Binary Interpolation Tree (B.I.T)

Binary Interpolation Tree is a machine learning model i have been working on mostly out of curiosity. Please let me know if my terminology is off, i made any other errors, or something very similar already exists.

The main idea was to conceptionalize a model that provides:

- fast runtime on cpu's in relation to model complexity (O(n), Ω(1), input dependent)
- the ability to further simplify computations based on a maximum inaccuracy
- simple and fast training
- very low initial model complexity that increases during training
- multithreading capabilities for maximum cpu usage
- increased transparency compared to other ML models
- a parameter regarding determinism of the algorithm

1. The Model

1.1 The basic model

As the name suggests, this model is basically a binary tree. This binary tree consists of nodes and end-nodes. Nodes always have 2 distinct children except for end-nodes that have non. Every node has a reference to its parent, except for the root-node, which has no parent. Every node has a position vector and every end-node additionally contains an output vector. For the purpose of describing this system mathematically, every node has an index i and the following variables associated with it:

The index of the first child: $c_1(i)$.

The index of the second child: $c_2(i)$.

The index of the parent: $parent(i)$.

The position vector: $\overrightarrow{p}(i) \in V_i$.

The result of the calculation for nodes, or the output vector for end-nodes: $\overrightarrow{o}(i) \in V_o$.

For the root-node the index $i = 0$.

An index of -1 means there is no reference. For example $parent(0) := -1$.

To calculate the output $\overrightarrow{o}(0)$ of this model for a given input vector $\overrightarrow{x} \in V_i$, this binary tree is traversed recursively in my cpu implementation, from the root-node, to the end-nodes. The following calculations are performed for every node:

$$I(\overrightarrow{p_1}, \overrightarrow{p_2}, \overrightarrow{x}) := \frac{(\overrightarrow{p_2} - \overrightarrow{p_1}) \circ (\overrightarrow{x} - \overrightarrow{p_1})}{\left|\overrightarrow{p_2} - \overrightarrow{p_1}\right|^2} \qquad clamp(x) := \min(1, \max(0,x))$$

$a(c_1(i)) := clamp(I(\overrightarrow{p}(c_1(i)), \overrightarrow{p}(c_2(i)), \overrightarrow{x}))$ $a(c_2(i)) := 1 - a(c_1(i))$
$\overrightarrow{o}(i) := a(c_1(i)) \cdot \overrightarrow{o}(c_1(i)) + a(c_2(i)) \cdot \overrightarrow{o}(c_2(i))$
if $a(c_1(i)) = 0$ or $a(c_2(i)) = 0$, $\overrightarrow{o}(c_1(i))$ or $\overrightarrow{o}(c_2(i))$ doesn't have to be calculated, which can cause a great performance increase for sequential execution. the following statements are true and fully define $I(\overrightarrow{p_1}, \overrightarrow{p_2}, \overrightarrow{x})$ in an intuitive way:

$I(\overrightarrow{p_1}, \overrightarrow{p_2}, a * \overrightarrow{p_1} + (1 - a) * \overrightarrow{p_2}) = 1 - a \Rightarrow I(\overrightarrow{p_1}, \overrightarrow{p_2}, \overrightarrow{p_1}) = 0 \wedge I(\overrightarrow{p_1}, \overrightarrow{p_2}, \overrightarrow{p_2}) = 1$
$I(\overrightarrow{p_1}, \overrightarrow{p_2}, \overrightarrow{x_1}) = I(\overrightarrow{p_1}, \overrightarrow{p_2}, \overrightarrow{x_2}) \Leftrightarrow (\overrightarrow{p_2} - \overrightarrow{p_1}) \circ (\overrightarrow{x_1} - \overrightarrow{x_2}) = 0.$

1.2 Introducing indeterministic behavior

Swapping the definition of $a(c_1(i))$ from $a(c_1(i)) := clamp(I(\overrightarrow{p}(c_1(i)), \overrightarrow{p}(c_2(i)), \overrightarrow{x}))$ to
$a(c_1(i)) := clamp(I(\overrightarrow{p}(c_1(i)), \overrightarrow{p}(c_2(i)), \overrightarrow{x}) * d + (1 - d) * r)$
where $0 \leq d \leq 1 \wedge 0 \leq r \leq 1$, will allow indeterministic behavior, if a random number is assigned to r for each computation of $a(c_1(i))$. The parameter d is called determinism in my implementation, because $d = 1$ means that the model is 100% deterministic while $d = 0$ means the interpolation factor $a(c_1(i))$ is completely random between 0 and 1.

1.3 Simplifying computations by defining a maximum inaccuracy

For this purpose, the following variables are defined:
The number of dimensions of the input space: $m := \dim(V_i)$.
The number of dimensions of the output space: $n := \dim(V_o)$.
The Inaccuracy-scale vector $\vec{s} \in V_o$ allows different maximum inaccuracies for all different output dimensions and is meant to be immutable for every model. The maximum inaccuracy t is meant to be passed for every Forward pass, to be able to calculate the output for the same model with varying speed and accuracy. This step profits from normalizing the Data beforehand. The following variables are defined for each node:

The complexity: $complexity(i) := 1 + complexity(c_1(i)) + complexity(c_2(i))$ for all nodes and $complexity(i) := 1$ for all end-nodes specifically.
$complexity(i)$ is roughly proportional to the runtime of the forward pass off the node with the index i.

The contribution to the final result: $c(i) := \begin{cases} c(parent(i)) * a(i) & i \neq 0 \\ 1 & i = 0 \end{cases}$

The maximum contribution value $v(i) := \max_{1 \leq j \leq n} \left(\frac{\overrightarrow{o}(i)_j}{\vec{s}_j} \right)$ for all end-nodes and

$v(i) := \max(v(c_1(i)), v(c_2(i)))$ for every node that is no end-node. $v(i)$ can be cached for every node, because $\vec{s}$ is meant to be immutable for every model.

$$\sum_{k \in M} v(k) \cdot c(k) \leq t \Rightarrow \forall 1 \leq j \leq n : \sum_{k \in M} \overrightarrow{o}(k)_j * c(k) \leq \vec{s}_j * t$$

This Expression means, that you can set $a(k) := 0$ for $k \in M$ and can guarantee that the inaccuracy of $\overrightarrow{o}(0)$, caused by this simplification, will be less than $\vec{s} * t$ for each output dimension, which is a way of trading accuracy for speed. In my c# implementation, i wrote the recursive forward pass as an iterator method, to collect all nodes where $v(i) \cdot c(i) \leq t$ (potential minors) and choose as much nodes as possible,

that have the the lowest $\frac{v(i) \cdot c(i)}{complexity(i)}$ ratio from those potential minors to be in M, so that $\sum_{k \in M} v(k) \cdot c(k) \leq t$ is still satisfied. All nodes in M are called minors and get selected by

$\frac{v(i) \cdot c(i)}{complexity(i)}$ because of the following statements:

$v(i_1) \cdot c(i_1) < v(i_2) \cdot c(i_2) \wedge complexity(i_1) = complexity(i_2)$ means that it is beneficial to choose the node with the index i_1 as a minor, because this introduces less in inaccuracy, but roughly the same runtime.

$v(i_1) \cdot c(i_1) = v(i_2) \cdot c(i_2) \wedge complexity(i_1) < complexity(i_2)$ means that it is beneficial to choose the node with the index i_1 as a minor, because this introduces the same inaccuracy, but it tends run the forward pass faster.

$v(i_1) \cdot c(i_1) + v(i_2) \cdot c(i_2) = v(i_3) \cdot c(i_3) \wedge complexity(i_1) + complexity(i_2) = complexity(i_3)$ means that choosing the nodes with the index i_1 and i_2 as a minor, will have almost the same impact as choosing only the node with the index i_3, because this introduces the same inaccuracy, and roughly the same runtime.

2. Supervised training of the model

2.1 Training the parameters

For each input vector $\overrightarrow{x}(j)$ where $1 \leq j \leq b$ and b is the size of the batch, the result of the forward pass $\overrightarrow{o_j}(0)$ is calculated. $\overrightarrow{y}(j)$ represents the target $\overrightarrow{o_j}(0)$ is meant to approximate for the given input vector $\overrightarrow{x}(j)$. For every node that is no end-node, the index j is necessary for $\overrightarrow{o_j}(i)$, because $\overrightarrow{o_j}(i)$ depends on $\overrightarrow{x}(j)$, but for every end-node, $\overrightarrow{o}(i)$ remains constant within one training iteration and is therefore redundant. The function $error(\overrightarrow{w}_1, \overrightarrow{w}_2)$ represents the error metric and $c_j(i)$ is the contribution value for each node with the index i and input vector $\overrightarrow{x}(j)$. The set $S(i)$ contains all indices j for each node with the index i, where $c_j(i) \neq 0$. After each training iteration, $\alpha \cdot \Delta\overrightarrow{p}(i)$ is added to $\overrightarrow{p}(i)$, where α is the learning rate of this training process. Intuitively speaking, the position vector $\overrightarrow{p}(i)$ is adjusted towards those $\overrightarrow{x}(j)$ with $j \in S(i)$, the node is most „responsible" for and where the biggest error occurs. If $|S(i)| = 0$, all parameters that belong to the node with the index i remains unchanged.

$$\Delta\overrightarrow{p}(i) := \frac{1}{\left|S(i)\right|} \sum_{j \in S(i)} (\overrightarrow{x}(j) - \overrightarrow{p}(i)) * c_j(i) * \left(1 - \exp\left(-\frac{error(\overrightarrow{o_j}(0), \overrightarrow{y}(j))}{f(i)}\right)\right)$$

$$f(i) := \frac{1}{\left|S(i)\right|} \sum_{j \in S(i)} error(\overrightarrow{o_j}(0), \overrightarrow{y}(j))$$

When the position vector of all nodes was adjusted, the only parameters left unchanged are those within the output vector of each end-node. To adjust those, $\alpha \cdot \Delta\overrightarrow{o}(i)$ is added to $\overrightarrow{o}(i)$.

$$\Delta\overrightarrow{o}(i) := \frac{1}{\left|S(i)\right|} \sum_{j \in S(i)} (\overrightarrow{y}(j) - \overrightarrow{o_j}(0)) * c_j(i)$$

The preceding formulas also show that $\overrightarrow{o_j}(0)$ can be modified with functions like softmax for example, where the first derivative is always positive. This comes with the cost of possibly „overshooting“ the target by adding $\alpha \cdot \Delta\overrightarrow{o}(i)$ to $\overrightarrow{o}(i)$ and divergence, which causes the training to fail if alpha is too large.

2.2 Restructuring the binary tree

This model has the ability to begin its training with the lowest complexity possible that does not output a constant: a root-node, that has 2 end-nodes as its children. Over time, the complexity of the model can be increased with little computational effort and no negative impact on its accuracy. For this purpose, the standard deviation is used to determine which branches of this binary tree require the most complexity. Intuitively speaking, $(\overrightarrow{y}(j) - \overrightarrow{o_j}(0)) * c_j(i)$ ideally shouldn't deviate from $\alpha \cdot \Delta\overrightarrow{o}(i)$ for all $j \in S(i)$, because this means that this model can depict the Algorithm $f(\overrightarrow{x})$ it is meant to approximate perfectly. If it does deviate however, this branch is either not complex enough, meaning that $complexity(i)$ should be greater than it is, or $f(\overrightarrow{x})$ is nondeterministic and can not be depicted perfectly. If this deviation is small for any node, both children of this node are end-nodes and have very similar output vectors, this node can be replaced by an end-node that has the same position vector as this node and the mean of the output vectors of its children as the output vector, This way redundant complexity can be reduced, without $\overrightarrow{o}(0)$ changing to much for any input as a result.