CORA Cork Open Research Archive
Cartlann Taighde Oscailte Chorcaí

Title	NimbleCache - low cost, dynamic cache allocation in constrained edge environments
Authors	Chilukuri, Shanti;Pesch, Dirk
Publication date	2021-03
Original Citation	Chilukuri, S. and Pesch, D. (2021) 'NimbleCache - Low Cost, Dynamic Cache Allocation in Constrained Edge Environments', 2021 IEEE Wireless Communications and Networking Conference (WCNC), Nanjing, China, 29 March-1 April, (7 pp). doi: 10.1109/WCNC49053.2021.9417473
Type of publication	Conference item
Link to publisher's version	https://ieeexplore.ieee.org/document/9417473 - 10.1109/WCNC49053.2021.9417473
Rights	© 2021 IEEE. Personal use is permitted, but republication/ redistribution requires IEEE permission.
Download date	2026-11-17 11:28:30
Item downloaded from	https://hdl.handle.net/10468/11038



UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

NimbleCache - Low Cost, Dynamic Cache Allocation in Constrained Edge Environments

Shanti Chilukuri and Dirk Pesch
School of Computer Science and IT, University College Cork, T12 K8AF Cork, Ireland
Email: s.chilukuri@cs.ucc.ie, d.pesch@cs.ucc.ie

***Abstract*—Edge computing and caching of data in the Internet of Things (IoT) has several benefits such as reduced energy consumption by IoT end devices and increased availability of data and Quality of Service (QoS). In typical IoT scenarios, edge nodes (gateways) support several end devices, each of which may produce data in different patterns. In addition, data generated by different types of end devices varies in the application QoS requirements while also widely varying in the data access patterns by IoT services. Managing the data storage resources at edge nodes in such scenarios is a difficult task, especially since the edge nodes themselves may have limited computation capability and storage space. In this paper, we propose a dynamic, differentiated edge cache allocation strategy called NimbleCache that has low computational requirements and performs efficient cache allocation at edge nodes. Based on a Mixture Density Network (MDN), NimbleCache allocates varying portions of the edge cache to traffic of different IoT applications to achieve cache hit ratios very close to the target hit ratio. Simulation results show that NimbleCache achieves good average cache hit ratio with low cache space requirement and small computational overhead.**

I. Introduction

Edge computing and caching of data generated by IoT (Internet of Things) end devices has shown promise in improving quality of service and availability of data and reducing traffic in the core network [1]. In IoT edge networks, end devices (producers) generate sensor data that is sent to a nearby edge nodes (e.g., gateways), which store and process it before it is requested by consumers. Here, consumers are IoT services that process the data, control processes, or visualise data for human consumption. In the IoT, edge nodes are often miniature computers (e.g., the RaspberryPi) connected to multiple end devices and have limited computing power and storage capacity compared to data centre servers.

Commonly, producers generate bursty data due to sleep-wake cycles [2]. The data may be requested by IoT services running elsewhere in the network. QoS requirements for such services may vary widely, from being very tight in terms of availability, delay and accuracy (e.g., gas sensors) to quite lax (e.g., weather monitoring), making resource provisioning in IoT a challenging task. Differentiated cache space allocation at edge nodes can help meeting requirements for data availability and speedy access. Cache management involves assigning different portions of the edge cache to different services. It facilitates meeting QoS requirements (most metrics such as delay, availability of data etc. are directly effected by the hit ratio) for services even with dynamic, diverse data generation and request patterns [3]. The goal of differentiated cache space allocation is to allocate cache space to achieve a cache hit ratio (CHR) as close to the desired hit ratio as possible. In constrained edge environments, the additional goals are minimizing both the space allocated and the computational cost for allocation.

Machine Learning(ML), especially Neural Networks (NNs) and Reinforcement Learning (RL), have been successfully applied to problems where it is difficult to model data generation and demand and allocate resources accordingly [4]–[6]. However, optimal resource allocation using NN or RL can be expensive in terms of memory and computing power required, especially when done dynamically and frequently to suit ever-changing network conditions and service requirements. RL may also take a long time to converge. As such, these techniques may not be suitable for constrained edge nodes.

In this paper, we propose a resource allocation mechanism called NimbleCache which learns the cache space allocation to yield the desired performance for given data generation and service request patterns using Mixture Density Networks (MDNs) and allocates different fractions of edge cache space to different services accordingly. The aim of NimbleCache is to achieve a probabilistic CHR which is close to the desired CHR for all IoT services served by an edge node, with minimal computational overhead and better cache space utilization at the edge. MDNs [7] are a type of NNs that solve the problem of non-Gaussian distributed data by predicting the probability density of the output as a combination of kernel functions. Our evaluation results show that NimbleCache results in close to the desired hit ratios for different data generation and request patterns. They also show that NimbleCache yields in close to optimal cache space utilization with reduced computation at the edge, compared to allocation based on conventional NNs. The main contributions of this paper are:

- identification of a set of features that effect the CHR in IoT edge networks, to serve as a basis for cache allocation using ML in light of bursty data generation and lack of accurate models of data request patterns.
- NimbleCache, a dynamic, differentiated cache allocation scheme for edge nodes that yields the desired CHR for different applications with minimal space allocation and computational cost.

This work has received funding in part, from the European Unions Horizon 2020 Research and Innovation Programme under the EDGE COFUND Marie Sklodowska Curie grant agreement No. 713567 and from the Science Foundation Ireland under CONNECT Centre grant no. 13/RC/2077.

- comparison of a conventional, optimal NN-based cache resource allocation approach with NimbleCache in terms of cache hit ratio and computational expense.

II. Resource Allocation in Constrained Networks

In this section, we establish the rationale for finding the probability density function (PDF) in resource allocation for constrained network environments followed by a brief introduction to MDNs (Table I gives the notation). While machine learning for constrained networks can be done by off-loading the *training* phase to the cloud [8], the burden of *inference from the resulting model* lies with the (edge) node performing dynamic resource allocation. It is this cost that we focus on.

TABLE I
Notation

p_i^d	desired performance at the edge node for the i^{th} application
p_i	actual performance at the edge node for the i^{th} application
γ	minimum allocation unit of the cache
β	maximum number of blocks possible per application
a_i	fraction of cache allocated to the i^{th} application
$\mathcal{C}$	total resource (cache space) available at the edge node
Δ	overall difference between actual and desired performance
θ	computation cost for allocation
n	number of applications

In general, the goal of differentiated resource allocation is to allocate resources such that the overall difference between the desired performance (p_i^d for the i^{th} application) and actual performance (p_i) is minimized. If a_i is the amount of resource allocated to the i^{th} application, the goal is -

$$\min\{\Delta\},\ s.t.\ \sum_{i=1}^{n} a_i \leq \mathcal{C},\ \text{where}\ \Delta = \sum_{i=1}^{n} (p_i^d - p_i) \quad (1)$$

where n is the number of applications and $\mathcal{C}$ is the total amount of resource available. Several optimization techniques including those using deep neural networks (DNN) have been proposed to meet the goal in Eq. 1 [9], [10]. In constrained environments such as edge nodes, however, the goal is to maximize the performance *while* minimizing both resources allocated *and* computation cost θ necessary for finding the optimal allocation. Hence, the goal is -

$$\min\{\Delta, \sum_{i=1}^{n} a_i, \theta\}, \quad s.t. \quad \sum_{i=1}^{n} a_i \leq \mathcal{C} \quad (2)$$

Minimizing the overall resources allocated while minimizing Δ helps mitigate over-allocation beyond performance saturation. This is particularly useful in edge nodes with storage space limitations. Considering an edge cache of size $\mathcal{C}$ (which is the resource in this case) and *granularity* (minimum allocation unit) of γ, the total number of allocation blocks is $\mathcal{C}/\gamma$. The maximum number of possible allocation blocks for each application (or producer, if data generated by each producer is used by a separate application) assuming that every application is allocated at least one block is β, where -

$$\beta = \frac{\mathcal{C}}{\gamma} - n + 1, \quad \text{and} \quad \frac{\mathcal{C}}{\gamma} \geq n \quad (3)$$

A. The Case for Finding the Probability Density Function

When supervised learning with NNs is used, the network learns a function to map a set of input features $\mathbf{x}=\{x_1, x_2, .., x_d\}$ to one or more output variables $\mathbf{y}=\{y_1, y_2, .., y_c\}$. Once the function is learned, it can be used to predict the output for any input from the same distribution. A simple approach for network resource allocation with supervised learning to minimize both Δ and the overall resources allocated involves two steps -

1) Treat the network state $\mathbf{S}$ and a possible allocation vector $\mathbf{a}$ as input and predict the performance Δ as the output. That is, the goal is to find $\Delta|\mathbf{S}, \mathbf{a}$. For β possible allocation blocks and n applications, the number of predictions to be performed is $\binom{\beta}{n}$.
2) Carry out an exhaustive search of the predicted values and choose the minimal allocation that satisfies some goal (e.g., maximize fairness or efficiency of allocation).

While this is a feasible approach in computing environments with a large amount of resources, the number of predictions to be carried out and an exhaustive search of all possible allocations may be too much to handle in constrained environments, as $\binom{\beta}{n}$ grows exponentially with β. Specifically, for a given number of applications, while a smaller granularity γ results in higher β and hence higher complexity, a larger γ may result in over-allocation and waste of space.

An alternative method is to combine the three goals in Eq. 2 into a single loss function with different weights for each goal. This is the well-studied weighted-sum scalarization approach for multi-objective optimization [11]. However, since application QoS requirements may vary widely in IoT, the preference given for each of the goals in the optimization may vary accordingly. This requires training and using different models for each possible weight combination [12], which may also not be feasible for constrained edge nodes.

Reinforcement learning [13] is another approach that has been used widely to solve resource allocation problems. In RL, the agent (network resource allocator) senses the network's current state and takes an action (allocates resources) to observe a reward (the network performance). By trying out all possible actions, the agent arrives at an allocation policy that yields maximum reward. The number of possible actions (possible resource allocations), may be too large to handle for constrained devices.

In control applications, NN-based prediction can be used for the *inverse problem* where the goal is to choose an input that gives a desired output. This can be applied to resource allocation to find the allocation given a desired performance. Thus, we invert the problem to train a NN to find $\mathbf{a}|\mathbf{S}, \Delta$. This enables the allocation decision with a single prediction, greatly reducing the resources consumed by the allocator.

However, inverting the NN input and output is not sufficient in this case. In the cache allocation problem, there may be several possible allocations that result in the same hit ratio for the same network conditions. Here, over-allocation yields the same performance as an optimal allocation. For multi-

valued functions, the predicted value by a conventional NN is generally an average of the target values (allocations that result in a desired performance) [7]. This gives no insight into the optimal allocation for a desired performance, usually resulting in excessive allocation.

Instead, we propose that *the PDF of the allocation for a given network state and desired performance should be found*. From the PDF, an allocation that most probably results in the desired performance can be selected with constant (just n) predictions for the inverse problem. This is ideal for constrained resource allocation as it does not require performance prediction for $\binom{\beta}{n}$ allocations or multiple models to be checked for selecting an optimal strategy. Also, it does not result in over-allocation like the inverse case where just the allocation for a state is predicted. We use Mixture Density Networks which combine a NN model with mixture density models to predict the conditional PDF of the output variable [7].

B. MDN Concepts

An MDN has two components - the neural network and the mixture model [7]. The NN can be standard with input **x**. The outputs of the NN are the parameters of the Gaussian distributions denoted by **z(x)**. The mixture model takes these as input and generates the conditional probability density $p(\mathbf{y}|\mathbf{x})$. In an MDN, the probability density of the output variable is denoted by a mixture of m kernels as [7] -

$$p(\mathbf{y}|\mathbf{x}) = \sum_{i=1}^{m} \pi_i(\mathbf{x})\phi_i(\mathbf{y}|\mathbf{x}) \tag{4}$$

$\pi_i(\mathbf{x})$ are called the *mixing coefficients* and $\phi_i(\mathbf{y}|\mathbf{x})$ is the conditional density of i^{th} kernel. The parameters of a mixture function are denoted by **z(x)** and consist of three m dimensional vectors:

- the means of the distributions of the c output features denoted by the vector $\boldsymbol{\mu}_i(\mathbf{x})$,
- the variance of the distributions $\sigma_i(\mathbf{x})$, assuming a common variance for components of **y** exists, and
- the mixing coefficients $\pi_i(\mathbf{x})$

The mixing coefficients $\pi_i(\mathbf{x})$ must be chosen such that $\sum_{i=1}^{m} \pi_i(\mathbf{x}) = 1$. While any kernel function can be chosen, a common choice is a Gaussian function [7]. With a proper choice of mixing coefficients and Gaussian parameters i.e., the means and coefficients, the Gaussian mixture model with such a kernel can represent any density function quite accurately [14]. The NN part of an MDN is trained to make this choice. The outputs of the NN are the parameters of the distributions denoted by **z(x)**. The number of outputs of the neural network is $(c+2)*m$. The output of an MDN is [15]:

$$p(\mathbf{y}|\mathbf{x},\mathbf{w}) = \sum_{i=1}^{m} \pi_i(\mathbf{x},\mathbf{w})\mathcal{N}(\boldsymbol{\mu}_i(\mathbf{x},\mathbf{w}),\sigma_i(\mathbf{x},\mathbf{w})) \tag{5}$$

where **w** is vector of the NN weights and $\mathcal{N}(\boldsymbol{\mu}(\mathbf{x},\mathbf{w}),\sigma(\mathbf{x},\mathbf{w}))$ is the Gaussian component density. When a single output is required from the MDN, the mean $\boldsymbol{\mu}_i$ of the component with the largest central value provides a very good approximation [7]:

$$\mathbf{y} = \boldsymbol{\mu}_i(\mathbf{x}) \quad s.t. \max_i \{\frac{\pi_i(\mathbf{x})}{\sigma_i(\mathbf{x})^c}\} \tag{6}$$

MDNs have been successfully used in applications such as parametric speech synthesis [16] and spatio-temporal vision attention [17]. For more details on MDNs (kernel, loss function etc.), refer to [15] and [7].

III. NimbleCache

Considering a typical IoT scenario, all the data gathered by end devices (producers) is pushed to an edge node which caches them. The consumers of data (web or mobile applications or actuators) request data from producers, which are ideally satisfied by respective edge node caches. We make the following reasonable assumptions for IoT networks:

- Data generated by producers is bursty. Dynamic allocation is done periodically based on the data rate during a (small) unit of time. The average number of data items generated for the i^{th} application in this time unit is f_i.
- The cache receives r_i requests per time unit for items of the i^{th} application.
- The first request for a data item is received after an average of t_i seconds after the data item enters the cache.
- Each cached item of the i^{th} application has an average lifetime l_i for which it is useful.
- The cache receives an average of τ_i requests for a data item during its lifetime.

These assumptions allow modelling the effect of request and data arrivals on the CHR fairly well by our chosen NN. This is established by the results of simulations presented in Section IV. The chosen parameters are easily measurable by the edge cache in real-time.

A. NimbleCache Principles

With differentiated caching, the CHR p_i of the i^{th} application depends on the amount of cache space allotted for that application, the rate at which data enters the cache and the rate and pattern with which it is requested [3]. Web and content caching solutions generally increase the CHR p_i of an application by focusing on the cache replacement mechanism, so that caching of items is done based on the popularity of data. For IoT data, a simple FIFO replacement policy suffices, as more recent data is more valuable compared to older data. However, the request patterns for different applications are diverse and data inflow to the edge cache is bursty, with mean to peak traffic rates that can be more than tenfold [2]. Hence, the focus of NimbleCache is on allocation of cache space rather than cache replacement.

To deal with the changing traffic and request patterns, NimbleCache splits the available cache space $\mathcal{C}$ at an edge node into sections - one for each of the n applications (producers) it serves. Traffic of the i^{th} application is allotted a_i cache blocks (each of size γ), where $a \in [1, \beta]$. As the data generation and request rates for each application vary, a_i

is also varied so that CHR is maximized over a period. We denote the desired (minimum) hit ratio for the i^{th} application by p_i^d. For given traffic patterns of a set of n applications, let the overall gap between the actual and desired hit ratios of all applications be Δ as defined in Eq. 1. With conventional NN-based allocation, the focus is on minimizing the first two terms of Eq. 2 at the stake of the third. With NimbleCache, the cost of computation θ is constant and an allocation that most probably gives the desired CHR is found.

B. NimbleCache in Practice

NimbleCache works in three phases, a test phase for data gathering, the training phase where the ML agent is trained and the operations phase where cache allocation is performed using the ML agent for inference.

1) Data Gathering Phase: NimbleCache starts with edge nodes measuring the CHR (average during a set time unit) for varying network and traffic states and different allocations and building dataset D. The state of the edge traffic $\mathbf{S}^t$ at the t^{th} time unit is a n element vector, where n is the number of producers (applications). Each item in this vector is in turn a state vector $\mathbf{s}_i^t$ of the i^{th} application. Table II lists the features of an application, chosen so that each is easily measurable by the edge node and reflects the data generation pattern of the producer and the request pattern of the consumer for an application. The values of these features are captured and stored with the allocation a_i during time t at the edge cache.

TABLE II
FEATURES OF THE i^{th} APPLICATION

r_i	avg. number of requests per sec. for all data items
τ_i	avg. number of requests for the same data item during its lifetime
f_i	number of data items generated by the device per sec.
t_i	time between first data item from device and request for the same
l_i	avg. data item lifetime

2) Training Phase: As nodes may see diverse network traffic conditions during training, a model built solely based on the data gathered by one edge node may not be very accurate. Training can be done in one of the two following ways:

- Data is sent to a server that centrally trains an agent based on all the data received from the edge nodes.
- Alternately, a federated learning algorithm such as proposed in [3] can be used to train the agents without transferring data. This may yield less accurate models, but has the benefit of lower communication.

Model parameters are then transferred to edge nodes to make resource allocation decisions. The data gathering and training phases may be repeated periodically to refine the model whenever major changes occur in the network as in [3].

3) Operations Phase: Irrespective of how training happens, prediction of CHR based on the model and cache allocation based on prediction has to be done by edge nodes. This cost of inference is much less for NimbleCache than for a NN-based method. In a conventional NN-based allocation, the NN is trained with the set of features in Table II with cache allocation a_i as input and the resulting hit ratio as output. During the operations phase, each edge node periodically measures network state parameters, request and data pattern given in Table II. It then predicts the hit ratio for different possible allocations and chooses the minimum allocation that minimizes Δ from the $\binom{\beta}{n}$ possible combinations.

In contrast, NimbleCache works on the inverse problem. The input to the NimbleCache agent during training is the set of features in Table II and *the observed hit ratio* p_i and the output is the cache allocation a_i. During the operations phase, NimbleCache periodically measures features in Table II and predicts values of a_i for any application with a given *desired hit ratio* p_i^d. Note that the observed hit ratio p_i that is part of the training set is replaced by the desired hit ratio p_i^d for prediction. NimbleCache yields the PDF of the allocation that leads to the desired hit ratio. As discussed in Section II-B, the solution can be taken as the mean of the component with the largest central value predicted by the NimbleCache agent. If this allocation is not feasible (there is not enough cache space to be allocated), a different allocation can be chosen as the PDF of the allocation that leads to the desired hit ratio is known. This requires no extra training or predictions.

IV. SIMULATION RESULTS

To evaluate the performance of NimbleCache, we simulated our approach using ndnSIM, an ns-3-based network simulator for ICN networks. Information Centric Networking (ICN) is a networking paradigm where the routing is based on the name of requested data rather than IP addresses. As routing is based on the name, no or minimal resolution is necessary. Name-based routing suits applications where the number of devices is large and dynamic as device registration is not needed. In-network caching increases availability of data and reduces data retrieval times. These features make ICN an attractive choice for IoT applications [18]. Named Data Networking (NDN, [19]) is a popular ICN architecture that we use for NimbleCache. However, the principle behind NimbleCache can be used for host-centric network architectures as well.

TABLE III
PARAMETERS OF THE NETWORKS TRAINED

Parameter	**FwdNN**	**RevNN**	**MDN**
number of hidden layers	1	2	3
activation function	sigmoid	sigmoid, ReLU	tanh
units in hidden layers	8x8	8x8x8	16x32x32x32
initial learning rate	0.01	0.01	0.0001
batch size	16	16	128

We simulated typical scenarios with different types of devices common in smart campuses/homes. Bursty device traffic was generated using the traffic patterns taken from [2]. To generate sample training data, we considered three applications (producers) in different scenarios with diverse data generation patterns and network topologies. The device locations were random in an area of 100x100 m, with the edge node roughly at the centre. The link between the edge node and

producers was WiFi (IEEE 802.11a) with loss as per the ns-3 HybridBuildingsPropagationLossModel. The t_i and l_i values were varied from 1 to 10 seconds with τ_i varying from 1 to 5 (Table II). The value of r_i was varied from 5 to 20 per second and request arrivals were Poisson. The features of each application were captured after every 100 simulation seconds. We considered two edge cache sizes, 250 (with a granularity γ of 25) and 1000 items (with two granularities of 25 and 100). While these numbers may be much larger in real world, it does not effect the efficacy of NimbleCache as long as enough training data is gathered from similar scenarios as those for which allocation needs to be done. After data gathering, we trained three networks:

- FwdNN, to predict the CHR given the allocation and system state to know the optimal allocation in terms of performance and allocated space,
- RevNN, to predict the allocation for a desired CHR and
- NimbleCache (using MDN), to predict the PDF of space allocation for a desired CHR.

The model parameters for each of these networks are given in Table III. First, we evaluated performance of FwdNN in predicting CHR for given traffic conditions and allocation. The mean-squared-log-error (MSLE) after training was ≈5%. Figure 1 illustrates this by showing values predicted in comparison with actual values for eight network states and allocations. It can be be seen that FwdNN can predict the hit ratio in all scenarios (states) with reasonable accuracy.

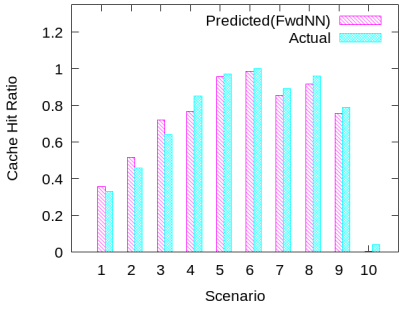


Fig. 1. Predictions of CHR for different network states by FwdNN

Next, we considered two values of γ (minimum allocation unit of the cache) - 25 and 100 for the same cache size (1000 items). For γ = 25, the maximum number of units possible per application (β) is 38 and for γ = 100, β = 8 from Eq. 3. FwdNN was used for predicting the value of Δ for all possible allocations (8436 and 56 for γ = 25 and 100 respectively) and the allocation that leads to the minimum value of Δ with minimum possible allocated space is chosen. We call this the optimal policy as this is the minimal allocation that can give the best value of Δ.

RevNN and NimbleCache were then used for predicting the allocation for each state with a desired CHR p^d of 1 for all applications. In the case of NimbleCache since the output is a set of mixing component parameters (32 components, based on the suggestions in [7]), the mean of the component with the largest central value was taken as the output as discussed in Section II-B. Figure 2 depicts the total cache space allocated (i.e.,$\sum_{i=1}^{n} a_i$) by each of these schemes as a percentage of the total space allocated by the optimal policy (which uses FwdNN) for γ = 100. It can be seen that NimbleCache allocates less space compared to RevNN in all scenarios.

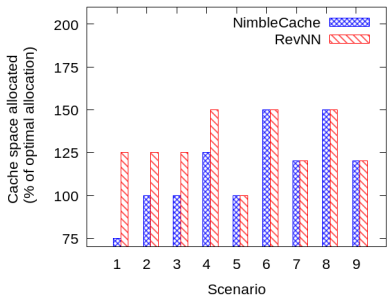


Fig. 2. Cache Space Allocated for Different Scenarios ($\mathcal{C}$=1000, γ=100)

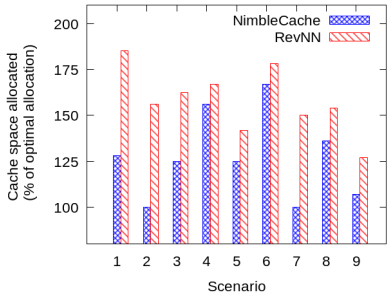


Fig. 3. Cache Space Allocated for Different Scenarios ($\mathcal{C}$=1000, γ=25)

Figure 3 shows the total space allocated for γ = 25 for the same set of scenarios as in Figure 2. As this is a finer granularity, the effects of over-allocation are reduced for all policies. The benefit of NimbleCache is clearer here. While optimal allocation would require 8436 possible combinations to be checked, NimbleCache allocates space closer to the optimal allocation with just one prediction per application. Simulating the network with the allocations suggested by each of the three schemes confirms that all three schemes result in a Δ of 0 (CHR of 1 for all applications) for this cache size. Hence, NimbleCache can achieve a CHR comparable to that of the optimal allotment policy with much less computation. The relative performance of the three schemes in meeting the optimization goal (Eq. 2) can be seen in Figure 4.

In some cases, it may not be possible to allocate space for all applications to have a CHR of 1. In such cases, the allocation can be done by choosing a different (lower) CHR for one or more applications. With NimbleCache, this can be done by choosing a different space allocation value from the PDF of the allocation predicted for a target CHR of 1. We chose an allocation value (for each application) of

$$a_i = \mu_i(\mathbf{s}) - \sigma_i(\mathbf{s}) \quad s.t.\ \max_i\{\frac{\pi_i(\mathbf{s})}{\sigma_i(\mathbf{s})}\} \tag{7}$$

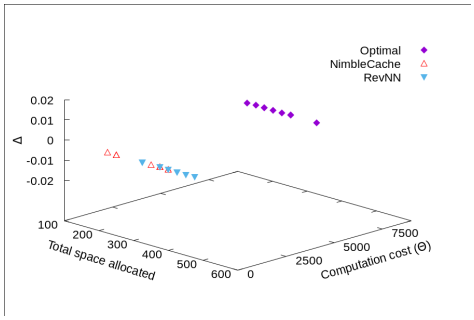


Fig. 4. Optimization Goal Achievement ($\mathcal{C}$=1000, γ=25)

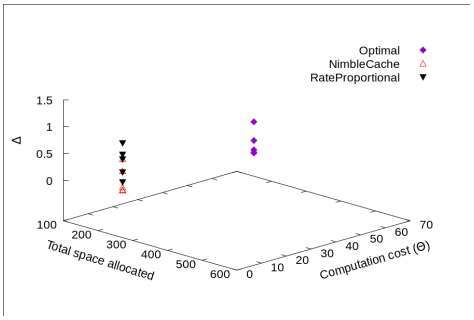


Fig. 6. Optimization Goal Achievement ($\mathcal{C}$=250, γ=25)

Here, the suffix i refers to the value of the i^{th} mixture component. The resulting value of Δ is plotted in Figure 5. In addition to the values of Δ for the optimal and NimbleCache schemes, the values for a simple rate-proportional space allocation (proportional to the rate of incoming data rate for each application) is also plotted. The RevNN scheme is not plotted as the values suggested by the scheme are not feasible to be allotted (for a desired hit ratio of 1). This is because RevNN gives the average of several possible allocations that result in the desired performance and it is infeasible to allocate these predicted values (i.e., $\sum_{i=1}^{n} a_i > \mathcal{C}$) for all applications.

In Figure 5 it can be seen that NimbleCache gives a value of Δ close to the optimal allocation in all cases. The total area allocated in all cases is the entire cache (250 items). NimbleCache and rate-proportional allocation require a single prediction per application, while the optimal allocation requires 56 predictions. Hence, NimbleCache gives a good balance between the three optimization goals (Eq. 2), as shown in Figure 6.

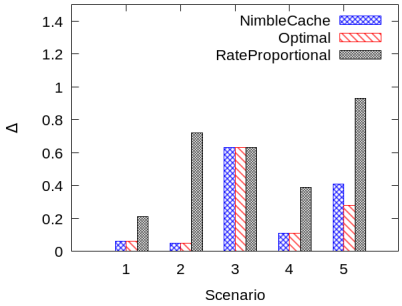


Fig. 5. Δ for different allocation strategies ($\mathcal{C}$=250, γ=25)

In Table III it can be seen that the MDN used for NimbleCache requires more hidden layers than FwdNN or RevNN. Also, while FwdNN and RevNN predict a single value, NimbleCache predicts the parameters of each Gaussian component (3 parameters each for 32 mixture components for this simulation setup). Hence the model (the weights and biases of the MDN) that has to be stored at the edge cache requires more space compared to FwdNN and RevNN. However, the saving in *cache* space can be quite substantial with NimbleCache as shown in Figures 2 and 3. Moreover, the PDF predicted by NimbleCache makes a myriad of choices possible for probabilistic space allocation with a single model.

V. Related Work

Differentiated caching for web servers was explored by [20] and many others. Differentiated caching with diverse QoS requirements using a control theoretic approach to allocate cache space in proxy servers such that the a CHR close to the target CHR is achieved for each application has been proposed in [21]. The crux of these works is that cache partitioning can support differentiated QoS for different applications. Drawing on this, we propose a cache partitioning mechanism that is particularly suited to IoT networks with bursty data generation, varying request patterns and unpredictable channel conditions.

An excellent overview of machine learning techniques for next generation wireless networks is given in [22], while [8] identifies the main impediments in applying machine learning to IoT. Low computational capabilities of IoT devices, the need for quick convergence and decision-making are among the challenges identified. Cloud-based ML frameworks are often used for training in resource constrained IoT networks. While we agree that training can be off-loaded to the cloud, we focus not on the training cost, but on the *cost of inference*. As inference from the model needs to be done frequently and dynamically at the edge node performing the resource allocation, the cost can be quite substantial.

Deep learning for resource allocation in networks has been studied by several researchers as an optimization problem with a single goal(e.g, [4]), while multi-objective optimization was studied by a few. Of particular relevance to our problem is [23], which focuses both on cache allocation and transmission rate in content-centric IoT networks for improving the Quality of Experience dynamically using deep RL. However, deep RL maybe too complex and slow to converge for IoT nodes [8]. Pareto Q-learning [24] proposes the use of Q-learning for finding the entire Pareto front, but requires a Q-table to store quality of the state-action pairs. This is very memory-intensive, especially when the number of possible (network) states and actions (possible allocations) is large and may not be feasible at edge nodes. In [25], the authors explore deep reinforcement learning for making

caching decisions adaptively. Their focus is efficient delivery of data in content delivery networks (CDNs). CDNs differ a lot from IoT networks in the data and request patterns and QoS requirements such as delay. In this paper, we focus on those aspects that play a major role in IoT networks.

A deep learning-based model to predict object popularity and replace cache contents is proposed in [26]. However, web page access requests and QoS requirements are very different from those of IoT. We also propose a cache allocation policy to competing applications. While we consider FIFO as the most apt policy for most IoT applications, any replacement policy including [26] can be used in NimbleCache. In [27], authors propose service differentiation in IoT caches by dividing applications in vehicular networks into two types, infotainment applications and safety-critical applications. They propose dynamically allocating different portions of the cache to each application type and applying different replacement schemes to suit each request pattern. In contrast, we consider per-application allocation of cache space and a much wider set of features that takes into account varying request and bursty data generation patterns typical in IoT scenarios. In addition, our goal is to optimize resources at the edge. While predicting PDFs (Probability Density Functions) has been studied for other applications [16], [17], to the best of our knowledge, this is the first paper that studies the feasibility of resource allocation using PDF prediction in constrained networks.

VI. Conclusions and Future Work

In this paper, we presented a set of parameters that effect the edge cache hit ratio in IoT networks and an MDN-based dynamic, differentiated cache space allocation scheme called NimbleCache. NimbleCache predicts the PDF of the cache space that results in a desired hit ratio for a given network state with constant, low computation cost. This is used to achieve a desired cache hit ratio with minimal cache space allocation. Simulation results show that NimbleCache results in performance and space savings close to the optimal allocation policy with much less computation cost. In future, we plan to compare NimbleCache with deep RL and multi-objective RL methods and include prioritization of data.

References

[1] S. Wang, X. Zhang, Y. Zhang, L. Wang, J. Yang, and W. Wang, "A survey on mobile edge networks: Convergence of computing, caching and communications," *IEEE Access*, vol. 5, pp. 6757–6779, 2017.

[2] A. Sivanathan, D. Sherratt, H. H. Gharakheili, A. Radford, C. Wijenayake, A. Vishwanath, and V. Sivaraman, "Characterizing and classifying iot traffic in smart cities and campuses," in *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, May 2017, pp. 559–564.

[3] Shanti Chilukuri and Dirk Pesch, "Achieving optimal cache utility in constrained wireless networks through federated learning," in *The 21st IEEE International Symposium on A World of Wireless, Mobile and Multimedia Networks (IEEE WOWMOM 2020)*, 2020.

[4] H. Sun, X. Chen, Q. Shi, M. Hong, X. Fu, and N. D. Sidiropoulos, "Learning to optimize: Training deep neural networks for wireless resource management," in *2017 IEEE 18th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, 2017, pp. 1–6.

[5] H. Ye, G. Y. Li, and B. F. Juang, "Deep reinforcement learning based resource allocation for v2v communications," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 4, pp. 3163–3173, 2019.

[6] Z. Xu, Y. Wang, J. Tang, J. Wang, and M. C. Gursoy, "A deep reinforcement learning based framework for power-efficient resource allocation in cloud rans," in *2017 IEEE International Conference on Communications (ICC)*. IEEE, 2017, pp. 1–6.

[7] C. M. Bishop, "Mixture density networks," 1994.

[8] T. Park, N. Abuzainab, and W. Saad, "Learning how to communicate in the internet of things: Finite resources and heterogeneity," *IEEE Access*, vol. 4, pp. 7063–7073, 2016.

[9] H. Sun, X. Chen, Q. Shi, M. Hong, X. Fu, and N. D. Sidiropoulos, "Learning to optimize: Training deep neural networks for interference management," *IEEE Transactions on Signal Processing*, vol. 66, no. 20, pp. 5438–5453, 2018.

[10] M. Eisen, C. Zhang, L. F. Chamon, D. D. Lee, and A. Ribeiro, "Learning optimal resource allocations in wireless systems," *IEEE Transactions on Signal Processing*, vol. 67, no. 10, pp. 2775–2790, 2019.

[11] S. Boyd, S. P. Boyd, and L. Vandenberghe, *Convex optimization*. Cambridge University press, 2004.

[12] A. Dosovitskiy and J. Djolonga, "You only train once: Loss-conditional training of deep networks," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=HyxY6JHKwr

[13] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.

[14] G. J. McLachlan and D. Peel, *Finite Mixture Models*. Wiley Series in Probability and Statistics, 2000.

[15] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer.

[16] H. Zen and A. Senior, "Deep mixture density networks for acoustic modeling in statistical parametric speech synthesis," in *2014 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, 2014, pp. 3844–3848.

[17] L. Bazzani, H. Larochelle, and L. Torresani, "Recurrent mixture density network for spatiotemporal visual attention," *arXiv preprint arXiv:1603.08199*, 2016.

[18] J. Quevedo, D. Corujo, and R. Aguiar, "A case for icn usage in iot environments," in *2014 IEEE Global Communications Conference*, Dec 2014, pp. 2770–2775.

[19] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, P. Crowley, C. Papadopoulos, L. Wang, B. Zhang *et al.*, "Named data networking," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 66–73, 2014.

[20] Wenzhong Chen, P. Martin, and H. Hassanein, "Differentiated caching of dynamic content using effective page classification," in *IEEE International Conference on Performance, Computing, and Communications, 2004*, 2004, pp. 293–298.

[21] Y. Lu, T. F. Abdelzaher, C. Lu, and G. Tao, "An adaptive control framework for qos guarantees and its application to differentiated caching," *IEEE 2002 Tenth IEEE International Workshop on Quality of Service (Cat. No.02EX564)*, pp. 23–32, 2002.

[22] J. Wang, C. Jiang, H. Zhang, Y. Ren, K.-C. Chen, and L. Hanzo, "Thirty years of machine learning: The road to pareto-optimal next-generation wireless networks," *arXiv preprint arXiv:1902.01946*, 2019.

[23] X. He, K. Wang, H. Huang, T. Miyazaki, Y. Wang, and S. Guo, "Green resource allocation based on deep reinforcement learning in content-centric iot," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2018.

[24] K. Van Moffaert and A. Nowé, "Multi-objective reinforcement learning using sets of pareto dominating policies," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3483–3512, 2014.

[25] A. Sadeghi, G. Wang, and G. B. Giannakis, "Deep reinforcement learning for adaptive caching in hierarchical content delivery networks," *IEEE Transactions on Cognitive Communications and Networking*, vol. 5, no. 4, pp. 1024–1033, 2019.

[26] A. Narayanan, S. Verma, E. Ramadan, P. Babaie, and Z.-L. Zhang, "Deepcache: A deep learning based framework for content caching," 08 2018, pp. 48–53.

[27] V. S. Varanasi and S. Chilukuri, "Adaptive differentiated edge caching with machine learning for v2x communication," in *2019 11th International Conference on Communication Systems Networks (COMSNETS)*, Jan 2019, pp. 481–484.