June 12, 2018

BRANCHING NEURAL NETWORKS

Ufuk Can Biçici

Cem Keskin

Shahram Izadi

BRANCHING NEURAL NETWORKS

Abstract

A conditional deep learning model that learns specialized representations on a decision tree is described. Unlike similar methods taking a probabilistic mixture of experts (MoE) approach, a feature augmentation based method is used to jointly train all network and decision parameters using back–propagation, which allows for deterministic binary decisions at both training and test time, specializing subtrees exclusively to clusters of data. Feature augmentation involves combining intermediate representations with scores or confidences assigned to branches. Each representation is augmented with all of the scores assigned to the active branch on the computational path to encode the entire path information, which is essential for efficient training of decision functions. These networks are referred to as Branching Neural Networks (BNNs). As this is an approach that is orthogonal to many other neural network compression methods, such algorithms can be combined to achieve much higher compression rates and further speedups.

Background

Deep learning has become the dominant approach for challenging Computer Vision (CV) tasks. Now in almost every benchmark dataset, convolutional neural networks (CNN) consistently reach the lowest error rates, and even achieve human-level recognition rates for many image recognition tasks. However, the use of CNNs in real-time CV applications and products has lagged behind other classical approaches. This is primarily because the most important metric for real world applications and products is the accuracy that can be achieved within a small predefined computational budget.

Unfortunately, evaluation of CNN models remains computationally intensive. Even on modern GPUs, it can be orders of magnitude higher than that of other alternatives such as Randomized Decision Forests (RDFs) for CV tasks. Most mobile devices such as phones or standalone Augmented and Virtual Reality (AR/VR) headsets have limited computational resources for these tasks, but also typically require high accuracy.

There is now significant recent attempts at increasing the speed of CNNs, for instance by pruning layers in an energy-aware manner, by substitution of filters with smaller ones, or by quantizing or binarizing weights. These techniques whilst effective, rely on compressing the networks by optimizing weights and filter sizes, without a drastic reduction in accuracy.

An orthogonal approach focuses on architectures for CNNs exploiting branching and tree-like structures without losing the benefits of deep networks, by maximizing speed and filter specialization through conditional branching, specifically for highly efficient real-time scenarios. The biggest challenge for these methods is effectively training such networks and their branching functions, while optimizing the network for binary decision making to optimize runtime speeds.

Description

Conditional Computing, as described herein, relies on conditional activation or deactivation of individual neurons or sub-networks on a per-example basis. Such a network can be thought of as a decision tree, where each node transforms its input via a transformation function contained within, and a decision function attached to the node selects the next node from a set of valid children, conditioned on some feature maps. The transformation functions can be any differentiable mapping, including convolutions.

Other related work has common characteristics, such as employing a set of parametric functions for learning representations that can be specialized by conditioning on the input by selection of parametric routers at training or test time. Their differences lie in how the decisions are made (stochastic, deterministic, binary, sparse, dense) and how these parameters are learned (jointly, independently, iteratively, via RL, via back-propagation). One known limitation is mixing hard, binary decisions with back-propagation, while still allowing for specialized representation learning.

A method to train transformation and decision parameters jointly via back-propagation using only binary decisions at both training and test time is described. A conditional decision function assigns real valued confidences to each valid child node as in the case of probabilistic MoE approaches. Only the highest scoring branch is activated, and its corresponding confidence value is concatenated to the feature map that was produced by the current node. Each such confidence is used to augment every feature map on the rest of the computational path. At leaves, the final representation contains all the confidences of each decision, which effectively encodes the path taken by the input as a real valued vector. A standard objective function is then used to train the network, which jointly estimates decision and transformation parameters via back-propagation. The loss function is differentiable with respect to every parameter everywhere except for the decision boundaries. Similar methods for tackling hard decisions have been proposed for decision trees without transformation functions. BNNs improve on such methods by (i) allowing conditional representation learning, (ii) removing the back-propagation bottleneck for decision parameters by a skip-connection trick, and (iii) smoothing the energy landscape around decision boundaries by linearly blending two branches when the confidences are small.

The BNNs described herein include:

- a feature augmentation method to train conditional neural networks with binary decisions,
- a skip connection method to encode the entire path information in a feature map for efficient training of decision parameters,
- a regularization method to estimate gradients for examples close to decision boundaries, and
- discussion of ways to initialize and regularize decision parameters

A BNN is a full k-ary tree of depth $D \geq 1$, with k^{D-1} leaf nodes and $k^{D-1} - 1$ split nodes. For $D = 1$, it reduces to a regular CNN. A differentiable and parametric transformer function $\Phi_s \colon \mathbb{R}^{\dim(x'_d)} \to \mathbb{R}^{\dim(x_{d+1})}$ contained in a node with index s in level d transforms its augmented (defined below) input feature map x'_d to x_{d+1}, such that:

$$x_{d+1} = \Phi_s(x'_d;\, \theta_s). \tag{1}$$

The rank and dimensionality of each feature map is determined by the type of operations applied as in regular CNNs. A differentiable and parametric decision function (*a router*) Ψ_s :

$\mathbb{R}^{\dim(x_{d+1})} \rightarrow \mathbb{R}_{+}^{k}$ also contained in the node (except for leaf nodes) conditionally assigns non-negative scores h_d to each of the k children, such that:

$$h_d = \Psi_s(x_{d+1}; \rho_s). \tag{2}$$

A non-parametric and non-differentiable function (an *activator*) $\Upsilon : \mathbb{R}^k \rightarrow \{0,1\}^k$ maps the scores h_d to the activation vector r_d, which is a one-hot encoding of the activated branch. This can easily be achieved by an arg max operation followed by one hot encoding, or by using the following function:

$$r_d = \Upsilon(h_d) = \left\lfloor \frac{h_d}{\max(h_d)} \right\rfloor. \tag{3}$$

r_d is explicitly used to activate the next node. Also, the set of activations $\{r_d\}_1^D$ can be used to determine which nodes were active during the forward pass for each sample, such that:

$$A_s = \begin{cases} 0 & \text{node is inactive} \\ 1 & \text{node is active} \end{cases} \tag{4}$$

A_1 is always 1 for every sample, and activity of other nodes are determined from r_d at each level. The children of inactive nodes are also inactive.

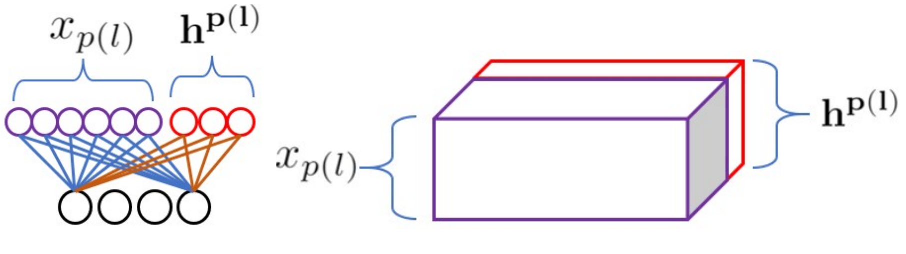

Figure 1: **Feature augmentation:** *Left*: when feature map rank is 1, confidences are simply concatenated to the end, *Right*: For higher rank feature maps, confidences are duplicated and concatenated to each *pixel*.

The output representation is then augmented with all of the confidence vectors $\{h_i\}_{i=1}^{d}$ produced on the path with a feature map specific function that concatenates the confidences to the feature map in a suitable manner. For a flat representation $x_{d+1} \in \mathbb{R}^m$, augmentation is defined as follows:

$$x'_{d+1} = x_{d+1} \oplus_i^d h_i \tag{5}$$

Where $\oplus$ is the concatenation function, and $x'_{d+1} \in \mathbb{R}^{m+kd}$. For convolutional transformations, where x_{d+1} has a rank > 1, the confidences are concatenated to each pixel on the last dimension independently. A visual explanation is given in Figure 1. Finally, the leaf nodes map their input feature maps into the network output $\hat{y}$:

$$\hat{y}_l = x_{D+1} = A_l \Phi_l(x'_D; \theta_l) \tag{6}$$

$$\hat{y} = \sum_l^L y_l \tag{7}$$

Only one of the leaf activations A_l is 1 for an example at a given run. A binary BNN example is depicted in Figure 2.

To train a BNN via back propagation, we need the derivative of the loss function with respect to all the weights in the model. For a generic loss $\mathcal{L}(y, \hat{y})$ and weight ω_i , the chain rule gives:

$$\frac{\partial \mathcal{L}(y,\hat{y})}{\partial \omega_i} = \frac{\partial \mathcal{L}(y,\hat{y})}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \omega_i} \tag{8}$$

$$= \frac{\partial \mathcal{L}(y,\hat{y})}{\partial \hat{y}} \sum_l \frac{\partial y_l}{\partial \omega_i} \tag{9}$$

$$= \frac{\partial \mathcal{L}(y,\hat{y})}{\partial \hat{y}} \sum_l \left(\frac{\partial \Phi_l}{\partial \omega_i} A_l + \Phi_l \frac{\partial A_l}{\partial \omega_i} \right). \tag{10}$$

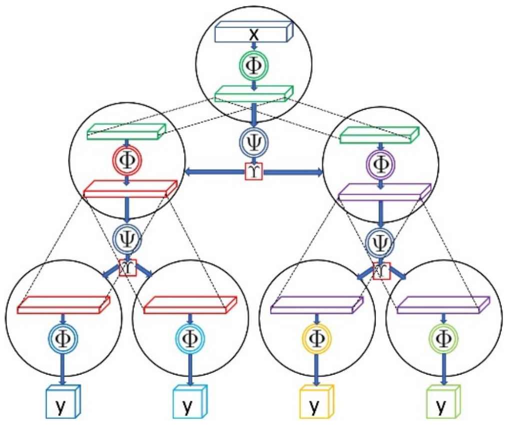

Figure 2: Branching Neural Network: Each split node has a *transformer* Φ_s, a *router* Ψ_s, and an *activator* Υ. Each leaf only has one *transformer*.

Here, $\frac{\partial A_l}{\partial \omega_i}$ is always 0, and only one of $\frac{\partial \Phi_j}{\partial \omega_i}$ is non-zero, where $A_j = 1$. This shows that all derivatives for the inactive branches disappear, making each sample to contribute only to its own active branch.

Augmenting each feature map at level d with $\{h_i\}_{i=1}^{d}$ as opposed to just h_d creates direct channels from each node to all of the decision parameters of its grandparents. This can be seen as the skip-connection trick applied to just the decisions, helping with the training of decisions that appear early on the computational path. These back-channels are visualized in Figure 3.

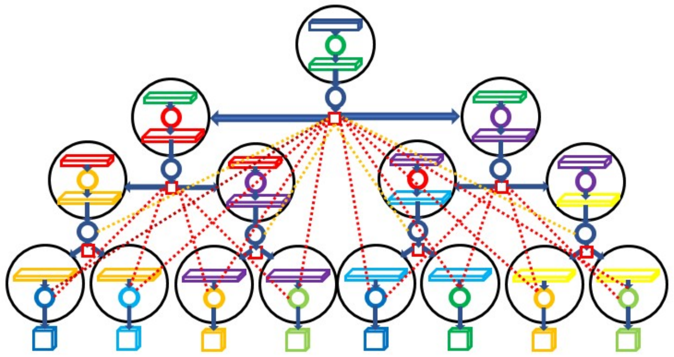

Figure 3: Augmenting with all ancestral confidences creates direct back–channels in the back–propagation step.

Making hard decisions ensures specialization of each subtree to its own cluster only. However, this can potentially increase the generalization error due to samples close to decision boundaries ending up on the wrong branch because of noise. Since datasets are finite and limited in number of samples, it is a good idea to simulate nearby samples by adding zero-mean noise to decisions.

This ensures that branches are not ignorant to samples that can potentially end in them. Another approach is using dropout as a regularizer throughout the network, as it has a similar effect on decisions.

Time complexity of making an inference with BNNs is very close to a regular CNN that has the same architecture as the active branch of the BNN, with the only overhead being making decisions. As these are typically linear projections in the feature space, they are negligible compared to the much more expensive convolutional operations.

To improve the load balance, it is possible to use some of the existing information gain maximization techniques as a regularizer. Although such methods already exist, they rely on this regularizer only to train decision parameters.

It is possible to extend the formulation to allow making decisions on the raw input feature space, as in the case of RDFs. In this case, decision parameters can be initialized by first training an RDF greedily, and then the network can be refined by global optimization.

BNNs can easily be implemented and trained in auto-differentiation frameworks that support masking in the batch dimension. After each decision, a masked version of the mini-batch is sent to each child node, meaning that the batch size gets smaller after each decision.

For other frameworks that do not support batch-masking, the entire mini-batch can be sent to every node, and the results can be multiplied with the activations at the leaves, although it is a more inefficient solution.

Another important consideration is automatic gradient scaling, which needs to be off to make sure the leaf contributions are properly scaled.

Each leaf is assigned a separate loss, and the minimized objective is the sum over all losses. This gives the model the flexibility to use different types of losses at each leaf.

To showcase BNNs, it is applied to a complex semantic segmentation task. Given a dataset with depth images of people performing gestures and label masks for left hand, right hand and body classes, a fully convolutional CNN is trained first to do classification, which achieves 85.5% accuracy. Figure 4 shows how to add a decision layer to convert the CNN to a BNN of depth 2. It is straightforward to add more decisions to further increase the depth. In this simple case, BNNs perform better by reaching 86.7%.

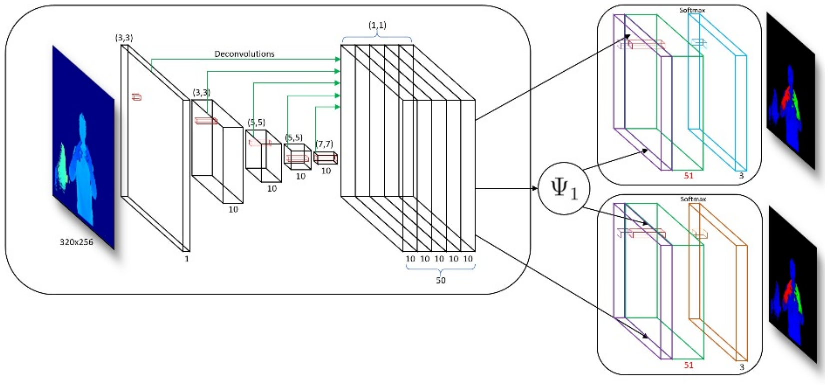

Figure 4: BNNs on Hand Segmentation A BNN that employs a router after the deconvolutions of the original CNN. The last convolutions and classification are specialized to each branch.

Here, a conditional deep learning model is described that combines divide-and-conquer approaches with representation learning, specifically allowing for binary decisions to be used at both training and test time for better speed and specialization. Feature augmentation is used as a method to jointly train all network parameters using back-propagation. Furthermore, new issues that arise from this approach were identified and solutions for each presented. As this is an approach that is orthogonal to many other DNN compression methods, such algorithms can be combined to achieve much higher compression rates.