SubFlow: A Dynamic Induced-Subgraph Strategy Toward Real-Time DNN Inference and Training

Seulki Lee and Shahriar Nirjon
Department of Computer Science
Univeristy of North Carolina at Chapel Hill
{seulki, nirjon}@cs.unc.edu

***Abstract*—We introduce *SubFlow*—a dynamic adaptation and execution strategy for a deep neural network (DNN), which enables real-time DNN inference and training. The goal of SubFlow is to complete the execution of a DNN task within a timing constraint that may dynamically change while ensuring comparable performance to executing the full network by executing a subset of the DNN at run-time. To this end, we propose two online algorithms that enable SubFlow: 1) *dynamic construction* of a sub-network which constructs the best sub-network of the DNN in terms of size and configuration, and 2) *time-bound execution* which executes the sub-network within a given time budget either for inference or training. We implement and open-source SubFlow by extending TensorFlow with full compatibility by adding SubFlow operations for convolutional and fully-connected layers of a DNN. We evaluate SubFlow with three popular DNN models (LeNet-5, AlexNet, and KWS), which shows that it provides flexible run-time execution and increases the utility of a DNN under dynamic timing constraints, e.g., 1x–6.7x range of dynamic execution speed with average -3% of performance (inference accuracy) difference. We also implement an autonomous robot as an example system that uses SubFlow and demonstrate that its obstacle detection DNN is flexibly executed to meet a range of deadlines that varies depending on its running speed.**

I. Introduction

Recently, DNNs (deep neural networks) [1]–[3] have been increasingly used in many real-life applications due to their superiority in solving complex machine learning problems [4]–[6], e.g., autonomous cars [7]–[10], natural language processing [11]–[13], and healthcare applications [14]–[16]. However, their long and unpredictable execution time resulting from a significant amount of computation often limits their deployment on real-time systems. Although high-performance hardware such as multi-core CPUs or GPUs efficiently process the massive workload of a DNN in parallel, the complexity and proprietary architecture of these platforms make effective scheduling of deadline-aware DNN tasks challenging, as shown in many previous works [17]–[27].

Moreover, the time constraints of many practical systems dynamically change at run-time, making DNNs more challenging to be executed as a real-time task. Such dynamic time constraints are found in many modern embedded systems such as autonomous cars [28]–[30], drones [31]–[33], and smartphones [34]–[36] where the system must deal with online changes such as run-time application requirements, resource availability, energy level, failures, and re-configurations. Such changes consequently cause variations in the time requirements of related-tasks [37]; e.g., data-dependent requirements where the periods depend on the input sensor data; time-dependent requirements where the actual deadline becomes known only at run-time when setting the actuators. For example, autonomous vehicles impose dynamic time constraints on tasks in reaction to a variety of road situations—a lower latency for obstacle detection is expected when traveling at higher speeds or when a pedestrian makes a sudden appearance. Failure of a scheduler also introduces variability in timing constraints, which reduces the amount of allowed execution time. A task scheduler in a complex and dynamic system may fail to start a task at its latest allowed start time and miss the deadline.

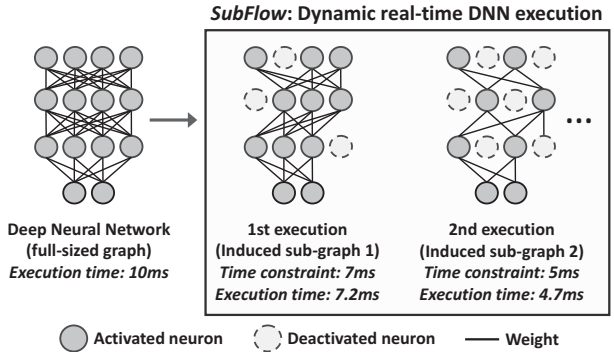


Fig. 1: *SubFlow* enables real-time inference and training of a DNN by *dynamically executing* a sub-graph of the DNN according to the timing constraint changing at run-time. For each inference or training execution, an *induced sub-graph* of the DNN, whose execution is completed in time, is constructed and executed by activating only necessary neurons, enabling *time-aware utilization* of the DNN.

Although DNN compression techniques such as [38]–[48] reduce the execution time to some extent, they are not directly applicable to DNNs having dynamic timing constraints since 1) they generate only one compressed network from the original DNN, which does not dynamically adapt once deployed, 2) most of them primarily focus on reducing memory usage as opposed to speedup, and 3) most compression methods are time-consuming as they require multiple training iterations and fine-tuning, and are limited to specific types of DNNs.

To enable the execution of DNNs with dynamic deadline constraints, we introduce *SubFlow*—an online DNN sub-graph

2642-7346/20/$31.00 ©2020 IEEE
DOI 10.1109/RTAS48715.2020.00-20

Authorized licensed use limited to: TONGJI UNIVERSITY. Downloaded on May 02,2021 at 01:58:06 UTC from IEEE Xplore. Restrictions apply.

strategy that constructs and executes a sub-graph of a DNN called the *sub-network* that completes the inference or training tasks within timing constraints that may change at run-time. The process is shown in Figure 1. For each execution of the DNN inference/training task, a different sub-network of different size and composition is constructed on-the-fly and executed within the time budget. In this way, the system ensures time-aware execution of a DNN with a flexible time budget–which also improves the CPU utilization and schedulability.

SubFlow consists of two run-time algorithms: 1) *dynamic construction* and 2) *time-bound execution* of a sub-network. For construction, it composes the best sub-network based on *the importance of neurons* in such a way that the execution time is expected to match the time budget while the performance loss due to the reduced size of sub-network is minimized. For execution, we propose *time-bound inference* and *training* of convolutional and fully-connected layers of a DNN, which are two main building blocks of a DNN. We name them time-bound since their inference and training times are bounded by the architecture and size of the sub-network.

We implement SubFlow by extending TensorFlow [49], one of the most popular DNN frameworks, and we open-source it[1]. We develop four custom operations and libraries for time-bound feed-forward and back-propagation [50] of dynamic sub-networks, i.e., *sub-convolution*, *sub-multiplication*, *sub-convolution-gradient*, and *sub-multiplication-gradient*. They support both CPUs and GPUs, and are implemented by using Eigen [51] and CUDA [52] libraries, respectively. DNNs designed with TensorFlow are easily adapted to SubFlow by simply applying SubFlow operations to the model, without requiring any architectural modifications, which makes SubFlow universal and applicable to existing DNNs. Since a new sub-network is constructed at run-time as opposed to constructing and saving a set of sub-networks offline, no additional memory is needed in SubFlow for this purpose.

We evaluate SubFlow for three standard DNN architectures, i.e., LeNet-5 [53], AlexNet [6], and KWS (Key-Word Spotting) [54] on three popular datasets, i.e., MNIST [53], CIFAR-10 [55], and GSC (Google Speech Commands) [56], respectively. Experiments are conducted on various hardware platforms: CPU (x86 and ARM) and GPU (RTX 2080 Ti and Jetson Nano [57]). The evaluation results show that both inference and training time of DNNs change dynamically according to the size and configuration of the sub-networks while achieving comparable performance to the full-sized network. For instance, the execution speed of AlexNet [6] dynamically changes between 1x and 6.7x while only a 3% inference accuracy drop is observed on average.

We also implement a mobile robot using an embedded GPU platform (Jetson Nano [57]) as an example real system that uses SubFlow where the latency requirement for obstacle detection changes due to the traveling speed of the robot. In this real-life experiment, the robot runs at various speeds and the execution of the DNN [58] that detects obstacles is adapted dynamically depending upon varying deadlines.

[1]SubFlow Project: https://github.com/learning1234embed/SubFlow

The main contributions of this paper are:

- We introduce *SubFlow*, a real-time DNN execution strategy enabling flexible time-bound inference and training that is completed within a dynamic time budget by constructing and executing a sub-network of the DNN at run-time.
- We propose an online sub-network construction algorithm to determine the best sub-graph of a DNN with a minimum performance loss based on induced sub-graph [59] method whose execution time is matched to a dynamic time budget.
- We propose time-bound feed-forward and back-propagation of convolutional and fully-connected layers of a DNN, where the total computation time is bounded by the size and configuration of the sub-network.
- We implement and open-source SubFlow[1] by developing four custom operations of TensorFlow with full compatibility, which allows the DNN designers to transform their DNNs into real-time dynamic DNNs easily.
- We develop and demonstrate a mobile robot as an example real system that uses SubFlow. The robot executes a depth estimation DNN for obstacle detection with its time constraint that dynamically changes based on the speed of the robot.

II. Overview

The goal of SubFlow is to enable execution of DNN inference and training tasks in such a way that the task is completed under dynamically varying time constraints while retaining comparable performance to executing the original full-size DNN. The flexible execution increases the utility of a DNN by letting it meet a range of deadlines at run-time, which conventional DNNs cannot. SubFlow also facilitates flexible scheduling of multitask learning where new tasks can be accommodated by dynamically updating the deadline of existing ones. The schedulability of a system running multiple DNNs can be improved by taking into account the flexible execution time of DNNs in the scheduling decision at run-time, which increases the total system utilization.

An unbounded trade off of inference accuracy for real-time execution of a DNN is not desirable in most systems. Hence, to limit the maximum loss of accuracy above a certain level, SubFlow limits the execution of sub-networks whose expected accuracy is lower than the desired level. SubFlow enables this by imposing a limit on the minimum network utilization parameter (defined in Section III) that essentially defines the size of the sub-network. The minimum network utilization parameter is empirically determined and is set by the developer or the system admin.

A. SubFlow Operations

SubFlow has three major operations, which are shown in Figure 2. A brief description of each operation follows.

1) Ranking Neurons. Given a trained DNN on which we want to apply SubFlow, the utility/contribution of each neuron to the performance (inference accuracy) of the DNN is computed.

Authorized licensed use limited to: TONGJI UNIVERSITY. Downloaded on May 02,2021 at 01:58:06 UTC from IEEE Xplore. Restrictions apply.

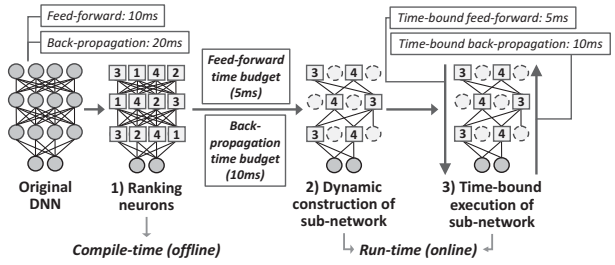


Fig. 2: **SubFlow operations**: SubFlow consists of three steps: ranking neurons, dynamic construction, and time-bound execution of a sub-network. Ranking of neurons is done at compile-time. At run-time, inference or training jobs with different time budgets are executed by forming and executing dynamic sub-networks.

This is calculated only once at compile-time. The details of this step are described in Section IV.

2) Dynamic Construction of Sub-Network. At run-time, a sub-network of the DNN is dynamically constructed for each job of a DNN task according to the given time budget. For example, an image classification task releases a job (say, every 500ms) where the job is to classify an image taken with the camera. For every job, an induced sub-graph [59] with a different subset of neurons (vertices) is constructed based on their importance calculated at compile-time. The construction of a sub-network is described in Section IV.

3) Time-Bound Execution of Sub-Network. Each sub-network corresponding to a job is executed and completed within the given time budget. The execution time of a job is bounded by the size and configuration of sub-networks. To enable the time-bound execution of the sub-network, we propose time-bound feed-forward and back-propagation [50] algorithms, which are described in Section V.

B. An Example Application

As an application of SubFlow, we describe an autonomous mobile robot which is one of many application-specific systems where SubFlow is applicable. We identify two real-time inference tasks of the robot (i.e., obstacle detection and sensor-based control) that have dynamic timing constraints. In section VIII, as a proof of concept system, we implement and evaluate the obstacle detection task on an embedded GPU-enabled autonomous mobile robot.

Obstacle Detection. In most autonomous cars and robots of today, data captured by cameras and other on-board sensors are processed by convolutional neural networks (CNNs) [60]–[64] to detect obstacles and to take timely measures to avoid collisions. For example, Tesla's autopilot [65] constructs depth maps using cameras to create 3D point maps of their surroundings, measuring objects' distance [9], [58], [66]–[68]. The real-time requirement of obstacle detection task in these systems becomes tighter when the vehicle is moving at a relatively higher speed – requiring the CNN to complete its processing faster. In contrast, when the vehicle is moving at a lower speed, the timing requirements are relaxed, allowing more time for the CNN to complete execution.

Sensor-based Control. Real-time requirements for sensor-based control systems of a mobile robot may change dynamically at run-time [37]. For example, tactile sensors on a mobile robot measure the force (and torque) exerted on its body [69], [70], which helps collision avoidance [70]. Depending on whether the robot is likely to be in contact with an object, it can adapt its sampling frequency of the sensors and thus scale its computation accordingly. Such dynamics not only changes the timing requirements of the tactile sensing and collision inference task but also affects the timing requirements of other inference tasks that are concurrently running in the system.

C. Programmability

SubFlow provides a set of DNN operations fully compatible with the existing TensorFlow operations, which allows a programmer to easily design and execute a DNN with SubFlow. Listing 1 and 2 show an example code of TensorFlow and SubFlow written for a convolutional layer, respectively. The implementation of SubFlow can be found in our GitHub[1].

Listing 1: TensorFlow programming example.

```
1  # DNN designing (a convolution layer)
2  output = tensorflow.nn.conv2d(input, filters, ...)
3  # DNN execution
4  sess.run(..., feed_dict={...})
```

Listing 2: SubFlow programming example.

```
1  # DNN designing (a convolution layer)
2  output = subflow.conv2d(input, filters, ..., activation)
3  # DNN execution
4  activation_vector = get_activation(network_utilization)
5  sess.run(..., feed_dict={..., activation: activaiton_vector})
```

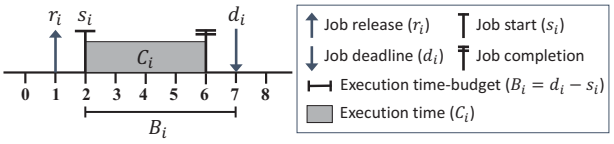
III. Background and Terminologies

SubFlow regards an inference or training task of a DNN as a real-time task τ with period T, execution time C, release time r, and relative and absolute deadline D and d, which is scheduled along with other tasks in the system. A DNN task, τ releases a sequence of jobs, J corresponding to the execution of a single iteration of inference or training that needs to be completed within the deadline, D as shown in Figure 3a.

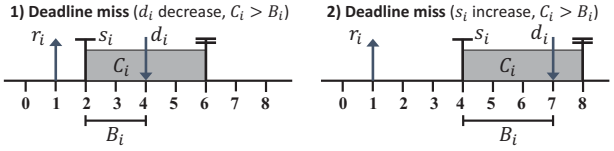
Dynamic Execution Time Budget. We define *execution time budget* for the i-th job J_i as $B_i = d_i - s_i$, where d_i is the absolute deadline and s_i is the start time of J_i. Since B_i for different J_i may be different, we call it a dynamic execution time budget. It is equivalent to the maximum allowed execution time for J_i to meet the deadline. Obviously, J_i meets its deadline if $B_i \geq C_i$, where C_i is the execution time of J_i. On the other hand, if $B_i < C_i$, J_i cannot meet the deadline. The latter case, i.e., $B_i = d_i - s_i < C_i$ happens in two situations: 1) d_i has decreased due to the system or application induced run-time variations in timing requirements, and 2) s_i has increased due to a scheduling failure or unavailable resources, causing J_i to be executed too late to complete within the deadline. Figure 3b illustrates an example of these two cases.

Sub-Network and Execution Time. SubFlow enables a DNN task, τ to complete J_i within B_i even if $B_i < C_i$ by reducing the execution time to the given time budget, i.e., $C_i \rightarrow B_i$.

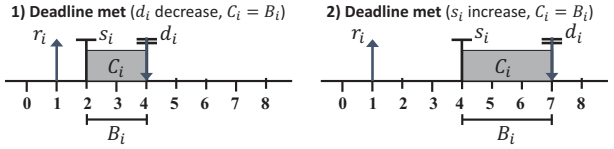
Authorized licensed use limited to: TONGJI UNIVERSITY. Downloaded on May 02,2021 at 01:58:06 UTC from IEEE Xplore. Restrictions apply.



(a) An example job execution of a task with $T = 6$ and $C = 4$.



(b) Two cases of dynamic execution time budget, which causes deadline miss unless execution time, C_i is adapted.



(c) The job meets the deadline by adjusting C_i to B_i for both cases.

Fig. 3: **An example of a dynamic execution time budget**: Given a dynamic execution time budget, SubFlow allows the job to meet the deadline by adjusting its execution time according to the budget.

For each J_i, SubFlow constructs and executes a sub-network, τ_i^s that is a subset of the full-size network, τ, which is able to complete its execution within B_i. Figure 3c shows an example in which the job meets the deadline by adjusting C_i to B_i.

Network Utilization. We quantify the size of a sub-network using *network utilization*, $u_i \in (0, 1]$ which is the relative size ratio of a sub-network constructed for the job, J_i to the full-size network as defined in Equation 1. Note that it is different from the task utilization used in scheduling, i.e., $U_i = C_i/T_i$.

Properties of DNNs. The construction of sub-networks having different execution times is enabled by three unique properties of DNNs: 1) many different configurations and sizes of DNN graphs often result in a similar performance [71], [72], 2) the oversized architecture of modern DNNs, i.e., the massive number of neurons and parameters allows incorporating multiple sub-networks into a single larger network [73], and 3) most DNNs repeat the same computation for each layer, i.e., convolution, matrix multiplication, etc., which makes the estimation of computation time for a sub-network feasible [74].

Induced Sub-Graph. SubFlow constructs a sub-network based on the induced sub-graph [59] of a DNN graph. Given a graph $G = (V, E)$ where $S \subset V$ be any subset of vertices of G, the induced sub-graph $G[S]$ is defined as the graph whose vertex set is S and whose edge set consists of all of the edges in E that have both endpoints in S.

In SubFlow, the neurons at each layer are considered as vertices, and connections between two neurons with a weight parameter are considered as edges. By activating the right subset of neurons and using only the edges connected to them as an induced sub-graph, a sub-network with the desired size and configuration is constructed. We use induced sub-graph since it is based on the selection of vertices (neurons), not edges (weight parameters). For most DNNs, the number of neurons is several orders of magnitude smaller than weight parameters, which enables efficient construction of sub-networks.

IV. Dynamic Construction of Sub-Network

The first two steps of SubFlow are the ranking of neurons and the dynamic construction of a sub-network. This section first describes the construction step starting from the definition of sub-network and then discusses how neurons are ranked for the construction of a sub-network.

A. Definition of Sub-Network

Basic DNN Operation. Before defining the sub-network, we describe the basic operation of DNNs. Given a training or test dataset for a DNN, τ having n instances, we denote the entire dataset as $\{(\mathbf{x}_j, \mathbf{y}_j)\}_{j=1}^{n}$. For the j-th instance, the input and output neurons of layer l is denoted as $\mathbf{o}_j^{l-1} \in \mathbb{R}^{m^{l-1}}$ and $\mathbf{o}_j^{l} \in \mathbb{R}^{m^{l}}$, respectively. The output of layer l, $\mathbf{o}_j^l$ is obtained by performing $\mathbf{o}_j^l = \sigma\left(\mathbf{y}_j^l\right)$, where $\mathbf{y}_j^l = \mathbf{o}_j^{l-1^\top}\mathbf{W}^l + \mathbf{b}^l$ with $\mathbf{W}^l \in \mathbb{R}^{m^{l-1} \times m^l}$ being the weight parameter and $\mathbf{b}^l \in \mathbb{R}^{m^l}$ being the bias for layer l. For nonlinearity, an nonlinear function $\sigma(\cdot)$ such as sigmoid function [75] is applied to $\mathbf{y}_j^l$. Following the recent trend in state-of-the-art DNNs such as [6], [76], [77], we use the rectified linear unit (ReLU) [78] as our $\sigma(\cdot)$. Although this is a formulation for a fully-connected layer, it also applies to convolution layers by converting a kernel operation with input into a matrix product as in [79].

Sub-Output Neuron. To construct a sub-network, τ_i^s for the job J_i from a DNN task, τ, we first compose *sub-output neuron*, $\tilde{\mathbf{o}}_j^l \in \mathbb{R}^{m^l}$ for each layer l, which is a sparse vector of the same length with the output neuron, $\mathbf{o}_j^l \in \mathbb{R}^{m^l}$ at the l-th layer of τ. It consists of a subset of $\mathbf{o}_j^l$ and zeros. Having $\tilde{\mathbf{o}}_j^l$ for all layers, we create a sub-network by connecting only the non-zero elements of $\tilde{\mathbf{o}}_j^l$ (activated vertices) based on the induced sub-graph construction [59]. Depending on $\tilde{\mathbf{o}}_j^l$, only a subset of weight parameter elements (edges) in $\mathbf{W}^l$ connected between $\tilde{\mathbf{o}}_j^{l-1}$ and $\tilde{\mathbf{o}}_j^l$ is activated for the sub-network. The elements of $\tilde{\mathbf{o}}_j^l$ are obtained by multiplying $\mathbf{o}_j^l$ with a binary vector, $\mathbf{a}_i^l \in \mathbb{R}^{m^l}$ called *activation vector* that determines whether the corresponding neuron element of $\mathbf{o}_j^l$ is activated or not in $\tilde{\mathbf{o}}_j^l$ as a vertex of induced sub-graph. In summary, the sub-output neuron of layer l, $\tilde{\mathbf{o}}_j^l$ for the j-th input instance is given by:

$$
\begin{aligned}
&\mathbf{a}_i^l \in \{0,1\}^{m^l} \;\text{ s.t. }\; \left\|\mathbf{a}_i^l\right\|_1 = \left\lfloor u_i^l \cdot m^l \right\rfloor \;\text{ or }\; u_i^l = \frac{\left\|\mathbf{a}_i^l\right\|_1}{m^l} \\
&\tilde{\mathbf{y}}_j^l = \tilde{\mathbf{o}}_j^{l-1^\top}\mathbf{W}^l + \mathbf{b}^{l^\top} \\
&\tilde{\mathbf{o}}_j^l = \mathbf{a}_i^l \circ \sigma\left(\tilde{\mathbf{y}}_j^l\right) = \sigma\left(\mathbf{a}_i^l \circ \tilde{\mathbf{y}}_j^l\right) \;\text{ since } \sigma(\cdot) \text{ is ReLU}
\end{aligned}
\tag{1}
$$

where $\mathbf{a}_i^l$ is the activation vector, m^l is the length of $\tilde{\mathbf{o}}_j^l$ and $\mathbf{o}_j^l$, $\|\cdot\|_1$ is ℓ_1-norm, u_i^l is the network utilization, $\mathbf{W}^l$ is the weight parameter, $\mathbf{b}^l$ is the bias, $\sigma(\cdot)$ is the nonlinear function (ReLU), and $\circ$ is Hadamard (element-wise) product [80].

Definition of Sub-Network. We define sub-network, τ_i^s for the job J_i as an induced sub-graph of a DNN, τ with total L layers, where its vertices are composed of the sub-output neurons of all layers, $\{\tilde{\mathbf{o}}_j^l | 1 \le l \le L\}$ defined in Equation 1.

Authorized licensed use limited to: TONGJI UNIVERSITY. Downloaded on May 02,2021 at 01:58:06 UTC from IEEE Xplore. Restrictions apply.

B. Construction of Sub-Network

Construction Objectives. Given a dynamic execution time budget B_i for the job J_i, a sub-network having total layers L, τ_i^s is constructed to achieve two objectives: 1) finding the maximum network utilization for each layer, $\mathbf{u}_i = [u_i^1, u_i^2, ..., u_i^L]$ in such a way that their total execution time is equivalent to or less than B_i, and 2) finding the activation vector, $\mathbf{a}_i^l$ for each layer l to determine sub-output neuron, $\tilde{\mathbf{o}}_j^l$ in which the number of elements of value one in $\mathbf{a}_i^l$, i.e., $\left\|\mathbf{a}_i^l\right\|_1$ is equivalent to $\left\lfloor u_i^l \cdot m^l \right\rfloor$ as defined in Equation 1 in such a way that the error between $\tilde{\mathbf{o}}_j^l$ and $\mathbf{o}_j^l$ is minimized.

Finding Network Utilization. To find the network utilization $\mathbf{u}_i$, we define execution time $C(\mathbf{u}_i)$ of a sub-network, τ_i^s as:

$$C(\mathbf{u}_i) \propto \kappa \sum_{l=1}^{L} u_i^l f^l\left(\mathbf{o}_j^l\right) \propto \kappa N_i \sum_{l=1}^{L} f^l\left(\mathbf{o}_j^l\right) \text{ if } u_i^l = N_i \quad (2)$$

where κ is a constant, $u_i^l \in (0, 1]$ is the network utilization of layer l, and $f^l\left(\mathbf{o}_j^l\right)$ is the execution time function of layer l, i.e., the time required to compute $\mathbf{o}_j^l$. We apply the same network utilization, N_i to all layers, i.e., $u_i^l = N_i$ since finding a set of optimal u_i^l is NP-hard, i.e., the search space for all combinations of u_i^l is exponential. Also, by activating the same proportion of neurons at each layer, the resulting sub-network is not to be cut or broken. Hence, $\mathbf{u}_i$ can be obtained by finding the maximum N_i satisfying $C(N_i) \leq B_i$ in Equation 2.

Layer-Wise Error. To obtain sub-output neuron, $\tilde{\mathbf{o}}_j^l$ of layer l that minimizes the error from $\mathbf{o}_j^l$, we define layer-wise error between $\tilde{\mathbf{o}}_j^l$ and $\mathbf{o}_j^l$ for both the j-th training instance and the total n of training instances, similar to [81]. They are denoted as $E_j^l(\tilde{\mathbf{o}}_j^l)$ and E^l, respectively, which are defined as:

$$\begin{aligned} \hat{\mathbf{o}}_j^l &= \tilde{\mathbf{o}}_j^l - \mathbf{a}_i^l \circ \mathbf{o}_j^l \\ E_j^l(\tilde{\mathbf{o}}_j^l) &= \hat{\mathbf{o}}_j^{l^\top} \hat{\mathbf{o}}_j^l = \left\|\tilde{\mathbf{o}}_j^l - \mathbf{a}_i^l \circ \mathbf{o}_j^l\right\|_2^2 \\ E^l &= \frac{1}{n}\sum_{j=1}^{n} E_j^l(\tilde{\mathbf{o}}_j^l) = \frac{1}{n}\sum_{j=1}^{n} \left\|\tilde{\mathbf{o}}_j^l - \mathbf{a}_i^l \circ \mathbf{o}_j^l\right\|_2^2 \end{aligned} \quad (3)$$

where $\circ$ is Hadamard (element-wise) product, $\|\cdot\|_2$ is ℓ_2-norm, and $\hat{\mathbf{o}}_j^l = \tilde{\mathbf{o}}_j^l - \mathbf{a}^l \circ \mathbf{o}_j^l$ is the output error vector of layer l.

Error Bound. From Equation 1 and $\|\sigma(\mathbf{x}) - \sigma(\mathbf{y})\|_2 \leq \|\mathbf{x} - \mathbf{y}\|_2$, the property of the ReLU, $E_j^l(\tilde{\mathbf{o}}_j^l)$ is bounded by:

$$\begin{aligned} E_j^l(\tilde{\mathbf{o}}_j^l) &= \left\|\tilde{\mathbf{o}}_j^l - \mathbf{a}^l \circ \mathbf{o}_j^l\right\|_2^2 \\ &= \left\|\sigma\left(\mathbf{a}^l \circ \tilde{\mathbf{y}}_j^l\right) - \sigma\left(\mathbf{a}^l \circ \mathbf{y}_j^l\right)\right\|_2^2 \\ &\leq \left\|\mathbf{a}^l \circ \tilde{\mathbf{y}}_j^l - \mathbf{a}^l \circ \mathbf{y}_j^l\right\|_2^2 \\ &= \left\|\mathbf{a}^l \circ \left(\left(\tilde{\mathbf{o}}_j^{l-1} - \mathbf{a}^{l-1} \circ \mathbf{o}_j^{l-1}\right.\right.\right. \\ &\qquad \left.\left.\left. + \mathbf{a}^{l-1} \circ \mathbf{o}_j^{l-1} - \mathbf{o}_j^{l-1}\right)^\top \mathbf{W}^l\right)\right\|_2^2 \\ &\leq \left\|\mathbf{a}^l \circ \left(\hat{\mathbf{o}}_j^{l-1^\top} \mathbf{W}^l\right)\right\|_2^2 \\ &\qquad + \left\|\mathbf{a}^l \circ \left(\left(\left(\mathbf{1} - \mathbf{a}^{l-1}\right) \circ \mathbf{o}_j^{l-1}\right)^\top \mathbf{W}^l\right)\right\|_2^2 \end{aligned} \quad (4)$$

where $\|\cdot\|_2$ is ℓ_2-norm, and $\mathbf{1}$ is a vector whose all elements are equal to 1. As shown in the last inequality, the upper bound of $E_j^l(\tilde{\mathbf{o}}_j^l)$ is determined by two results of the previous layer $l-1$, i.e., 1) the error vector, $\hat{\mathbf{o}}_j^{l-1}$ in the first term, and 2) the not activated elements of the output neuron, $\left(\mathbf{1} - \mathbf{a}^{l-1}\right) \circ \mathbf{o}_j^{l-1}$ in the second term. Hence, the bound of $E_j^l(\tilde{\mathbf{o}}_j^l)$ of layer l can be obtained by recursively computing them for the previous layers, i.e., from the first to the $(l-1)$-th layer.

Minimizing Error. Since sub-output neuron of the last layer L, $\tilde{\mathbf{o}}_j^L$ determines the performance of a sub-network, we minimize E^L. From Equation 1, 2 and 3, minimizing E^L given time budget, B_i is reduced to finding the activation vector, $\mathbf{a}_i^L$.

$$\underset{\mathbf{a}_i^L \in \{0,1\}^{m^L}}{\operatorname{argmin}} \; E^L \;\; \text{s.t. } C(N_i) \leq B_i \text{ and } \left\|\mathbf{a}_i^L\right\|_1 = \left\lfloor N_i \cdot m^L \right\rfloor \quad (5)$$

Since E^L is obtained from the errors of previous layers as shown in Equation 4, it is also minimized by finding $\{\mathbf{a}_i^l | 1 \leq l \leq L\}$ that minimizes the error of each layer l, i.e., $E_j^l(\tilde{\mathbf{o}}_j^l)$, which is achieved by minimizing their summation, i.e., $\sum_{l=1}^{L} E_j^l(\tilde{\mathbf{o}}_j^l)$. Hence, from Equation 1 and 3, the problem of constructing a sub-network given B_i is reformulated as:

$$\begin{aligned} &\underset{\mathbf{a}_i^l \in \{0,1\}^{m^l}}{\operatorname{argmin}} \; \frac{1}{n}\sum_{j=1}^{n}\sum_{l=1}^{L} E_j^l(\tilde{\mathbf{o}}_j^l) \\ &= \underset{\mathbf{a}_i^l \in \{0,1\}^{m^l}}{\operatorname{argmin}} \; \frac{1}{n}\sum_{j=1}^{n}\sum_{l=1}^{L} \left\|\mathbf{a}_i^l \circ \sigma\left(\tilde{\mathbf{y}}_j^l\right) - \mathbf{a}_i^l \circ \sigma\left(\mathbf{y}_j^l\right)\right\|_2^2 \\ &\qquad \text{s.t. } C(N_i) \leq B_i \text{ and } \left\|\mathbf{a}_i^l\right\|_1 = \left\lfloor N_i \cdot m^l \right\rfloor \end{aligned} \quad (6)$$

Hence, a sub-network for the job J_i which is completed within B_i with minimum error is dynamically constructed by finding N_i and $\{\mathbf{a}_i^l | 1 \leq l \leq L\}$ in Equation 6.

C. Neuron Ranking for Sub-Network Construction

Importance-based Ranking. While the network utilization, N_i is easily obtained by finding the maximum N_i satisfying $C(N_i) \leq B_i$ in Equation 2, the activation vector, $\mathbf{a}_i^l$ that selects the elements of sub-output neuron, $\tilde{\mathbf{o}}_j^l$ from $\mathbf{o}_j^l$ is required to minimize the error $E_j^l(\tilde{\mathbf{o}}_j^l)$ in Equation 3. We determine $\mathbf{a}_i^l$ based on the importance of each neuron of $\mathbf{o}_j^l$, which represents the increased error when it is removed. Then, $E_j^l(\tilde{\mathbf{o}}_j^l)$ is minimized by composing the binary elements of $\mathbf{a}_i^l$ such that $\tilde{\mathbf{o}}_j^l$ consists of $\left\lfloor N_i \cdot m^l \right\rfloor$ number of neurons in $\mathbf{o}_j^l$ having largest importance and zero for all the other elements.

To measure the importance, we compute the second-order derivatives of the error $E_j^l(\mathbf{o}_j^l)$ w.r.t. the output neuron, $\mathbf{o}_j^l$ for each layer using Optimal Brain Surgeon algorithm [82]. We use it since the heuristic methods such as magnitude-based method [46], [83], [84] may eliminate wrong neurons [44], [82], resulting in large error and poor performance [85].

Error Approximation. For a DNN trained to a local minimum, the error, $E_j^l(\tilde{\mathbf{o}}_j^l)$ can be approximated with Taylor series

Authorized licensed use limited to: TONGJI UNIVERSITY. Downloaded on May 02,2021 at 01:58:06 UTC from IEEE Xplore. Restrictions apply.

as in [82] and [86] w.r.t. the output neuron, $\mathbf{o}_j^l$ as follows:

$$\begin{aligned}\delta E_j^l &= E_j^l(\tilde{\mathbf{o}}_j^l) - E_j^l(\mathbf{o}_j^l) \\ &= \left(\frac{\partial E_j^l}{\partial \mathbf{o}_j^l}\right)^\top \delta \mathbf{o}_j^l + \frac{1}{2}\delta \mathbf{o}_j^{l^\top} \mathbf{H}^l \delta \mathbf{o}_j^l + O(\|\delta \mathbf{o}_j^l\|^3) \\ &\approx \frac{1}{2}\delta \mathbf{o}_j^{l^\top} \mathbf{H}^l \delta \mathbf{o}_j^l\end{aligned} \tag{7}$$

where δ is a perturbation of corresponding variable, $\mathbf{H}^l \equiv \partial^2 E_j^l / \partial (\mathbf{o}_j^l)^2$ is the Hessian matrix [87], and $O(\|\delta \mathbf{o}_j^l\|^3)$ is the third and all higher-order terms. With the error function defined in Equation 3, the first (linear) and third terms are ignored [81]. To minimize the increase in error, δE_j^l, we set the q-th element of $\mathbf{o}_j^l$, denoted as o_{jq}^l, to zero, expressed as:

$$\delta o_{jq}^l + o_{jq}^l = 0 \text{ or more generally } \mathbf{e}_q^{l^\top} \delta \mathbf{o}_j^l + o_{jq}^l = 0 \tag{8}$$

where $\mathbf{e}_q^l$ is the unit vector whose q-th element is 1 and others are all 0. With o_{jq}^l being removed from $\mathbf{o}_j^l$ as shown in Equation 8, we minimize Equation 7 as follows:

$$\min_q \frac{1}{2}\delta \mathbf{o}_j^{l^\top} \mathbf{H}^l \delta \mathbf{o}_j^l \quad \text{s.t. } \mathbf{e}_q^{l^\top} \delta \mathbf{o}_j^l + o_{jq}^l = 0 \tag{9}$$

Computing Importance. To compute the importance of the q-th element of $\mathbf{o}_j^l$ at layer l, o_{jq}^l, we solve Equation 9 with a Lagrangian function, $\mathcal{L}^l$, which is given by:

$$\mathcal{L}^l = \frac{1}{2}\delta \mathbf{o}_j^{l^\top} \mathbf{H}^l \delta \mathbf{o}_j^l + \lambda(\mathbf{e}^{l^\top} \delta \mathbf{o}_j^l + o_{jq}^l) \tag{10}$$

where λ is a Lagrange multiplier. By taking derivatives, employing Equation 8, and using matrix inversion, the optimal increase in error and output change of o_{jq}^l are obtained by:

$$s_q^l = \frac{1}{2}\frac{(o_{jq}^l)^2}{[\mathbf{H}^{-1}]_{qq}^l} \quad \text{and} \quad \delta \mathbf{o}_j^l = -\frac{o_{jq}^l}{[\mathbf{H}^{-1}]_{qq}^l}[\mathbf{H}^{-1}]^l \mathbf{e}_q^l \tag{11}$$

We call s_q^l as the *importance* of the q-th neuron element of layer l, o_{jq}^l– the amount of increase in error when it is removed from a DNN. Based on s_q^l, the activation vector, $\mathbf{a}_i^l$ is determined to activate $\lfloor N_i \cdot m^l \rfloor$ number of neuron elements in $\mathbf{o}_j^l$ having the largest importance by setting the corresponding elements of $\mathbf{a}_i^l$ to 1 and others to 0, which provides the sub-output neuron, $\tilde{\mathbf{o}}_j^l$ for construction of a sub-network with minimum error. s_q^l is computed only once at compile-time.

V. Time-Bound Execution of Sub-Network

The last step of SubFlow is to execute a newly-constructed sub-network within the execution time budget. This section describes the two run-time execution operations of SubFlow, i.e., time-bound feed-forward and back-propagation, which enables the time-bound completion of a sub-network.

A. Time-bound Feed-Forward

Time-Bound Feed-Forward. The inference of a DNN is achieved by executing the DNN layer by layer, which is called the feed-forward. Given a sub-network, an inference job, J_i completes the feed-forward within the time budget, B_i by performing computation in Equation 1 only for the non-zero elements of a sub-output neuron, $\tilde{\mathbf{o}}_j^l$. The amount of computation, as well as the execution time, are expected to be proportional to the number of non-zero elements of $\tilde{\mathbf{o}}_j^l$, which is determined by the activation vector, $\mathbf{a}_i^l$. Computation related to zero neuron elements is skipped since multiplication by zero results in a zero. We name it as *time-bound feed-forward* since the feed-forwarding time is bounded by the size and configuration of a sub-network depending on a set of $\mathbf{a}_i^l$.

Existing Feed-Forward. Unfortunately, the current feed-forward algorithms, such as the lowering method [88] do not support the sparse-neuron-aware feed-forwarding. They always perform the same amount of computation based on the fixed sequence of calculation regardless of the number of non-zero neuron elements. To enable time-bound feed-forward, we propose *sub-convolution* and *sub-multiplication* for a convolutional and fully-connected layer, respectively in a similar way to the direct sparse convolution [89].

Sub-Convolution. For a convolutional layer l, layer output, $\mathbf{O}^l \in \mathbb{R}^{n \times c' \times h' \times w'}$ is computed by taking input, $\mathbf{O}^{l-1} \in \mathbb{R}^{n \times c \times h \times w}$ from the previous layer $l-1$, where n is the size of the input batch; c', h', and w' denote the channel, height, and width of the output. For input $\mathbf{O}^{l-1}$, c, h, and w denote its channel, height, and width. For convolution, convolutional filter (weight parameter) denoted as $\mathbf{W}^l \in \mathbb{R}^{c' \times c \times y \times x}$ is applied to input, $\mathbf{O}^{l-1}$, where c', c, y, and x denote the size of the output channel, input channel, height, and width of filter.

Given sub-input neuron, $\tilde{\mathbf{O}}^{l-1}$ and sub-output neuron, $\tilde{\mathbf{O}}^l$ composed by $\mathbf{a}_i^{l-1}$ and $\mathbf{a}_i^l$, respectively, the computational complexity of convolution operation at layer l with filter $\mathbf{W}^l$, denoted as $f_c^l(\cdot)$, is given by:

$$\begin{aligned}&f_c^l(\mathbf{a}_i^{l-1}, \mathbf{a}_i^l, \mathbf{W}^l) \\ &\quad = \mathcal{O}(|\mathbf{a}_i^l|\,|\mathbf{W}^l| - \|\mathbf{1} - \mathbf{a}_i^{l-1}\|_1 - \|\mathbf{1} - \mathbf{a}_i^l\|_1 |\mathbf{W}^l|)\end{aligned} \tag{12}$$

where $\|\cdot\|_1$ denotes ℓ_1-norm, $|\cdot|$ denotes the number of elements, and $\mathbf{1}$ is a vector whose all elements are 1. The first term indicates the total amount of computation at the l-th layer of the full-size network, while the second and third term indicates the amount of computation reduced by the sub-input and sub-output neuron, respectively. The activation vector of the previous layer, $\mathbf{a}_i^{l-1}$ also determines the complexity since an output of one layer is the input of the next layer.

We define *sub-convolution* as the convolution of a sub-network whose computational complexity is determined by $\mathbf{a}_i^{l-1}$ and $\mathbf{a}_i^l$ as shown in Equation 12. Equation 13 is an example of sub-convolution with $\tilde{\mathbf{O}}^{l-1} \in \mathbb{R}^{1\times1\times3\times3}$, $\tilde{\mathbf{O}}^l \in \mathbb{R}^{1\times1\times2\times2}$, and $\mathbf{W}^l \in \mathbb{R}^{1\times1\times2\times2}$, where only two and five elements are activated as sub-output and sub-input neuron.

$$\begin{bmatrix} \cancel{o_{11}^l} & o_{12}^l \\ o_{21}^l & \cancel{o_{22}^l} \end{bmatrix} = \begin{bmatrix} \cancel{o_{11}^{l-1}} & o_{12}^{l-1} & o_{13}^{l-1} \\ o_{21}^{l-1} & \cancel{o_{22}^{l-1}} & \cancel{o_{23}^{l-1}} \\ o_{31}^{l-1} & \cancel{o_{32}^{l-1}} & o_{33}^{l-1} \end{bmatrix} * \begin{bmatrix} w_{11}^l & w_{12}^l \\ w_{21}^l & w_{22}^l \end{bmatrix} \tag{13}$$

Here, $*$ denotes the convolution, $o_{ij}^{l-1} \neq 0$ and $o_{ij}^l \neq 0$ are the non-zero neuron elements, whereas $\cancel{o_{ij}^{l-1}} = 0$ and $\cancel{o_{ij}^l} = 0$

Authorized licensed use limited to: TONGJI UNIVERSITY. Downloaded on May 02,2021 at 01:58:06 UTC from IEEE Xplore. Restrictions apply.

are the zero neuron elements. With vectorization, Equation 13 can be rewritten as matrix multiplication, which is given by:

$$\begin{bmatrix} \cancel{o_{11}^l} & o_{12}^l & o_{21}^l & \cancel{o_{22}^l} \end{bmatrix} = \begin{bmatrix} w_{11}^l & w_{12}^l & w_{21}^l & w_{22}^l \end{bmatrix} \begin{bmatrix} \cancel{o_{11}^{l-1}} & o_{12}^{l-1} & o_{21}^{l-1} & \cancel{o_{22}^{l-1}} \\ o_{12}^{l-1} & o_{13}^{l-1} & \cancel{o_{22}^{l-1}} & \cancel{o_{23}^{l-1}} \\ o_{21}^{l-1} & o_{22}^{l-1} & o_{31}^{l-1} & \cancel{o_{32}^{l-1}} \\ \cancel{o_{22}^{l-1}} & \cancel{o_{23}^{l-1}} & \cancel{o_{32}^{l-1}} & \cancel{o_{33}^{l-1}} \end{bmatrix} \tag{14}$$

Figure 4 illustrates the sub-convolution of Equation 14, where only four out of sixteen multiplications are performed. It efficiently performs only the necessary computation by

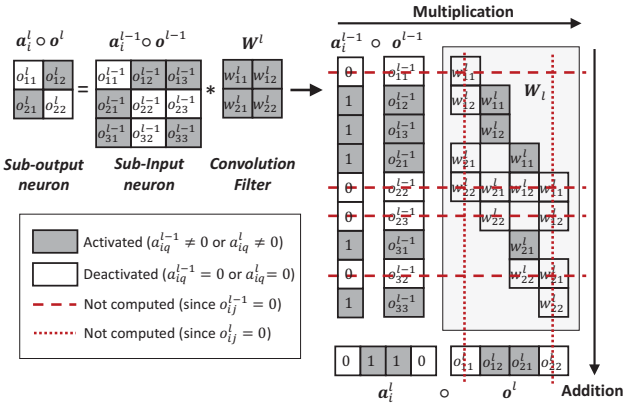


Fig. 4: **An example of sub-convolution**: Given a sub-input, sub-output neuron, and convolution filter, the sub-convolution is performed by walking through the sub-input (vertical direction) and sub-output (horizontal direction) only once to see if the elements are zero or not. By skipping computation related to zero-elements, the total computation time becomes proportional to the number of non-zeros.

checking the sub-input and sub-output neurons only once to see whether they are zero or not with linear complexity, i.e., $\mathcal{O}(|\tilde{\mathbf{O}}^{l-1}|+|\tilde{\mathbf{O}}^{l}|)$, while a naive algorithm takes $\mathcal{O}(|\tilde{\mathbf{O}}^{l}||\mathbf{W}^{l}|)$. For example, a sub-convolution between 100×100 input and 10×10 filter, which results in 91×91 output, requires only $18,281$ zero-element checks. On the contrary, a naive algorithm requires $828,100$ zero-check (i.e., $45\times$ less efficient).

Sub-Multiplication. For a fully-connected layer l, layer output $\mathbf{O}^l \in \mathbb{R}^{n \times m^l}$ is computed by taking the input, $\mathbf{O}^{l-1} \in \mathbb{R}^{n \times m^{l-1}}$ from the previous layer $l-1$, where n is the size of the input batch, m^{l-1} and m^l are the input and the output size, respectively. The output, $\mathbf{O}^l$ is obtained by multiplying a weight parameter $\mathbf{W}^l \in \mathbb{R}^{m^{l-1} \times m^l}$ to the input, $\mathbf{O}^{l-1}$.

Given sub-input $\tilde{\mathbf{O}}^{l-1}$ and sub-output $\tilde{\mathbf{O}}^{l}$ composed by $\mathbf{a}_i^{l-1}$ and $\mathbf{a}_i^l$, respectively, the computational complexity of matrix multiplication with weight parameter $\mathbf{W}^l$, $f_m^l(\cdot)$ is given by:

$$f_m^l(\mathbf{a}_i^{l-1}, \mathbf{a}_i^l, \mathbf{W}^l) = \mathcal{O}\left(\left\|\mathbf{a}_i^{l-1}\right\|_1 \left|\mathbf{a}_i^l\right| + \left\|\mathbf{a}_i^l\right\|_1 \left|\mathbf{W}^l\right| - \left\|\mathbf{a}_i^l\right\|_1 \left\|\mathbf{a}_i^{l-1}\right\|_1\right) \tag{15}$$

Equation 15 is proportional to the number of non-zero elements in the weight matrix used for multiplication. The first and second term indicate the number of row-wise and column-wise elements in the matrix, respectively. The last term cancels out the overlapped elements between the first and second term.

We define *sub-multiplication* as the matrix multiplication of a sub-network whose computational complexity is determined by $\mathbf{a}_i^{l-1}$ and $\mathbf{a}_i^l$ as shown in Equation 15. Equation 16 is an example of sub-multiplication with 1×3 sub-input, 1×3 sub-output neuron, and 3×3 weight.

$$\begin{bmatrix} \cancel{o_{11}^l} & o_{12}^l & o_{13}^l \end{bmatrix} = \begin{bmatrix} o_{11}^{l-1} & \cancel{o_{12}^{l-1}} & o_{13}^{l-1} \end{bmatrix} \begin{bmatrix} w_{11}^l & w_{12}^l & w_{13}^l \\ w_{21}^l & w_{22}^l & w_{23}^l \\ w_{31}^l & w_{32}^l & w_{33}^l \end{bmatrix} \tag{16}$$

With o_{12}^{l-1} and o_{11}^l being zero in sub-input and sub-output neuron, respectively, (1×3) by (3×3) matrix multiplication reduces to (1×2) by (2×2), as follows:

$$\begin{bmatrix} o_{12}^l & o_{13}^l \end{bmatrix} = \begin{bmatrix} o_{11}^{l-1} & o_{13}^{l-1} \end{bmatrix} \begin{bmatrix} w_{12} & w_{14} \\ w_{32} & w_{34} \end{bmatrix} \tag{17}$$

B. Time-Bound Back-Propagation

Time-Bound Back-Propagation. The training of a DNN is achieved by the compute-intensive process called back-propagation [90]. The goal of back-propagation is to update weight parameter, $\mathbf{W}^l$ of each layer l by computing the gradient [91] of a loss function, denoted as L, w.r.t. $\mathbf{W}^l$. The back-propagation is repeated with multiple iterations until the loss function, L converges to a particular criterion.

Given sub-output, $\tilde{\mathbf{o}}_j^l$ composed by $\mathbf{a}_i^l$, the gradient of the loss, L w.r.t. $\mathbf{W}^l$ for the j-th training instance, ∇L is:

$$\nabla L = \frac{\partial L}{\partial \mathbf{W}^l} = \frac{\partial L}{\partial \tilde{\mathbf{o}}_j^l} \cdot \mathbf{J}^l = \frac{\partial L}{\partial \left(\mathbf{a}_i^l \circ \sigma\left(\tilde{\mathbf{y}}_j^l\right)\right)} \cdot \mathbf{J}^l \tag{18}$$

where $\mathbf{J}^l \equiv \partial\left(\mathbf{a}_i^l \circ \sigma\left(\tilde{\mathbf{y}}_j^l\right)\right)/\partial \mathbf{W}^l$ is the Jacobian matrix [92], and $\tilde{\mathbf{y}}_j^l$ is defined as the same in Equation 1. By computing ∇L only for the non-zero elements of a sub-output neuron, $\tilde{\mathbf{o}}_j^l$, which is determined by the activation vector, $\mathbf{a}_i^l$, a back-propagation job, J_i is completed within the execution time budget, B_i. We name it as *time-bound back-propagation* since the gradient computation time is bounded by the size and configuration of a sub-network depending on a set of $\mathbf{a}_i^l$.

Since the sparse-neuron-aware gradient is also not supported by the existing back-propagation [90], we propose *sub-convolution-gradient* and *sub-multiplication-gradient* for convolutional and fully-connected layers, respectively.

Sub-Convolution-Gradient. Given sub-input neuron, $\tilde{\mathbf{O}}^{l-1}$ and sub-output neuron, $\tilde{\mathbf{O}}^l$ of a convolutional layer composed by $\mathbf{a}_i^{l-1}$ and $\mathbf{a}_i^l$, respectively, the complexity of computing convolution gradient with filter $\mathbf{W}^l$, $g_c^l(\cdot)$ is given by:

$$g_c^l(\mathbf{a}_i^{l-1}, \mathbf{a}_i^l, \mathbf{W}_i^l) = \mathcal{O}\left(\left\|\mathbf{a}_i^l\right\|_1 \left|\mathbf{W}^l\right| - \max\left(\left\|\mathbf{1} - \mathbf{a}_i^{l-1}\right\|_1 - \left\|\mathbf{a}_i^l\right\|_1 \left|\mathbf{W}^l\right|, 0\right)\right) \tag{19}$$

The first term depends on the number of non-zero elements in sub-output, and the second term depends on the number of non-zero elements in sub-input.

We define *sub-convolution-gradient* as the gradient of the convolution of a sub-network whose computational complexity is determined by $\mathbf{a}_i^{l-1}$ and $\mathbf{a}_i^l$ as shown in Equation 19. Equation 20 is an example of a sub-convolution-gradient for

Authorized licensed use limited to: TONGJI UNIVERSITY. Downloaded on May 02,2021 at 01:58:06 UTC from IEEE Xplore. Restrictions apply.

Equation 13, which shows only four out of sixteen differentiations are performed for the computation of ∇L.

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial w_{11}^l} & \frac{\partial L}{\partial w_{12}^l} & \frac{\partial L}{\partial w_{21}^l} & \frac{\partial L}{\partial w_{22}^l} \end{bmatrix} = \begin{bmatrix} \cancel{\frac{\partial L}{\partial o_{11}^l}} & \frac{\partial L}{\partial o_{12}^l} & \frac{\partial L}{\partial o_{21}^l} & \cancel{\frac{\partial L}{\partial o_{22}^l}} \end{bmatrix} \begin{bmatrix} \cancel{\frac{\partial o_{11}^l}{\partial w_{11}^l}} & \cancel{\frac{\partial o_{11}^l}{\partial w_{12}^l}} & \cancel{\frac{\partial o_{11}^l}{\partial w_{21}^l}} & \cancel{\frac{\partial o_{11}^l}{\partial w_{22}^l}} \\ \frac{\partial o_{12}^l}{\partial w_{11}^l} & \frac{\partial o_{12}^l}{\partial w_{12}^l} & \cancel{\frac{\partial o_{12}^l}{\partial w_{21}^l}} & \cancel{\frac{\partial o_{12}^l}{\partial w_{22}^l}} \\ \frac{\partial o_{21}^l}{\partial w_{11}^l} & \cancel{\frac{\partial o_{21}^l}{\partial w_{12}^l}} & \frac{\partial o_{21}^l}{\partial w_{21}^l} & \frac{\partial o_{21}^l}{\partial w_{22}^l} \\ \cancel{\frac{\partial o_{22}^l}{\partial w_{11}^l}} & \cancel{\frac{\partial o_{22}^l}{\partial w_{12}^l}} & \cancel{\frac{\partial o_{22}^l}{\partial w_{21}^l}} & \cancel{\frac{\partial o_{22}^l}{\partial w_{22}^l}} \end{bmatrix} \quad (20)$$

Here, the matrix on the right-hand side is the Jacobian, $\cancel{\frac{\partial o}{\partial w}} = 0$ and $\frac{\partial \cancel{o}}{\partial w} = 0$ denote that the computation related to $\frac{\partial o}{\partial w}$ is skipped since the corresponding sub-input and sub-output neuron element is zero, respectively. The zero elements of sub-output neuron eliminate the corresponding rows of the Jacobian, e.g., the first and last rows of the Jacobian become unnecessary since o_{11}^l and o_{22}^l are zeros as shown in Equation 13. The zero elements of sub-input neuron result in scattered elimination of individual derivatives in the Jacobian, e.g., $\frac{\partial o_{12}^l}{\partial w_{21}^l}$ is not computed since the corresponding sub-input neuron element, o_{22}^{l-1} is zero, i.e., $\frac{\partial o_{12}^l}{\partial w_{21}^l} = o_{22}^{l-1} = 0$.

Sub-Multiplication-Gradient. Given sub-input neuron, $\tilde{\mathbf{O}}^{l-1}$ and sub-output neuron, $\tilde{\mathbf{O}}^l$ of a fully-connected layer composed by $\mathbf{a}_i^{l-1}$ and $\mathbf{a}_i^l$, respectively, the complexity of computing multiplication gradient with weight $\mathbf{W}^l$, $g_m^l(\cdot)$ is:

$$g_m^l(\mathbf{a}_i^{l-1}, \mathbf{a}_i^l, \mathbf{W}^l) = \mathcal{O}\left(\left\|\mathbf{a}_i^{l-1}\right\|_1 \left|\mathbf{a}_i^l\right| + \left\|\mathbf{a}_i^l\right\|_1 \left|\mathbf{W}^l\right| - \left\|\mathbf{a}_i^l\right\|_1 \left\|\mathbf{a}_i^{l-1}\right\|_1\right) \quad (21)$$

We define *sub-multiplication-gradient* as the gradient of matrix multiplication of a sub-network whose computation complexity is determined by $\mathbf{a}_i^{l-1}$ and $\mathbf{a}_i^l$ as shown in Equation 21, which is equivalent to the sub-multiplication (Equation 15).

VI. Implementation

We implement SubFlow as an extended version of the TensorFlow library [49], which is fully compatible with the existing TensorFlow operations. The programmers can easily apply SubFlow to their DNNs in the same way they design DNNs without SubFlow. SubFlow is implemented to run on both CPU and GPU, which makes it adaptable to a wide range of platforms. For CPU, it is implemented based on the Eigen library [51] that is optimized to perform matrix operations in CPU. For GPU, it is implemented with CUDA library [52], [93] to support the parallel computation of sub-networks like the other GPU operations.

In constructing and executing sub-networks, SubFlow does not generate and save multiple versions of sub-networks a priori. Based on a single DNN designed by the programmer, SubFlow is implemented to construct and execute sub-networks at run-time based on time-bound sparse execution.

Figure 5 shows the SubFlow framework, along with the TensorFlow, which consists of a sub-network library, Python client operations, and kernel implementations.

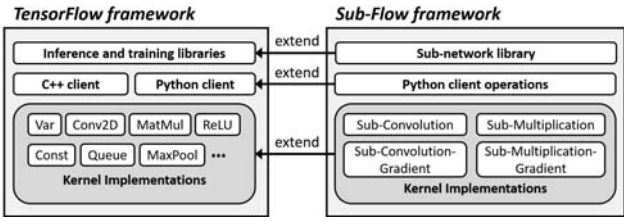


Fig. 5: **SubFlow Framework**: The SubFlow framework consists of the sub-network library, python client operations, and kernel implementations. It is fully compatible with TensorFlow and provides all the necessary components of the TensoFlow hierarchy.

Sub-Network Library. It is a high-level module that computes the importance of output neurons in the DNN for the construction of sub-networks. Since the computation of the Hessian matrix in Equation 11 is intractable with DNNs of considerable size, an approximation of the Hessian using sample covariance is computed [82] instead.

This module is also responsible for the online construction of sub-networks. Based on the importance of neurons and time budget of the i-th job, J_i, it produces $\mathbf{a}_i^l$ in Equation 1 for all layers at run-time, except the last layer where all the final output neurons of the DNN should be selected.

Python Client Operations. Python client operations provide the programmer with a set of wrapper APIs that help design a DNN model using SubFlow, which is fully compatible with other existing operations of TensorFlow. Each API represents and corresponds to its kernel implementation that is executed with higher efficiency when a sub-network runs.

Kernel Implementations. They are the lower-level implementation of four operations used in SubFlow, i.e., sub-convolution, sub-multiplication, sub-convolution-gradient, and sub-multiplication-gradient. These operations are responsible for executing a sub-network of the DNN designed with Python client operations. Written in C and C++, they are optimized to hardware platforms and able to perform the efficient time-bound execution of sub-network with minimum overhead.

VII. Experiment

A. Experimental Setup

Hardware and Software. We conduct experiments on a system consisting of Intel Core i9-9900K CPU, NVIDIA RTX 2080 Ti GPU with 11 GB of memory, and 32 GB of system memory (RAM). We use TensorFlow 1.13.1 with Eigen 3.3.90 and CUDA 10.0 (CUDNN 7.4.2) for implementation.

Datasets and DNN Models. We use three standard machine learning datasets in our evaluation, i.e., MNIST [53] (handwritten digits), CIFAR-10 [55] (image classification), and GSC (Google Speech Commands V2) [56]. For each dataset, the state-of-the-art DNN model that provides the best performance for the dataset is designed with SubFlow, i.e., LeNet-5 [53], AlexNet [6], and KWS (Key-Word Spotting) architecture [54]. Table I summarizes the DNN architectures and datasets.

Time Measurement. The execution time of a sub-network, including individual operations, is measured by using Ten-

Authorized licensed use limited to: TONGJI UNIVERSITY. Downloaded on May 02,2021 at 01:58:06 UTC from IEEE Xplore. Restrictions apply.

	LeNet-5 (MNIST)	AlexNet (CIFAR-10)	KWS (GSC)
Layer 1	Input: 28×28×1	Input: 32×32×3	Input: 61×13×1
Layer 2	Conv1: 5×5×1×6	Conv1: 3×3×3×64	Conv1: 12×6×1×64
Layer 3	Conv2: 5×5×6×16	Conv2: 3×3×64×192	Conv2: 6×3×64×64
Layer 4	FC1: 400	Conv3: 3×3×192×384	FC1: 1024
Layer 5	FC2: 84	FC1: 4096	FC2: 512
Layer 6	FC3 (Output): 10	FC2: 2048	FC3 (Output): 35
Layer 7		FC3 (Output): 10	

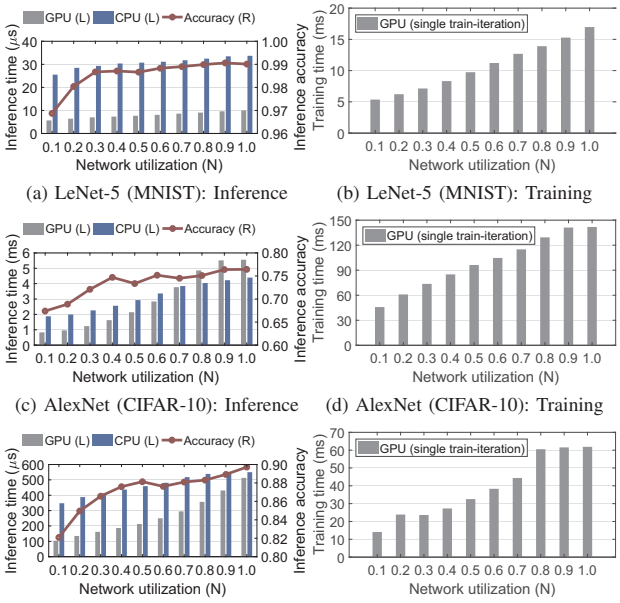
* KWS: Key-Word Spotting, Conv: Convolution layer, FC: Fully-connected layer

TABLE I: The DNN models and datasets used in the evaluation.

sorFlow's Timeline tool [94] that traces and records the execution time of all the operations of a DNN in the unit of microseconds. We analyze and compare the execution time of different sub-networks by analyzing their tracing log files saved in the JSON (JavaScript Object Notation) format [95].

B. End-to-End Execution Time and Performance

We evaluate the end-to-end execution time and performance (i.e., inference accuracy) of sub-networks of different sizes determined by the network utilization, N. The inference time is measured on both CPU and GPU by calculating the average feed-forward time on the entire test samples as one input batch. The training time is evaluated on GPU by measuring the execution time of one training iteration for both feed-forward and back-propagation with a mini-batch size of 96 samples. We use separate datasets for training and testing.



(a) LeNet-5 (MNIST): Inference (b) LeNet-5 (MNIST): Training

(c) AlexNet (CIFAR-10): Inference (d) AlexNet (CIFAR-10): Training

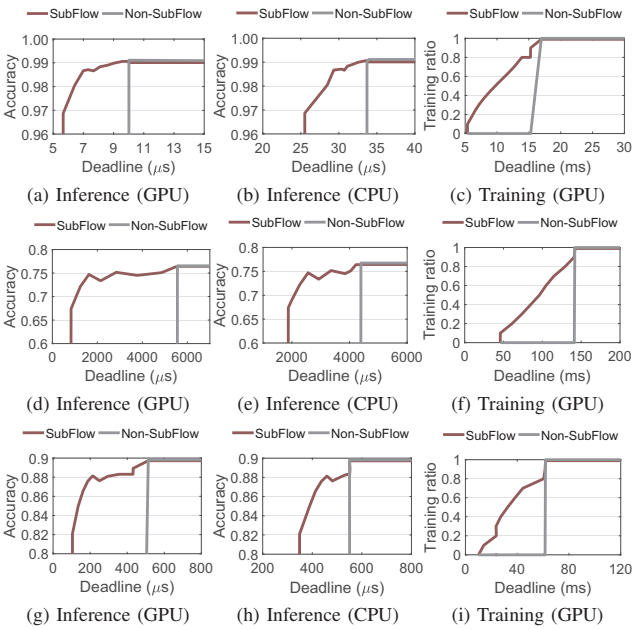
(e) KWS (GSC): Inference (f) KWS (GSC): Training

Fig. 6: **The end-to-end execution time and inference accuracy over the network utilization (N)**: The inference time is measured on both GPU and CPU, and training time is measured on GPU.

Figure 6 shows the end-to-end inference and training time of the three DNNs for different network utilizations, i.e., from $N = 0.1$ to 1.0. All three DNNs show that their inference time decreases as N decreases without significant loss of inference accuracy. For example, the sub-network of AlexNet with $N = 0.1$ achieves 6.7x speedup with only 9% drop of inference accuracy, i.e., from 76% to 67%. The inference accuracy of LeNet-5 stays almost the same, i.e., a maximum 2% drop while providing 2x speedup when $N = 0.1$. The training time also decreases as N decreases, e.g., the training time of KWS is reduced by 4.4x with a sub-network of $N = 0.1$. However, their speedup is not linear to N since 1) all the neurons in the first and the last layers are activated for all sub-networks in our implementation, and 2) run-time overhead occurs.

C. Usefulness (Utility) of DNN

We next evaluate how SubFlow improves the usefulness of a DNN given dynamic deadlines against the same original DNNs that run without SubFlow. We measure the usefulness of inference and training tasks based on the inference accuracy and the training ratio that indicates the ratio of the DNN components trained within the deadline, respectively. Figure 7 shows the usefulness of the three DNNs, i.e., the inference accuracy (GPU and CPU) and training ratio (GPU) over a range of dynamic deadlines. Unlike the non-SubFlow DNNs, SubFlow makes the best use of the DNNs for a given deadline by flexibly utilizing them, and completing the inference or training task in time. For example, SubFlow AlexNet achieves 74% average inference accuracy, which is 2% lower than the original DNN (76%), for the deadlines ranging between 1800μ and 5700μs, while the non-SubFlow DNN achieves 0% accuracy for the same set of deadlines as shown in Figure 7d. As the deadline gets closer to the execution time of the original DNN, the accuracy of SubFlow AlexNet approaches 76% since the full network is executed. On the other hand, the non-SubFlow DNN results in zero usefulness unless the deadline is equivalent to or larger than the execution time. For deadlines smaller than that, its usefulness is zero since they are not even executed, or the execution completed after the deadline.



(a) Inference (GPU) (b) Inference (CPU) (c) Training (GPU)

(d) Inference (GPU) (e) Inference (CPU) (f) Training (GPU)

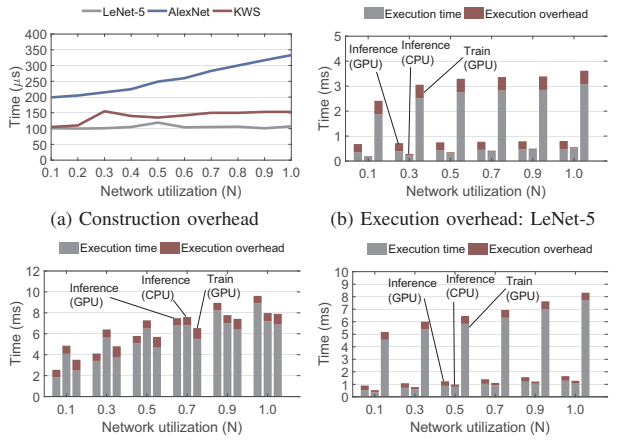
(g) Inference (GPU) (h) Inference (CPU) (i) Training (GPU)

Fig. 7: **The usefulness (inference accuracy and training ratio) over dynamic deadline**: SubFlow vs. non-SubFlow. (a)-(c): LeNet-5 (MNIST), (d)-(f): AlexNet (CIFAR-10), and (g)-(i): KWS (GSC).

Authorized licensed use limited to: TONGJI UNIVERSITY. Downloaded on May 02,2021 at 01:58:06 UTC from IEEE Xplore. Restrictions apply.

D. Run-time Overhead

We measure two types of run-time overheads of SubFlow: 1) the sub-network construction overhead, and 2) the sub-network execution overhead for one single input sample which is the additional computation time required to run a DNN with SubFlow. These two overheads are obtained 1) by measuring the time needed to generate the activation vector, $\mathbf{a}_i^l$ in Equation 6 for all layers, which is required to construct a sub-network, and 2) by measuring the execution overhead time and the actual execution time separately during the feed-forward and back-propagation.



(a) Construction overhead

(b) Execution overhead: LeNet-5

(c) Execution overhead: AlexNet

(d) Execution overhead: KWS

Fig. 8: **The run-time overhead**: (a): the construction overhead for all the three DNNs. (b), (c), and (d): the actual execution time (gray) vs. execution overhead (red) of each DNN.

Figure 8a shows the sub-network construction overhead of the three DNNs for different network utilizations, which stays the same (LeNet-5 and KWS) or increases slightly (AlexNet) with increased network utilization. They do not tend to change significantly with different network utilization settings since the same length of activation vector is generated for sub-network of any size; the only difference between sub-networks is the composition of ones and zeros. Also, their absolute time costs are low since an activation vector is efficiently generated from the rank of neurons that is pre-computed at compile-time. Figure 8b, 8c, and 8d show the sub-network execution overhead, which is higher than the construction overhead. For example, the inference overhead of AlexNet on GPU, $709\mu s$, which is 7% of the total inference time ($8915\mu s$), is almost twice higher than the construction overhead of the full-size sub-network ($333\mu s$, $N = 1.0$). The ratio of execution overhead to the total execution time increases as the sub-network size decreases, e.g., from 7% to 27% in AlexNet with network utilization settings 1.0 and 0.1, respectively. The execution overhead ratio increases for smaller sub-networks since the overhead remains similar for all sizes of sub-network while the actual computation time decreases with the size. It shows that the execution overhead is critical to small sub-networks and should be further decreased so that they can be efficiently executed with tighter time constraints.

E. Comparison with the State-of-the-Art

We compare SubFlow with two state-of-the-art DNN execution algorithms: 1) BranchyNet [96] that makes an early exit of the DNN for fast inference and 2) AdapDeep [97] that accelerates a DNN with a combination of compression techniques. Table II provides their inference and training speeds on CPU and/or GPU, and the inference accuracy of the two DNNs, i.e., LeNet-5 (MNIST) and AlexNet (CIFAR-10). We observe that SubFlow achieves comparable speedup and inference accuracy to the other two. Also, it achieves flexible execution for both inference and training, unlike the other two methods that lack such flexibility (AdaDeep) and training speedup (BranchyNet). For example, SubFlow AlexNet on GPU achieves dynamic speedup for both inference (1.0x–6.7x) and training (1.0x–3.1x), while BranchyNet achieves 1.0x–2.4x speedup only for inference without providing dynamic training speedup. AdaDeep achieves a fixed speedup for inference (2.3x on CPU), but does not achieve training speedup at all.

LeNet-5 (MNIST)	**Inference Speed (CPU / GPU)**	**Training Speed (GPU)**	**Inference Accuracy**
SubFlow	**1.0x–1.3x / 1.0x–1.8x**	**1.0x–3.2x**	**0.97–0.99**
BranchyNet [96]	1.0x–5.4x / 1.0–4.7x	N/A	0.98–0.99
AdaDeep [97]	1.8x / N/A	N/A	0.97
AlextNet (CIFAR-10)	**Inference Speed (CPU / GPU)**	**Training Speed (GPU)**	**Inference Accuracy**
SubFlow	**1.0x–2.4x / 1.0x–6.7x**	**1.0x–3.1x**	**0.67–0.76**
BranchyNet [96]	1.0–1.5x / 1.0–2.4x	N/A	0.75–0.79
AdaDeep [97]	2.3x / N/A	N/A	0.72

* For SubFlow, the network utilization is set as $N = [0.1, 1.0]$.

TABLE II: Comparison between SubFlow, BranchNet, and AdaDeep.

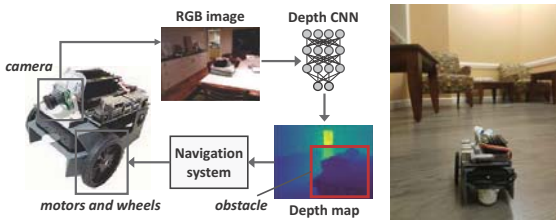
VIII. Application

We implement an autonomous mobile robot as an example application of SubFlow, which detects obstacles by generating depth maps from a camera image in real-time [9], [58], [67], [68]. While driving, a CNN (convolutional neural network) transforms an RGB image into a depth map where the required latency of transformation changes based on the traveling speed of the robot. The faster it runs, the quicker the transformation should be performed to detect an obstacle in time. Figure 9a shows our mobile robot that executes a depth-estimation CNN [58] with SubFlow on its GPU for obstacle detection. It is implemented using Jetson Nano [57], an embedded GPU platform having NVIDIA Maxwell GPU, ARM A57 CPU, and 4 GB of RAM. The robot has a camera in the front, and two motors and wheels on both sides installed on the skeleton that we printed on a 3D printer. Table III shows the architecture of the depth estimation CNN [58] executed by the mobile robot. We use NYU depth dataset V2 [98] for training and testing.

A. End-to-End Execution Time and Performance

Figure 10 shows the execution time and depth estimation error of the CNN over the network utilization, N. The execution time is measured the same way as in Section VII, and the estimation error is measured with linear RMSE [67], which is calculated by $\sqrt{\frac{1}{n}\sum_{j=1}^{n}\left\|\tilde{\mathbf{y}}_j - \mathbf{y}_j^*\right\|_2^2}$, where $\tilde{\mathbf{y}}_j$ and $\mathbf{y}_j^*$ is the j-th output of a sub-network and ground truth, respectively.

Authorized licensed use limited to: TONGJI UNIVERSITY. Downloaded on May 02,2021 at 01:58:06 UTC from IEEE Xplore. Restrictions apply.



(a) Depth-based obstacle detection (b) The mobile robot

Fig. 9: SubFlow robot performing depth-based obstacle detection.

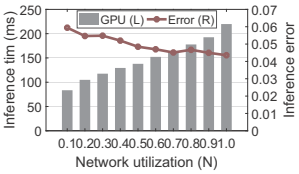
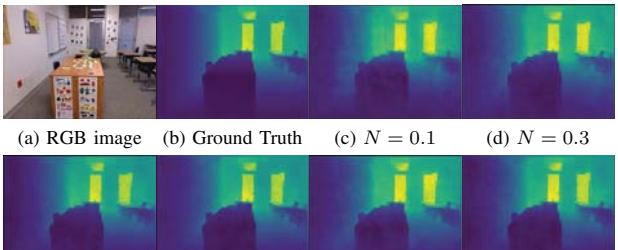


Fig. 10: Execution time and error over the network utilization.

	Depth CNN (NYU2)
Layer 1	Input: 60×80×3
Layer 2	Conv1: 11×11×3×96
Layer 3	Conv2: 5×5×96×256
Layer 4	Conv3: 3×3×256×384
Layer 5	Conv4: 3×3×384×384
Layer 6	FC1: 2048
Layer 7	FC2 (Output): 4800

TABLE III: The depth estimation CNN architecture [58].

Figure 11 shows example depth maps generated from different sub-networks with $N = 0.1,\ 0.3,\ 0.5,\ 0.9$, and 1.0.



(a) RGB image (b) Ground Truth (c) $N = 0.1$ (d) $N = 0.3$

(e) $N = 0.5$ (f) $N = 0.7$ (g) $N = 0.9$ (h) $N = 1.0$

Fig. 11: Depth map images generated from different settings of N.

B. Real-World Deployment

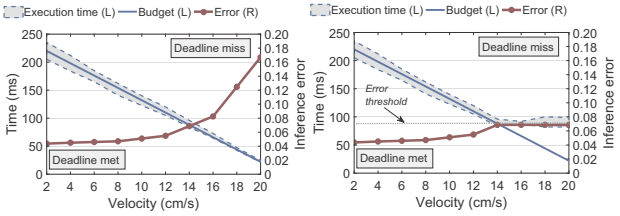
As a real-life experiment, we evaluate the execution time and depth estimation error by running the mobile robot at various speeds that impose different execution time budgets (latency) on the depth CNN. We deploy the robot in the corridor, kitchen, and bedroom of an apartment that has typical furniture such as chairs, desks, and a bed (Figure 9b). The robot runs for three hours and executes 50,000 CNN jobs. We randomly change the speed of the robot (2cm/s–20cm/s) to enable dynamic deadlines that we empirically obtain during preliminary experiment. The result, summarized in Table IV, shows that the execution of the CNN completes within the time budget with a small variance.

Since obstacle detection is critical to safe driving, the robot may want to execute only the sub-networks generating depth map with an error lower than a threshold, which makes it navigate without a collision. To experiment in this scenario, we limit the execution of sub-networks that cause large errors (0.068). Figure 12 shows the execution time and error over velocity with and without the error threshold. The execution

Velocity	**20 cm/s**	**16 cm/s**	**12 cm/s**	**8 cm/s**	**4 cm/s**	**2 cm/s**
Budget	**22 ms**	**66 ms**	**110 ms**	**154 ms**	**198 ms**	**220 ms**
Avg-ET	23.1 ms	68.2 ms	112.6 ms	152.2 ms	197.6 ms	219.8 ms
Min-ET	22.5 ms	66 ms	108 ms	147 ms	192 ms	212.5 ms
Max-ET	23.62 ms	69.7 ms	114 ms	155.7 ms	202 ms	225 ms
N	0.01	0.04	0.25	0.61	0.92	1.00
Error	0.1669438	0.082438	0.054829	0.046986	0.044965	0.04365

* **Velocity**: Traveling speed of the robot (centimeters per second), **Budget**: Execution time budget (milliseconds), **Avg-ET**: Average execution time (milliseconds), **Min-ET**: Minimum execution time (milliseconds), **Max-ET**: Maximum execution time (milliseconds), **N**: Network utilization, **Error**: Depth estimation error

TABLE IV: Execution time budget, actual execution time, network utilization, and depth estimation error of the mobile robot with various running speeds.



(a) Error threshold: None (b) Error threshold: 0.068

Fig. 12: **The execution time and error over velocity with and without an error threshold**: The execution time budget that changes based on the velocity is drawn with a diagonal line. The deadline is met if the execution time is under the diagonal line, missed otherwise.

time budget is shown as a diagonal line, implying that the execution time above the line is a deadline miss. Without the threshold, Figure 12a shows that the robot meets all the deadlines in the entire speed range, but generates a depth map with a high error at high speed. On the other hand, Figure 12b shows that the error is limited to 0.068 for all the speeds by executing only the sub-networks with an error below the threshold, which ensures the desired level of performance. In consequence of not performing the sub-networks resulting in error higher than the threshold, the robot misses the deadlines when running faster than 14 cm/s in return for the low error.

IX. Discussion

Scalability to Larger DNNs. Although the DNNs used in the evaluation, e.g., AlexNet [6] (15M parameters), are smaller than ResNet [99] (26M parameters), we expect SubFlow to achieve better results with larger networks like ResNet since they have more room for optimization [100]. An induced sub-graph can be constructed with residual connections, and SubFlow supports both convolution and fully-connected layers. In this paper, we followed to design our workload based on many recent works [79], [97], [101]–[103] for embedded systems, which demonstrates that results hold for both low-end CPUs and embedded GPUs. We hypothesize that smaller DNNs like LeNet-5 [53] used in the evaluation are harder cases for SubFlow as there is little scope for speedup and/or compression.

Accuracy Requirements. While SubFlow minimizes the loss of inference accuracy when constructing a sub-network for the time-bound execution, the accuracy drop is expected to increase in general as the size of a sub-network decrease. Since

Authorized licensed use limited to: TONGJI UNIVERSITY. Downloaded on May 02,2021 at 01:58:06 UTC from IEEE Xplore. Restrictions apply.

the accuracy is critical for many safety-critical applications, SubFlow limits the maximum loss of accuracy above a certain level by controlling the network utilization parameter that limits the construction and execution of sub-networks whose expected accuracy is lower than the desired level. The expected accuracy over network size is obtained by running various sizes of sub-networks offline before the DNN is deployed on the system. For some applications where both accuracy and real-time execution are critical, SubFlow can provide intermediate inference results faster than the full-size network, which serves as preliminary guidance before getting the high-accuracy result from the full-size network.

X. Related Work

Real-Time DNNs. RTDNN [104] adapts the parameters and structure of DNN to a dataset in real-time conditions. Although it performs an adaptation without requiring a significant number of samples, it does not support convolutional DNN and relies on competitive learning [105] that is not widely used in many DNNs. Based on the constructive network model [106], [107] proposed a real-time learning algorithm, which can automatically select appropriate values of neural quantizers and determine the parameters of the network. However, their learning is performed without any real-time constraints, which is different from SubFlow having definite time budgets. Above all, none of them provide timing guarantee of DNN execution.

Imprecise Computing. The imprecise computation [108]–[110] divides a time-critical task into two sub-tasks: mandatory and optional. The mandatory sub-task is executed to completion to produce an acceptable result. The optional sub-task refines the result to reduce the error in the result. The milestone, sieve function, and multiple version method [111]–[114] are the popular algorithms for it. However, the division of a task is not trivial and increases the complexity of scheduling by adding optional tasks to the system. Also, dividing a task into only two parts does not provide flexible execution. SubFlow does not require an artificial division of a task and automatically executes the proper amount of computation based on flexible construction and execution of sub-networks.

DNN Compression/Prunning. The need to deploy DNNs on resource constrained systems motivated techniques that can reduce the storage and computational costs, including knowledge distillation [38]–[40], low-rank factorization [41]–[43], pruning [44]–[48], quantization [115]–[117], compression with structured matrices [118], [119], network binarization [120]–[122], and hashing [123]. However, they do not provide real-time guarantee due to their primary focus on size reduction. Also, the significantly compressed DNNs do not run nearly as significantly faster since most parameters are pruned in fully-connected layers while convolutional layers consume most computation time, as shown in [48], [117], [124]. Although some algorithms, such as DeepIoT [79], [125] compress DNNs achieving less execution time, the final network is not dynamically changed once it is compressed offline. Moreover, they lack easy-to-follow procedures and require significant effort, e.g., architecture modification, multi-rounds of retraining, fine-tuning. In contrast, SubFlow enables the run-time execution of multiple sub-networks of the DNN instead of compressing the DNN into one single network without requiring such an effort. SubFlow also supports time-bound training, which is missing in most compression works that only focus on the inference.

Improving Inference Speed. To improve the inference speed, parallel techniques such as SIMD [126] have been used [127], which is also employed in the implementation of SubFlow. Also, faster algorithms specifically for 3x3 convolutional filters have been studied [128] for VGGNet [77] and ResNet [99]. The early exit is another approach. CDL [129] adds classifiers to each layer and monitors the output to decide whether a sample can be exited early. BranchyNet [96] enables more general branches with additional layers at each exit point. In contrast, SubFlow executes all the layers without exiting in the middle. Instead, some neurons of each layer are selected and executed for speedup. To speed up sparse convolution, efficient sparse DNNs such as [130], [46], and [131] have been proposed. Escoin [132] applies the direct sparse convolution [89] to GPU in optimizing parallelism and locality. SparseSep [133] leverages the sparsification of fully-connected layers and the separation of convolutional kernels for wearable devices. Although SubFlow uses the direct sparse convolution, it does not rely on CSR (compressed sparse row) format that incurs overhead of decoding the sparse format, unlike them.

Improving Training Speed. Dropout [134] and DropConnect [135] can be used not only to increase the performance with reduced overfitting but also to reduce training time by performing back-propagation only for a part of DNN. StochasticDepth [136] starts with deep networks, but during training, randomly drops a subset of layers and bypasses them. Highway networks [137] proposes to modify the architecture of deep feed-forward networks such that information flow across layers becomes easier. In meProp [138], only a small subset of the gradient is computed to update the model parameters in back-propagation. MSBP [139] proposes to store unpropagated gradients in memory for the next learning. They either change the DNN architecture or select weight parameters to be trained based on the magnitude, which may eliminate wrong parameters [44], [82], unlike SubFlow that does not modify architecture and use the second-order derivative for selection.

XI. Conclusion

We propose SubFlow that enables real-time inference and training of a DNN by dynamically executing an induced sub-graph of the DNN according to varying time budget. We implement SubFlow by extending TensorFlow, which allows a programmer to design time-aware DNNs based on SubFlow. Our empirical evaluation result shows that time-bound inference and training are achieved without experiencing significant performance loss. We implement an autonomous robot as an application of SubFlow, which demonstrates that the object detection task is completed within the time budget that dynamically changes based on the running speed of the robot.

Authorized licensed use limited to: TONGJI UNIVERSITY. Downloaded on May 02,2021 at 01:58:06 UTC from IEEE Xplore. Restrictions apply.

ACKNOWLEDGEMENT

This paper was supported, in part, by NSF grants CNS-1816213 and CNS-1704469 and NIH grant 1R01LM013329-01.

REFERENCES

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[2] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[3] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[4] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing," *ieee Computational intelligenCe magazine*, vol. 13, no. 3, pp. 55–75, 2018.

[5] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 815–823.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[7] M. Bojarski, P. Yeres, A. Choromanska, K. Choromanski, B. Firner, L. Jackel, and U. Muller, "Explaining how a deep neural network trained with end-to-end learning steers a car," *arXiv preprint arXiv:1704.07911*, 2017.

[8] X. Chen, H. Ma, J. Wan, B. Li, and T. Xia, "Multi-view 3d object detection network for autonomous driving," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 1907–1915.

[9] X. Chen, K. Kundu, Z. Zhang, H. Ma, S. Fidler, and R. Urtasun, "Monocular 3d object detection for autonomous driving," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2147–2156.

[10] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 2722–2730.

[11] R. Socher, Y. Bengio, and C. Manning, "Deep learning for nlp," *Tutorial at Association of Computational Logistics (ACL)*, 2012.

[12] L. Deng and Y. Liu, *Deep Learning in Natural Language Processing*. Springer, 2018.

[13] W. Khan, A. Daud, J. A. Nasir, and T. Amjad, "A survey on the state-of-the-art machine learning models in the context of nlp," *Kuwait journal of Science*, vol. 43, no. 4, 2016.

[14] R. Miotto, F. Wang, S. Wang, X. Jiang, and J. T. Dudley, "Deep learning for healthcare: review, opportunities and challenges," *Briefings in bioinformatics*, vol. 19, no. 6, pp. 1236–1246, 2017.

[15] F. Jiang, Y. Jiang, H. Zhi, Y. Dong, H. Li, S. Ma, Y. Wang, Q. Dong, H. Shen, and Y. Wang, "Artificial intelligence in healthcare: past, present and future," *Stroke and vascular neurology*, vol. 2, no. 4, pp. 230–243, 2017.

[16] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. Van Der Laak, B. Van Ginneken, and C. I. Sánchez, "A survey on deep learning in medical image analysis," *Medical image analysis*, vol. 42, pp. 60–88, 2017.

[17] Z. Dong, Y. Liu, H. Zhou, X. Xiao, Y. Gu, L. Zhang, and C. Liu, "An energy-efficient offloading framework with predictable temporal correctness," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. ACM, 2017, p. 19.

[18] G. A. Elliott and J. H. Anderson, "Exploring the multitude of real-time multi-gpu configurations," in *2014 IEEE Real-Time Systems Symposium*. IEEE, 2014, pp. 260–271.

[19] H. Zhou and C. Liu, "Task mapping in heterogeneous embedded systems for fast completion time," in *2014 International Conference on Embedded Software (EMSOFT)*. IEEE, 2014, pp. 1–10.

[20] G. A. Elliott, B. C. Ward, and J. H. Anderson, "Gpusync: A framework for real-time gpu management," in *2013 IEEE 34th Real-Time Systems Symposium*. IEEE, 2013, pp. 33–44.

[21] G. A. Elliott and J. H. Anderson, "An optimal k-exclusion real-time locking protocol motivated by multi-gpu systems," *Real-Time Systems*, vol. 49, no. 2, pp. 140–170, 2013.

[22] ——, "Robust real-time multiprocessor interrupt handling motivated by gpus," in *2012 24th Euromicro Conference on Real-Time Systems*. IEEE, 2012, pp. 267–276.

[23] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt, "Gdev: First-class {GPU} resource management in the operating system," in *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, 2012, pp. 401–412.

[24] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "Timegraph: Gpu scheduling for real-time multi-tasking environments," in *Proc. USENIX ATC*, 2011, pp. 17–30.

[25] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "Ptask: operating system abstractions to manage gpus as compute devices," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 233–248.

[26] C. Augonnet, S. Thibault, and R. Namyst, "Starpu: a runtime system for scheduling tasks over accelerator-based multicore machines," Ph.D. dissertation, INRIA, 2010.

[27] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2009, pp. 45–55.

[28] Ö. Ş. Taş, F. Kuhnt, J. M. Zöllner, and C. Stiller, "Functional system architectures towards fully automated driving," in *2016 IEEE Intelligent vehicles symposium (IV)*. IEEE, 2016, pp. 304–309.

[29] A. Pongpunwattana and R. Rysdyk, "Real-time planning for multiple autonomous vehicles in dynamic uncertain environments," *Journal of Aerospace Computing, Information, and Communication*, vol. 1, no. 12, pp. 580–604, 2004.

[30] Z. Shiller, Y.-R. Gwo *et al.*, "Dynamic motion planning of autonomous vehicles," *IEEE Transactions on Robotics and Automation*, vol. 7, no. 2, pp. 241–249, 1991.

[31] P. Chen, Y. Dang, R. Liang, W. Zhu, and X. He, "Real-time object tracking on a drone with multi-inertial sensing data," *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, no. 1, pp. 131–139, 2017.

[32] T. Nägeli, L. Meier, A. Domahidi, J. Alonso-Mora, and O. Hilliges, "Real-time planning for automated multi-view drone cinematography," *ACM Transactions on Graphics (TOG)*, vol. 36, no. 4, p. 132, 2017.

[33] M. Soto, P. Nava, and L. Alvarado, "Drone formation control system real-time path planning," in *AIAA Infotech@ Aerospace 2007 Conference and Exhibit*, 2007, p. 2770.

[34] S. He, Y. Liu, and H. Zhou, "Optimizing smartphone power consumption through dynamic resolution scaling," in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*. ACM, 2015, pp. 27–39.

[35] C. Wanpeng and B. Wei, "Adaptive and dynamic mobile phone data encryption method," *China Communications*, vol. 11, no. 1, pp. 103–109, 2014.

[36] H. Balog, D. Salmon, P. DeVries, M. Saks, B. Jansen, and S. Arnison, "Dynamic protocol selection and routing of content to mobile devices," Feb. 21 2002, uS Patent App. 09/823,654.

[37] D. B. Stewart and P. K. Khosla, "Real-time scheduling of sensor-based control systems," *IFAC Proceedings Volumes*, vol. 24, no. 2, pp. 139–144, 1991.

[38] G. Chen, W. Choi, X. Yu, T. Han, and M. Chandraker, "Learning efficient object detection models with knowledge distillation," in *Advances in Neural Information Processing Systems*, 2017, pp. 742–751.

[39] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[40] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, "Fitnets: Hints for thin deep nets," *arXiv preprint arXiv:1412.6550*, 2014.

[41] Y. Ioannou, D. Robertson, J. Shotton, R. Cipolla, and A. Criminisi, "Training cnns with low-rank filters for efficient image classification," *arXiv preprint arXiv:1511.06744*, 2015.

[42] C. Tai, T. Xiao, Y. Zhang, X. Wang *et al.*, "Convolutional neural networks with low-rank regularization," *arXiv preprint arXiv:1511.06067*, 2015.

[43] T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran, "Low-rank matrix factorization for deep neural network training with high-dimensional output targets," in *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 2013, pp. 6655–6659.

[44] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in neural information processing systems*, 1990, pp. 598–605.

Authorized licensed use limited to: TONGJI UNIVERSITY. Downloaded on May 02,2021 at 01:58:06 UTC from IEEE Xplore. Restrictions apply.

[45] A. Polyak and L. Wolf, "Channel-level acceleration of deep face representations," *IEEE Access*, vol. 3, pp. 2163–2175, 2015.

[46] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *arXiv preprint arXiv:1608.08710*, 2016.

[47] R. Yu, A. Li, C.-F. Chen, J.-H. Lai, V. I. Morariu, X. Han, M. Gao, C.-Y. Lin, and L. S. Davis, "Nisp: Pruning networks using neuron importance score propagation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 9194–9203.

[48] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient dnns," in *Advances In Neural Information Processing Systems*, 2016, pp. 1379–1387.

[49] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[50] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[51] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[52] NVIDIA, "Nvidia cuda home page," https://developer.nvidia.com/cuda-zone, 2019.

[53] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[54] T. Sainath and C. Parada, "Convolutional neural networks for small-footprint keyword spotting," 2015.

[55] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.

[56] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," *arXiv preprint arXiv:1804.03209*, 2018.

[57] NVIDIA, "Jetson nano," https://developer.nvidia.com/embedded/jetson-nano-developer-kit, 2019.

[58] P. Chakravarty, K. Kelchtermans, T. Roussel, S. Wellens, T. Tuytelaars, and L. Van Eycken, "Cnn-based single image obstacle avoidance on a quadrotor," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2017, pp. 6369–6374.

[59] R. Diestel, "Graph theory (graduate texts in mathematics). 3rd," *Ed Springer*, pp. 17–18, 2006.

[60] P. Li, X. Chen, and S. Shen, "Stereo r-cnn based 3d object detection for autonomous driving," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 7644–7652.

[61] X. Pan, J. Shi, P. Luo, X. Wang, and X. Tang, "Spatial as deep: Spatial cnn for traffic scene understanding," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[62] H. Gao, B. Cheng, J. Wang, K. Li, J. Zhao, and D. Li, "Object classification using cnn-based fusion of vision and lidar in autonomous vehicle environment," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 9, pp. 4224–4231, 2018.

[63] M. Al-Qizwini, I. Barjasteh, H. Al-Qassab, and H. Radha, "Deep learning algorithm for autonomous driving using googlenet," in *2017 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2017, pp. 89–96.

[64] P. Drews, G. Williams, B. Goldfain, E. A. Theodorou, and J. M. Rehg, "Aggressive deep driving: Model predictive control with a cnn cost model," *arXiv preprint arXiv:1707.05303*, 2017.

[65] S. Ingle and M. Phute, "Tesla autopilot: semi autonomous driving, an uptick for future autonomy," *International Research Journal of Engineering and Technology*, vol. 3, no. 9, 2016.

[66] Y. Wang, W.-L. Chao, D. Garg, B. Hariharan, M. Campbell, and K. Q. Weinberger, "Pseudo-lidar from visual depth estimation: Bridging the gap in 3d object detection for autonomous driving," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 8445–8453.

[67] M. Mancini, G. Costante, P. Valigi, and T. A. Ciarfuglia, "Fast robust monocular depth estimation for obstacle detection with fully convolutional networks," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2016, pp. 4296–4303.

[68] D. Eigen, C. Puhrsch, and R. Fergus, "Depth map prediction from a single image using a multi-scale deep network," in *Advances in neural information processing systems*, 2014, pp. 2366–2374.

[69] Z. Kappassov, J.-A. Corrales, and V. Perdereau, "Tactile sensing in dexterous robot hands," *Robotics and Autonomous Systems*, vol. 74, pp. 195–220, 2015.

[70] E. Badreddin, "Obstacle avoidance using tactile sensing for an autonomous mobile robot," *IFAC Proceedings Volumes*, vol. 25, no. 29, pp. 325–329, 1992.

[71] R. Hecht-Nielsen, "Theory of the backpropagation neural network," in *Neural networks for perception*. Elsevier, 1992, pp. 65–93.

[72] H. J. Sussmann, "Uniqueness of the weights for minimal feedforward nets with a given input-output map," *Neural networks*, vol. 5, no. 4, pp. 589–593, 1992.

[73] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2223–2232.

[74] D. Justus, J. Brennan, S. Bonner, and A. S. McGough, "Predicting the computational cost of deep learning models," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 3873–3882.

[75] J. Han and C. Moraga, "The influence of the sigmoid function parameters on the speed of backpropagation learning," in *International Workshop on Artificial Neural Networks*. Springer, 1995, pp. 195–201.

[76] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[77] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[78] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011, pp. 315–323.

[79] S. Yao, Y. Zhao, A. Zhang, L. Su, and T. Abdelzaher, "Deepiot: Compressing deep neural network structures for sensing systems with a compressor-critic framework," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. ACM, 2017, p. 4.

[80] C. Davis, "The norm of the schur product operation," *Numerische Mathematik*, vol. 4, no. 1, pp. 343–344, 1962.

[81] X. Dong, S. Chen, and S. Pan, "Learning to prune deep neural networks via layer-wise optimal brain surgeon," in *Advances in Neural Information Processing Systems*, 2017, pp. 4857–4867.

[82] B. Hassibi and D. G. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," in *Advances in neural information processing systems*, 1993, pp. 164–171.

[83] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang, "Network trimming: A data-driven neuron pruning approach towards efficient deep architectures," *arXiv preprint arXiv:1607.03250*, 2016.

[84] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[85] A. Sharma, N. Wolfe, and B. Raj, "The incredible shrinking neural network: New perspectives on learning representations through the lens of pruning," *arXiv preprint arXiv:1701.04465*, 2017.

[86] G. B. Arfken and H. J. Weber, "Mathematical methods for physicists," 1999.

[87] G. Upton and I. Cook, *A dictionary of statistics 3e*. Oxford university press, 2014.

[88] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[89] J. Park, S. Li, W. Wen, P. T. P. Tang, H. Li, Y. Chen, and P. Dubey, "Faster cnns with direct sparse convolutions and guided pruning," *arXiv preprint arXiv:1608.01409*, 2016.

[90] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[91] D. Bachman, *Advanced calculus demystified*. McGrawHill, 2007.

[92] W. Kaplan, "Advanced calculus. redwood city," 1984.

[93] S. Cook, *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[94] Google, "Timeline visualization for tensorflow using chrome trace format," https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/client/timeline.py, 2018.

[95] Ecma International, "Standard ecma-404 the json data interchange syntax," https://www.ecma-international.org/publications/standards/Ecma-404.htm, 2017.

Authorized licensed use limited to: TONGJI UNIVERSITY. Downloaded on May 02,2021 at 01:58:06 UTC from IEEE Xplore. Restrictions apply.

[96] S. Teerapittayanon, B. McDanel, and H.-T. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," in *2016 23rd International Conference on Pattern Recognition (ICPR)*. IEEE, 2016, pp. 2464–2469.

[97] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du, "On-demand deep model compression for mobile devices: A usage-driven model selection framework," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2018, pp. 389–400.

[98] N. Silberman, D. Hoiem, P. Kohli, and R. Fergus, "Indoor segmentation and support inference from rgbd images," in *European Conference on Computer Vision*. Springer, 2012, pp. 746–760.

[99] 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.

[100] Y. Wang, C. Xu, C. Xu, and D. Tao, "Beyond filters: Compact feature map for portable deep model," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 3703–3711.

[101] S. Lee and S. Nirjon, "Neuro. zero: a zero-energy neural network accelerator for embedded sensing and inference systems," in *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, 2019, pp. 138–152.

[102] G. Gobieski, B. Lucia, and N. Beckmann, "Intelligence beyond the edge: Inference on intermittent embedded systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 199–213.

[103] C. Jiang, G. Li, C. Qian, and K. Tang, "Efficient dnn neuron pruning by minimizing layer-wise nonlinear reconstruction error." in *IJCAI*, vol. 2018, 2018, pp. 2–2.

[104] Á. S. Miralles and M. Á. S. Bobi, "Real time dynamic neural network (rtdnn)."

[105] T. Martinetz and K. Schulten, "Topology representing networks," *Neural Networks*, vol. 7, no. 3, pp. 507–522, 1994.

[106] G.-B. Huang, "Learning capability and storage capacity of two-hidden-layer feedforward networks," *IEEE Transactions on Neural Networks*, vol. 14, no. 2, pp. 274–281, 2003.

[107] G.-B. Huang, Q.-Y. Zhu, and C. K. Siew, "Real-time learning capability of neural networks," *IEEE Trans. Neural Networks*, vol. 17, no. 4, pp. 863–878, 2006.

[108] J. W. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung, "Imprecise computations," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 83–94, 1994.

[109] J. W.-S. Liu, K.-J. Lin, W. K. Shih, A. C.-s. Yu, J.-Y. Chung, and W. Zhao, "Algorithms for scheduling imprecise computations," in *Foundations of Real-Time Computing: Scheduling and Resource Management*. Springer, 1991, pp. 203–249.

[110] K.-J. Lin, S. Natarajan, and J. W.-S. Liu, "Imprecise results: Utilizing partial computations in real-time systems," 1987.

[111] K.-J. Lin and S. Natarajan, "Expressing and maintaining timing constraints in flex," in *Proceedings. Real-Time Systems Symposium*. IEEE, 1988, pp. 96–105.

[112] W.-K. Shih, J. W. Liu, and J.-Y. Chung, "Algorithms for scheduling imprecise computations with timing constraints," *SIAM Journal on Computing*, vol. 20, no. 3, pp. 537–552, 1991.

[113] J.-Y. Chung, J. W.-S. Liu, and K.-J. Lin, "Scheduling periodic jobs that allow imprecise results," *IEEE transactions on computers*, vol. 39, no. 9, pp. 1156–1174, 1990.

[114] K. B. Kenny and K.-J. Lin, "Structuring large real-time systems with performance polymorphism," in *[1990] Proceedings 11th Real-Time Systems Symposium*. IEEE, 1990, pp. 238–246.

[115] H. Li, S. De, Z. Xu, C. Studer, H. Samet, and T. Goldstein, "Training quantized nets: A deeper understanding," in *Advances in Neural Information Processing Systems*, 2017, pp. 5811–5821.

[116] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4820–4828.

[117] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[118] Y. Cheng, F. X. Yu, R. S. Feris, S. Kumar, A. Choudhary, and S.-F. Chang, "An exploration of parameter redundancy in deep networks with circulant projections," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 2857–2865.

[119] V. Sindhwani, T. Sainath, and S. Kumar, "Structured transforms for small-footprint deep learning," in *Advances in Neural Information Processing Systems*, 2015, pp. 3088–3096.

[120] Z. Li, X. Wang, X. Lv, and T. Yang, "Sep-nets: Small and effective pattern networks," *arXiv preprint arXiv:1706.03912*, 2017.

[121] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*. Springer, 2016, pp. 525–542.

[122] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[123] W. Chen, J. Wilson, S. Tyree, K. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," in *International Conference on Machine Learning*, 2015, pp. 2285–2294.

[124] J. Park, S. R. Li, W. Wen, H. Li, Y. Chen, and P. Dubey, "Holistic sparsecnn: Forging the trident of accuracy, speed, and size," *arXiv preprint arXiv:1608.01409*, vol. 1, no. 2, 2016.

[125] S. Yao, Y. Zhao, H. Shao, S. Liu, D. Liu, L. Su, and T. Abdelzaher, "Fastdeepiot: Towards understanding and optimizing neural network execution time on mobile and embedded devices," in *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2018, pp. 278–291.

[126] D. A. Patterson and J. L. Hennessy, *Computer organization and design MIPS edition: the hardware/software interface*. Newnes, 2013.

[127] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," 2011.

[128] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4013–4021.

[129] P. Panda, A. Sengupta, and K. Roy, "Conditional deep learning for energy-efficient and enhanced pattern recognition," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 475–480.

[130] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Pensky, "Sparse convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 806–814.

[131] V. Lebedev and V. Lempitsky, "Fast convnets using group-wise brain damage," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2554–2564.

[132] X. Chen, "Escoin: Efficient Sparse Convolutional Neural Network Inference on GPUs," *arXiv e-prints*, p. arXiv:1802.10280, Feb 2018.

[133] 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*. ACM, 2016, pp. 176–189.

[134] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[135] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, and R. Fergus, "Regularization of neural networks using dropconnect," in *International conference on machine learning*, 2013, pp. 1058–1066.

[136] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger, "Deep networks with stochastic depth," in *European conference on computer vision*. Springer, 2016, pp. 646–661.

[137] R. K. Srivastava, K. Greff, and J. Schmidhuber, "Highway networks," *arXiv preprint arXiv:1505.00387*, 2015.

[138] X. Sun, X. Ren, S. Ma, and H. Wang, "meprop: Sparsified back propagation for accelerated deep learning with reduced overfitting," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 3299–3308.

[139] Z. Zhang, P. Yang, X. Ren, and X. Sun, "Memorized sparse backpropagation," *arXiv preprint arXiv:1905.10194*, 2019.

Authorized licensed use limited to: TONGJI UNIVERSITY. Downloaded on May 02,2021 at 01:58:06 UTC from IEEE Xplore. Restrictions apply.