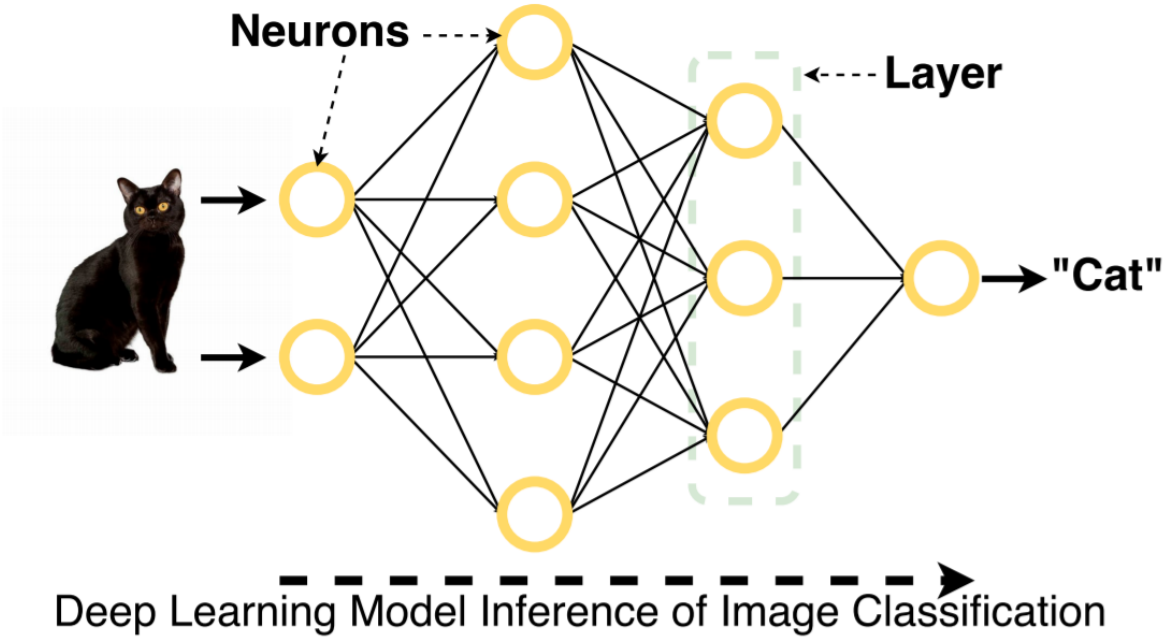
Edge Intelligence: On-Demand Deep Learning Model Co-Inference with Device-Edge Synergy

Hailiang Zhao
http://hliangzhao.me

This paper was published in **ACM SIGCOMM 2018 Workshop on Mobile Edge Communications (MECOMM'18)**.

A primer on *DNN*

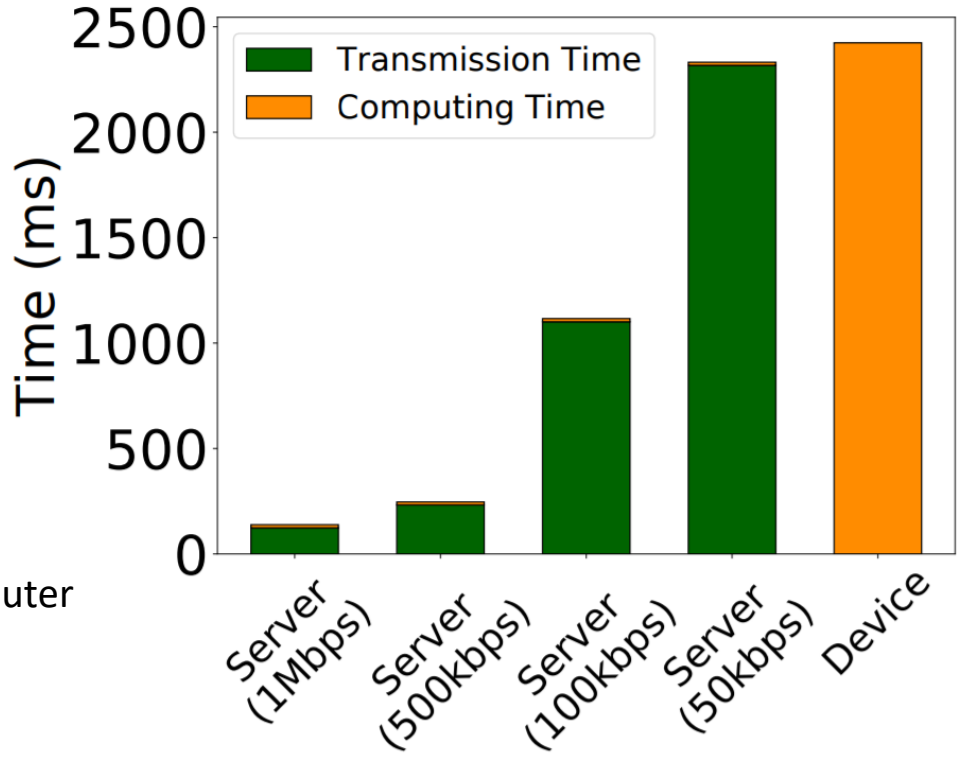


A typical *DNN* contains tens of layers and hundreds of nodes per layer, i.e., the number of parameters can easily reach the scale of millions. As a result, *DNN inference* is **computational intensive**.

———— cannot be well supported by today's mobile device.

Status quo of mobile *DNN inference*: two models

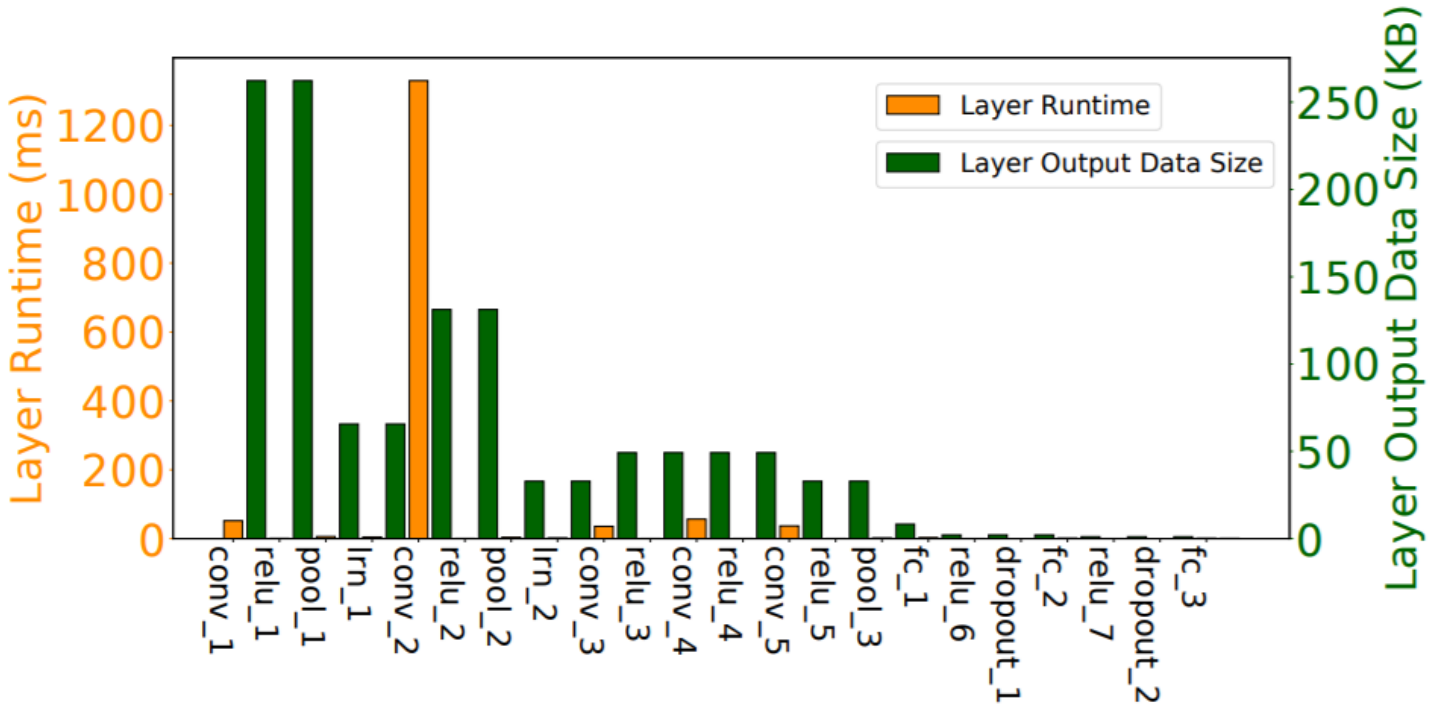
- direct execution on the mobile devices
- offloading to the cloud/edge server for execution



The Classical AlexNet DNN on Cifar-10 dataset.

* Raspberry Pi tiny computer → mobile device
* Desktop PC → MEC server

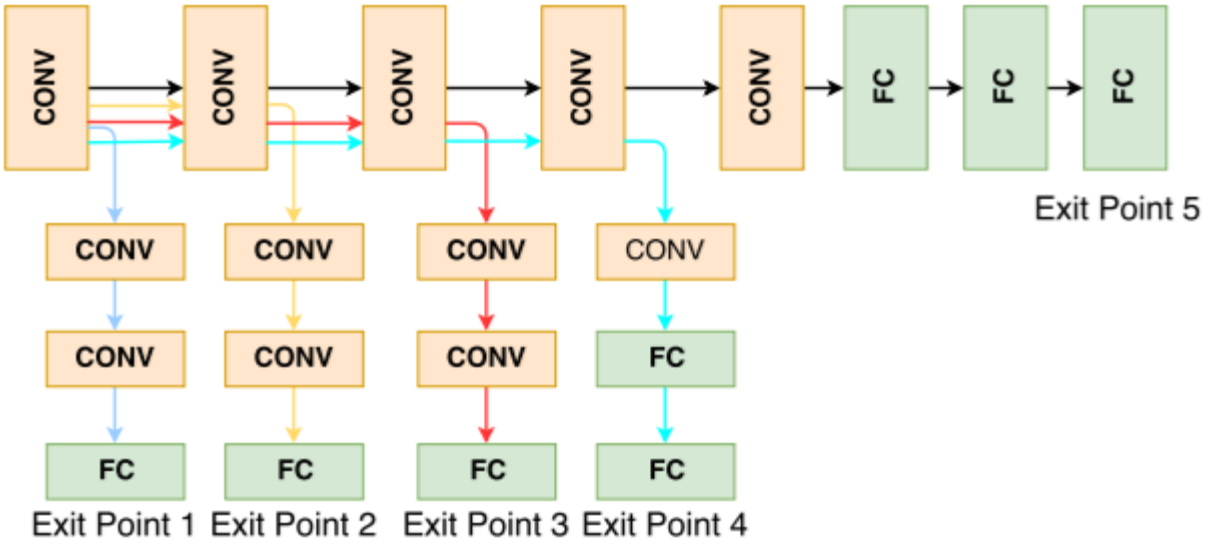
DNN Partitioning



Layers with long runtime do not necessarily have a large output data size.

—— partitioning the DNN into two parts and offloading the computational intensive one to the server with low latency while running the rest layers on the device

DNN Right-sizing

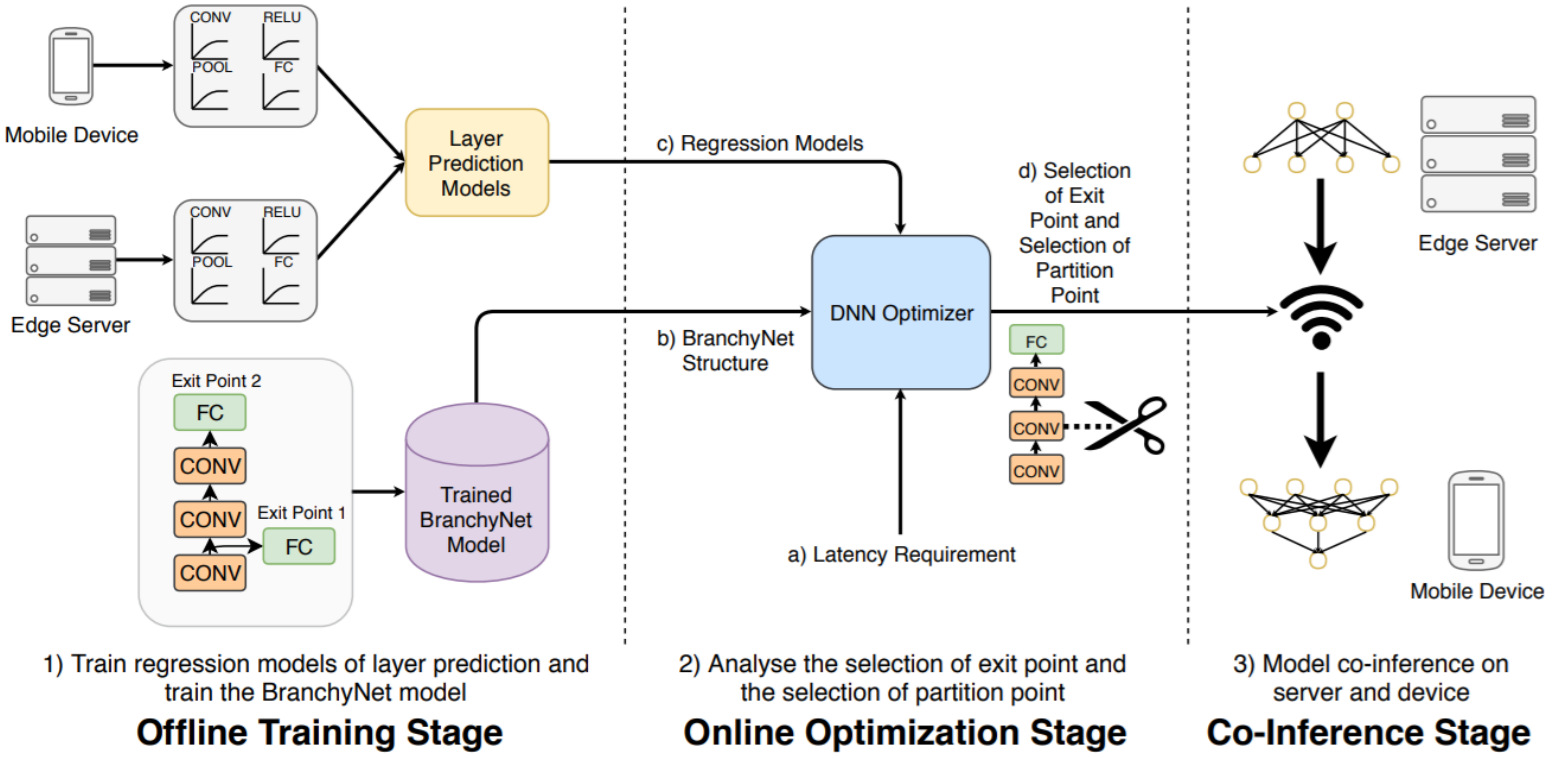


The early-exit mechanism can be obtained by the open source framework BranchNet.

—— early-exit reduces the computing time and device side but deteriorates the accuracy of DNN inference

Problem Formulation

Given a predefined latency requirement, how to jointly optimize the decisions of partitioning and right-sizing, in order to maximize DNN inference accuracy?



Framework Overview

- Offline training stage*
  (1) generate regression-based performance prediction models
  (2) use Banchynet to train DNN models with different exit points

Layer Type	Independent Variable
Convolution	amount of input feature maps, (filter size/stride)^2*(num of filters)
Relu	input data size
Pooling	input data size, output data size
Local Response Normalization	input data size
Dropout	input data size
Fully-Connected	input data size, output data size
Model Loading	model size

- Online optimization stage
- Co-inference stage

*Noting that we only focus on DNN inference, DNN training can be conducted in an off-line manner using powerful cloud resources.

Framework Stage #1

- Offline training stage*
  (1) generate regression-based performance prediction models
  (2) use Banchynet to train DNN models with different exit points

Layer	Mobile Device model	Edge Server model
Convolution	y = 6.03e-5 ∗ x1 + 1.24e-4 ∗ x2 + 1.89e-1	y = 6.13e-3 ∗ x1 + 2.67e-2 ∗ x2 - 9.909
Relu	y = 5.6e-6 ∗ x + 5.69e-2	y = 1.5e-5 ∗ x + 4.88e-1
Pooling	y = 1.63e-5 ∗ x1 + 4.07e-6 ∗ x2 + 2.11e-1	y = 1.33e-4 ∗ x1 + 3.31e-5 ∗ x2 + 1.657
Local Response Normalization	y = 6.59e-5 ∗ x + 7.80e-2	y = 5.19e-4 ∗ x+ 5.89e-1
Dropout	y = 5.23e-6 ∗ x+ 4.64e-3	y = 2.34e-6 ∗ x+ 0.0525
Fully-Connected	y = 1.07e-4 ∗ x1 - 1.83e-4 ∗ x2 + 0.164	y = 9.18e-4 ∗ x1 + 3.99e-3 ∗ x2 + 1.169
Model Loading	y = 1.33e-6 ∗ x + 2.182	y = 4.49e-6 ∗ x + 842.136

- Online optimization stage
- Co-inference stage

*Noting that we only focus on DNN inference, DNN training can be conducted in an off-line manner using powerful cloud resources.

Framework Stage #2

- Offline training stage*
- Online optimization stage

Based on:

(1) the regression prediction models & Branchynet-trained DNN with various sizes

(2) available bandwidth

(3) pre-defined latency requirement,

this stage select the best partition point and early-exit point to maximize the accuracy while providing performance guarantee on latency.

- Co-inference stage

The edge server executes the layer before the partition point and the rest will run on the mobile device.

*Noting that we only focus on DNN inference, DNN training can be conducted in an off-line manner using powerful cloud resources.

Exit Point and Partition Search Algorithm

Algorithm 1 Exit Point and Partition Point Search

Input:

M: number of exit points in a branchy model

$\{N_i | i = 1, \cdots, M\}$: number of layers in each exit point

$\{L_j | j = 1, \cdots, N_i\}$: layers in the i-th exit point

$\{D_j | j = 1, \cdots, N_i\}$: each layer output data size of i-th exit point

$f(L_j)$: regression models of layer runtime prediction

B: current wireless network uplink bandwidth

$Input$: the input data of the model

$latency$: the target latency of user requirement

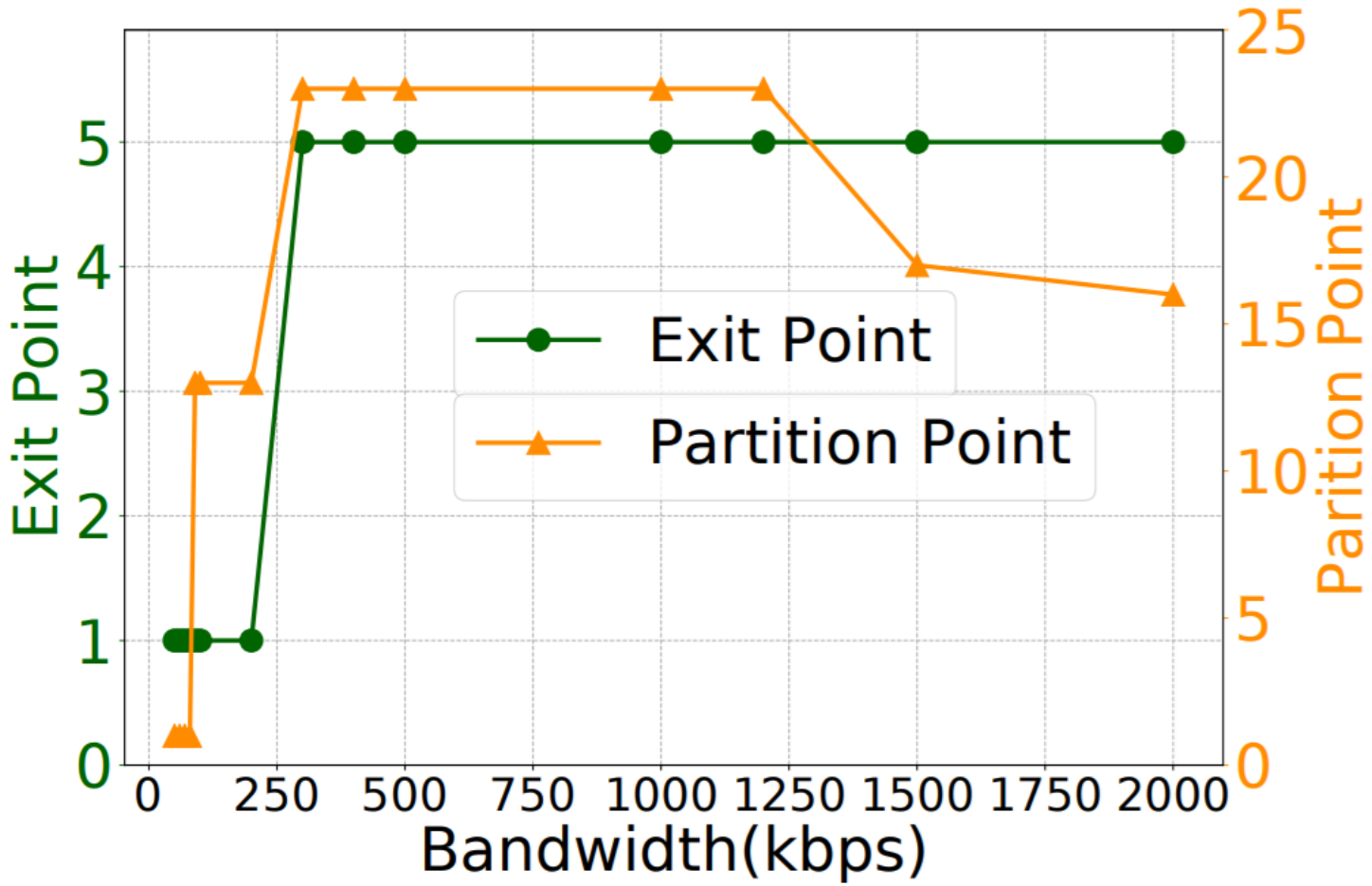
Output:

Selection of exit point and its partition point

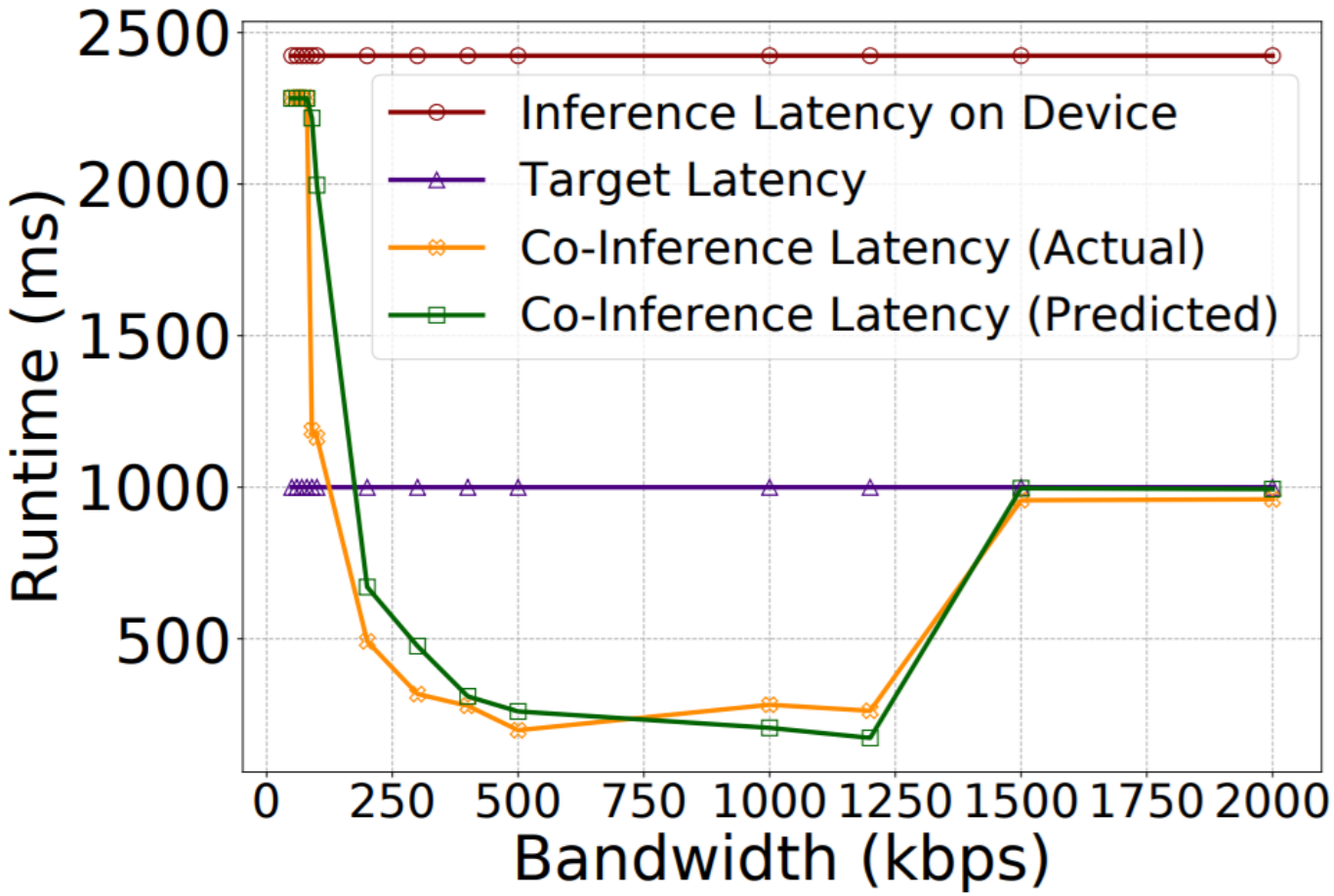
Exit Point and Partition Search Algorithm

Procedure
for $i = M, \cdots, 1$ **do**
    Select the i-th exit point
    for $j = 1, \cdots, N_i$ **do**
        $ES_j \leftarrow f_{edge}(L_j)$
        $ED_j \leftarrow f_{device}(L_j)$
    end for
    $A_{i,p} = \underset{p=1,\cdots,N_i}{\operatorname{argmin}} (\sum_{j=1}^{p-1} ES_j + \sum_{k=p}^{N_i} ED_j + Input/B + D_{p-1}/B)$
    if $A_{i,p} \leq latency$ **then**
        return Selection of Exit Point i and its Partition Point p
    end if
end for
return NULL

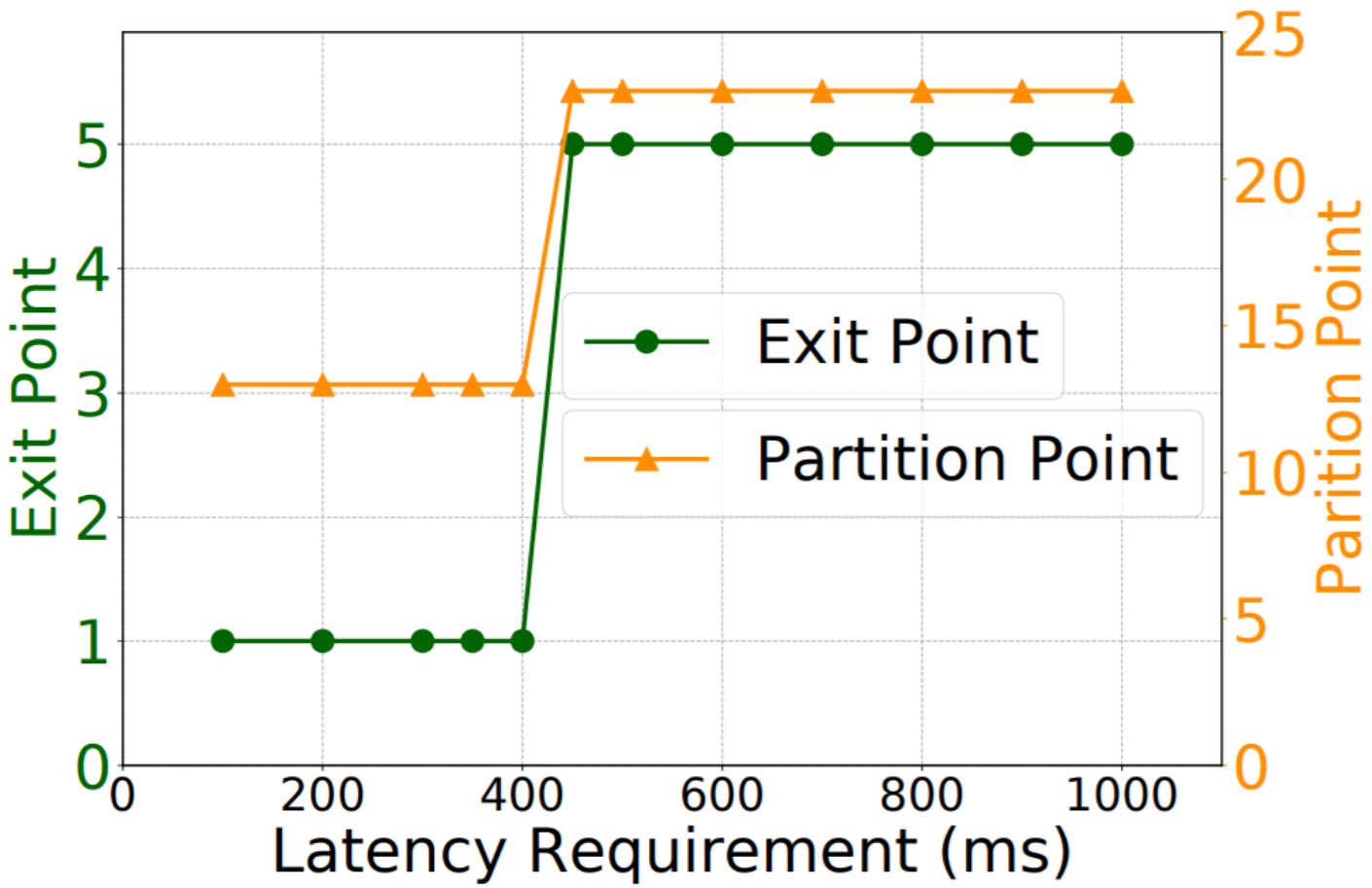
Selection under different bandwidths



Model runtime under different bandwidths



Selection under different latency requirements



- Actually nothing new has been proposed in this paper.
- It's not easy to simulate the framework because much of Machine Learning experience is necessary.
- The Raspberry Pi tiny computer can be taken as mobile device and the desktop can be taken as the MEC server, while the bandwidth between them is controlled by the WonderShaper tool.
- How can we dig further on this framework?
  * Is the layer prediction models necessary? Can we find a better agent to do this?
  * The algorithm proposed assumes that model with larger size is more accuracy.
  * The algorithm just linear search.
- …