Dynamic Hierarchical Neural Network Offloading in IoT Edge Networks

Wassim Seifeddine, Cédric Adjih, Nadjib Achir

Dynamic Hierarchical Neural Network Offloading in IoT Edge Networks

Wassim Seifeddine
Inria
Saclay Center, Palaiseau, France
wassim.seifeddine@inria.fr

Cedric Adjih
Inria
Saclay Center, Palaiseau, France
cedric.adjih@inria.fr

Nadjib Achir
L2TI, Institut Galillee
Universite Sorbonne Paris Nord
Villetaneuse, France
nadjib.achir@univ-paris13.fr

***Abstract*—In recent developments in machine learning, a trend has emerged where larger models achieve better performance. At the same time, deploying these models in real-life scenarios is difficult due to the parallel trend of pushing them on end-users or IoT devices with strong resource limitations. In this work, we develop a novel technique for executing parts of a single model successively through multiple devices (IoT, edge, cloud) while respecting each device's resource limitations. For that, we introduce a new offloading mechanism where, during computation, a decision can be made to offload work, together with the ability to exit early in the computation with intermediate results. The decision itself is tuned through Deep Q-Learning.**

I. Introduction

One of the main reasons for the success of Deep Learning in the past decade is its ability to learn hidden patterns in datasets and use them to achieve the desired task. However, this success comes at an important cost. Indeed, Deep Learning neural networks models have been growing from a few hundred parameters to hundreds of millions [3] and even hundreds of billions of parameters [1]. This ever-increasing number of parameters also causes the computational abilities required for training to increase dramatically. A large number of parameters impacts both the training phase and the use of the trained model (inference), which are both computationally expensive: the training time becomes large, as is the necessary time and latency for performing inference. For achieving faster inference, methods have been proposed in the area of *model compression*; for instance, through parameter quantization or model pruning techniques [2]. Although such techniques help to reduce the model size and speed up inference, there remain issues when using deep models on devices with resource constraints, as commonly found in embedded systems and in IoT.

In this work, we provide a solution for embedded IoT systems, that uses support from edge or cloud servers, and we propose a hierarchical execution model. Our approach combines three main ideas: first, this idea of splitting the same neural network model across two or more devices according to their computational abilities; second, the idea of *early exit* [15], where execution of the model can be stopped at some intermediate layers, trading accuracy for computational cost; and third, the idea of *off-loading*, where, after a few intermediate layers, the computation can be continued from the device to the edge server. The cornerstone of our technique is a method to decide which action should be taken at every possible exit of the intermediate layers: *stay* (on device), *offload, exit*. To do so, we formulate this decision problem as a *Markov Decision Process*, and our method uses a reinforcement learning approach; precisely, we use deep reinforcement learning, where we train a smaller model, outputting a decision based on the output of the previously executed layers. Our method is capable of preserving the accuracy of the model while respecting the device's computational capabilities. It can also trade accuracy for matching various constraints in terms of energy cost, latency, etc. We validated our approach on a modified version of the VGG Network [13] trained on the CIFAR10 dataset [6], and observed excellent results.

The rest of the paper is organized as follows. Section II presents related work. Section III discusses our approach in detail. In Section IV we present our results. Finally, a conclusion is drawn in Section V, where we also look at the future directions.

II. Related Work

Apart from model compression techniques [2], putting "AI on the Edge" has proposals about finding ways of improving the execution model on devices and at the edge; computation offloading techniques and conditional computation techniques fall under this category as described in [14].

A. Conditional Computation

Conditional computation is a very recent and active research field in deep learning. It comes from the idea that one does not need to run the whole network in most cases to get decent, reliable predictions from an input x. In other words, given an input x, can we know which part of the model to run to get correct predictions with decent accuracy?

Many approaches are used to accomplish this. Some of the most notable ones are BranchyNets [15] which illustrated for instance, that for more than 90% of that CIFAR10 [6], it is not necessary to run all the layers of the neural network model (VGG16[13] in their case) for accurate predictions. SkipNets [16], and Slimmable Neural Networks [17] are also other more complex approaches for conditional computation. However, these networks usually require an external entity,

normally a sequential decision-making agent, to decide what actions to take.

Our work is focusing on conditional computation due to its simplicity and its high potential for efficient deep learning on edge devices.

B. Offloading Techniques

Offloading is a technique where an application or an end device delegates some of its computationally intensive tasks to a more capable remote entity. Offloading is considered an active research topic, which continues to attract an increasing number of contributions. Several recent works in the literature addressed the offloading decision problem. The main objective is to determine whether a task should be offloaded, on not. Some other work also covers the choice of the corresponding execution location, including either the terminal or an established set of edge or cloud servers, as represented in Fig. 1. One of the main criteria for such a decision is the completion time of the application. In addition, other constraints such as the terminal's processing or battery capacity, the available network bandwidth, or the remote servers' processing capacity and energy consumption were also considered [8], [7]. Another important aspect regarding offloading decisions is related to the application's single-task or multitasking architecture type.

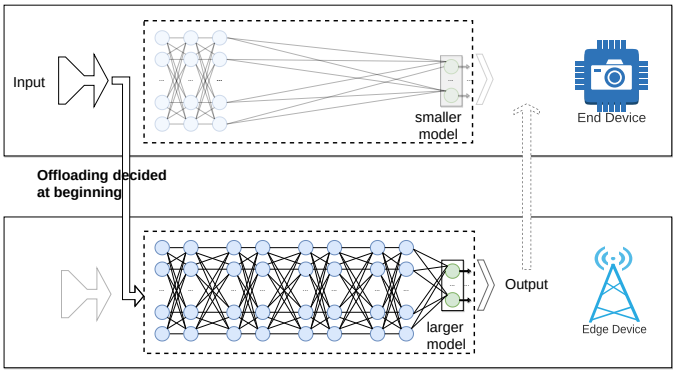

Fig. 1. Offloading of model inference: approaches with different models on different devices, depending on the device capability

More recently, some works focus on modern implementations of deep learning applications along the continuum device-edge-cloud. Instead of the old design of simply sending all the data in the Cloud, they propose to offload part or all of the inference on one dedicated edge server or across several remote servers [11], [12]. One of the most recent works, proposed in [14], the offloading decision relies on two objective functions. A first function (*Max-Accuracy*) aims to maximize the accuracy on an input x, and a second objective function (*Max-Utility*) that seeks to optimize the utilization of the resources at the end device.

The large research activity on the subject of how to deploy machine learning models on resource constraint devices, is an indicator of how important this field is in practice. However, the focus was not directed on how to run one model efficiently across multiple devices. For example, FastVA [14] focuses on how to balance the execution of several models with different capabilities across multiple devices depending on the need of the user. BranchyNet [15] and SkipNets [16] did not use the modularity characteristic of neural networks to divide the network on multiple devices. To our understanding, all previous work for "AI on the edge" did not tackle this problem.

III. Dynamic Hierarchical Neural Network Offloading (DHN²O) in IoT Edge Networks

Our work, Dynamic Hierarchical Neural Network Offloading (DHN²O), focuses on improving the use of neural network models for scenarios where a hierarchy of computational devices are present: the end-device itself, some edge server, and/or some cloud server. We combine the use of such a hierarchy with the framework of early exit networks, where the computation can be stopped after the execution of only some of the (first) layers. DHN²O proposes to divide the layers of a single neural network model in blocks as in early exit networks but then deploy them on the device, the edge, and/or the cloud, as represented in Fig. 2

Overall the inference operates as follows: the inference is performed going successively through each block of layers of the model starting from the first one. After the execution of each block of layers, a decision is made: it consists in selecting one of the possible actions: *stay* (continue inference to through the next set of layers), *offload* (send the intermediate output of the current layer to the edge or to the server, to continue the computation there), *exit* (stop the computation here and use the current output of the intermediate layers to make a classification decision). This improves on traditional early exit networks, which only consider the execution on a single device because we introduce a new kind of action "offload": deciding to offload the computation to another more capable computational device.

The other specificity of our method is that we are able to formulate different optimization targets as an optimization problem with constraints. Different performance is obtained depending on what decision is made at each block of layers. As described later, we cast this problem to a reinforcement learning framework and obtain heuristic solutions through Deep Reinforcement Learning. Combining early exit with offloading opens the avenue for many new possibilities, for numerous optimizations, and choices of tradeoffs. Indeed, the early exits give the ability to trade energy for precision, e.g., executing fewer layers, hence reducing the accuracy while reducing the energy cost of running the model, or the opposite. The offloading gives the ability to trade some communication cost for precision, e.g., by transmitting the output of the intermediate layers to the edge or cloud. Finally, some additional tradeoffs in terms of model execution time (i.e., latency) are interesting to analyze. In our system model, all of the various tradeoffs result from a simple set of decisions. This design also makes models very modular, leading to applying the same model for different devices but by splitting it differently.

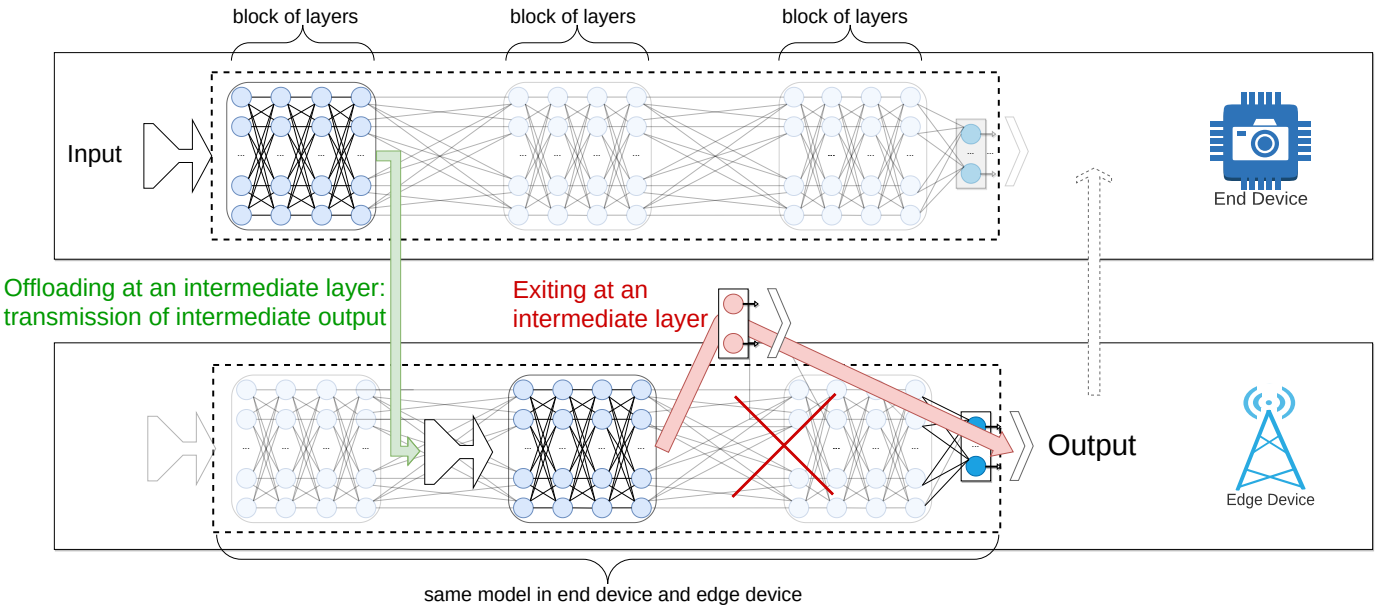

Fig. 2. Our approach DHN²O: same model divided on multiple devices. Offloading and early exit are possible after each block of layers

Another natural side benefit for our approach is also in terms of privacy. In previous work on offloading, the decision was either to run the whole model on the device or to offload all the data to be analyzed remotely by a model. In our case, we can force the model to complete several layers before offloading the computation. This may correspond to the feature extraction performed by the first layers, which leads to improved privacy, e.g., when the data contains personally identifying information. In this work, we do not attempt to optimize for some privacy metric, however, our system design allows us to simply consider privacy constraints.

A. Model Design

Our method applies to all neural networks models without memory[1]. In this article, we choose to apply it to the classical vision task of the object recognition: given a picture, the model has to identify the class of the main object appearing in the picture, from a predefined set of classes (e.g., for CIFAR-10: airplane, cat, dog, etc.). The input are the pixels of the image, and the output is a *prediction* vector, with one numerical value for each class, each representing the confidence of the model on the fact that the image represents an object from this class. The final classification result of the model is the class corresponding to the higher confidence in this prediction vector. In this section, we provide more details on the model that we built for our experiments.

We start by considering and designing an image classification model, inspired by the one already defined in the work on early exit networks [15]. The network is composed of two main parts. The *feature extractor* part which is responsible for extracting hidden features in an image is composed of a set of convolutional, max pooling, batch normalization layers with a ReLU as the activation function. The second part which we call the *classifier*, takes as input the output of the *feature extractor* and tries to classify these patterns into the appropriate class. *The classifier* is composed of a single linear layer that maps a vector of size 512, corresponding to the output of the *feature extractor*, to the prediction vector, an n sized vector where n is the number of classes we have. The "exits" of the model have the same architecture as the classifier part. Thus, we will consider the output of the model as being the last exit and will count it as an exit. We tried different combinations of how and where to add exit points to the model. We ended up adding 5 exits on different levels of the model graph. The first exit is on layer 3 and the last one is on layer 25. The model architecture is similar to VGG network [13]. The layer combination (convolutional, batch normalization, ReLu) are stacked one after another. We used a stack of 16 of these layers which we decided is a balance between model complexity and accuracy. We chose this architecture because we know that the VGG network[13] works well for image classification and is simple enough to experiment with. The same design can be applied to other architectures such as ResNet [4].

The key idea is finding the ideal placement of the exit points. To our understanding, there is are no straightforward automatic methods to achieve this, hence we proceeded experimentally. Fig. 3 is a simplified representation of the final model.

The model was trained and evaluated on the CIFAR10 vision dataset [6]. We used the recommended training setup for VGG [13] network from PyTorch [10]. The optimizer used is Adam [5] with a learning rate of 0.0003. The optimizer also had a weight decay factor of 0.001. The training was done on an A100 NVIDIA GPU with 40GB of VRAM. This amount of VRAM allowed us to have a relatively big batch size of

[1]Hence, we exclude Recurrent Neural Networks (RNN), etc.

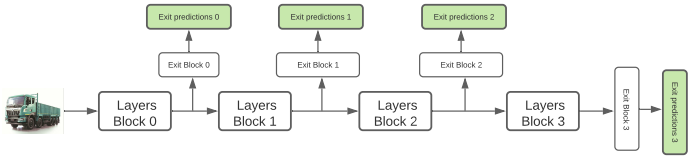

Fig. 3. Early Exit Network with 4 exits

4096 images for training and $8,192$ for validation. Training early exit networks [15] is not identical to training regular neural networks, the difference is that in early exit networks each exit outputs a vector of predictions (or logits) and this vector comes with its own loss due to the error in predictions. Usually in *normal* deep neural networks, one has one output layer which has these predictions, so one calculates the loss and back-propagates the loss back to all of the layers using the chain rule to adjust the weights using a gradient descent technique. However, in our design we have multiple exits thus multiple losses: by computing a weighted sum of these losses as the final loss, the autograd engine of PyTorch [10] can be used seemlessly to automatically perform classical backpropagatation.

Our goal behind training this network is not to achieve state-of-the-art on CIFAR10 [6] but to have a decent classification model as a proof of concept, that we can use later and validate our approach. We measured the training accuracy, training loss, validation accuracy, validation loss on every exit in the network. The results are presented in table I.

Exit Number	Training Loss	Training Accuracy %	Validation Loss	Validation Accuracy %
0	0.6921	78.664	1.072	65.8
1	0.4519	87.78	0.8281	74.44
2	0.02463	99.972	0.8131	79.31
3	0.0006934	100	0.7755	82.59
4	0.00001995	100	1.081	82.51
5	0.0001207	100	0.8971	82.48

TABLE I
PERFORMANCE RESULTS OF OUR TRAINED IMAGE CLASSIFICATION MODEL (AT EACH EXIT)

B. Optimization Problem

After training the image classification model, we need to determine the best decisions to select, corresponding to solving an optimization problem. At each exit point, the system has to choose an action from the set actions: {*Stay, Exit, Offload*}. *Stay* means that the network should continue to be executed on the same device. *Exit* means that the network should exit execution completely and returns the current results if any. *Offload* means that the current device should delegate the execution of the model to a device higher in the device hierarchy and stop executing it locally. The problem is complex since it depends on multiple factors and considerations, summarized below:

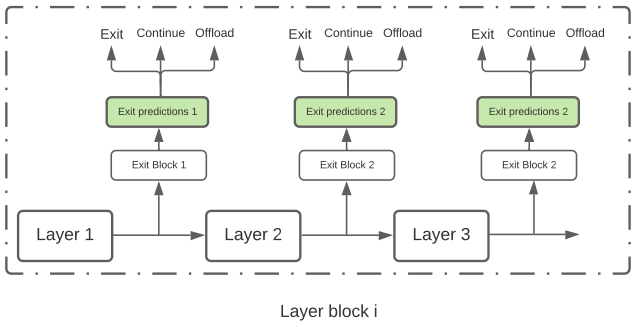

Fig. 4. Sample exit points: a decision has to be taken at every branch

- *Stay*:
  - Does the device have enough energy to continue executing the model to the next exit?
  - Does the accuracy achieved so far seem to be good enough or should one wait until the next exit, hoping the accuracy will be higher?
  - Is the current prediction likely to be correct? or is the model confident but selects the wrong class?
  - If offload is selected, can more accuracy be obtained with less energy consumption?
- *Exit*:
  - Should the inference exit because of sufficient accuracy or because the time deadline is approaching?
  - Should the inference exit even though low accuracy seems low, or maybe offloading is better?
  - How confident is the model about the prediction?
- *Offload*:
  - Does the device have enough energy to offload the data now?
  - Can the device offload the current data with the network bandwidth available without depleting my energy?
  - When offloading, can the time deadline still be respected?

As shown, the set of decisions and consideration is large, hence one has to formulate a proper optimization problem, with objective functions and constraints.

The solution technique presented later can be applied to any optimization problem with a scalar objective function and with arbitrary constraints. In the following, we use the following formulation:

$$\begin{aligned} \max \ & objective\ function\ f(energy\ cost, accuracy) \\ \text{s.t. } & time\ consumed < time\ deadline \\ & energy\ consumed < energy\ limit \end{aligned} \quad (1)$$

It matches perfectly the problem of inference on devices that have energy constraints: it expresses a tradeoff between consumed energy and the accuracy of the model. Hence we want to maximize a function f of $energy\ cost$ and $accuracy$. At the same time, there might be additional constraints, such

as maximum latency, and constraints in terms of the energy budget.

The decision variables of this optimization problem are the decisions taken at each exit. Notice that importantly, that the accuracy is never really known by the model. An estimate can only be guessed from the prediction vector at the current exit (and the vectors of the prior exits). Determining the probability that the current classification is correct is not straightforward. The decisions are also taken in sequence: this allows us to model the decision process as a simple *Markov Decision Process*, where the observed state consists of the current exit index, the prediction vectors of the current exit, and of the previous ones. Optionally, dynamically varying parameters (such as transmission energy costs, etc.) could be added to this state.

When solving a Markov Decision Process, it is also crucial to understand the nature of the variables, and whether they are continuous or discrete, finite or infinite. In our case, the states are continuous mainly due to the fact the vectors of predictions v_n is a vector of floating numbers bounded in $[0, 1]$. We have the following components of the *observed state*:

- Predictions: a vector p of size n where p_i states how confident the model is that this data is of class i at e_i
- Energy consumption until e_i
- Time consumed until exit e_i
- Network bandwidth available
- (Optionally) any other dynamically changing parameters

C. Reinforcement Learning Solution

We decided to solve this Markov Decision Process (MDP) through reinforcement learning. Due to the non-trivial observed state, and its continuous nature, we opted for using one deep reinforcement learning algorithm, Deep-Q Network (DQN) [9]. Hence DHN²O uses a (much) smaller neural network model (denoted *DQN model*), that takes as input the specified observed state and outputs a decision for the current exit (stay, exit, offload).

This choice has an important, very powerful, property: the DQN model actually takes the prediction vectors as part of its input, and hence we are training this model to also implicitly evaluate how reliable the decision vectors are. Notice that [15] had to resort to heuristics (with arbitrary thresholds), to decide on early exits: we implicitly use a deep learning approach to this.

For our practical experiments, we selected a simple objective function f that weights linearly (energy) cost and accuracy:

$$f(energy\ cost, accuracy) = -\alpha \times energy\ cost + \beta \times (prediction\ is\ correct) \quad (2)$$

where α and β are constant determined from an actual system.

Our MDP is episodic: an episode starts with an initial image and consists in taking successively one decision at each possible exit until a final decision to actually exit is selected. Since the episodes are short (i.e. the number of steps is at most the number of exits), we compute a reward for the whole episode, at the end of the episode. We do not use discounting. The reward itself is computed as follows: it is a large negative number if the constraints are not met at the end of the episode, otherwise, it is simply the objective function:

$$Reward = f(cost, accuracy) \quad (3)$$

Note that obviously, by design, the training will tend to maximize our objective function while satisfying the constraints.

The training data for this DQN is collected during the training of the model discussed in III-A. Prior to training the DQN, at the end of the training of the image classification model, the image model is run on all images. For each image, we collect the prediction vector of all exits (along with the correctness of the decision).

IV. Experimental Results

We did not experiment DHN²O on real systems, but as a proof of concept, we ran experiments with simplified assumptions to illustrate its good behavior. We used an existing DRL framework with an implementation of DQN, obtained from `stable-baselines` (itself derived from the `baselines` of OpenAI).

For the objective function, we start by setting the coefficient α to 1 and β becomes directly the tradeoff between accuracy and energy efficiency. The β parameter is application-related, should be obtained from a real system, and is an input for solving the optimization problem. We also ignore all the constraints. We equate the energy cost to the exit index (i.e. processing one block of layers costs exactly 1 energy unit).

For our proof of concept, we only want to illustrate that some good decisions are taken by the DQN model. Thus we artificially set β to some values, based on the results of the validation accuracy found in table I. It is easy to see that the DQN model should stop approximately at exit i and exit $i+1$ when β has the following value:

$$\beta_{i,i+1} = \frac{1}{(error\ rate_i - error\ rate_{i+1})} \quad (4)$$

After selecting β, the DQN model is then trained for 10,000 episodes across all the training data. Fig. 5 represents the training loss (a measure of the error made by the DQN model), during the training phase. We can see that it converges quickly after a few thousand iterations.

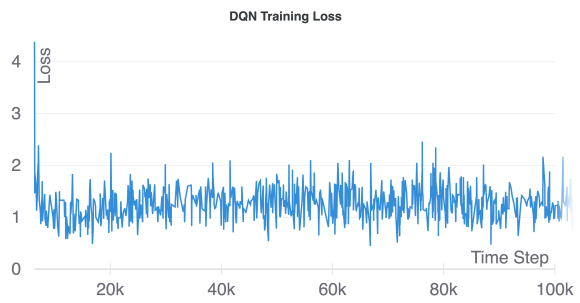

Fig. 5. DQN training loss monitored

Then, we trained the DQN model for different values of β, selected according to (4). We are presenting the results for a value of β where we can expect to exit at exit one or two.

In order to check if the DQN model is performing as expected (hence, is close to optimizing the given objective function), we simply need to check after how many exits, on average, the DQN model is selecting the action "exit". Incidentally, the DRL software that we use also records the average episode length during training, which corresponds to exactly this number. Hence, they are shown in the Fig. 6. We

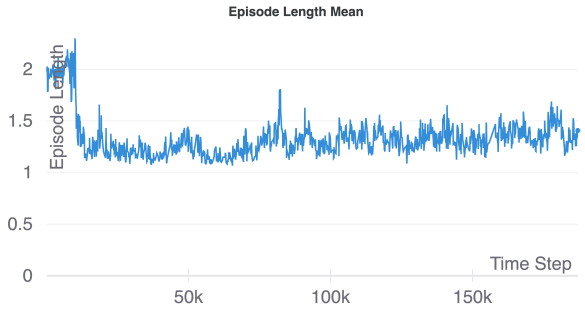

Fig. 6. For β with expected exit betweem first and second

can see that for the selected value of β in Fig. 6, the model is indeed exiting on average at exit 1 and at exit 2.

V. Conclusion and Future Work

In this paper, we presented DHN²O, a novel approach for model splitting across multiple devices. Our contribution lies in a new framework that would allow users to deploy a single model split into multiple devices having multiple computational capabilities instead of using multiple models based on the capabilities of the hosting device. We also introduced an optimization problem that we solve for reinforcement learning heuristics. We obtain a policy that decides the action at each possible dynamic exit: *Stay*, *Exit*, or *Offload*. We solved this optimization problem using a DQN [9] and the good obtained results validated our approach.

As future work, we intend to explore how to introduce dynamically accuracy weights for classes, and more generally other parameters, as additional inputs to the DQN model. Having different accuracy requirements for different classes allows to adapt for different applications (e.g. accurate human detection is more important in intrusion detection applications). Future work also includes the use of parameters, cost, and constraints from real systems in the optimization problem, and includes real implementation.

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[2] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *CoRR*, abs/1710.09282, 2017.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[5] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[6] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

[7] Houssemeddine Mazouzi, Nadjib Achir, and Khaled Boussetta. Maximizing mobiles energy saving through tasks optimal offloading placement in two-tier cloud. In *Proceedings of the 21st ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pages 137–145. ACM, 2018.

[8] Houssemeddine Mazouzi, Nadjib Achir, and Khaled Boussetta. Dm2-ecop: An efficient computation offloading policy for multi-user multi-cloudlet mobile edge computing environment. *ACM Transactions on Internet Technology (TOIT)*, 19(2):24, 2019.

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[10] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[11] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen. Deepdecision: A mobile deep learning framework for edge video analytics. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 1421–1429, 2018.

[12] Xukan Ran, Haoliang Chen, Zhenming Liu, and Jiasi Chen. Delivering deep learning to mobile devices via offloading. In *Proceedings of the Workshop on Virtual Reality and Augmented Reality Network*, VR/AR Network '17, page 42–47, New York, NY, USA, 2017. Association for Computing Machinery.

[13] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[14] Tianxiang Tan and Guohong Cao. Fastva: Deep learning video analytics through edge processing and npu in mobile, 2020.

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

[16] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E. Gonzalez. Skipnet: Learning dynamic routing in convolutional networks, 2018.

[17] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. Slimmable neural networks, 2018.