DYNAMIC SPLIT COMPUTING FOR EFFICIENT DEEP EDGE INTELLIGENCE

Arian Bakhtiarnia*, Nemanja Milošević†, Qi Zhang*, Dragana Bajović‡, Alexandros Iosifidis*

ABSTRACT
Deploying deep neural networks (DNNs) on IoT and mobile devices is a challenging task due to their limited computational resources. Thus, demanding tasks are often entirely offloaded to edge servers which can accelerate inference, however, it also causes communication cost and evokes privacy concerns. In addition, this approach leaves the computational capacity of end devices unused. Split computing is a paradigm where a DNN is split into two sections; the first section is executed on the end device, and the output is transmitted to the edge server where the final section is executed. Here, we introduce dynamic split computing, where the optimal split location is dynamically selected based on the state of the communication channel. By using natural bottlenecks that already exist in modern DNN architectures, dynamic split computing avoids retraining and hyperparameter optimization, and does not have any negative impact on the final accuracy of DNNs. Through extensive experiments, we show that dynamic split computing achieves faster inference in edge computing environments where the data rate and server load vary over time.

1. INTRODUCTION
The combination of deep learning and Internet of Things (IoT) has tremendous applications in fields such as healthcare, smart homes, transportation and industry [1]. However, deep learning models typically contain millions or even billions of parameters, making it difficult to deploy these models on resource-constrained devices. One solution is to offload the computation to an edge or cloud server [2], as shown in Figure 1 (b). However, since the size of the inputs to deep learning models can be massive, particularly images and videos, this approach consumes a lot of bandwidth and energy, and leads to delays. Moreover, even though IoT devices are limited, they still possess computational capabilities that remain unused when the entire computation is offloaded, and utilizing these capabilities would reduce the load on the servers. In addition, in applications that process personal data such as health records, or in audio or visual streams with voice activity or human presence, privacy regulations such as European Union’s GDPR [3] or United States’ HIPAA [4] may apply. These regulations typically forbid direct access to non-anonymized data, leaving the options to either anonymize the data at the cost of additional computation and higher latency, or process the data at the source.

Split computing, depicted in Fig. 1 (c), alleviates these issues by splitting the deep model into a head section and a tail section [5]. The head model is executed on the device, and its output (the intermediate representation at that particular layer of the deep network) is transmitted to the server, then processed by the tail model to obtain the final output. In a way, split computing is a partial offloading of the computation, as opposed to the full-offloading approach. Another benefit of split computing over full-offloading is that it can be used as a privacy preserving technique since intermediate representations are being transmitted instead of the actual inputs, and the original inputs cannot be easily reconstructed from the intermediate representations [6]. In addition, split computing can be combined with early exiting in order to obtain an early result on the device [7, 5], as illustrated in Figure 1 (c), which is useful when transmission takes longer than expected.

Since split computing aims to decrease the communication cost, natural bottlenecks, which are the layers of the deep network where the size of the intermediate representation is smaller than the input size, can be used as splitting points for deep learning models. In this paper, we show that unlike older popular models, state of the art models such as EfficientNet [8, 9] possess many natural bottlenecks. Based on this fact, we propose a method called dynamic split computing where the best splitting point is automatically and dynamically determined based on input and channel conditions, as shown in Figure 1 (d). Since the underlying deep learning model is not modified, dynamic split computing can be used as a plug-and-play method, meaning it can be employed without domain knowledge about the particular deep learning models that are being used. It is important to note that dynamic split computing is a complimentary efficient inference method that can be used in combination with other approaches, including model compression techniques such as pruning and quantization [10], as well as dynamic inference methods such as early exiting [11]. Our code is available at https://gitlab.au.dk/maleci/dynamicsplitcomputing.
Several approaches for speeding up the inference of deep neural networks (DNNs) on resource-constrained devices exist in the literature. Local computing performs the entire computation on the device, yet modifies the architecture of the neural network in order to decrease the required computation, while causing a minimal negative impact on the accuracy. Lightweight models such as MobileNet [12, 13, 14] are specifically designed to be deployed on such limited devices, whereas model compression techniques [15] alter existing architectures in order to make them more lightweight, for instance, pruning removes the less impactful parameters (weights) of the neural network; quantization uses less bits to represent each parameter [16]; and knowledge distillation aims to train a more compact model to reproduce outputs similar to a given larger neural network [17].

Dynamic inference methods [18] can alter the architecture of existing neural networks to adapt their inference time at the cost of accuracy, meaning they will produce more accurate outputs the longer they are allowed to execute. Various dynamic inference approaches exist, such as early exiting [7], where early exit branches are added after intermediate layers of a DNN that produce an output similar to the final output; layer skipping [19, 20, 21], where the execution of some DNN layers are skipped; and channel skipping [22], where less impactful channels of convolutional layers are ignored.

Even with local computing techniques, many high performing DNNs exceed the computational capacity of devices, especially when the output is expected within a strict deadline. In such cases, the computation can be offloaded to external servers. When the computation of a DNN is offloaded, the inputs must be transmitted from a device to a server, yet this can introduce massive delays during data transmission, particularly when the input size is large, which may defeat the original purpose of speeding up the computation. This has led to a recent emerging paradigm called edge computing [23] where the computation is offloaded to edge servers located much closer to end devices compared to cloud servers which are often located in data centers. Even though edge computing reduces the transmission delay, it still has some drawbacks. First, since the original inputs are being transmitted over a network, privacy issues arise. Furthermore, since typically multiple end devices are connected to the same edge server, if all of them offload their computation simultaneously, the edge server may experience a high load while the computational resources of each end device remain unused.

Split computing [5] (also known as collaborative intelligence) is an alternative approach that provides a balance between local computing and full-offloading, where some layers of the DNN are executed on the end device and the intermediate output is then sent over to the edge server where it is processed by the rest of the DNN layers. When the splitting point is chosen such that the size of the intermediate representation is lower than the input size, the transmission delay will consequently be lower than that of full-offloading. However, not all deep learning models possess such natural bottlenecks, and even if they exist, they may be located in the final layers of the network where the bulk of the computation has already been carried out, and it would not be sensible to offload the remaining computation. For instance, widely used models such as ResNet [24] and Inception [25] do not contain natural bottlenecks in their early layers [5]. In such cases, bottleneck injection can be used, where the architecture of the network is modified to artificially insert a bottleneck [5]. However, this approach requires time-consuming operations such as retraining the model and optimizing hyperparameters such as the size of the inserted bottleneck. Furthermore, there is no guarantee that the new architecture can obtain an accuracy comparable to that of the original architecture, particularly when a limitation such as a small bottleneck is imposed. Therefore, bottleneck injection is far from ideal.

The idea of dynamically determining splitting points has been explored in the literature [26, 27]. However, these methods do not consider bottlenecks and treat all layers as viable splitting points, adding unnecessary overhead to the decision process. In addition, using bottlenecks reduces the possibility of inferring the raw data from compressed representations, which is crucial in privacy-preserving applications.

3. DYNAMIC SPLIT COMPUTING

We assume a trained high-performing DNN is to be deployed on a device with access to a server, where the data rate of the communication channel and the number of inputs in the batch may vary. The variations in the data rate may be due
to fluctuations in wireless channel or traffic congestion, and the variations in batch size may occur due to different workloads at different times. The goal of our method is to optimize the end-to-end inference time by dynamically detecting the best splitting point for a given DNN based on the communication channel state and batch size. Since we aim to design our method in a “plug-and-play” manner, such that it can be deployed in edge computing systems without creating trade-offs involving the accuracy or the hassle of retraining, we avoid altering the underlying architecture or lossy compression techniques that may affect the accuracy.

Formally, neural networks can be formulated as \( f(x) = f_L(f_{L-1}(\ldots f_1(x))) \) where \( x \) is the input, \( L \) is the total number of layers in the neural network and \( f_i \) is the operation performed at layer \( i \). The intermediate representation at layer \( i \), which is the output of the \( i \)-th layer is recursively formulated as \( h_i = f_i(h_{i-1}) \) where \( h_0 = x \) is the input. Based on this notation, with split computing at layer \( j \), the head and tail parts of the DNN are denoted by \( f^h_i(x) = f_j(f_{j-1}(\ldots f_1(x))) \) and \( f^t_i(x) = f_{L}(f_{L-1}(\ldots f_{j+1}(h_j))) \), respectively, and \( h_j \) is the intermediate representation that is transmitted.

The first step is to find the natural bottlenecks of the DNN by calculating the compression ratio \( c_l = |h_l|/|x| \) for each layer \( l \) where \( |h_l| \) and \( |x| \) denote the size of intermediate representation at layer \( l \) and the input size, respectively. If \( c_l < 1 \), layer \( l \) is a natural bottleneck of the DNN. However, not all natural bottlenecks are useful in split computing. We define \( T^h_{i,j} \) and \( T^t_{i,j} \) as the inference time from layer \( i \) up to and including layer \( j \) \((i < j)\) of the deep neural network measured on the device and the server, respectively. Note that \( T^h_{i,j} \) may vary with server load, and thus might need to be measured and updated in regular intervals. When layers \( m \) and \( n \) \((m < n)\) have the same compression ratio, in other words when \( c_m = c_n \), the total end-to-end inference time with split computing at layer \( m \) and layer \( n \) are

\[
T^m = T^h_{1,m} + c_m T_{\text{IO}} + T^h_{m+1,n} + T^t_{n+1,L},
\]

\[
T^n = T^h_{1,m} + T^h_{m+1,n} + c_n T_{\text{IO}} + T^t_{n+1,L}.
\]

where \( T_{\text{IO}} \) is the transmission time of the entire input in full-offloading. Assuming the computational resources of the server are greater than that of the device, then \( T^m_{m+1,n} > T^n_{m+1,n} \), thus it is favorable to choose the earlier layer as splitting point. Consequently, only natural bottlenecks with compression ratio lower than all previous natural bottlenecks are useful. We call such bottlenecks compressive. Compressive natural bottlenecks are defined by

\[
C = \{ j | c_j < 1, c_j < c_i \ \forall i < j \}.
\]

The total end-to-end inference time for a given batch of inputs when the splitting point of the network is \( l \) is

\[
T_l = T^h_{1,l} + \frac{D c_l}{r} + T^t_{l+1,L},
\]

where \( D \) is the data size of the original input, \( c_l \) is the compression ratio of the intermediate representation at layer \( l \) and \( r \) is the data rate of the communication channel. When inputs are images or video frames, the total load in bytes can be calculated as \( D = BWHC \), where \( B \) is the batch size, \( W \) and \( H \) are the width and height of the images, and \( C \) is the number of channels in the images, for instance, \( C = 3 \) for color images and \( C = 1 \) for grayscale.

We define the end-to-end inference time in case of no-offloading as \( T_L = T^h_{1,L} \) and in case of full-offloading as

\[
T_0 = \frac{D}{r} + T^t_{1,L}.
\]

Therefore, the optimal splitting point \( s_{\text{opt}} \) can be determined by optimizing for

\[
s_{\text{opt}} = \arg \min_{l \in \{0\ldots L\}}(T_l).
\]

Dynamic split computing finds the optimal split location for a given data rate and batch size based on Eq. (6) at each time step, and switches to that configuration. When full-offloading cannot be used, for instance, due to privacy requirements, the range in Eq. (6) is reduced to \( \{0 \ldots L - 1\} \). Note that based on previous arguments, only compressive natural bottlenecks need to be investigated, therefore once all compressive natural bottlenecks are identified, we calculate the optimal splitting point for each batch size and data rate by measuring the inference time of head and tail models for each compressive bottleneck. It is important to note that the relationship between inference time of head or tail model and batch size is not strictly linear, therefore it needs to be measured for each batch size. Additionally, when the data rate is too low, it may not be sensible to use any form of offloading since it introduces too much delay. Therefore, dynamic split computing considers the no-offloading option alongside the optimal splitting point and switches between split computing and no-offloading when necessary.

Since different applications and environments may have different ranges for data rate and batch size and a unique pattern for their variations, we need a method to measure how beneficial dynamic split computing is in each specific case. We define a scenario as a sequence of the state of the environment throughout time, i.e., \( S = ((B_1, r_1), (B_2, r_2), \ldots, (B_T, r_N)) \), where \( B_i \) and \( r_i \) are the batch size and data rate at time step \( i \), respectively. The relative average gain of dynamic split computing in terms of end-to-end inference time over a specific method, for instance, static split computing at a specific location, can then be calculated by

\[
G = \frac{1}{N} \sum_{1 \leq i \leq N} \frac{|T^i_{s_{\text{opt}}(B_i, r_i)} - T^{SS}(B_i, r_i)|}{T^{SS}(B_i, r_i)},
\]

where \( T^i_{s_{\text{opt}}(B_i, r_i)} \) and \( T^{SS}(B_i, r_i) \) are the end-to-end inference time using dynamic and static split computing, respectively, with batch size \( B_i \) and data rate \( r_i \).
We investigate 14 modern DNN architectures: 7 variations of EfficientNetV2 [9] and 7 variations of EfficientNetV1 [8]. All these architectures were originally designed for image classification and have since been applied to various other problems such as speech recognition [28]. The accuracy of these architectures on ImageNet [29] ranges from 77.1% to 85.7%.

First, we find the compressive natural bottlenecks for each architecture. The number of natural bottlenecks in these architectures ranges from 15 to 68, three to four of which are compressive. For comparison, VGG-16 [30], which is an older architecture, has only 5 natural bottlenecks. Then, we find the optimal splitting point for each architecture in a wide range of states. We check data rates ranging from 1 MBps to 128 MBps and batch sizes of 1 to 64. Some larger models such as EfficientNetV2-L run into memory issues with large batch sizes, therefore, we reduce the maximum batch size to 32 or 24 in such cases. For the edge server, we use an Nvidia 2080 Ti GPU, and in order to simulate a resource-constrained device, we underclock the same type of GPU to 300 MHz (the normal GPU frequency is around 1800 MHz). Since the only factor that impacts $s_{opt}$ (in Equation 6) is the inference speed of the GPU, this simulation is very realistic for our purposes. The inference speed of this underclocked GPU is 2.37 TFLOPS, similar to the Nvidia Jetson TX2 and Xavier family, popular in edge computing systems, which range from 1.33 to 5.25 TFLOPS.

Figures 2, 3 and 4 show the results for the EfficientNetV1-B4, EfficientNetV2-S and EfficientNetV2-L architectures, respectively, where for each state (data rate and batch size), the optimal split location derived based on Equation 6 is specified. Observe that each compressive bottleneck is an optimal split location in several states. Moreover, no-offloading is the optimal solution in some other states. Therefore, dynamically switching between split locations (as well as no-offloading) based on the state of the communication channel improves inference speed. This is also the case with the other 10 investigated architectures. Figure 5 depicts the average relative gain of dynamic split computing in terms of end-to-end inference time in the EfficientNetV1-B4 architecture, averaged over all other methods (static split computing at different blocks as well as no offloading).

Fig. 4. Optimal split location based on batch size and data rate for the EfficientNetV2-L architecture.

Fig. 5. The relative gain of dynamic split computing in terms of end-to-end inference time in the EfficientNetV1-B4 architecture, averaged over all other methods (static split computing at different blocks as well as no offloading).

Fig. 2. Optimal split location based on batch size and data rate for the EfficientNetV1-B4 architecture.

Fig. 3. Optimal split location based on batch size and data rate for the EfficientNetV2-S architecture.

Fig. 4. Optimal split location based on batch size and data rate for the EfficientNetV2-L architecture.

5. CONCLUSION

In this paper, we showed that dynamic split computing improves in inference time over both no-offloading and split computing with a fixed split location. Moreover, as opposed to full-offloading, dynamic split computing can decrease the load on the server by performing parts of the computation on the device. Finally, by transmitting intermediate representations instead of inputs, dynamic split computing circumvents privacy issues that arise when using full-offloading.
6. REFERENCES

[1] X. Ma, T. Yao, M. Hu, Y. Dong, W. Liu, F. Wang, and J. Liu, “A survey on deep learning empowered iot applications,” IEEE Access, vol. 7, pp. 181721–181732, 2019.

[2] X. Wang, Y. Han, V. C. M. Leung, D. Niyato, X. Yan, and X. Chen, “Convergence of edge computing and deep learning: A comprehensive survey,” IEEE Communications Surveys Tutorials, vol. 22, no. 2, pp. 869–904, 2020.

[3] European Commission, “2018 reform of EU data protection rules,” Online at https://ec.europa.eu/info/law/law-topic/data-protection/data-protection-eu_en.

[4] Centers for Medicare & Medicaid Services, “The Health Insurance Portability and Accountability Act of 1996 (HIPAA),” Online at http://www.cms.hhs.gov/hipaa/, 1996.

[5] Y. Matsubara, M. Levorato, and F. Restuccia, “Split computing and early exiting for deep learning applications: Survey and research challenges,” arXiv:2103.04505, 2021.

[6] H.-J. Jeong, I. Jeong, H.-J. Lee, and S.-M. Moon, “Computation offloading for machine learning web apps in the edge server environment,” in 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), 2018, pp. 1492–1499.

[7] S. Scardapane, M. Scarpiniti, E. Baccarelli, and A. Uncini, “Why should we add early exits to neural networks?,” Cognitive Computation, vol. 12, no. 5, pp. 954–966, Sep 2020.

[8] M. Tan and Q. Le, “EfficientNet: Rethinking model scaling for convolutional neural networks,” in International Conference on Machine Learning, K. Chaudhuri and R. Salakhutdinov, Eds. 09–15 Jun 2019, vol. 97 of Proceedings of Machine Learning Research, pp. 6105–6114, PMLR.

[9] M. Tan and Q. V. Le, “Efficientnetv2: Smaller models and faster training,” arXiv:2104.00298, 2021.

[10] T. Choudhary, V. Mishra, A. Goswami, and J. Sarangapani, “A comprehensive survey on model compression and acceleration,” Artificial Intelligence Review, vol. 53, no. 7, pp. 5113–5155, Oct 2020.

[11] A. Bakhtiariqia, Q. Zhang, and A. Iosifidis, “Multi-exit vision transformer for dynamic inference,” The 32nd British Machine Vision Conference (BMVC 2021), 2021.

[12] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” arXiv:1704.04861, 2017.

[13] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “Mobilenetv2: Inverted residuals and linear bottlenecks,” in Computer Vision and Pattern Recognition (CVPR), June 2018.

[14] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, and H. Adam, “Searching for mobilenetv3,” in International Conference on Computer Vision (ICCV), October 2019.

[15] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, “Model compression and acceleration for deep neural networks: The principles, progress, and challenges,” IEEE Signal Processing Magazine, vol. 35, no. 1, pp. 126–136, 2018.

[16] T. Liang, J. Glosnner, L. Wang, S. Shi, and X. Zhang, “Pruning and quantization for deep neural network acceleration: A survey,” Neurocomputing, vol. 461, pp. 370–403, 2021.

[17] J. Gou, B. Yu, S. J. Maybank, and D. Tao, “Knowledge distillation: A survey,” International Journal of Computer Vision, vol. 129, no. 6, pp. 1789–1819, Jun 2021.

[18] X. Han, G. Huang, S. Song, L. Yang, H. Wang, and Y. Wang, “Dynamic neural networks: A survey,” arXiv:2102.04906, 2021.

[19] A. Graves, “Adaptive computation time for recurrent neural networks,” arXiv:1603.08983, 2016.

[20] A. Banino, J. Balaguer, and C. Blundell, “Pondernet: Learning to ponder,” arXiv:2107.05407, 2021.

[21] X. Wang, F. Yu, Z.-Y. Dou, T. Darrell, and J. E. Gonzalez, “Skipnet: Learning dynamic routing in convolutional networks,” in European Conference on Computer Vision (ECCV), September 2018.

[22] X. Gao, Y. Zhao, L. Dudziak, R. D. Mullins, and C. Xu, “Dynamic channel pruning: Feature boosting and suppression,” in International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6–9, 2019. 2019, OpenReview.net.

[23] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, “Mobile edge computing: A survey,” IEEE Internet of Things Journal, vol. 5, no. 1, pp. 450–465, 2018.

[24] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in Computer Vision and Pattern Recognition (CVPR), June 2016.

[25] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the inception architecture for computer vision,” in Computer Vision and Pattern Recognition (CVPR), June 2016.

[26] J. Karjee, K. Anand, P. N. S, R. B. V. Dabbiru, C. S. Byagdi, and S. N, “Dynamic split computing of posenet inference for fitness applications in home iot-edge platform,” in 2022 14th International Conference on Communication Systems & Networks (COMSNETS), 2022, pp. 430–432.

[27] J. Karjee, P. Naik S, K. Anand, and V. N. Bhargav, “Split computing: Dnn inference partition with load balancing in iot-edge platform for beyond 5g,” Measurement: Sensors, vol. 23, pp. 100409, 2022.

[28] Q. Lu, Y. Li, Z. Qin, X. Liu, and Y. Xie, “Speech recognition using efficientnet,” in International Conference on Multimedia Systems and Signal Processing, New York, NY, USA, 2020, ICMSSP 2020, p. 64–68, Association for Computing Machinery.

[29] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in Computer Vision and Pattern Recognition, 2009, pp. 248–255.

[30] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” arXiv:1409.1556, 2014.