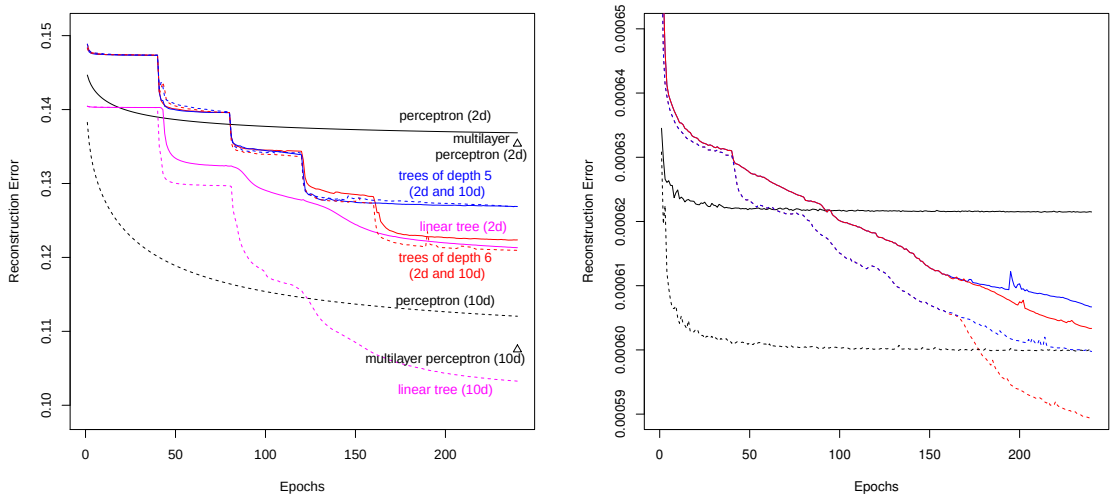

Figure 2: Reconstruction errors of different autoencoder architectures on MNIST (left) and 20News (right) datasets. Black, blue and red denote the autoencoder perceptron, the autoencoder tree with a depth of five, and the autoencoder tree with a depth of six, respectively. Purple denote autoencoder tree with linear map at the leaves (section 4.3). Solid and dash denote dimensionality reduction to two and ten, respectively. Points denoted by triangle show multilayer perceptron which first reduces the dimensionality to 50 and then reduces once more to 2 or 10.

mapped to the four corners. For autoencoder trees, most instances converge on a single leaf response but we also observe convex combinations of multiple leaves. Note that the hidden representations are more local rather than distributed for trees: Rather than assigning a global meaning to different directions in the hidden space, the hidden representation assigns regions (of different sizes in different levels, in a multi-resolution or multi-granular fashion) of the space to different digits, and closeness becomes more important. This behavior is similar to clustering. Indeed, autoencoder trees can be considered to do a dimensionality reduction alongside hierarchical soft clustering. Aforementioned relatively small gains by increasing the dimensionality of the hidden layer is another evidence to this local behavior, since distributed representations gain much more from a higher number of dimensions.

In Figure 4, an encoder tree with a depth of six is shown. Histograms at each node shows the class distributions. Since we use a soft decision tree, every instance has a *soft membership* at a given node, computed by the sigmoidal gating function, which is used as the soft count when counting the instances which belong to a node. Although training is unsupervised, we see that some leaf nodes capture single classes, such as the sixth leaf including mostly only the digit '0' (light blue) or the eight leaf including only the digit '2' (light green). Others capture two, or more classes but learn a locality and the combination of gating functions from the root to that leaf defines the regions of that locality.

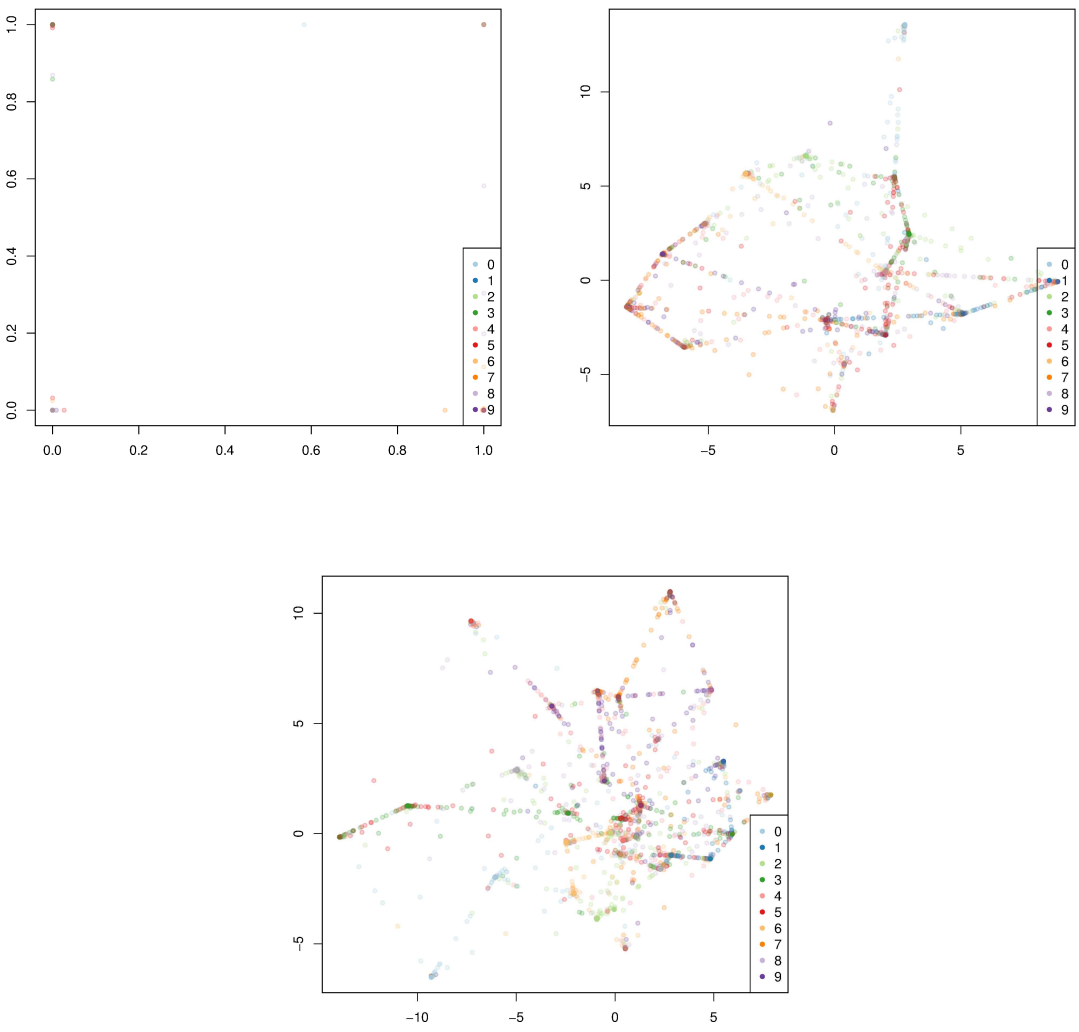

Figure 3: Dimensionality reduced representations of digits for MNIST for autoencoder perceptron (top left), autoencoder tree of depth five (top right) and depth six (bottom).

In Figure 5, we show a decoder tree learned over the MNIST dataset. Nodes show their response values with internal nodes depicting the latest value before they are split and another level is added. We observe hierarchies captured by the decoder tree: To the left, nodes disentangle different variations of digits '9' and '7' with different slopes. At certain nodes, digits '0' and '6' are represented together, then separated at children. A similar phenomenon occurs with digits '3' and '8', as well as '2' and '8'.

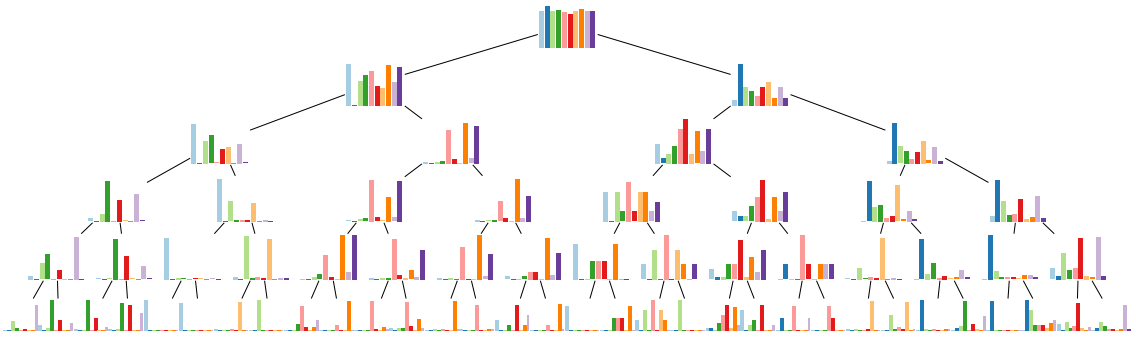

Figure 4: Class distributions for the encoder tree on MNIST. Different colors represent the ten classes.

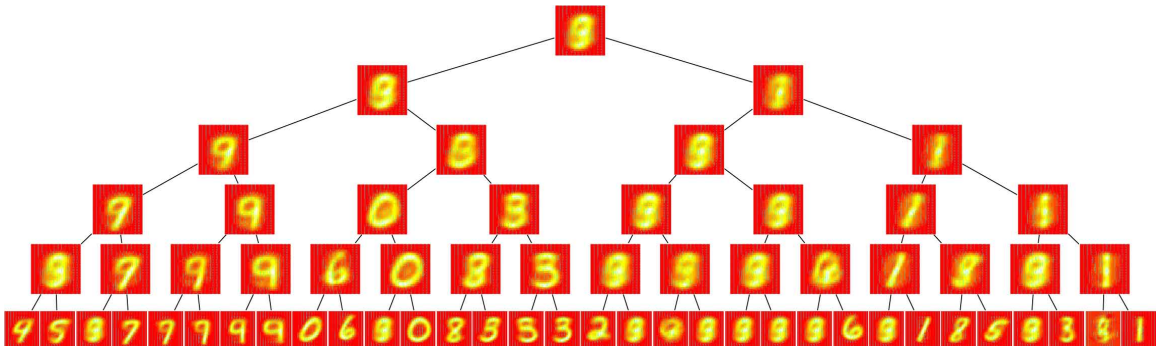

Figure 5: Response values for the decoder tree on MNIST. Internal nodes show the latest response values before splitting them.

We show some examples of original and reconstructed digit images in Figure 6, for autoencoder trees with depth six and hidden dimensionality of two and ten. We observe that most reconstructions are faithful, and we see that some of the errors done by 2d reducing tree are corrected by the 10d reducing tree.

Similarly in Figure 7, we show part of the decoder tree over 20News dataset. Since the response vector is a bag-of-words representations, we sort the words by their coefficients and show only the top words. We show some of the paths and omit others to avoid clutter. Again, we see hierarchies captured by the tree as seen by word distributions which resemble topics at finer and finer grain as we split the nodes further. This behavior is similar to a hierarchical topic model, since every document can be mapped to a distribution on tree leaves (with the gating function) and every leaf can be mapped to a distribution on words (by normalizing leaf responses). For instance, leftmost leaf roughly encapsulates concepts about internet anonymity, next leaf is about programs and output methods (file, printf, etc.), and the last two are about internet publishing of images. As we go down from the top to bottom, word activations become more and more specific.

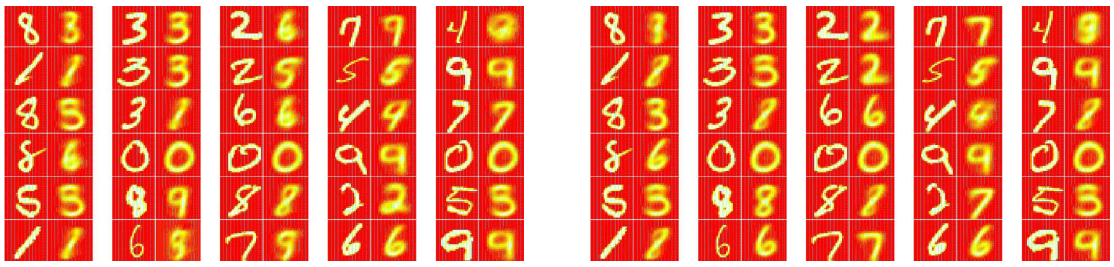

Figure 6: A sample of original (left columns) and reconstructed (right columns) images using autoencoder trees. Selection is random. Left: Reconstructed from 2d reduction. Right: Reconstructed from 10d reduction.

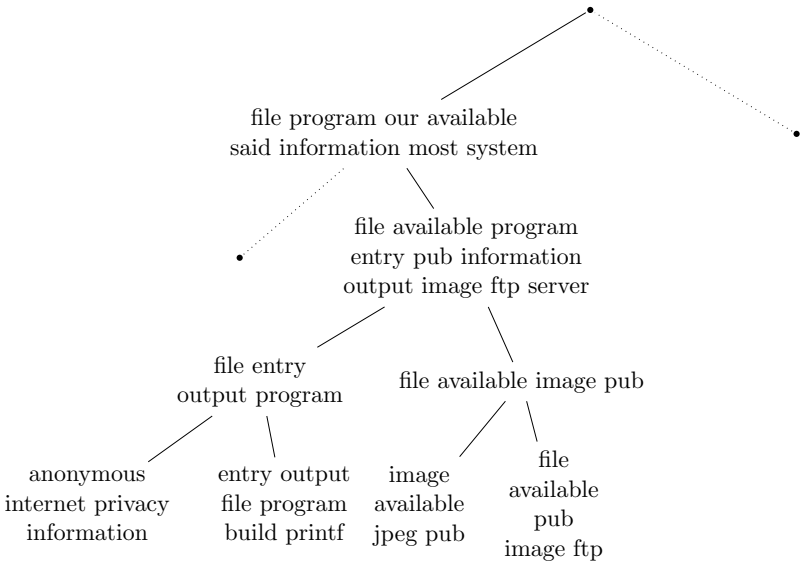

Figure 7: Response values for a subtree of the decoder tree on 20News. Internal nodes show latest response values before splitting them.

4.3. Extension to Model Trees

A simple extension to the aforementioned autoencoder tree model can be done by more complex leaf models, such as a linear map over the inputs, instead of a constant vector valued response. This can be done by modifying Equation 1 so that in a leaf node m, we define $\boldsymbol{\rho}_m = \boldsymbol{V}_m \boldsymbol{x}$ and gradient-descent rules are modified accordingly to update $\boldsymbol{V}_m$. Hence, the value stored in a leaf is no longer constant but parameters of a linear model and the response in a leaf varies linearly based on the input.

This results in an autoencoder tree model in which leaf nodes make local linear projections in the input space. This also provides some degree of distributed representational

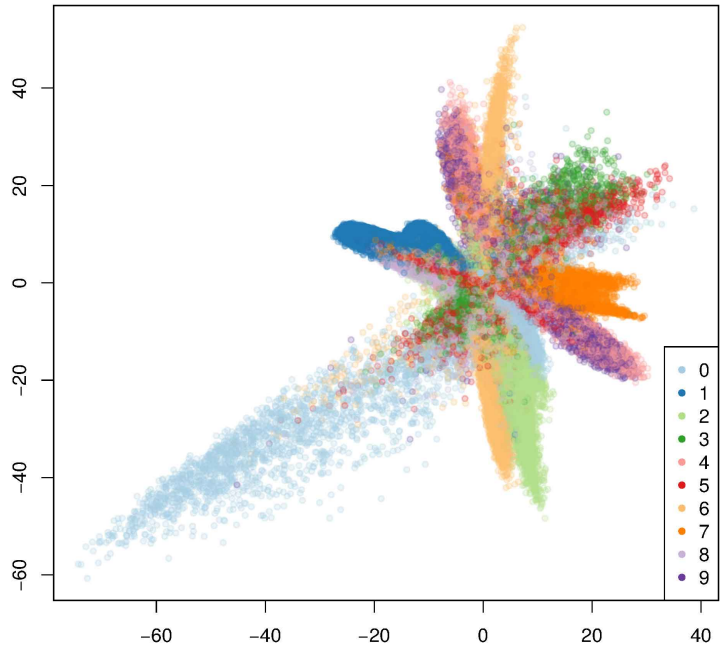

Figure 8: Dimensionality reduced representations of digits for MNIST for autoencoder linear model tree.

power to the autoencoder tree, corresponding to locally partitioning the space and assigning a distributed model to every (soft) partition.

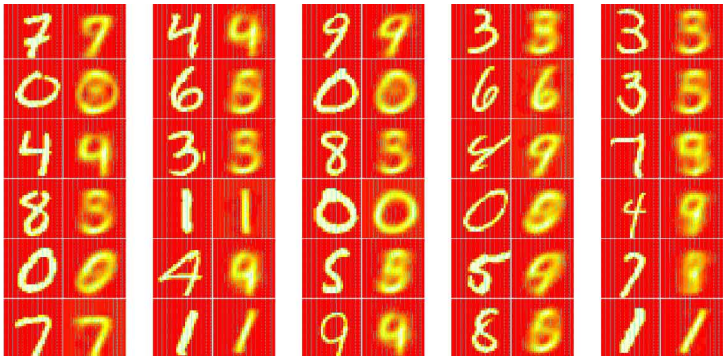

Figure 9: A sample of original (left columns) and reconstructed (right columns) images. Selection is random. Reconstructed from 2d reduction with autoencoder linear model trees.

We experiment with this extension on the MNIST dataset. Figure 8 shows the resulting two-dimensional representations for different digits where we see that classes are very well-separated even in two dimensions, indicating that the model has effectively captured the underlying distribution. In contrast to Figure 3, autoencoder linear model trees provide a much smoother distribution in the hidden representation space, and move away from the clustering-like behavior to a degree. This validates our intuition that linear models would help incorporate a distributed representation in addition to locality. We see in Figure 2 (left) that with such trees we can get smaller reconstuction error on MNIST data, as also observed on a sample of reconstructed images in Figure 9.

5. Conclusions and Discussion

We discuss an autoencoder model with soft decision trees as encoder and decoder. We apply our model in a dimensionality reduction setting. The model is shown to have comparable or better reconstructive power than autoencoder perceptrons, when reducing to a small number of dimensions. Autoencoder trees provide a hierarchical decomposition in the input space and the space of the hidden representation, by making both the encoding and decoding hierarchical.

Autoencoder trees can be conceptualized as doing a soft hierarchical clustering on the data, and doing a dimensionality reduction within the clusters. As opposed to applying a hierarchical clustering algorithm and then reconstructing each cluster by its centroid, autoencoder trees provide dimensionality reduction coupled with a clustering-like behavior. Furthermore, decoding process is also hierarchical in autoencoder trees. The use of a soft convex combination of leaves implies that an instance can be efficiently modeled as a combination of multiple leaf nodes, allowing to model things such as a background factor that is added to many instances.

The hidden representation learned by an autoencoder tree can be fed to a supervised learner for classification or regression. We see that though the training is unsupervised, the autoencoder tree finds internal nodes or leaves that become increasingly responsive to single or few classes. The combination of gating values from the root to a node defines the (soft) boundaries of a locality and hidden representations learned at different levels can be considered as representations at different resolutions or granularities. Feeding these representations at multiple resolutions may improve prediction performance in a supervised setting—this will be an interesting future research direction.

Acknowledgments

This work is partially supported by Boğaziçi University Research Funds with Grant Number 14A01P4.

References

20Newsgroups. http://qwone.com/~jason/20Newsgroups/.

Yoshua Bengio. Learning deep architectures for AI. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

Garrison W Cottrell, Paul Munro, and David Zipser. Learning internal representations from gray-scale images: An example of extensional programming. In *Ninth annual conference of the cognitive science society*, pages 462–473, 1987.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12: 2121–2159, 2011.

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

Yann LeCun and Corinna Cortes. The MNIST database of handwritten digits, 1998.