Reproducibility Challenge: Differentiable Branching in Deep Networks for Fast Inference

elena.franchini@usi.ch roberto.neglia@usi.ch bole.ma@usi.ch

jamgbe@usi.ch

Abstract

In recent years, one notable advancement in the deep learning field has been the introduction of models that incorporate multiple auxiliary branches for early exiting during inference. This project aims to reproduce the findings of a seminal paper that proposed a novel algorithm for end-to-end training of neural networks, without having to fine-tune manually all the parameters of the auxiliary branches. We will re-implement the algorithm as described in the paper and apply it to a range of classification tasks, using the publicly available datasets that are also used in the original paper. Through this reproducibility project, we aim to provide valuable insights into the practical applicability and generalizability of the proposed algorithm in real-world classification tasks.

1 Introduction

The target of this project is to reproduce the results of the scientific paper "Differentiable branching in deep networks for fast inference"[1] developed by Simone Scardapane, Danilo Comminiello, Michele Scarpiniti, Enzo Baccarelli, Aurelio Uncini, from the DIET department, Sapienza University of Rome, Italy.

The authors explains that there is a lack of focus in the area of early exit as a technique used for developing energy efficient neural networks (since early exit decreases inference time), so they decided to focus on its implementation. There are different techniques to reduce inference time (compression and pruning of the neural network weights[2], regularisation[3], low-precision arithmetic[4], ...), and among them it has been shown that the early exit can be very efficient, especially if optimised.

There are many scenarios in which energy efficiency and compact models are required, such as Internet of Things[5], embedded devices[6] and Fog computing[7]. The requirement of this kind of models in these usecases highlights the importance of our paper.

The authors developed a deep neural network with low inference time[1]. Among all possible alternatives, the one chosen was to extend the model with auxiliary classifiers (branches) which are used to perform early-exit at different layers. This sort of neural networks are mainly efficient in cases where the classification can be performed at very early stages (since it is possible to perform the early exit and reducing the inference time) and can be distributed within a distributed architecture (e.g. in Fog Computing[7] as we said before).

2 Related works

There are different methods used in other research paper to improve the efficiency of the different parts of a neural network such as deployment, inference etc, but in the case of fast interference, where we need fast and accurate predictions, we can say early exit is one of the best method.

Techniques that other papers have used for fast inference are compression and pruning of the neural network weights [2], regularization [3], using low-precision arithmetic [4], or exploiting ad-hoc components for fast inference [8]. Although all these method resulted in the reduction of inference time based on their perspective paper result

section, the use of these methods can depend on what you need it for, the type of hardware used and whether you would like to choose either to have more accuracy or faster predictions. In the case of the authors proposed method with the optimisation of the entire process with the focus on early exiting, the problem of the trade off between speed and accuracy is reduced.

The work presented in this paper we reproduced explores ways of deciding when to exit in the neural network in a more efficient way, which is by approximating each of the auxiliary classifier with a soft conditioned version (that is differentiable) and then will optimised through back-propagation [1]. The difference between the paper we reproduced and others is that there is more focused place on the exploration, implementation and the use of deciding when the neural network should early exit.

3 Background

Consider a complex neural network called $f(x)$. Unlike traditional setups, this network includes smaller branches, $f_1(x)$ through $f_B(x)$, each capable of making varying accuracy predictions about input patterns. To make these branches work well, we need to solve two main challenges: how to train all these branches together efficiently and how to decide when to stop processing an input and make a prediction, especially when energy is limited. The state-of-the-art is achieved by BranchyNet (B-NET) [9].

3.1 Training multi-branch neural networks

We will explore how the standard multi-branch neural network is built. Multi-branch neural network is used because instead of using one feature of the input to make predictions, it will use different thing for example colours in an image input. Images are used as input to train the multi-branch neural network because of convenience. It has a lot of complex information, so various features can be used for the multi-branch to work on in order to learn from the model. This is useful in the case of this paper as that is exactly what the authors are looking to demonstrate.

The input image is denoted like this

$$\{x_i, y_i\}_{i=1}^{N}$$

where x_i and y_i is the pair of images and N signifies the number of dataset samples. Each of the branch is used to focus on detecting those different things and then they are used together in the neural network to make more accurate predictions. A cross entropy loss is calculated for each branch b to get the predicted probability of each term, therefore allowing for optimisation of what each of the branch is learning. This can be calculated as shown below

$$L_b = \frac{1}{N} \sum_{i=1}^{N} y_i \log(f_b(x_i)) \tag{1}$$

where L_b is the loss calculated for each branch, N is the number of samples in each dataset, $f_b(x_i)$ is the predicted probability of the ith example belonging to the positive class as given by the model and y_i is the true class for the data point. Although we have the loss of each branch, we actually would need the sum of the losses calculated.

The sum of losses can be calculated using the formula below

$$L = \sum_{b=1}^{B} \alpha_b L_b \tag{2}$$

The reason for calculating the sum of losses is so that the model will be able to combine the information from each branch, therefore leading to better performance of the model. The neural network model is able to learn more from the input data because each of the branches is focusing on different things and using the combination of the information learnt.

3.2 Early-exit strategy based on classifier's entropy

After training, one way to decide whether to stop processing at a certain branch is to measure its entropy H_b, which is defined as:

$$H_b(x) = -\frac{1}{C}\sum_j f_b(x)\log(f_b(x)) \tag{3}$$

where C is the size of $f_b(x)$. Entropy shows how confident the network is about a branch's decision. Lower values indicate higher confidence. If the entropy H_b falls below a certain threshold γ_b, we may decide to stop there and make a prediction. This is defined using the formula below.

$$H_b(X) < \gamma_b$$

The value $H_b(x)$ has to be between the range of 0 and 1. The number shows how strong the belief that the neural network has in the branch b. However, determining these thresholds is a challenge and often involves manual adjustments[9] or heuristics[10]. This method is effective for a small number of branches but isn't straightforward to adapt to specific energy constraints.

These strategies aim to optimize neural networks with multiple branches by training them efficiently together and deciding when to stop processing inputs based on the confidence levels of each branch. However, finding the right confidence thresholds remains challenging, especially for networks with many branches or in scenarios with specific energy limitations.

4 Authors Proposed Methodology

The authors introduce a novel approach aimed at improving deep learning models by allowing them to dynamically decide when to stop processing inputs and make predictions based on their confidence levels. Their goal is to develop a method where a deep learning model can decide whether to make a prediction or exit early during inference.

They propose training a logic within the network that controls the decision to exit or continue processing inputs. This logic is trained alongside the rest of the network in an end-to-end manner. Each auxiliary classifier within the network is equipped with a mechanism called $g_b(x)$, determining whether to make an early exit. This mechanism is part of the same auxiliary network ($g_b(x)$ and $f_b(x)$) and shares common processing.

One new idea brought by this paper is the differentiable formulation for training [1]. A new soft-conditional output is proposed to make the training process differentiable. This output is formulated using the gate mechanism ($g_b(x)$) to control the branching logic of the network:

$$\tilde{f}_b(x) = g_b(x) * f_b(x) + (1 - g_b(x)) * \tilde{f}_{b+1}(x) \tag{4}$$

Instead of training on individual branch losses, a cross-entropy loss with respect to the final output $\tilde{f}_{B+1}(x)$ is used to train all branches simultaneously.

During inference, the recursive output $\tilde{f}_{B+1}(x)$ provides a means to adaptively utilize all auxiliary outputs, acting similar to an ensemble technique, enhancing accuracy. The auxiliary values $h_1(x)$, $h_2(x)$, etc., can serve as standard binary classifiers to make early-exit decisions based on their probabilities.

Besides accuracy, the authors aim to improve energy efficiency by minimizing the computational cost of the network. They introduce a scheme to minimize the average computational cost of the network, considering the complexity of early exits at different branches (γ_b). This cost is a non-differentiable function, so it's approximated by a differentiable proxy function ($\tilde{\gamma}_{B+1}$) that includes early exit decisions as a regularization term during training.

5 Implementation

The paper that we reproduced did not come with a code repository, so we had to write the code from scratch based on the methodology that they described. Our results although similar differ from the results achieved by the developers and we didn't implement all parts since the paper was not fully clear on different concepts. This sections discusses the technical details on how we implemented the code.

5.1 Datasets

The authors of the paper used three different datasets to compare the accuracy of the neural network with and without early exit: CIFAR-10, CIFAR-100 and CINIC-10. CIFAR-10[11] consists of 10 classes, CIFAR-100[11] consists of 100 classes, but both contain in total 60'000 32x32 images of which 50'000 are training images and 10'000 are test images. CINIC-10[12] consists of 10 classes and contains in total 270'000 32x32 images. We decided to use the CIFAR-10 and CIFAR-100 datasets, and try to reproduce their results on these two. Additionally, we further divided the training set in 20% validation and 80% training subsets in all datasets.

5.2 Hyperparameters

For the training and the architecture of the early exit branches, we used all the hyperparameters mentioned in the paper[1], but some of them were missing. We did this because the results of the experiments can differ if we use different hyperparameters, so we thought that for comparison reasons it would be easier when comparing the results we got with the one in the paper. The below hyperparameters were consistent over all the experiments we conducted:

- Batch size = 128;
- Total epochs = 30 (not mentioned in the paper);
- Optimizer: Adam optimizer;
- Learning rate = 0.0005 (not mentioned in the paper);
- Early stopping patience = 5 (not mentioned in the paper).

To run the CIFAR-100 dataset, the number of classes was changed to 100.

5.3 Experimental setup

We didn't have access to a cluster to run our experiments, so instead we used Google Colab. Google Colab has a limit of how much GPU can be used in a day by a single user, so we had to take turns to run the code. In our code, we put that if the GPU can be found, the code should use GPU and if not it uses CPU. Training and running a model on the GPU is the best option in this case since it's optmized for matrix calculus, which is fundamental for neural networks (especially convolutional neural networks). To change from CPU to GPU in Colab, click "Runtime" at the top of the Colab workbook, then "Change runtime type" and select T4 GPU (the only one available for free).

Our code repository can be found here.

5.4 Computational requirements

Since we used Google Colab to run our experiments, the GPU resources varied with each run, so we just used the last run as the information to show the computational requirements used during the running of the experiments.

- GPU: Tesla T4;
- GPU Driver: 525.105.17;
- GPU Memory: 16GB.

We mostly used the GPU to run experiments, but because there was a limit on Colab, we sometimes had to use the CPU with a reduction of epochs and batch size, just to see what was working and what wasn't.

After all the training was done, we checked the memory usage and as can be seen in Figure 1, 3293 MiB were used and would be required to run the code (this is the memory usage for running on CIFAR-10 dataset).

5.5 Neural network

Among all the options used in the scientific paper as neural networks (AlexNet, VGG-13, ResNet-18) we decided to implement the VGG-13 only (due to time limits). The VGG-13 is a standard deep convolutional neural network architecture which consists of 13 layers: 10 convolutional layers and 3 fully connected layers divided as following:

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.105.17   Driver Version: 525.105.17   CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   74C    P0    32W /  70W |   3293MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
```

Figure 1: Memory Usage while running whole code

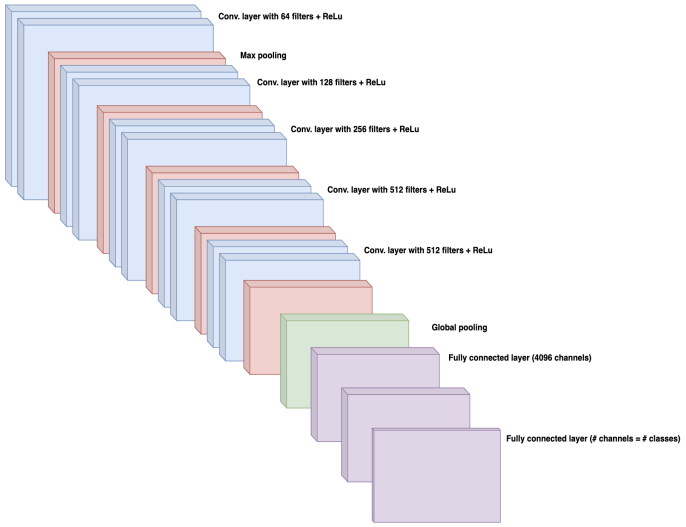


Figure 2: VGG13 architecture

Each convolutional layer has a kernel size of 3x3. Between each layer we implemented the ReLU activation function (typical for the VGG architecture) and there is the possibility to add batch normalisation layers (with the target to speed up the training and make learning easier). At the end of each convolutional block (after the ReLU activation of a pair of convolutional layers) a 2x2 max pooling is implemented. With the VGG-13 architecture and the shapes of the images $32\times32\times3$ equal for all the three datasets used, at the end of all the convolutional layers we reached a feature map of dimension $1\times1\times512$, thus in order to move to the fully connected layers we just flattened it into an array. These fully connected layers have optionally dropout layers in between, to help prevent overfitting.

The model is trained using the standard Adam optimizer and Cross-Entropy as loss function (typical for image classification). We introduced the option to add early stopping to avoid overfitting and to improve performance of the model on validation and test set. For the multi-branching models we used a joint Cross-Entropy loss function instead of the standard Cross-entropy, that is the sum of the Cross-Entropy for all the outputs that we'll have.

5.6 Early exit

We added to the basic VGG-13 model small convolutional models (classifiers) consisting of:

- one convolutional layer
- one ReLU activation function
- one max pooling
- one fully connected layer

A total of 9 auxiliary classifiers are used and placed after the max pooling of each convolutional layer.

Our implementation of the early exit decision criterion relies on the concept of entropy that measures the confidence or certainty of the network's prediction for a given input. The function `entropy(x)` calculates the entropy of the output probabilities using the softmax function to normalize the output tensor x. The entropy H is computed as the negative mean of the element-wise multiplication between the normalized probabilities and the logarithm of these probabilities with a small epsilon value added to prevent numerical instability. Specifically, the model may exit early if the entropy of the output probabilities falls below a certain predefined threshold (γ) (if H is lower than γ, the model may exit early.)

The EarlyStopping class provides functionality to halt training when certain conditions related to the validation loss improvement are not met. It keeps track of the best model based on the validation loss and stops training if the validation loss does not improve for a defined number of epochs (`patience`). If enabled, it restores the weights of the best model and stops the training process if the conditions are met.

Initially, if no best loss is set, the `best_loss` is updated, and the `best_model` is copied. Then, the function checks if the difference between the current `val_loss` and the `best_loss` is greater than the `min_delta`, which specifies the minimum change in validation loss to be considered as an improvement. If the difference is significant (greater than `min_delta`), it updates the `best_loss`, resets the counter, and saves the current model's state as the best model. If the difference is insignificant, it increments the counter to keep track of epochs without improvement. If the counter exceeds the patience value, training is stopped and the status is updated. If `restore_best_weights` is enabled, it restores the best model's weights before stopping the training. Finally, it updates the status to reflect the current counter and patience.

5.7 Challenges faced with code implementation

There were a number of challenge we faced while trying to reproduce the code.

One of the most important decision was deciding whether to use PyTorch or TensorFlow. Some of the implementations we saw online used TensorFlow for the early exiting, but most of our team members were more comfortable in using PyTorch. At first, we decided to use TensorFlow as there were more examples to refer to, but towards the end, we decided to switch to PyTorch.

The results showed overfitting occurred, but we thought that the early exiting implementation and training would serve as a regularization, and hence that this would prevent overfitting. To prevent this, we added batch normalisation and dropout, but this strategies were not mentioned in the paper, so we weren't sure how the authors handled overfitting in their experiments. In the end, we decided to not train the model with the strategies above, since the main goal of the proposed method was not to prevent overfitting, but to speed up the inference time.

Another difficulty that we faced, was in understanding exactly the proposed training algorithm. With our interpretation, by computing the loss of only the last soft-conditional output, we were not training the early exits at all, and hence we had to use the joint cross entropy loss function from Formula 2. By doing this the auxiliary classifiers were trained correctly. We also had to change the criterion that we used to early exit and to compute the soft-conditional outputs because the results with our interpretation, that used the entropy to compute it, were very poor. Thus, we moved to a different criterion, based on the highest probability of the predicted class of each image, and by doing this the results are much more similar to the ones obtained in the paper, and to the standard B-NET.

6 Results

This section discusses the results achieved in the paper. We only implemented the VGG-13 architecture on CIFAR-10 and CIFAR-100 datasets.

Table 1 shows the results of the test accuracy on the VGG-13 architecture using CIFAR-10 and CIFAR-100 datasets. Likewise to the paper, better test accuracy were obtained using the CIFAR-10 dataset, while the CIFAR-100 achieved the worst results. We were only able to obtain similar test accuracy to the one in the paper for CIFAR-10. We believe a number of factor comes into play, such the paper not giving enough details for some of the architecture implementation they used.

Table 1: Test accuracy on the datasets for the VGG-13 architecture

Architecture	Implementation	CIFAR-10	CIFAR-100
VGG-13	Baseline	77.40%	32.89%
	B-Net	78.94%	39.98%
	Proposed	77.43%	31.38%
	Prop+DifferentCriterion	79.51%	35.51%

We wanted to go into more detail like shown in the paper, so we decided to detail the average test accuracy, inference time and speed up based on the experiment of VGG-13 using CIFAR-10 dataset. This is shown in Table 2. We would like to emphasise that the reason our inference time differs from by a large amount is because of the difference in machine with the paper. The speed up was calculated by getting the inference time of a random image from the test set. The formula we used for the speed-up using the inference time taken is

$$\frac{\text{baseline inference time}}{\text{chosen implementation inference time}}$$

These values of course depends on the random image taken, since each model may early exit at a different stage, with a different accuracy. It's the case for example for the inference time using the proposed model (second entry of Table 2), where the model takes the second exit, but the result is wrong.

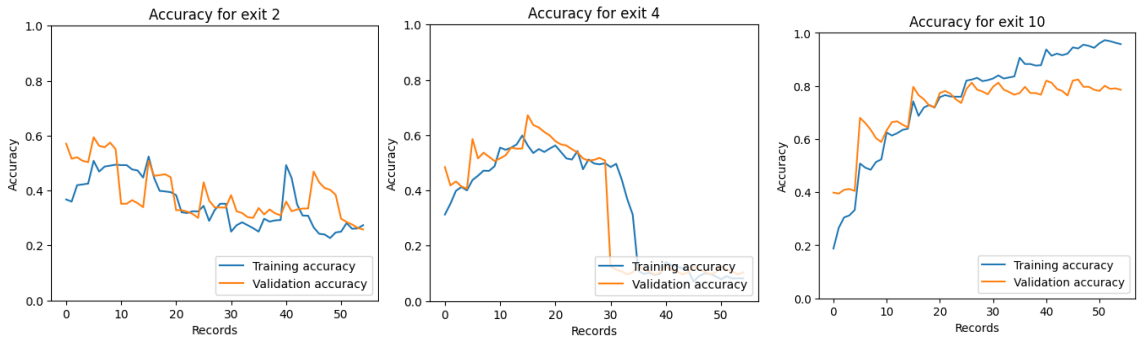
Table 2: Results of the proposed algorithm when enabling the early-exit strategy

Architecture	Implementation	Test Acc	Inference Time	Speed-up
VGG-13	Baseline	78.94%	6.06108 ms	-
	Proposed	78.94%	2.92706 ms	107.07%
	Prop+DifferentCriterion	78.11%	2.60544 ms	128.34%

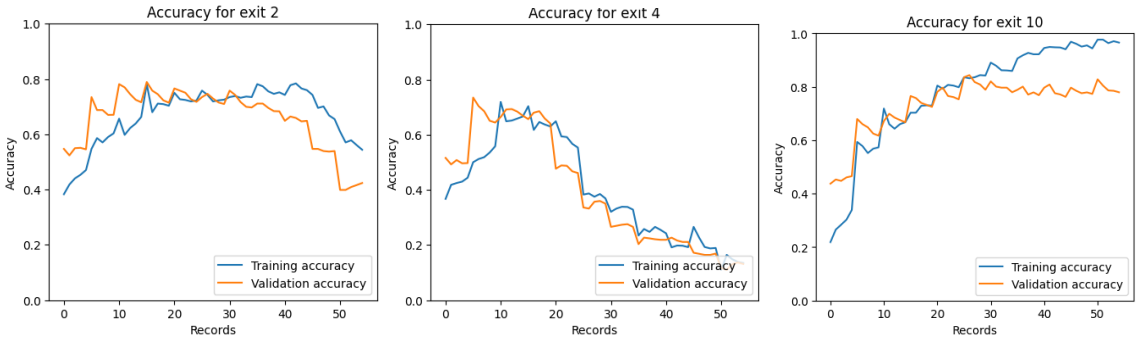
Figure 3 shows the image of the accuracy of exit branch 2, 4 and 10 for the authors proposed methodology and the author's proposed methodology but using a different decision criterion trained on VGG-13 architecture on CIFAR-10. We picked exit branch 2 because that was the branch we got to get the prediction and recorded inference time noted in Table 2, exit branch 4 because it was the only one that had a significant high drop in accuracy for both implementation and exit branch 10 as it is the last exit branch. For the proposed method early exit 2, the accuracy is much lower than the proposed method with a different criterion and as can be seen from the image, it records accuracy for the proposed method from approximately range 0.2 - 0.6. We noted that because of the effect in the reduction in accuracy, the proposed method gave a wrong prediction of airplane instead of ship, while the proposed method with a different criterion predicted it correctly (refer to VGG-13_Cifar10.py code).

In addition to the early exit, we had to fine tune our hyperparameters and conduct some experiments. We decided to leave the proposed method with a different decision criterion because we achieved higher and better results for that method compared to the implementation we did using their proposed method. Although even with the best solution we got, it did not achieve their results for CIFAR-10 and CIFAR-100.

The authors did not detail their computational requirements as well as go into as much detail as we would have liked, so the exact setup wasn't possible and not as straight forward to implement.



(a) Proposed early exit



(b) Proposed+DifferentCriterion early exit

Figure 3: Early exits Accuracy for 3 branches from the trained VGG-13 architecture on CIFAR-10

7 Discussion and conclusion

Although the results are not the exactly the same as the one in the paper, we were able to base our success of reproducing the code based on the similarity of the results. The second version where we implemented a different criterion gave better results. We think the reason we didn't achieve their results is that the paper is not very clear on their architecture specification. The paper did not specify what they used as a decision criterion, so this is one of the reasons we believe why we didn't achieve exactly their results. We would also like to note that the paper is not well detailed about what they have done and how they have done it and therefore it is difficult to perfectly execute to obtain their results what they are detailing in the proposed methodology.

We concluded that we were successfully able to reproduce the result based on the similarity and we believe that if we had more time, we would have been able to implement the other two models as well. We were able to reproduce some of the results from the paper based on the architecture that we chose, which is the VGG-13. The methods implemented by the authors of the paper, shows the importance of early branching in making more faster and accurate predictions as can be seen from even our own results where we were able to get a lower inference time and higher accuracy.

References

[1] Simone Scardapane, Danilo Comminiello, Michele Scarpiniti, Enzo Baccarelli, and Aurelio Uncini. Differentiable branching in deep networks for fast inference. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4167–4171, May 2020.

[2] Tao Jiang, Xiangyu Yang, Yuanming Shi, and Hao Wang. Layer-wise deep neural network pruning via iteratively reweighted optimization. pages 5606–5610, 05 2019.

[3] Simone Scardapane, Danilo Comminiello, Amir Hussain, and Aurelio Uncini. Group sparse regularization for deep neural networks. *Neurocomputing*, 241:81–89, 2017.

[4] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations, 2016.

[5] Alexios Lekidis and Panagiotis Katsaros. Model-based design of energy-efficient applications for iot systems. *Electronic Proceedings in Theoretical Computer Science*, 272:24–38, June 2018.

[6] Adam Seewald, Ulrik Pagh Schultz, Julius Roeder, Benjamin Rouxel, and Clemens Grelck. Component-based computation-energy modeling for embedded systems. SPLASH Companion 2019, page 5–6, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Enzo Baccarelli, Paola G. Vinueza Naranjo, Michele Scarpiniti, Mohammad Shojafar, and Jemal H. Abawajy. Fog of everything: Energy-efficient networked computing architectures, research challenges, and a case study. *IEEE Access*, 5:9882–9910, 2017.

[8] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size, 2016.

[9] Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. Branchynet: Fast inference via early exiting from deep neural networks, 2017.

[10] Xin Wang, Yujia Luo, Daniel Crankshaw, Alexey Tumanov, Fisher Yu, and Joseph E. Gonzalez. Idk cascades: Fast deep learning by learning not to overthink, 2018.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[12] Luke N. Darlow, Elliot J. Crowley, Antreas Antoniou, and Amos J. Storkey. Cinic-10 is not imagenet or cifar-10, 2018.