Distributed Inference with Deep Learning Models across Heterogeneous Edge Devices

Chenghao Hu, Baochun Li
ch.hu@mail.utoronto.ca, bli@ece.toronto.edu
Department of Electrical and Computer Engineering, University of Toronto

***Abstract*—Recent years witnessed an increasing research attention in deploying deep learning models on edge devices for inference. Due to limited capabilities and power constraints, it may be necessary to distribute the inference workload across multiple devices. Existing mechanisms divided the model across edge devices with the assumption that deep learning models are constructed with a chain of layers. In reality, however, modern deep learning models are more complex, involving a directed acyclic graph (DAG) rather than a chain of layers.**

In this paper, we present *EdgeFlow*, a new distributed inference mechanism designed for general DAG structured deep learning models. Specifically, *EdgeFlow* partitions model layers into independent execution units with a new progressive model partitioning algorithm. By producing near-optimal model partitions, our new algorithm seeks to improve the run-time performance of distributed inference as these partitions are distributed across the edge devices. During inference, *EdgeFlow* orchestrates the intermediate results flowing through these units to fulfill the complicated layer dependencies. We have implemented *EdgeFlow* based on PyTorch, and evaluated it with state-of-the-art deep learning models in different structures. The results show that *EdgeFlow* reducing the inference latency by up to 40.2% compared with other approaches, which demonstrates the effectiveness of our design.

I. Introduction

As deep learning models are used in a wide variety of tasks such as image recognition, video analysis, and natural language processing, they are typically deployed at remote cloud servers and require users to upload local data for inference, incurring considerable overhead with respect to the time needed for transferring large volumes of data over the Internet. An intuitive solution to reduce such overhead is to “offload” these inference tasks from the cloud server to the edge devices. Unfortunately, edge devices are typically resource-constrained while the inference process is extremely computation-intensive [1]. Directly using a deep learning model for inference on devices with limited computation power may result in an even longer inference time. For this reason, it is desirable to design distributed inference mechanisms that accelerate the inference process by partitioning the workload and distributing them to a cluster of edge devices for cooperative inference [2].

Though distributed inference has received much attention in the recent literature, existing works generally assume that deep learning models are constructed as a chain of sequentially executed layers. Unfortunately, such an assumption is too simplified to hold with modern deep learning models: besides stacking deeper layers, structural modifications are also applied to the models to pursue the best performance, such as residual blocks in ResNet [3] and PANet [4]. Generally, modern models are constructed as DAGs (directed acyclic graphs) instead of chains to represent complicated layer dependencies, *e.g.*, a layer may require outputs from multiple preceding layers, or its output has to be fed into multiple subsequent layers.

This work was supported by Huawei Canada Technologies Co., Ltd.

Such a DAG structure comes with new challenges in the distributed deployment of deep learning models. For example, DAG structured models require new layers, like upsampling and concatenation, to maintain the consistency of the intermediate results, which are barely discussed in existing distributed inference mechanisms. The execution sequence of the layers is undetermined as there could be parallel paths in the computation graph. Also, the fact that one layer may depend on multiple preceding layers increases the complexity of model partitioning, since the partitioning of a specific layer should consider how its preceding layers are partitioned.

Unfortunately, existing works failed to provide adequate support for DAG structured models. Most of them tried to turn a DAG back into a chain with two intuitive methods: ignoring the branches and manually fix the dependency [5]; or finding cut points — whose removal adds the connected components of the graph — such that the branchy layers between two cut points can be treated as a single one [6], [7]. Though these methods may work well with simple DAG structures like ResNet, they cannot deal with more complex models, such as Yolo V5 [8], a state-of-the-art deep learning model for object detection, which contains no cut points except the input and output layers, and there are so many branches that it will be almost impossible to manually fix the layer dependency.

To address these challenges, this paper introduces *EdgeFlow*, a new distributed inference mechanism specifically designed for general DAG structured deep learning models. Rather than turning a DAG into a chain, *EdgeFlow* breaks layers of the computation graph into a set of execution units, which contain a list of input requirements, the computation operator, and a forwarding table. During the inference, when the required input is ready, the execution unit will apply the computation operator on the input to get an intermediate result. According to the forwarding table, it will send the intermediate result to other units to fulfill their input requirements. Intuitively, the intermediate results *flow* among execution units that are distributed among edge devices according to the

dependency to finish a logically equivalent inference as the original computation graph.

EdgeFlow achieves acceleration by partitioning a layer into multiple independent execution units, such that they can be assigned to different devices for parallel execution. One of the major challenges, therefore, is how these layers should be partitioned. Since we have to decide the partitioning scheme for each layer, the decision space could be too large to find the optimum solution. In addition, the partitioning decision for a specific layer is related to how its preceding layers are partitioned, which further adds complexity. Considering both the run-time efficiency and partition optimality, we propose a new progressive algorithm that partitions the computation graph layer by layer in topological order, such that we can optimally partition the workload of a specific layer when the partition schemes for its preceding layers are fixed.

Though such a partitioning problem can be formulated as an integer programming problem, such a formulation may not be practical to solve as solving integer programming problems is NP-hard in general. In this context, another highlight in our contributions is that we are able to transform such an integer programming problem to a linear programming (LP) problem as an approximation, such that it can be solved to optimality efficiently, without resorting to heuristics. Our transformed LP problem can then be solved efficiently using off-the-shelf LP solvers. We have implemented *EdgeFlow* with PyTorch, and evaluate it on various deep learning model structures, including the latest Yolo V5 model. Our comparison results show that *EdgeFlow* outperforms the latest distributed inference works, reducing the inference latency by up to 40.2%.

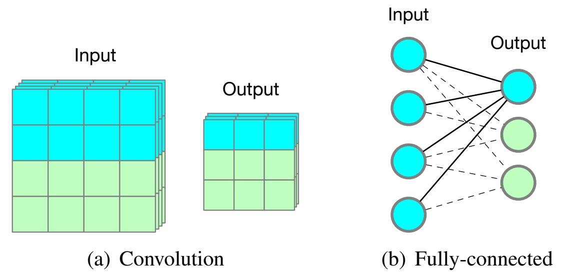
II. Background and Motivation

In this section, we briefly introduce the structure of deep learning models and their inference process, and show how the inference can be distributed among edge devices. We then analyze the challenges brought by the DAG structure, which directly motivates the design of *EdgeFlow*.

A. Deep Learning Model Structures

With modern deep learning models, their structure is becoming more and more complicated. Generally, they can be modeled as a DAG structured computation graph $\mathcal{G} = \{\mathcal{L}, \mathcal{E}\}$, where each vertex $l \in \mathcal{L}$ represents a layer in a deep learning model and defines a specific algorithmic operation. The edges $\mathcal{E}$ dictates the execution order of the inference process. An edge $(m, l) \in \mathcal{E}$ means layer l takes the output of m as its input, and m has to be finished before the execution of l. Therefore the layers should be executed in the topological order of the graph during the inference.

In the context of this paper, the layers can be classified into two groups according to the dependency property. For example, convolution layer can be regarded as a *partially dependent layer*, which means, as illustrated in Fig. 1(a), the computation of each output only requires parts of the input. Similar layers include pooling, batch normalization, and element-wise activations like RELU. In contrast, the other



(a) Convolution (b) Fully-connected

Fig. 1. Input and output dependency among convolution and fully-connected layers.

layers can be grouped as *fully dependent layers*, where the computation of every single element of the output requires the entire input from the previous layers, such as the flatten and fully-connected layers, which is exemplified in Fig. 1(b).

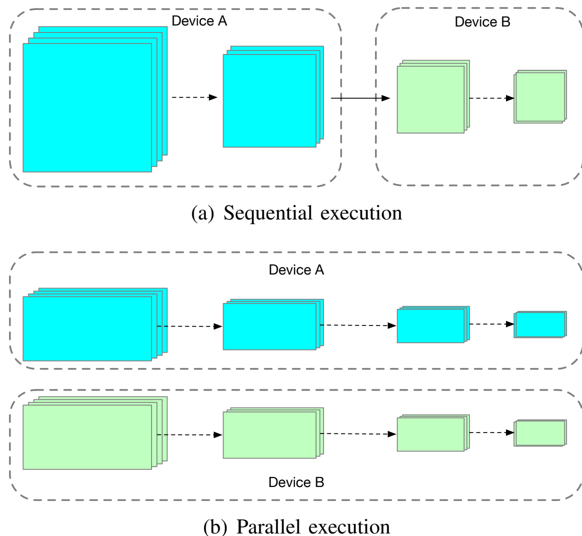
B. Distributed Inference across Edge Devices

The simplest way to distribute the inference workload can be borrowed from the model parallelism in distributed machine learning, where layers are atomically assigned to different devices for computation [6], [9]. As illustrated in Fig. 2(a), the input data is firstly processed by two convolution layers at device A, and then the intermediate result will be passed to device B for the following computation. In this case, the inference process is sequentially executed even though there could be multiple devices, and the computing resources are underutilized unless there are enough inference requests to fulfill the pipeline.

To fully exploit the computing resources of devices, an important observation is that inference can be processed in a parallel manner by taking advantage of the partial feature dependency of some layers like convolution and pooling as we described in Fig. 1(a). Since parts of the output of a convolution layer only require a subset of the input, one can partition the input into multiple pieces and feed to different devices, such that each of them can compute parts of the output which can then be assembled back to get the original output.

Motivated by this intuition, DeepThings [10] proposed a Fused Tile Partitioning method, which partitions the last 2-dimension output feature map into small non-overlapping tiles, whose computation task will be assigned to devices. For each partition, DeepThings computes the feature dependency all the way back to the input to find the required partition of the input, which means by feeding this specific input partition into the model, the device can obtain the target output tile independently. As a result, the computation of the final result can be parallelized and hence accelerated as each device only has to compute a subset of the original output.

However, the adjacent partition may require overlapping intermediate input, which could be redundantly computed at different devices, and the overlapping area grows when the model becomes deeper. According to [11], the redundant computation overhead can be up to $3\times$ $5\times$ with only 12 layers. To address this issue, CoEdge [5] partitions the output



(a) Sequential execution

(b) Parallel execution

Fig. 2. Two different distributed inference mechanisms.

of each layer with no overlapping, and pulls the required data when necessary. Since the range of the overlapping area is tiny for two consecutive layers, CoEdge significantly reduces the redundant computation overhead with the cost of relatively small margin data transmission, therefore demonstrates the state-of-the-art performance regarding distributed inference.

C. Challenges with DAG Structures

Though distributed inference has gained broad research attention, most of them assume the model is in the chain structure, which strongly hinders the applicability since most modern deep learning models are constructed as complicated DAGs. The adaptation to the DAG structure is non-trivial, and the challenges can be summarized in the following two aspects.

In distributed inference, it is important to maintain the layer dependencies because the layer dependencies indicate the proper execution order of the layers and guarantee that the correct results are obtained. With the chain structure, the relationship of the layers is very straightforward as one layer only has one preceding and one succeeding layer, respectively. However, in a DAG structure, one layer may be needed by multiple subsequent layers or require the results from multiple preceding layers as input. Compared with the chain structure, the layer dependencies inside a DAG are much more complicated, adding complexity to ensure the correct execution, especially after the layers can be partitioned and distributed among different devices.

Also, the fact that a layer may have multiple preceding layers has an important impact on the partition scheme for the current layer. Since the preceding layers could be partitioned and distributed among different devices, the output of these layers, which is also the required input of the current layer, will be scattered around. Therefore an inappropriate partition scheme of the current layer may result in a considerable overhead with respect to network traffic, in order to collect the required input from other devices.

III. *EdgeFlow* Design

In this section, we introduce *EdgeFlow*, a distributed inference system with natural support for DAG structured models, and then show how *EdgeFlow* partitions and distributes the model among different devices while maintaining the complicated layer dependencies to ensure the correct inference results.

A. *EdgeFlow Overview*

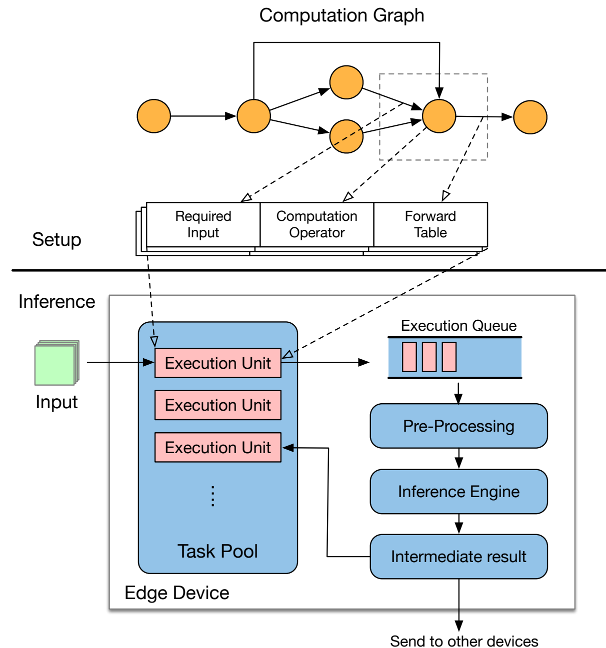


Fig. 3. System overview of *EdgeFlow*.

The architecture overview of *EdgeFlow* is presented in Fig. 3. It can be divided into a setup and an inference phase. During the setup phase, a *partitioner* takes the computation graph of a deep learning model as an input and partitions its layers into *execution units*. Specifically, *EdgeFlow* partitions the output feature map of the layers along the height dimension as illustrated in Fig. 2(b) and assign to these execution units, which means each of the units will be responsible for calculating part of the output features of the current layer. The execution units should ensure the completeness of the layer. In other words, the original layer output can be obtained by combining all the outputs of the execution units.

The execution unit is defined by three fields: (1) *Required input* indicating all the input features needed to compute the assigned output partition. The required input range of the preceding layers is determined by both the edges pointing to this layer in the computation graph and the mathematical property, which we will discuss in the following subsection; (2) *Computation Operator* including the operation type (e.g., convolution, pooling and etc.) of the current layer, as well as

the parameters of the layer; and lastly, (3) *Forward Table* tells the execution units where to forward its output features since they can be demanded by other execution units. Notice that the next execution unit might only require parts of the results to complete its task. Therefore the forward table should include both the target and the required range of its output.

After partition, the execution units will be deployed on the device for inference. Initially, all the execution units hibernate in the task pool since they have to wait for the required input. When an inference request arrives, the input data will be delivered to the execution units representing the first layer. In the following inference process, whenever the required inputs of an execution unit are ready, the execution unit will be moved into an execution queue waiting to be processed (e.g., there could be multiple active execution units representing the layers in the parallel paths of the computation graph). The received input feature maps need to be preprocessed before feeding into the inference engine, e.g., assemble the data from different execution units in the correct order. Then do the computation to obtain the result. When the current execution unit finishes computing, the results will be partitioned and forwarded according to the forward table. If the target execution units are not deployed at the same device, the corresponding data slice "flows" to the other devices through the network connection. In this way, the flow of the intermediate result continues until the inference finishes when the last execution unit sends the inference result back to the requester.

Intuitively, *EdgeFlow* maintains the connection between the layers in the computation graph by directing the flow of the intermediate result using the forward table. Also, the required input guarantees the correct execution order of the execution units.

B. Calculating the Feature Dependency

To ensure the execution units generate the expected partition of the output features, we need to calculate the range of the input required to calculate the desired output. Assume for layer l with output height H, we let integers (o_s, o_e), which satisfies $0 \leq o_s < o_e \leq H$, denote the target output range. Then we can calculate the corresponding input range (i_s, i_e) according to the mathematical property of the layer.

The input and output features are bijective for element-wise operations like activation (e.g., Sigmoid and Relu), batch normalization, concatenate and etc. In these cases, the input range can be simply obtained by

$$(i_s, i_e) = (o_s, o_e). \tag{1}$$

In DAG structured models, the upsampling layer is used to enlarge the feature map by a scale_factor, and the input and output dependency has to been considered separately due to the different upsampling methods it takes. The most common way of upsampling is the nearest-neighbor, which directly copies the value from the nearest position. The input range of this upsampling layer can be calculated by

$$(i_s, i_e) = (o_s/\text{scale_factor}, o_e/\text{scale_factor}). \tag{2}$$

For convolution and pooling layers, their computation can be modeled as applying a sliding kernel with parameters of kernel_size and stride on the width and height dimension of the feature map. Sometimes the input feature map will be padded around if parameter $\text{padding} \neq 0$. The corresponding input range can be calculated by

$$i_s = o_s \times \text{stride} - \text{padding}, \tag{3}$$

$$i_e = (o_e - 1) \times \text{stride} + \text{kernel_size} - \text{padding}. \tag{4}$$

Now we have the required input, but directly feeding it to the convolution/pooling layer may not yield the desired output due to the stride and padding settings. A simple illustration is presented in Fig. 4, where a 4×4 input feature map is passed to a convolution layer with kernel size equals 2, stride 2, and padding 1. The pink region is the desired output, and we can easily obtain the required area in the original input feature map. But applying the layer on the required input will not generate the desired output because the upper and bottom paddings should not be added as the required area is in the middle of the original input.

To solve this problem, *EdgeFlow* adopts a pre-padding method, which set the padding parameter of the convolution/pooling layer to 0 and do the padding in the processing. Since the feature maps are partitioned along the height dimension, the left and right paddings can be automatically added to the input, and we have to determine the value of the upper and bottom paddings. Notice that according to Eqn. (3), i_s may have a negative value, meaning that the region starts in the padding region outside the upper bound of the original input feature map. Therefore, the upper padding can be calculated by

$$\text{upper_padding} = \begin{cases} -i_s, & i_s < 0 \\ 0, & \text{otherwise} \end{cases}. \tag{5}$$

And similarly, i_e could be larger than the input height H_i. It means that the required region ends in the padding region outside the bottom bound of the original input feature map, and the bottom padding is

$$\text{bottom_padding} = \begin{cases} i_e - H_i, & i_e > H_i \\ 0, & \text{otherwise} \end{cases}. \tag{6}$$

In this way, *EdgeFlow* automatically decides if the padding is necessary and how many paddings are required to obtain the desired output.

C. Dependency-Aware Layer Partitioning

Now we are ready to describe how to partition a layer into multiple execution units and maintain the dependencies. As illustrated in Fig. 5, the output features are firstly partitioned into some parts with no overlapping, and assigned to different execution units respectively. The execution units share the same computation operator as the original layer. But if the layer is convolution or pooling with non-zero padding, then the padding value will be set to 0 in the execution units and

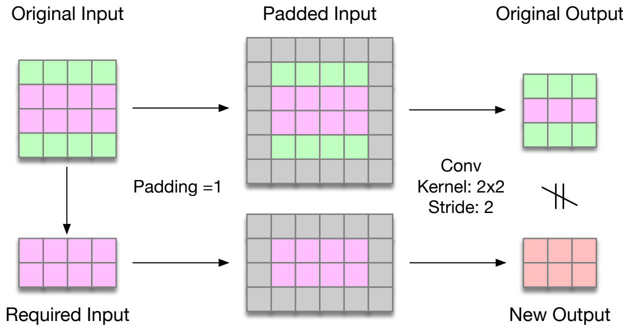


Fig. 4. A demonstration of stride and padding issue.

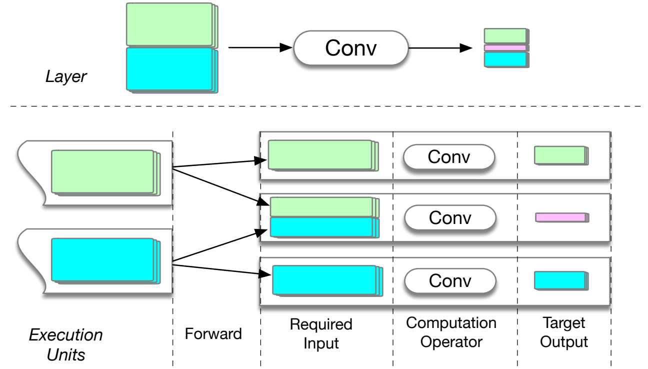


Fig. 5. Partition a layer into multiple execution units.

EdgeFlow will automatically add the paddings in the pre-processing phase as we described previously. Then for each unit, *EdgeFlow* will calculate the corresponding input range according to the type of the layer, which defines the required input of this execution unit.

Notice that the previous layers may also be partitioned, which means the required input might be distributed at multiple execution units too. For example, in Fig. 5, the input features have been divided into two parts and assigned to two different units. The second execution unit for the current layer, which is responsible for computing the pink region of the output, requires inputs from both parts. Therefore after obtaining the corresponding input range, the forwarding table of the preceding execution units needs to be updated about which part of their outputs are demanded so that the correct and complete input features will be forwarded to the next execution units. In a DAG-structured deep learning model, one layer may require inputs from multiple preceding layers, which means for these required layers, all of their corresponding execution units need to be notified.

During the inference, the execution units are assigned to different devices. Intuitively, we would prefer to put two units on the same device if they have a feature dependency, such that the demanding data can be delivered locally without network transmission. But unfortunately, the transmission still exists if we don't want any redundant computation, and some outputs are depending on the input features that cross the partition boundary, e.g., the pink region in Fig. 5. And this problem becomes even complicated with the DAG-structured model since the execution unit may have multiple dependencies on the previous layers. Therefore how to make proper partition decisions to balance the computation and transmission is the most critical problem of *EdgeFlow*, which we will discuss in the next section.

IV. Progressive Model Partitioning

As discussed previously, the partition scheme for a specific layer should consider how its preceding layers are distributed. In this section, we firstly formulate how to optimally partition a layer when its preceding layers have already been partitioned, then approximate it with an LP problem for run-time efficiency. And finally, we introduce how to partition the whole model layer by layer in a progressive manner.

A. Problem Formulation

Consider n available edge devices, and the output features of the current layer l range from 0 to H. Then the partition decision can be expressed as an integer vector $\mathbf{x}_l = (x_{l,0}, x_{l,1}, \ldots, x_{l,n})$, which means device i will be assigned with a execution unit responsible for calculating the output features that ranges from $x_{l,i-1}+1$ to $x_{l,i}$. The following constraints are required to make $\mathbf{x}_l$ a valid partitioning decision. Firstly, Eqn. (7) requires x_i to be non-negative integers as the features are indexed discretely. The decision range shall not exceed the output range of this layer, which is expressed in Eqn. (8). And lastly, constraint (9) restricts that there should be no overlapping between two different assignments to avoid redundant computation.

$$x_i \in \mathbf{Z}^+, i = 0, 1, \ldots, n \tag{7}$$

$$x_0 = 0, x_n = H \tag{8}$$

$$x_0 \leq x_1 \leq \cdots \leq x_n \tag{9}$$

Our target is to minimize the finish time of the current layer. Since the computation workload of this layer is distributed among different devices, we have to estimate the finish time on each device separately. During the inference process of layer l, device i has to wait for the required input before starting to compute the output features in the assigned range. Thus the finish time of the assigned workload of layer l on device i, denoted as $T_{l,i}$, can be estimated as follows

$$T_{l,i} = t_{\text{trans}}(i;l) + t_{\text{comp}}(i;l) \tag{10}$$

where $t_{tras}(i;l)$ is the finish time of the transmission needed to collect required input, and $t_{\text{comp}}(i;l)$ is the computation time.

We firstly estimate the transmission time. A key difference between chain and DAG structured models is that a layer may depend on multiple preceding layers. Thus how these layers are partitioned and assigned has an essential impact on the transmission time. Let M denote the set of the layers that are

required by l, *i.e.*, for each layer $m \in M$, we have $(m,l) \in \mathcal{E}$ and the partition scheme of m is denoted as $\mathbf{x}_m$. Given the output range $x_{l,i-1}$ and $x_{l,i}$, we can get the corresponding required regions starting from i_s to i_e in the input using the equation from (1) to (4).

Then device j, which has the output range $(x_{m,j-1}, x_{m,j})$ assigned previously, it has to send the overlapping area to device i. The length of the overlapping area can be calculated by

$$p_{m,i,j} = \min(i_e, x_{m,j}) - \max(i_s, x_{m,j-1}). \tag{11}$$

When $p_{m,i,j} \leq 0$, there is no overlapping between (s_i, e_i) and $(x_{m,j-1}, x_{m,j})$, which means device i doesn't require any input from device j, and there is no transmission between device i and j related to layer m and l. Thus to identify the valid transmission, we introduce an indicator function $\mathbb{1}_{\{p_{m,i,j}>0\}}$ whose value is 1 if $p_{m,i,j} > 0$ otherwise 0. The transmission phase ends when the last piece of the required input arrives. Therefore the finish time of transmission can be expressed as

$$t_{\text{trans}}(i;l) = \max_{j\in\{1,\dots,n\}, m\in M} \mathbb{1}_{\{p_{m,i,j}>0\}}(T_{m,j} + \frac{p_{m,i,j}}{B_{i,j}}) \tag{12}$$

where $T_{m,j}$ is the time that device j finished its execution unit of layer m, and $B_{i,j}$ is the available bandwidth between device i and j.

On the other hand, the estimation of computation time can be much easier. Previous works have successfully adopted regression models to predict the computation time of a layer with the hyper-parameters of the layer as input variables. Also, empirical studies of [7] have shown that for a specific layer and device, the computation time is proportional to the size of the input or output features. Hence we can confidently build a linear regression model Y_i for each device, which takes the output range as input to predict the computation time. Then for a specific layer l, the output range that device i is responsible for computing is $x_i - x_{i-1}$, and we can predict the computation time by

$$t_{comp}(i) = Y_i(x_i - x_{i-1}; l). \tag{13}$$

we take the longest finish time of these devices as our objective, and we can formulate the partition decision as the following optimization problem.

$$\begin{aligned} \min_{x} \quad & \max(T_{l,1}, T_{l,2}, \dots, T_{l,n}) \\ s.t. \quad & \text{Constraints (7), (8) and (9)} \end{aligned} \tag{14}$$

Problem (14) has non-convex optimization constraints and objective including indicator, minimum and maximum functions. Thus directly solving the above problem is not practical, not to mention this is only the partition problem for a single layer instead of the whole model. Next, we try to approximate the above problem with a linear programming problem such that we can solve it efficiently.

B. LP Approximation

The first step of finding a linear approximation of the original problem is to relax the integer constraint (7) by allowing continuous variables. And in this case, the linear constraints (8), (9) still guarantee the decision falls in the correct range. Also, the maximum function in the objective can be equivalently transformed by introducing an auxiliary variable λ, such that

$$\min_{x,\lambda} \quad \lambda \tag{15}$$

$$s.t. \quad T_{l,i} \leq \lambda, i \in \{1,\dots,n\} \tag{16}$$

$$\text{Constraints (8) and (9)}$$

We then focus on eliminating the nonlinear constraint (16), which, according to Eqn. (10), (12) and (13), can be expanded as a series of constraints

$$\mathbb{1}_{\{p_{m,i,j}>0\}}(T_{m,j} + \frac{p_{m,i,j}}{B_{i,j}}) + Y_i(x_i - x_{i-1}; l) \leq \lambda \tag{17}$$

where $i, j \in \{1,\dots,n\}, m \in M$. Constraints (17) and (16) are equivalent because of the maximum property.

Next, we show that as long as layer m is also partitioned with the same objective that minimizes the longest finish time of all the devices, the constraint whose indicator function values 0 will not be tight even when the value is set to 1, and therefore the indicator function can be removed safely without affecting the optimal solution. The consequence of removing the indicator function is that those devices that do not have to communicate with device i will also be considered during the computation of transmission time. Assume device i receives the last piece of the input from device j_0, representing the constraint

$$T_{m,j_0} + \frac{d_{m,i,j_0}}{B_{i,j_0}} + Y_i(x_i - x_{i-1}; l) \leq \lambda. \tag{18}$$

And $\widetilde{j}$ is any device that has no communication with device i such that $\mathbb{1}_{\{d_{m,i,\widetilde{j}}>0\}} = 0$. If we ignore the indicator function, the corresponding constraint becomes

$$T_{m,\widetilde{j}} + \frac{d_{m,i,\widetilde{j}}}{B_{i,\widetilde{j}}} + Y_i(x_i - x_{i-1}; l) \leq \lambda \tag{19}$$

which should not exists. But notice that λ is the upper bound. As long as constraint (19) is not tight, it won't affect the optimization result. if it was tight, then we have

$$T_{m,j_0} + \frac{d_{m,i,j_0}}{B_{i,j_0}} \leq T_{m,\widetilde{j}} + \frac{d_{m,i,\widetilde{j}}}{B_{i,\widetilde{j}}} \leq T_{m,\widetilde{j}}$$

where the last inequality holds because $d_{m,i,\widetilde{j}} \leq 0$. The above inequality indicates that device $\widetilde{j}$ didn't finish the assigned computation of the last layer m even when device i has received all the required input from other devices. In this case, the workload of device $\widetilde{j}$ is not assigned properly and should be offloaded to other devices since our optimization objective is to minimize the longest finish time. Therefore, as long as we partition the previous layers in the same way, the constraint

(19) will not be tight, and the indicator function can be safely removed, and the constraint (17) can be transformed into

$$T_{m,j} + \frac{p_{m,i,j}}{B_{i,j}} + Y_i(x_i - x_{i-1}; l) \le \lambda \tag{20}$$

The last step is to handle the computation of the overlapping length in (11), which can be transformed as

$$\begin{aligned} p_{m,i,j} &= \min(e_i, x_{m,j}) - \max(s_i, x_{m,j-1}) \\ &= \min(e_i - s_i, \\ &\qquad e_i - x_{m,j-1}, \\ &\qquad x_{m,j} - s_i, \\ &\qquad x_{m,j} - x_{m,j-1}) \end{aligned} \tag{21}$$

by simply expanding minimum and maximum function. Unfortunately, there is still a minimum function, and [12] has proved that solving a linear programming with minimum functions added to their constraints is NP-complete. But notice that the minimum function (21) can be expressed as an LP problem by

$$\begin{aligned} \min_{p_{m,i,j}} \quad & -p_{m,i,j} \\ s.t. \quad & p_{m,i,j} \le e_i - s_i, \quad p_{m,i,j} \le e_i - x_{m,j-1}, && (22) \\ & p_{m,i,j} \le x_{m,j} - s_i, \quad p_{m,i,j} \le x_{m,j} - x_{m,j-1}. && (23) \end{aligned}$$

By adding $-p_{m,i,j}$ to the objective of (15), we can try our best to enforce the Eqn. (21) holds in the feasible solution and eliminates the minimum function. And the problem becomes

$$\begin{aligned} \min_{x,\lambda,p} \quad & \lambda - \sum_{m,i,j} p_{m,i,j} && (24) \\ s.t. \quad & \text{Constraints (8) and (9),} \\ & \text{(20) ,(22) and (23)}, \forall i,j \in \{1,\dots,n\}, m \in M. \end{aligned}$$

Now all of the related functions of the above problem are linear. Thus problem (24) can be an LP approximation to the original problem (14), which we can use existing LP solvers to find the partitioning scheme for the current layer quickly. Considering the running efficiency, we use the solution of (24) to approximate the optimal partition decision.

C. Progressive Partitioning Algorithm

Based on the LP problem (24), we propose a model partitioning algorithm for *EdgeFlow* which partitions the model progressively layer by layer. The algorithm is presented in Algorithm 1. The input of the algorithm includes the layers $\mathcal{L}$ and edges $\mathcal{E}$ of the computation graph, the number of the available edge devices n, and their respective pre-established linear regression model Y_i to predict the computation time as well as the bandwidth between the devices B.

The idea of Algorithm 1 is that for each layer, we tried to find an optimal partition based on how its preceding layers are partitioned, and the partition decision for the current layer will then becomes the foundation for the subsequent layers. Therefore for each layer, the algorithm starts from picking out its preceding layers and initialize the variables of the LP problem (24). Then solve the problem with LP solvers (*e.g.*, Mosek 8.1 is adopted in evaluation) to obtain a solution $\widetilde{\mathbf{x}}_l$, which is a float vector and needs to be rounded to integers to become a valid partition decision $\mathbf{x_l}$. Then based on $\mathbf{x_l}$, we will estimate the finish time for l on each devices using (10) for the following loops. In our evaluation, the algorithm can be finished in a few seconds.

Notice that the entry layers, which directly accepts input data from the requesters, have no preceding layers in the computation graph, therefore cannot be process directly in Algorithm 1. But they can be adapted into the loop by assuming an input layer deployed at requester devices, and add the record of the input layer to the related variables.

Algorithm 1 Progressive Model Partitioning Algorithm.

Input:
$\mathcal{L}$: layers of the computation graph;
$\mathcal{E}$: layers of the computation graph;
n: number of available devices;
B: Bandwidth matrix of devices;
$Y_i, i \in \{1,\dots,n\}$: pre-established linear regression model for each devices;
Output: Partitioning decision $\mathbf{x}$ for each layer;
1: **for** layer l in $\mathcal{L}$ in topological order **do**
2: Obtain the preceding layer set $M = \{m \in \mathcal{L} | (m,l) \in \mathcal{E}\}$
3: Initializing the problem variables
4: Solve LP Problem (24) to obtain $\widetilde{\mathbf{x}}_l$
5: Round $\widetilde{\mathbf{x}}_l$ to the integer decision $\mathbf{x}_l$ for layer l
6: Calculate the finish time $T_{l,i}$ for each device with (10)
7: **end for**
8: **return** $\mathbf{x}$

V. Evaluation

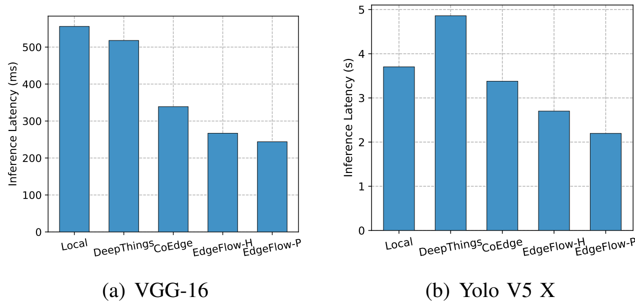
A. Evaluation Setup

Prototype Implementation: Consider that most of the edge devices do not have the privilege to be equipped with a GPU accelerator, we implement *EdgeFlow* based on PyTorch CPU version, and deploy it on the emulated testbed building upon Compute Canada. The testbed includes 6 virtual machines and the detailed settings are presented in Table. I. We also use Wondershaper tool to limit the available bandwidth to be 2000Mbps, which is the same as the wired connection setting used in [7].

TABLE I
Detailed settings of the virtual machine used in evaluation.

Type	Number of vCPUs	Memory	Number of Instances
C1	1	7.5GB	3
C2	2	7.5GB	2
C4	4	15GB	1

Inference workload: Though *EdgeFlow* is designed to tackle the complicated DAG-structured models, we take both the



(a) VGG-16 (b) Yolo V5 X

Fig. 6. Inference latency comparison with VGG-16 and Yolo V5 X.

chain and DAG structures into consideration in the evaluation to show the compatibility of *EdgeFlow*. For chain structure, we use VGG-16 [13], a traditional image classification model which organizes the layers sequentially. And for DAG-structure, the state-of-the-art object detection model, Yolo V5 X [8], is adopted. During the inference, we use another device as a requester to initiate the inference request. For VGG-16, the input data is a 224×224 image with 3 channels, and for Yolo V5, the image shape is 640×640.

Baselines: We compare the performance of *EdgeFlow* with the following baselines. (1) *DeepThings* partitions the output features of the last convolution layer and fuses all the necessary intermediate computation as independent tasks for the devices. (2) *CoEdge* splits the output features for each layer with no overlapping, and pulls the overlapped features when necessary. Also, CoEdge adopts a simple heuristic method to enforce that the data is transmitted between neighbor devices in a single direction to avoid mutual waiting. To demonstrate the effectiveness of our model partitioning method, we also compare with (3) *EdgeFlow-H*, which uses our proposed framework but the same heuristic scheme for the intermediate layers like CoEdge, and we label our proposed method as *EdgeFlow-P* for distinction. At last, we also take the (4) *Local* deployment into consideration, which deploys the model on every single device. In this case, the inference requests are distributed to these devices randomly and take the average latency as the result.

B. Evaluation Results

Inference Latency. The comparison of the overall inference latency is presented in Fig. 6. We firstly discuss the case of VGG-16. DeepThings is slightly faster than the local deployment, while CoEdge achieves a significant latency reduction compared with the local deployment, but not as good as the reported improvement in [5]. The reasons are that we deploy the model on all devices instead of only the slowest one. But with Yolo V5 X, CoEdge only has little improvement, and DeepThings is even slower than the local deployment.

Our proposed EdgeFlow-P achieves the fast inference speed in both cases, and even with the Yolo V5 X model, EdgeFlow-P reduces the inference latency by 40.2% compared with local deployment. With the same partition scheme of Yolo, EdgeFlow-H reduces 19.8% of the inference of CoEdge, which demonstrates the effectiveness of the architecture design of EdgeFlow. And compared with the heuristic method used in CoEdge, our model partitioning method further reduces the latency by 18.1%.

Workload Distribution. To have a better understanding of why EdgeFlow outperforms other methods, we plot their workload distributions for each layer of Yolo V5 X as a heat map in Fig. 7, where the value indicates the ratio of the output range over the complete output of the layer. Value 1.0 means that the device has computed 100% of the outputs of this layer.

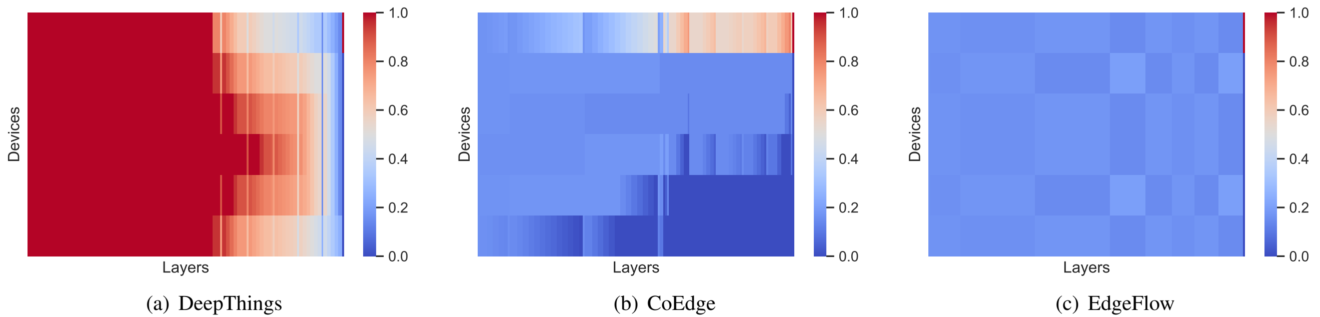
Firstly, DeepThings only considers the partitioning of the last convolution layer. The devices have to compute all the required input partitions locally to obtain these target pa independently. And the overlapping area in the input increases layer by layer until each device has to compute the whole layer as illustrated in Fig. 7(a). This explains why DeepThings could be even slower than the local deployment in Fig. 6(b) since more than half of the computations are redundant.

In contrast, CoEdge and EdgeFlow distribute the inference workload among the devices with no overlapping, and EdgeFlow achieves a better load balancing compared with CoEdge as shown in Fig. 7(c). This is because CoEdge enforces the transmission of the intermediate results in the same direction, which means the computation workload gathers in the same direction and finally concentrates on a single device, as illustrated in Fig. 7(b), leading to a bad utilization of the devices. And this is why EdgeFlow with our proposed progressive partition method is faster than the one with the same heuristic as CoEdge in Fig. 6.

Transmission Size. Fig. 8 shows the average communication size for each devices at different layers of Yolo V5 X. As we explained in Fig. 7(a), DeepThings treats the whole model as a single task unit and feed all the necessary data to the model in the input layer such that all the intermediate features required to compute the target slice of the output can be obtained locally. In other words, there is no transmission among the intermediate layers. Notice that DeepThings has three peaks of transmission in the final phase of the inference because the last layer of Yolo V5 is a non-dividable detection layer and requires three inputs from the previous layers.

In terms of the transmission size, DeepThings has a distinct advantage compared with the other two methods. But unfortunately, the advantage does not show up in the overall inference latency due to the heavily redundant computation, especially with the resource-constrained devices. In contrast, EdgeFlow and CoEdge sacrifice the transmission to avoid redundancy. For two consecutive layers, only the margin features need to be transmitted [5]. However, in a DAG structured model like Yolo V5, the output features of one layer might be required by a layer far behind itself. Due to the workload concentration we explained in Fig. 7(b), the layer could be assigned solely on a single device, while the required inputs are scattered at different places and need to be transmitted, and this causes the high transmission peak in the early phase of CoEdge in Fig. 8.

Also, an important design that differentiates EdgeFlow from



(a) DeepThings (b) CoEdge (c) EdgeFlow

Fig. 7. Inference workload distribution of Yolo V5 X among the devices.

CoEdge is that instead of pulling the required input from other devices when needed, which blocks the inference process of the current layer, the execution units actively push their output features to the subsequent layers as soon as the computation is finished. Therefore even if the transmission size of EdgeFlow is also huge at the fork point, the corresponding receiver units don't have to be blocked by requesting data that was ready long ago.

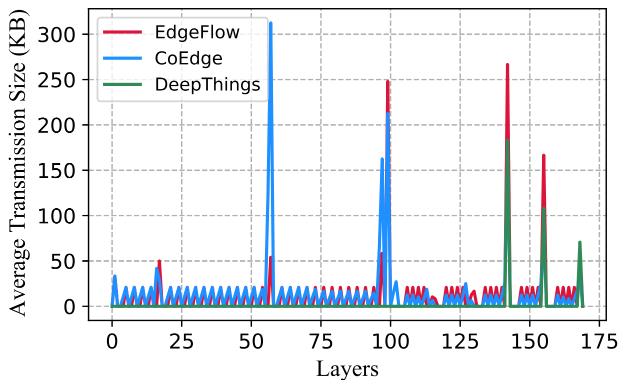


Fig. 8. Average communication size for each layer in the topological order of Yolo V5 X.

VI. Related Work

The exploration of deploying inference service at the edge side starts from fitting the model on the edge devices to keep all the computation locally. Bhattacharya *et al.* [14] reduced the size of the model by layer sparsification. Han *et al.* [15] jointly considered quantization and pruning approaches to compress the model for fast inference. Once-for-all [16] takes a big model as input and generates a large number of sub-models for different edge device configurations. These methods generally involve the modification of the original model structure, raising the concern of performance degradation like accuracy loss, while *EdgeFlow* keeps the model as it is.

When the model is too complicated to fit in a single edge device even after compression, a natural idea is to offload the computation to more powerful devices like cloud servers. Neurosurgeon [9] proposed to find a partition point between the layers such that the previous layers will be executed on edge and the remaining part will be offloaded to the cloud server. Hu *et al.* [17] proposed a dynamic adaptive method to partition the model under different network conditions, and [18] tried to reduce the transmission size by encoding the features. However, sending the intermediate results to the cloud server still incurs huge WAN transmission overhead, and sometimes the intermediate results can be even bigger than the original input data [6].

Besides offloading the workload to the cloud servers, cooperative inference among edge devices is another way to deploy deep learning models on the edge side, which is exactly the category that *EdgeFlow* falls in. DeepThings [10] proposed to divide the convolution layers into independent tasks, which allows the parallel execution of the distributed inference. MODNN [19] assigns workload according to the computation capacity. CoEdge [5] further considers the bandwidth and adopted a heuristic data transmission scheme to reduce the overhead. The highlight feature that characterizes *EdgeFlow* is the natural support of DAG structures, while the previous works try to adapt with heuristic methods. DeepSlicing [7] also considers the varieties of the model structure, but it requires finding specific points to cut the model into two sub-models that can be executed sequentially, which does not apply to the models whose main body is not separable, such as PANet [4] and UNet [20].

VII. Conclusion

The structure of deep learning models is becoming more and more complicated to pursue the best performance, exhibiting significant challenges to the edge deployment of the inference service. In this paper, we introduce *EdgeFlow*, a general distributed inference system with high applicability to support deep learning models in complicated structures. Instead of transforming the model into the chain structure, which is relatively simple and has been extensively studied. *EdgeFlow* encapsulates the layer dependencies inside carefully partitioned execution units, which are distributed among edge devices and collaboratively complete the inference task to achieve acceleration. The evaluation results show that *EdgeFlow* has distinct advantages compared with baselines, especially for complicated DAG structured model, which shows great potential to handle the ever increasing complexity of deep learning models.

References

[1] E. Li, L. Zeng, Z. Zhou, and X. Chen, “Edge ai: On-demand accelerating deep neural network inference via edge computing,” *IEEE Transactions on Wireless Communications*, vol. 19, no. 1, pp. 447–457, 2019.

[2] J. Chen and X. Ran, “Deep learning with edge computing: A review,” *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1655–1674, 2019.

[3] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[4] S. Liu, L. Qi, H. Qin, J. Shi, and J. Jia, “Path aggregation network for instance segmentation,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 8759–8768.

[5] L. Zeng, X. Chen, Z. Zhou, L. Yang, and J. Zhang, “Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices,” *IEEE/ACM Transactions on Networking*, vol. 29, no. 2, pp. 595–608, 2020.

[6] H. Li, C. Hu, J. Jiang, Z. Wang, Y. Wen, and W. Zhu, “Jalad: Joint accuracy-and latency-aware deep structure decoupling for edge-cloud execution,” in *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2018, pp. 671–678.

[7] S. Zhang, S. Zhang, Z. Qian, J. Wu, Y. Jin, and S. Lu, “Deepslicing: collaborative and adaptive cnn inference with low latency,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 9, pp. 2175–2187, 2021.

[8] G. Jocher, A. Stoken, J. Borovec, NanoCode012, A. Chaurasia, TaoXie, L. Changyu, A. V, Laughing, tkianai *et al.*, “ultralytics/yolov5: v5.0 - YOLOv5-P6 1280 models, AWS, Supervise.ly and YouTube integrations,” Apr. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4679653

[9] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, “Neurosurgeon: Collaborative intelligence between the cloud and mobile edge,” *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 615–629, 2017.

[10] Z. Zhao, K. M. Barijough, and A. Gerstlauer, “Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2348–2359, 2018.

[11] L. Zhou, M. H. Samavatian, A. Bacha, S. Majumdar, and R. Teodorescu, “Adaptive parallel execution of deep neural networks on heterogeneous edge devices,” pp. 195–208, 2019.

[12] T. M. Burks and K. Sakallah, “Min-max linear programming and the timing analysis of digital circuits,” in *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*. IEEE, 1993, pp. 152–155.

[13] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *3rd International Conference on Learning Representations, ICLR 2015*, Y. Bengio and Y. LeCun, Eds., 2015.

[14] S. Bhattacharya and N. D. Lane, “Sparsification and separation of deep learning layers for constrained resource inference on wearables,” in *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, 2016, pp. 176–189.

[15] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding,” in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016.

[16] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, “Toward collaborative inferencing of deep neural networks on internet-of-things devices,” *IEEE Internet of Things Journal*, vol. 7, no. 6, pp. 4950–4960, 2020.

[17] C. Hu, W. Bao, D. Wang, and F. Liu, “Dynamic adaptive DNN surgery for inference acceleration on the edge,” in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 1423–1431.

[18] J. H. Ko, T. Na, M. F. Amir, and S. Mukhopadhyay, “Edge-host partitioning of deep neural networks with feature space encoding for resource-constrained internet-of-things platforms,” in *2018 15th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*. IEEE, 2018, pp. 1–6.

[19] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, “Modnn: Local distributed mobile computing system for deep neural network,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1396–1401.

[20] X. Li, H. Chen, X. Qi, Q. Dou, C.-W. Fu, and P.-A. Heng, “H-denseunet: hybrid densely connected unet for liver and tumor segmentation from ct volumes,” *IEEE transactions on medical imaging*, vol. 37, no. 12, pp. 2663–2674, 2018.