A Simple Optimizer for Intra-Edge Data Movement

Lynn Zheng **Ted Shaowang**

1 Introduction

Over the last several years, we have witnessed the rise of *edge computing*, where computation is placed close to the point of data collection to reduce latency and make real-time decision systems possible. With the recent advances in highly-specialized hardware, such as NVIDIA Jetson [5], Edge TPU [3] and Intel VPU [4], we now have access to low-powered edge devices that actually deliver great computational performance. Now that we have enormous amount of data generated by sensors and smart home / IoT devices everyday, we can save a lot of network bandwidth if most of the computation can be done by the low-powered edge devices near data sources, without the need for transferring to cloud.

Existing edge computation frameworks fail to address the issue of intra-edge data movement, or *data serving*, which poses a new challenge of moving the data to the right place at the right time. In this paper, we build a simple optimizer that takes care of intra-edge data movement with the objective of minimizing total amount of data transferred within the edge network (Figure 1). Common constraints including storage size limitation, co-location and replication are considered.

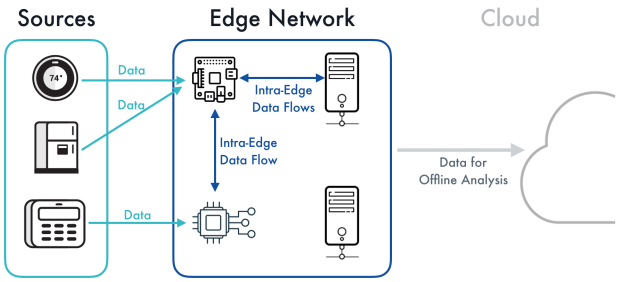

Figure 1: Edge computation systems today have limited support for data movement between edge nodes. We describe a vision where such data flows are declaratively specified to build robust and adaptive edge decision making systems.

2 Problem Formalization

We view the intra-edge data movement problem as a convex optimization problem and set up the problem parameters as follows:

- Suppose we have M devices and N types of data instances
- T: a $M \times N$ binary matrix, called the “transmit" matrix. It represents the end-state of our system. $T_{m,n} = 1$ signifies that the data type n exists on the device m.
- S: a $M \times N$ binary matrix, called the “source" matrix. It represents the start-state of our system. Following the same convention as T, $S_{m,n} = 1$ signifies that the data type n exists on the device m.
- C: a $N \times 1$ vector, called the “cost" vector. It represents the size of each type of data.

- X: a $N \times N$ binary matrix, called the “co-location constraint" matrix. It specifies whether two types of data need to be stored together on the same device.
- V: a $M \times 1$ vector, called the “volume" vector. It represents the volume, or, storage limit of each device.
- R: a $N \times 1$ vector (per data item), called the “replication" vector. It specifies how many times each data instance need to be replicated in the end-state system. To put another way, it specifies the number of devices each data instance need to be located on in the end-state system T.

2.1 Optimization Problem

We then define our optimization problem as follows:

2.1.1 Objective

Find a T s.t.

$$\min_T \sum (T - S) \times C$$

The rationale follows: $(T-S)\times C$, a $M \times 1$ vector gives the cost of data transferred to (or equivalently, replicated on) each device. Summing the vector elements results in the total cost of operations that transform our system from the start-state S into the end-state T.

2.1.2 Constraints

1. The end-state T is a binary matrix: Each $T_{m,n} \in \{0, 1\}$
2. We mustn't delete data from the system. In other words, all of the entries in S that are one must also be one in T. Element-wise, $T_{m,n} \geq S_{m,n} \quad \forall m, n$
3. The end-state T must satisfy the co-location constraint X: Element-wise, T' `&` $T \geq X$, where T' is the transpose of T and `&` is the bit-wise `AND` operator. In implementation, any pair of data types n_1, n_2 that needs to co-locate must co-locate on all of the devices: $T_{m,n_1} = T_{m,n_2} \quad \forall m$. Note that this also enforces transitivity of the pairwise co-location requirements.
4. The end-state T must satisfy the replication requirements R: Column-wise sum $T_{\bullet,n} = \sum_m T_{m,n}$ is greater or equal to R_n, the corresponding entry in R for the data type n.
5. The end-state T must satisfy the storage, or, volume constraint V: $T \times C \leq V$

We implemented and solved this optimization problem using the `cvxpy` [2] package.

3 Simulation of Real Scenarios

We consider several cases to simulate what might happen in real world scenarios. In a typical smart home environment, we deploy 3 Nest smart cameras to monitor suspicious activities in the kitchen, backyard and front yard respectively. Each camera is connected to a NVIDIA Jetson device [5] that specifically processes the streaming video and audio recorded by the camera. In addition, we also have a programmable router that takes all network traffic generated by laptops, phones, tablets and IoT devices such as Amazon Echo Dot [1] and Samsung Family Hub refrigerator. We further assume that each video stream has a size of 100 units while each audio stream has a size of 10 units. The size of network traffic stream captured by the programmable router is assumed to be 15 units.

3.1 Case 1

In the simplest case, we have a total of $M = 4$ devices consisting of 3 NVIDIA Jetson and 1 programmable router. Three cameras and one programmable router are generating $N = 7$ types of data instances (3 videos, 3 audios and 1 network traffic data). We require videos and audios recorded by the same camera to be co-located together at all times. In this case, we assume that each NVIDIA Jetson has 350 units of storage size and the programmable router has 15 units of storage size. All

kinds of data mentioned above are to be replicated at least 3 times. Formally, we have the following parameters for this case:

$$M = 4$$
$$N = 7$$
$$S = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$
$$C = [100 \quad 10 \quad 100 \quad 10 \quad 100 \quad 10 \quad 15]^T$$
$$X = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$
$$V = [350 \quad 350 \quad 350 \quad 15]$$
$$R = [3 \quad 3 \quad 3 \quad 3 \quad 3 \quad 3 \quad 3]$$

The solution T for this case given by the `cvxpy` package is

$$T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

and the minimized cost (objective) in this case is 690.

We can intuitively understand the solution as the fact that all video and audio streams are replicated to all three NVIDIA Jetson devices and the network traffic data is also replicated to two of them.

3.2 Case 2

In this case, we reduce the storage size of each NVIDIA Jetson device to 110 so it can only store one set of video and audio streams generated by a camera. In addition, we add a Raspberry Pi with 1,000 units of storage size and connect all three cameras to the Pi instead of NVIDIA Jetsons. We also relax the replication constraint from 3x to 2x. All other parameters remain unchanged. Ideally, we would like the video and audio streams to be replicated to NVIDIA Jetsons for faster processing. Formally, we have the following parameters for this case:

$$M = 5$$
$$N = 7$$
$$S = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$
$$C = [100 \quad 10 \quad 100 \quad 10 \quad 100 \quad 10 \quad 15]^T$$
$$X = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$
$$V = [1000 \quad 110 \quad 110 \quad 110 \quad 15]$$

$$R = [2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2]$$

The solution T for this case given by the `cvxpy` package is

$$T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

and the minimized cost (objective) in this case is 345.

As expected, the video and audio streams are transferred to three NVIDIA Jetsons separately and the network traffic data is replicated to the Pi for redundancy.

3.3 Case 3

In this case, we further reduce the storage size of both the Pi and each Jetson, but we get one more Jetson device. In other words, we now have 4 NVIDIA Jetsons with 100 units of storage each, and a Raspberry Pi with 330 units of storage. Due to reduced storage size, we drop the co-location constraint for video and audio recorded by the same camera. All other settings remain the same as **Case 2**. Formally, the parameters can be written as follows:

$$M = 6$$

$$N = 7$$

$$S = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$C = [100 \quad 10 \quad 100 \quad 10 \quad 100 \quad 10 \quad 15]^T$$

$$X = I_7$$

$$V = [330 \quad 100 \quad 100 \quad 100 \quad 100 \quad 15]$$

$$R = [2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2]$$

The solution T for this case given by the `cvxpy` package is

$$T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

and the minimized cost (objective) in this case is still 345.

In this case, three NVIDIA Jetsons store three video streams individually and the fourth Jetson stores all three audio streams as well as the redundant network traffic copy due to size constraints.

4 Simulation Studies of Randomized Hypothetical Situations

We perturb the parameters from **Case 2**, which has an optimal cost of 345. We visualize the effects of the perturbation on the optimal cost of our system.

4.1 Perturb Device Storage Constraints

Instead of a volume vector of $[1000, 110, 110, 110, 15]^T$, we randomize each entry to be uniformly between 50 and 500. Interestingly, this perturbation does not seem to vary the optimal cost of the system as much as it does to produce infeasible optimization problems or solutions that has infinite cost. Out of 1000 simulations, only 364 were solvable with a finite cost of 345, exactly the same as optimal cost in the original Case 2 settings. Three simulations produced infeasible problems and 633 resulted in infinite costs.

4.2 Perturb Data Type Replication Requirements

Instead of replicating each data type twice, we randomize each replication requirement to be uniformly between 1 to 3. Out of 1000 simulations, 177 were solvable and 823 resulted in infinite costs. The high number of infinite-cost solutions might caused by the restrictiveness of other constraints like the storage constraints or specifications like data type costs. The mode of the histogram below is still between 300 and 350, which is also the bin that could have contained 345, the optimal cost of the original system. In fact, 49 out of the 177 total solvable simulations have a cost of exactly 345. Within that same bin, 48 has a cost of 330. In other words, the majority of the feasible perturbed solutions have a perturbed replication requirement similar to the original one in Case 2. Because of this, these few perturbed situations remains solvable under the restrictiveness of Case 2 and achieved costs similar to, if not the same as, the optimal cost of Case 2.

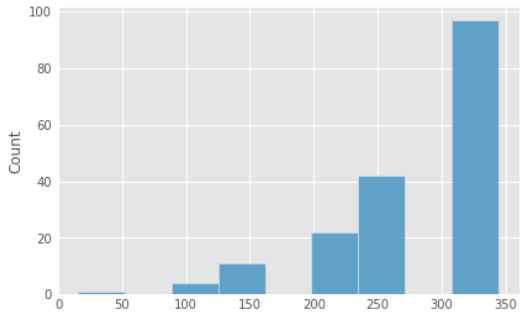

Figure 2: Distribution of optimal costs when we perturb the replication requirements

4.3 Perturb Data Type Costs

Instead of a data type cost vector of $[100, 10, 100, 10, 100, 10, 15]^T$, we randomize each entry to be between 10 and 60. We deliberately chose a smaller upper bound 60 on the data size than the max value 100 in our original Case 2 setting. This way, more simulations will produce solvable finite results for us to visualize the distribution. Out of the 1000 simulations, 101 were solvable, with costs distributed roughly bell-shaped as shown in the histogram below. Six were infeasible and 893 produced infinite costs.

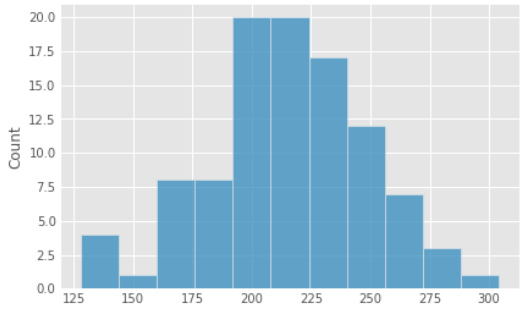

Figure 3: Distribution of optimal costs when we perturb the data type costs

5 Future Work and Extensions

5.1 Co-location Constraint

The co-location constraint we apply in this problem is the `ALL` version of co-location constraint: Any data type pair n_1, n_2 that needs to co-locate must co-locate on **all** of the devices. Alternatively, we may propose a more lenient `ANY` version of the co-location constraint: Any data type pair n_1, n_2

that needs to co-locate should co-locate on **any** (one or more) of the devices. This more lenient `ANY` version should solve to optimal costs smaller than the `ALL` version.

We may formalize this new constraint as follows. We define $\mathbf{N}$ as a set of data types that need to co-locate. Denote its cardinality as $|\mathbf{N}|$.

$$\begin{aligned} & T_{m,n_1} = T_{m,n_2} \quad \text{for some } m \\ \implies & \max\Big(\sum_{n\in\mathbf{N}}\sum_{m} T_{m,n}\Big) \geq |\mathbf{N}| \end{aligned}$$

Unfortunately, under this relaxation, our problem no longer follow DCP (Disciplined Convex Programming) or DQCP (Disciplined Quasiconvex Programming) rules. Therefore, for future explorations, we may need optimization packages more sophisticated than `cvxpy` to solve this relaxed `ANY` version of the co-location constraint.

5.2 Device Preference

In the problem we have formalized in Section 2, we did not consider the device preference for different types of data and workloads. In a real-world scenario, users might want a specific type of data to be processed (replicated) by a specific type of device. For example, videos are easily processed by GPUs or TPUs, but not programmable routers or a Raspberry Pi. We do not want video streams to be transferred to a Raspberry Pi, even if it has abundant empty space. As an extension to our existing formalization, a $M \times N$ preference weight matrix P can be added to denote how preferably a type of data instance i can be transferred to device j (0 means impossible to process, 1 means best placement).

6 Conclusion

In this paper, we formalize intra-edge data movement as a convex optimization problem with the objective of minimizing the total amount of data transferred within the edge network and constraints of storage size limitation, co-location and replication. We set up several hypothetical use cases and illustrate how the problem is modeled and solved in each case. In addition, we also discuss and visualize how our optimizer would respond to perturbations on parameters. Finally, we explore potential future extensions that can be done beyond our work, including another type of co-location constraint and additional parameters.

References

[1] Amazon Alexa. `https://developer.amazon.com/en-US/alexa`, 2014.

[2] Steven Diamond and Stephen Boyd. Cvxpy: A python-embedded modeling language for convex optimization. *The Journal of Machine Learning Research*, 17(1):2909–2913, 2016.

[3] Google. Edge tpu. `https://cloud.google.com/edge-tpu`, 2021.

[4] Intel. Intel movidius vision processing units. `https://www.intel.com/content/www/us/en/products/processors/movidius-vpu.html`, 2021.

[5] NVIDIA. Nvidia jetson. `https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/`, 2021.