2016 23rd International Conference on Pattern Recognition (ICPR)
Cancún Center, Cancún, México, December 4-8, 2016

BranchyNet: Fast Inference via Early Exiting from Deep Neural Networks

Surat Teerapittayanon
Harvard University
Email: steerapi@seas.harvard.edu

Bradley McDanel
Harvard University
Email: mcdanel@fas.harvard.edu

H.T. Kung
Harvard University
Email: kung@harvard.edu

***Abstract*—Deep neural networks are state of the art methods for many learning tasks due to their ability to extract increasingly better features at each network layer. However, the improved performance of additional layers in a deep network comes at the cost of added latency and energy usage in feedforward inference. As networks continue to get deeper and larger, these costs become more prohibitive for real-time and energy-sensitive applications. To address this issue, we present BranchyNet, a novel deep network architecture that is augmented with additional side branch classifiers. The architecture allows prediction results for a large portion of test samples to exit the network early via these branches when samples can already be inferred with high confidence. BranchyNet exploits the observation that features learned at an early layer of a network may often be sufficient for the classification of many data points. For more difficult samples, which are expected less frequently, BranchyNet will use further or all network layers to provide the best likelihood of correct prediction. We study the BranchyNet architecture using several well-known networks (LeNet, AlexNet, ResNet) and datasets (MNIST, CIFAR10) and show that it can both improve accuracy and significantly reduce the inference time of the network.**

I. Introduction

One of the reasons for the success of deep networks is their ability to learn higher level feature representations at successive nonlinear layers. In recent years, advances in both hardware and learning techniques have emerged to train even deeper networks, which have improved classification performance further [4, 8]. The ImageNet challenge exemplifies the trend to deeper networks, as the state of the art methods have advanced from 8 layers (AlexNet), to 19 layers (VGGNet), and to 152 layers (ResNet) in the span of four years [7, 13, 20]. However, the progression towards deeper networks has dramatically increased the latency and energy required for feedforward inference. For example, experiments that compare VGGNet to AlexNet on a Titan X GPU have shown a factor of 20x increase in runtime and power consumption for a reduction in error rate of around 4% (from 11% to 7%) [11]. The trade off between resource usage efficiency and prediction accuracy is even more noticeable for ResNet, the current state of the art method for the ImageNet Challenge, which has an order of magnitude more layers than VGGNet. This rapid increase in runtime and power for gains in accuracy may make deeper networks less tractable in many real world scenarios, such as real-time control of radio resources for next-generation mobile networking, where latency and energy are important factors.

To lessen these increasing costs, we present BranchyNet, a neural network architecture where side branches are added to the main branch, the original baseline neural network, to allow certain test samples to exit early. This novel architecture exploits the observation that it is often the case that features learned at earlier stages of a deep network can correctly infer a large subset of the data population. By exiting these samples with prediction at earlier stages and thus avoiding layer-by-layer processing for all layers, BranchyNet significantly reduces the runtime and energy use of inference for the majority of samples. Figure 1 shows how BranchyNet modifies a standard AlexNet by adding two branches with their respective exit points.

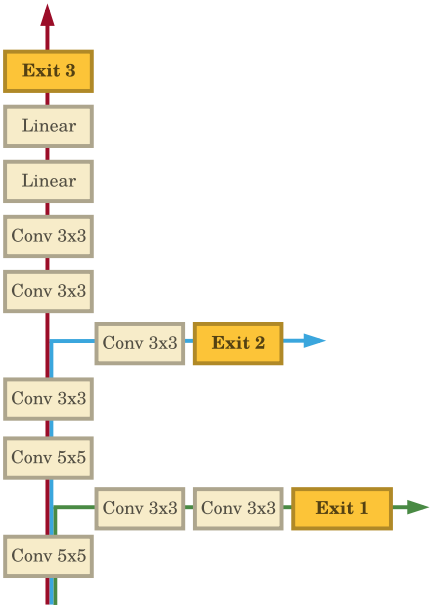

Fig. 1: A simple BranchyNet with two branches added to the baseline (original) AlexNet. The first branch has two convolutional layers and the second branch has 1 convolutional layer. The "Exit" boxes denote the various exit points of BranchyNet. This figure shows the general structure of BranchyNet, where each branch consists of one or more layers followed by an exit point. In practice, we generally find that it is not necessary to add multiple convolutional layers at a branch in order to achieve good performance.

BranchyNet is trained by solving a joint optimization

978-1-5090-4847-2/16/$31.00 ©2016 IEEE

problem on the weighted sum of the loss functions associated with the exit points. Once the network is trained, BranchyNet utilizes the exit points to allow the samples to exit early, thus reducing the cost of inference. At each exit point, BranchyNet uses the entropy of a classification result (e.g., by softmax) as a measure of confidence in the prediction. If the entropy of a test sample is below a learned threshold value, meaning that the classifier is confident in the prediction, the sample exits the network with the prediction result at this exit point, and is not processed by the higher network layers. If the entropy value is above the threshold, then the classifier at this exit point is deemed not confident, and the sample continues to the next exit point in the network. If the sample reaches the last exit point, which is the last layer of the baseline neural network, it always performs classification.

Three main contributions of this paper are:

- **Fast Inference with Early Exit Branches:** BranchyNet exits the majority of the samples at earlier exit points, thus reducing layer-by-layer weight computation and I/O costs, resulting in runtime and energy savings.
- **Regularization via Joint Optimization:** BranchyNet jointly optimizes the weighted loss of all exit points. Each exit point provides regularization on the others, thus preventing overfitting and improving test accuracy.
- **Mitigation of Vanishing Gradients:** Early exit points provide additional and more immediate gradient signal in back propagation, resulting in more discriminative features in lower layers, thus improving accuracy.

II. Background and Related Prior Work

LeNet-5 [15] introduced the standard convolutional neural networks (CNN) structure which is composed of stacked convolutional layers, optionally followed by contrast normalization and maxpooling, and then finally followed by one or more fully-connected layers. This structure has performed well in several image tasks such as image classification. AlexNet [13], VGG [20], ResNet [7] and others have expanded on this structure with their own innovative approaches to make the network deeper and larger for improved classification accuracy.

Due to the computational costs of deep networks, improving the efficiency of feedforward inference has been heavily studied. Two such approaches are network compression and implementation optimization. Network compression schemes aim to reduce the the total number of model parameters of a deep network and thus reduce the amount of computation required to perform inference. Bucilua et al. (2006) proposed a method of compressing a deep network into a smaller network that achieves a slightly reduced level of accuracy by retraining a smaller network on synthetic data generated from a deep network [3]. More recently, Han et al. (2015) have proposed a pruning approach that removes network connections with small contributions [5]. However, while pruning approaches can significantly reduce the number of model parameters in each layer, converting that reduction into a significant speedup is difficult using standard GPU implementations due to the lack of high degrees of exploitable regularity and computation intensity in the resulting sparse connection structure [6]. Kim et al. (2015) use a Tucker decomposition (a tensor extension of SVD) to extract shared information between convolutional layers and perform rank selection [11]. This approach reduces the number of network parameters, making the network more compact, at the cost of a small amount of accuracy loss. These network compression methods are orthogonal to the BranchyNet approach taken in this paper, and could potentially be used in conjunction to improve inference efficiency further.

Implementation optimization approaches reduce the runtime of inference by making the computation algorithmically faster. Vanhoucke et al. (2011) explored code optimizations to speed up the execution of convolutional neural networks (CNNs) on CPUs [25]. Mathieu et al. (2013) showed that convolution using FFT can be used to speed up training and inference for CNNs [17]. Recently, Lavin et al. (2015) have introduced faster algorithms specifically for 3x3 convolutional filters (which are used extensively in VGGNet and ResNet) [14]. In contrast, BranchyNet makes modifications to the network structure to improve inference efficiency.

Deeper and larger models are complex and tend to overfit the data. Dropout [21], L1 and L2 regularization and many other techniques have been used to regularize the network and prevent overfitting. Additionally, Szegedy et al. (2015) introduced the concept of adding softmax branches in the middle layers of their inception module within deep networks as a way to regularize the main network [23]. While also providing similar regularization functionalities, BranchyNet has a new goal of allowing early exits for test samples which can already be classified with high confidence.

One main challenge with (very) deep neural networks is the vanishing gradient problem. Several papers have introduced ideas to mitigate this issue including normalized network initialization [4, 16] and intermediate normalization layers [10]. Recently, new approaches such as Highway Networks [22], ResNet [7], and Deep Networks with Stochastic Depth [9] have been studied. The main idea is to add skip (shortcut) connections in between layers. This skip connection is an identity function which helps propagate the gradients in the backpropagation step of neural network training.

Panda et al. [18] propose Conditional Deep Learning (CDL) by iteratively adding linear classifiers to each convolutional layer, starting with the first layer, and monitoring the output to decide whether a sample can be exited early. BranchyNet allows for more general branch network structures with additional layers at each exit point while CDL only uses a cascade of linear classifiers, one for each convolutional layer. In addition, CDL does not jointly train the classifier with the original network. We observed in our paper that jointly training the branch with the original network significantly improve the performance of the overall architecture when compared to CDL.

III. BranchyNet

BranchyNet modifies the standard deep network structure by adding exit branches (also called side branches or simply branches for brevity), at certain locations throughout the

network. These early exit branches allow samples which can be accurately classified in early stages of the network to exit at that stage. In training the classifiers at these exit branches, we also consider network regularization and mitigation of vanishing gradients in backprogation. For the former, branches will provide regularization on the main branch (baseline network), and vice versa. For the latter, a relatively shallower branch at a lower layer will provide more immediate gradient signal in backpropagation, resulting in discriminative features in lower layers of the main branch, thus improving its accuracy.

In designing the BranchyNet architecture, we address a number of considerations, including (1) locations of branch points, (2) structure of a branch (weight layers, fully-connected layers, etc.) as well as its size and depth, (3) classifier at the exit point of a branch, (4) exit criteria for a branch and the associated test cost against the criteria, and (5) training of classifiers at exit points of all branches. In general, this "branch" notion can be recursively applied, that is, a branch may have branches, resulting in a tree structure. For simplicity, in this paper we focus a basic scenario where there are only one-level branches which do not have nested branches, meaning there are no tree branches.

In this paper, we describe BranchyNet with classification tasks in mind; however, the architecture is general and can also be used for other tasks such as image segmentation and object detection.

A. Architecture

A BranchyNet network consists of an entry point and one or more exit points. A branch is a subset of the network containing contiguous layers, which do not overlap other branches, followed by an exit point. The main branch can be considered the baseline (original) network before side branches are added. Starting from the lowest branch moving to highest branch, we number each branch and its associated exit point with increasing integers starting at one. For example, the shortest path from the entry point to any exit is exit 1, as illustrated in Figure 1.

B. Training BranchyNet

For a classification task, the softmax cross entropy loss function is commonly used as the optimization objective. Here we describe how BranchyNet uses this loss function. Let $\boldsymbol{y}$ be a one-hot ground-truth label vector, $\boldsymbol{x}$ be an input sample and $\mathcal{C}$ be the set of all possible labels. The objective function can be written as

$$L(\hat{\boldsymbol{y}}, \boldsymbol{y}; \theta) = -\frac{1}{|\mathcal{C}|} \sum_{c \in \mathcal{C}} y_c \log \hat{y_c},$$

where

$$\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{z}) = \frac{\exp(\boldsymbol{z})}{\sum_{c \in \mathcal{C}} \exp(z_c)},$$

and

$$\boldsymbol{z} = f_{\text{exit}_n}(\boldsymbol{x}; \theta),$$

where f_{exit_n} is the output of the n-th exit branch and θ represents the parameters of the layers from an entry point to the exit point.

The design goal of each exit branch is to minimize this loss function. To train the entire BranchyNet, we form a joint optimization problem as a weighted sum of the loss functions of each exit branch

$$L_{\text{branchynet}}(\hat{\boldsymbol{y}}, \boldsymbol{y}; \theta) = \sum_{n=1}^{N} w_n L(\hat{\boldsymbol{y}}_{\text{exit}_n}, \boldsymbol{y}; \theta),$$

where N is the total number of exit points. Section V-A discusses how one might choose weights w_n.

The algorithm consists of two steps: the feedforward pass and the backward pass. In the feedforward pass, the training data set is passed through the network, including both main and side branches, the output from the neural network at all exit points is recorded, and the error of the network is calculated. In backward propagation, the error is passed back through the network and the weights are updated using gradient descent. For gradient descent, we use Adam algorithm [12], though other variants of Stochastic Gradient Descent (SGD) can also be used.

C. Fast Inference with BranchyNet

Once trained, BranchyNet can be used for fast inference by classifying samples at earlier stages in the network based on the algorithm in Figure 2. If the classifier at an exit point of a branch has high confidence about correctly labeling a test sample $\boldsymbol{x}$, the sample is exited and returns a predicted label early with no further computation performed by the higher branches in the network. We use entropy as a measure of how confident the classifier at an exit point is about the sample. Entropy is defined as

$$\text{entropy}(\boldsymbol{y}) = \sum_{c \in \mathcal{C}} y_c \log y_c,$$

where $\boldsymbol{y}$ is a vector containing computed probabilities for all possible class labels and $\mathcal{C}$ is a set of all possible labels.

procedure BRANCHYNETFASTINFERENCE($\boldsymbol{x}, \boldsymbol{T}$)
  for $n = 1..N$ **do**
    $\boldsymbol{z} = f_{\text{exit}_n}(\boldsymbol{x})$
    $\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{z})$
    $e \leftarrow \text{entropy}(\hat{\boldsymbol{y}})$
    if $e < T_n$ **then**
      return $\arg\max \hat{\boldsymbol{y}}$
  return $\arg\max \hat{\boldsymbol{y}}$

Fig. 2: BranchyNet Fast Inference Algorithm. $\boldsymbol{x}$ is an input sample, $\boldsymbol{T}$ is a vector where the n-th entry T_n is the threshold for determining whether to exit a sample at the n-th exit point, and N is the number of exit points of the network.

To perform fast inference on a given BranchyNet network, we follow the procedure as described in Figure 2. The procedure requires $\boldsymbol{T}$, a vector where the n-th entry is the threshold used to determine if the input $\boldsymbol{x}$ should exit at

the n-th exit point. In section V-B, we discuss how these thresholds may be set. The procedure begins with the lowest exit point and iterates to the highest and final exit point of the network. For each exit point, the input sample is fed through the corresponding branch. The procedure then calculates the softmax and entropy of the output and checks if the entropy is below the exit point threshold T_n. If the entropy is less than T_n, the class label with the maximum score (probability) is returned. Otherwise, the sample continues to the next exit point. If the sample reaches the last exit point, the label with the maximum score is always returned.

IV. Results

In this section, we demonstrate the effectiveness of BranchyNet by adapting three widely studied convolutional neural networks on the image classification task: LeNet, AlexNet, and ResNet. We evaluate Branchy-LeNet (B-LeNet) on the MNIST dataset and both Branchy-AlexNet (B-AlexNet) and Branchy-ResNet (B-ResNet) on the CIFAR10 data set. We present evaluation results for both CPU and GPU. We use a 3.0GHz CPU with 20MB L3 Cache and NVIDIA GeForce GTX TITAN X (Maxwell) 12GB GPU.

For simplicity, we only describe convolutional and fully-connected layers of each network. Generally, these networks may also contain max pooling, non-linear activation functions (e.g., a rectified linear unit and sigmoid), normalization (e.g., local response normalization, batch normalization), and dropout.

For LeNet-5 [15] which consists of 3 convolutional layers and 2 fully-connected layers, we add a branch consisting of 1 convolutional layer and 1 fully-connected layer after the first convolutional layer of the main network. For AlexNet [13] which consists of 5 convolutional layers and 3 fully-connected layers, we add 2 branches. One branch consisting of 2 convolutional layers and 1 fully-connected layer is added after the 1st convolutional layer of the main network, and another branch consisting of 1 convolutional layer and 1 fully-connected layer is added after the 2nd convolutional layer of the main network. For ResNet-110 [7] which consists of 109 convolutional layers and 1 fully-connected layer, we add 2 branches. One branch consisting of 3 convolutional layers and 1 fully-connected layer is added after the 2nd convolutional layer of the main network, and the second branch consisting of 2 convolutional layers and 1 fully-connected layer is added after the 37th convolutional layer of the main network. We initialize B-LeNet, B-AlexNet and B-ResNet with weights trained from LeNet, AlexNet and ResNet respectively. We found the initializing each BranchyNet network with the weights trained from the baseline network improved the classification accuracy of the network by several percent over random initialization. To train these networks, we use Adam algorithm with a step size (α) of 0.001 and exponential decay rates for first and second moment estimates (β_1, β_2) of 0.99 and 0.999 respectively.

Figure 3 shows the GPU performance results of BranchyNet when applied to each network. For all of the networks, BranchyNet outperforms the original baseline network. The reported runtime is the average among all test samples. B-LeNet has the largest performance gain due to a more efficient branch which achieves almost the same level of accuracy as the last exit branch. For AlexNet and ResNet, we see that the performance gain is still substantial, but since more samples are required to exit at the last layer, smaller than B-LeNet. The knee point denoted as the green star represents an optimal threshold point, where the accuracy of BranchyNet is comparable to the main network, but the inference is performed significantly faster. For B-ResNet, the accuracy is slightly lower than the baseline. A different threshold could be chosen which gives accuracy higher than ResNet but with much less savings in inference time. The performance characteristics of BranchyNet running on CPU follow a similar trend to the performance of BranchyNet running on GPU.

Table I highlights the selected knee threshold values, exit (%) and gain in speed up, for BranchyNet for each network for both CPU and GPU. The $\boldsymbol{T}$ column denotes the threshold values for each exit branch. Since the last exit branch must exit all samples, it does not require an exit threshold. Therefore, for a 2-branch network, such as B-LeNet, there is a single $\boldsymbol{T}$ value and for a 3-branch network, such as B-AlexNet and B-ResNet, there are two $\boldsymbol{T}$ values. Further analysis of the sensitivity of the $\boldsymbol{T}$ parameters is discussed in Section V. The Exit (%) column shows the percentage of samples exited at each branch point. For all networks, we see that BranchyNet is able to exit a large percentage of the test samples before the last layer, leading to speedups in inference time. B-LeNet exits 94% of samples at the first exit branch, while B-AlexNet and B-ResNet exit 65% and 41% respectively. Exiting these samples early translate to CPU/GPU speedup gains of 5.4/4.7x over LeNet, 1.5/2.4x over AlexNet, and 1.9/1.9x over ResNet. The branch structure for B-ResNet mimics that of B-AlexNet.

TABLE I: Selected performance results for BranchyNet on the different network structures. The BrachyNet rows correspond to the knee points denoted as green stars in Figure 3.

	Network	**Acc.** (%)	**Time** (ms)	**Gain**	**Thrshld.** $\boldsymbol{T}$	**Exit** (%)
CPU	LeNet	99.20	3.37	-	-	-
	B-LeNet	99.25	0.62	5.4x	0.025	94.3, 5.63
	AlexNet	78.38	9.56	-	-	-
	B-AlexNet	79.19	6.32	1.5x	0.0001, 0.05	65.6, 25.2, 9.2
	ResNet	80.70	137.20	-	-	-
	B-ResNet	79.17	73.5	1.9x	0.3, 0.2	41.5, 13.8, 44.7
GPU	LeNet	99.20	1.58	-	-	-
	B-LeNet	99.25	0.34	4.7x	0.025	94.3, 5.63
	AlexNet	78.38	3.15	-	-	-
	B-AlexNet	79.19	1.30	2.4x	0.0001, 0.05	65.6, 25.2, 9.2
	ResNet	80.70	70.9	-	-	-
	B-ResNet	79.17	37.2	1.9x	0.3, 0.2	41.5, 13.8, 44.7

V. Analysis and Discussion

In this section, we provide additional analysis on key aspects BranchyNet.

A. Hyperparameter Sensitivity

Two important hyperparameters of BranchyNet are the weights w_n in joint optimization (Section III-B) and the exit thresholds $\boldsymbol{T}$ for the fast inference algorithm described in Figure 2. When selecting the weight of each branch, we observed that giving more weight to early branches improves the accuracy of the later branches due to the added regularization.

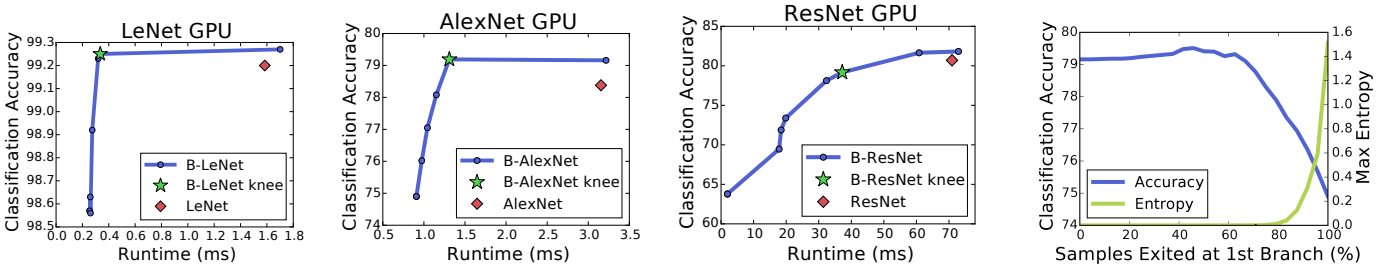

Fig. 3: GPU performance results for BranchyNet when applied to LeNet, AlexNet and ResNet. The original network accuracy and runtime are shown as the red diamond. The BranchyNet modification to each network is shown in blue. Each point denotes different combinations of entropy thresholds for the branch exit points (found via sweeping over $\boldsymbol{T}$). The star denotes a knee point in the curve, with additional analysis shown in Table I. The CPU performance results have similar characteristics, and can also be found in Table I. Runtime is measured in milliseconds (ms) of inference per sample. For this evaluation, we use batch size 1 as evaluated in [5, 11] in order to target real-time streaming applications. A larger batch size allows for parallelism, but lessens the benefit of early exit as all samples in a batch must exit before a new batch can be processed.

Fig. 4: The overall classification accuracy of B-AlexNet for varying entropy threshold for the first exit branch. For this experiment, all samples not exited in at the first branch are exited at the final exit. The entropy at a given value is the max entropy of all samples up to that point.

On a simplified version of BranchyAlexNet with only the first and last branch, weighting the first branch with 1.0 and the last branch with 0.3 provides a 1% increase in classification accuracy over weighting each branch equally. Giving more weight to earlier exit branches encourages more discriminative feature learning in early layers of the network and allows more samples to exit early with high confidence.

Figure 4 shows how the choice of $\boldsymbol{T}$ affects the number of samples exited at the first branch point in B-AlexNet. We observe that the entropy value has a distinctive knee where it rapidly becomes less confident in the test samples. Thus in this case it is relatively easy to identify the knee and learn a corresponding threshold. In practice, the choice of exit threshold for each exit point depends on applications and datasets. The exit thresholds should be chosen such that it satisfies the inference latency requirement of an application while maintaining the required accuracy.

An additional hyperparameter not mentioned explicitly is the location of the branch points in the network. In practice, we find the location of the first branch point depends on the difficulty of the dataset. For a simpler dataset, such as MNIST, we can place a branch directly after the first layer and immediately see accurate classification. For more challenging datasets, branches should be placed higher in order to still achieve strong classification performance. For any additional branches, we currently place them at equidistant points throughout the network. Future work will be to derive an algorithm to find the optimal placement locations of the branches automatically.

B. Tuning Entropy Thresholds

The results shown in Figure 3 provides the accuracy and runtime for a range of $\boldsymbol{T}$ values. These $\boldsymbol{T}$ values show how BranchyNet trades off accuracy for faster runtime as the entropy thresholds increase. However, in practice, we may want to set $\boldsymbol{T}$ automatically to met a specified runtime or accuracy constraint. One approach is to simply screen over $\boldsymbol{T}$ as done here and pick a setting that satisfies the constraints. We provided code used to generate the performance results which includes a method for performing this screening [24].

Additionally, it may be possible to use a Meta-Recognition algorithm [19, 26] to estimate the characteristics of unseen test samples and adjust $\boldsymbol{T}$ automatically in order to maintain a specified runtime or accuracy goal. One simple approach for creating such a Meta-Recognition algorithm would be to train a small Multilayer Perceptron (MLP) for each corresponding exit point on the output softmax probability vectors $\hat{\boldsymbol{y}}$ for that exit. The MLP at an exit point would attempt to predict if a given sample would be correctly classified at the specific exit. More generally, this approach is closely related to the open world recognition problem [2, 1], which is interested in quantifying the uncertainty of a model for a particular set of unseen or out of set test samples. We can expand on the MLP approach further by using a different formulation than SoftMax, such as OpenMax [2], which attempts to quantify the uncertainty directly in the probability vector $\hat{\boldsymbol{y}}$ by adding an additional uncertain class. These approaches could be used to tune $\boldsymbol{T}$ automatically to a new test set by estimating the difficulty of the test data and adapting $\boldsymbol{T}$ accordingly to meet the runtime or accuracy constraints. This work is outside the scope of this paper, which only provides the groundwork BranchyNet architecture, but will be explored in future work.

C. Effects of Structure of Branches

Figure 5 shows the impact on the accuracy of the last exit by adding additional convolutional layers in an earlier side branch for a modified version of B-AlexNet with only the first side branch. We see that there is a optimal number of layers to improve the accuracy of the main branch, and that adding too many layers can actually harm overall accuracy. In addition to convolutional layers, adding a few fully-connected layers after convolutional layers to a branch also proves helpful since this allows local and global features to combine and form more discriminative features. The number of layers in a branch and

the size of an exit branch should be chosen such that the overall size of the branch is less than amount of computation needed to do to exit at a later exit point. Generally, we find that earlier branch points should have more layers, and later branch points should have fewer layers.

D. Effects of cache

Since the majority of samples are exited at early branch points, the later branches are used more rarely. This allows weights at these early exit branches to be cached more efficiently. Figure 6 shows the effect of cache based on various $\boldsymbol{T}$ values for B-AlexNet. We see that the more aggressive $\boldsymbol{T}$ values have faster runtime on the CPU and also less cache miss rates. One could use this insight to select a branch structure that can fits more effectively in a cache, potentially speeding up inference further.

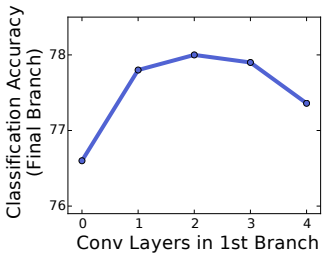

Fig. 5: The impact of the number of convolutional layers in the first branch on the final branch classification accuracy for B-AlexNet.

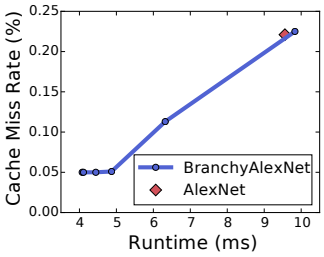

Fig. 6: The runtime and CPU cache miss rate for the B-AlexNet model as the entropy threshold is varied.

VI. Conclusion

We have proposed BranchyNet, a novel network architecture that promotes faster inference via early exits from branches. Through proper branching structures and exit criteria as well as joint optimization of loss functions for all exit points, the architecture is able to leverage the insight that many test samples can be correctly classified early and therefore do not need the later network layers. We have evaluated this approach on several popular network architectures and shown that BranchyNet can reduce the inference cost of deep neural networks and provide 2x-6x speed up on both CPU and GPU.

BranchyNet is a toolbox for researchers to use on any deep network models for fast inference. BranchyNet can be used in conjunction with prior works such as network pruning and network compression [3, 5]. BranchyNet can be adapted to solve other types of problems such as image segmentation, and is not just limited to classification problems. For future work, we plan to explore Meta-Recognition algorithms, such as OpenMax, to automatically adapt $\boldsymbol{T}$ to new test samples.

Acknowledgment

This work is supported in part by gifts from the Intel Corporation and in part by the Naval Supply Systems Command award under the Naval Postgraduate School Agreements No. N00244-15-0050 and No. N00244-16-1-0018.

References

[1] A. Bendale and T. Boult. Towards open set deep networks. *arXiv preprint arXiv:1511.06233*, 2015.
[2] A. Bendale and T. Boult. Towards open world recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1893–1902, 2015.
[3] C. Bucilu, R. Caruana, and A. Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 535–541. ACM, 2006.
[4] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pages 249–256, 2010.
[5] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.
[6] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, pages 1135–1143, 2015.
[7] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.
[8] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1026–1034, 2015.
[9] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Weinberger. Deep networks with stochastic depth. *arXiv preprint arXiv:1603.09382*, 2016.
[10] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.
[11] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015.
[12] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
[14] A. Lavin. Fast algorithms for convolutional neural networks. *arXiv preprint arXiv:1509.09308*, 2015.
[15] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
[16] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.
[17] M. Mathieu, M. Henaff, and Y. LeCun. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851*, 2013.
[18] P. Panda, A. Sengupta, and K. Roy. Conditional deep learning for energy-efficient and enhanced pattern recognition. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 475–480. IEEE, 2016.
[19] W. J. Scheirer, A. Rocha, R. J. Micheals, and T. E. Boult. Meta-recognition: The theory and practice of recognition score analysis. *IEEE transactions on pattern analysis and machine intelligence*, 33(8):1689–1695, 2011.
[20] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
[21] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.
[22] R. K. Srivastava, K. Greff, and J. Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.
[23] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.
[24] S. Teerapittayanon, B. McDanel, and H. Kung. Branchynet: Fast inference via early exiting from deep neural networks. https://gitlab.com/htkung/branchynet, 2016.
[25] V. Vanhoucke, A. Senior, and M. Z. Mao. Improving the speed of neural networks on cpus.
[26] P. Zhang, J. Wang, A. Farhadi, M. Hebert, and D. Parikh. Predicting failures of vision systems. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3566–3573, 2014.