A Geometric Approach to Real-time Quality of Experience Prediction in Volatile Edge Networks

Tom Goethals
Department of Information Technology
Ghent University - imec, IDLab
Gent, Belgium
ORCID 0000-0002-1332-2290

Bruno Volckaert
Department of Information Technology
Ghent University - imec, IDLab
Gent, Belgium
ORCID 0000-0003-0575-5894

Filip De Turck
Department of Information Technology
Ghent University - imec, IDLab
Gent, Belgium
ORCID 0000-0003-4824-1199

***Abstract*—In recent years, the continuing growth of the network edge, along with increasing user demands, has led to the need for increasingly complex and responsive management strategies for edge services. Many of these strategies are cloud-based, offering near-perfect solutions at the cost of requiring massive computational power, or edge-based, offering reactive strategies to changing edge conditions. This paper presents a decentralized, pro-active Quality of Experience (QoE) based architecture designed to run on edge nodes, which allows nodes to predict optimal service providers (fog nodes) in advance and request their services. The concepts behind the components of the architecture are explained, as well as geometry-inspired design decisions to limit model size. Evaluations on an NVIDIA Jetson Nano show that the architecture can predict optimal service providers for an edge node in real-time for 5 to 20 QoS (Quality of Service) and QoE parameters, with at least 50 potential fog nodes, and that overall QoE resulting from its use is improved by 1% to 18% over previous work such as SoSwirly, depending on the scenario.**

***Index Terms*—edge computing, edge intelligence, edge AI, quality of experience**

I. Introduction

The number and variety of edge devices has grown immensely in recent years, while changing user demands keep increasing the need for fog and edge services. Combined with a volatile network topology due to mobile nodes (e.g. Internet of Vehicles applications), edge services require increasingly responsive and intelligent management strategies.

Predictive models for service placement are generally designed for the cloud, and consist of tens of millions of parameters that generate near-perfect solutions at the scale of cloud data centers. However, the network edge is highly decentralized and consists of low-resource devices incapable of running such models, while the relevant parameters are radically different due to its scale, volatile topology, and variety. Edge service orchestrators often use basic reactive algorithms to determine service placement; however, conditions in the edge may be such that a proactive approach using Artificial Intelligence (AI) results in an overall preferable solution. Additionally, optimal selection of service providers for edge nodes depends on user experience, so subjective QoE parameters should be considered alongside QoS.

978-3-903176-51-5/22/$31.00 ©2022 IEEE

SoSwirly [1] is a decentralized edge service orchestrator which uses agents on edge devices to discover other nodes in their neighbourhoods, and request services from optimal providers depending on a generic distance metric. However, SoSwirly is reactive, and only changes its service topology when unacceptable QoS conditions are detected. This paper presents a compact, but effective, neural network architecture for real-time prediction of fog service QoE, designed to work with SoSwirly to replace its default reactive algorithm. As such, the proposed architecture is edge-driven, entirely decentralized, and aimed at low-resource edge devices. Although designed for decentralized operation with SoSwirly, the concepts and architecture developed in this paper are orchestrator-agnostic, and can be used for other solutions with minimal adaptation. Concretely, the contributions of this paper are:

- Confirming the feasibility of decentralized, edge-driven service orchestration with acceptable QoE.
- Illustrating the possibility of small but highly accurate neural network models based on geometrical concepts, and their performance on low-resource edge nodes.
- Building a neural network for QoE approximation based on the geometrical interpretation of neural networks as coordinate transformations on high-dimensional manifolds.
- Predicting near-future QoE changes through a (Recurrent Neural Network) RNN-based model borrowing from Q-learning concepts.

Considering that the solution is aimed at edge devices with $<$1GiB memory and low-power processors, there are some requirements:

Req1 Its added resource use should be less than that of default SoSwirly. CPU should not exceed 5% of a single core on average, and memory use should not exceed 10MiB.

Req2 It must be fast enough to react to edge topology changes in near real-time. A single update round, estimating the QoE of up to 50 nearby nodes, should take less than 50ms.

Req3 The efficiency of the model must be higher than that of default SoSwirly, measured as the range of QoE for a node for a prolonged period of time. The required efficiency improvement depends on the additional load caused by the model.

DOI: 10.23919/CNSM55787.2022.9964775

The rest of this paper is organized as follows: Section II presents existing research related to geometrical neural network interpretations and QoE management in the edge. Section III provides a theoretical framework and basic design for both components of the model, while Section IV discusses implementation details. In Section V, the evaluation setup and methodology are presented, while the results are presented in Section VI and discussed in Section VII. Finally, Section VIII draws high level conclusions from the paper.

II. Related Work

Although the classical geometric interpretation of neural networks is an optimization function finding a global minimum in an arbitrary n-dimensional space, often focused on the geometry of loss functions [2], some alternative approaches exist. Notably, Hauser et al. [3] provide a mathematical framework that shows how neural networks can determine Riemannian metric tensors for any parameter input space, and that those tensors can be transformed into Euclidian metrics.

Some studies present neural architectures that explicitly calculate non-linear functions. There are, for example, studies that present architectures for approximation of quadratic functions [4] and full polynomial fitting [5]. However, this paper relies on the properties of tensor algebra to learn non-linear (and possibly non-polynomial) coordinate transforms.

Magableh et al. [6] illustrate the use of a Deep Recurrent Q-Network (DRQN) for an effective self-adaptive service architecture, although the algorithm is not designed for decentralization and can not be used in real-time, unlike the solution presented in this paper. Lu et al. [7] use a solution based on double dueling Deep Q-networks (DQN) to determine optimal offloading policies in the edge. Although deployed in the edge, the algorithm is quite resource intensive and thus run on more powerful edge servers. QoE-DEER [8] uses a game-theoretic to edge resource allocation, thereby decentralizing resource allocation and giving end-users some degree of control over their QoE. However, it is focused on allocating scarce resources between the needs of various users, rather than service orchestration. Various other decentralized edge solutions exist, for example for offloading [9] and for minimizing the energy use of QoE-constrained services [10]. Many studies, for example 5G-QoE [11], are focused on modeling QoE in the edge for the purposes of reliable and qualitative media streaming, as a particularly QoE-sensitive application. Such domain-specific strategies can be integrated into the proposed solution. Finally, Barakabitze et al. [12] provide a survey on the subject of intelligent QoE management for multimedia in the edge. An overview of useful embeddings related to QoE parameters is given by Potdar et al. [13]. Tokuyama et al. [14] show how timestamps and traffic volume data can be encoded for IoV models, and their findings indicate that day of week is an important factor in addition to a daily timestamp.

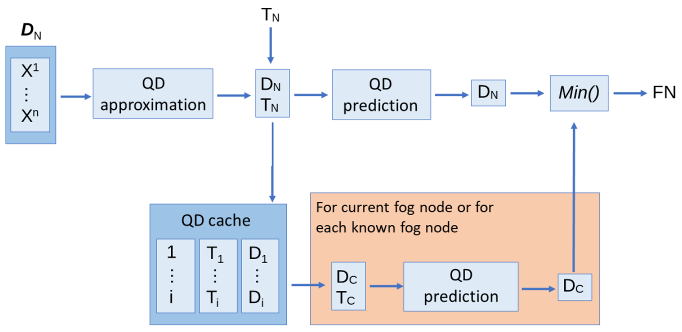

Fig. 1: Complete architecture of QD-based service node selection.

III. Architecture Components

For the purposes of this paper, Qualitative Distance (QD) is used rather than QoE, as a lower QD value means better (i.e. higher) QoE. QD is equated to "distance" in arbitrary manifolds, simplifying the notation of several equations. One option of calculating a useful QoE from QD is

$$QoE = 1 - tanh(QD) \tag{1}$$

which approaches 0 QoE asymptotically as QD increases.

The architecture, shown in full in Fig. 1, depends on a QD approximation model and a QD prediction model. The edge node running the architecture is assumed to regularly discover its neighbourhood and query nearby fog nodes (i.e. service providers) for their properties and locations. QoS-related input parameters (e.g. network, memory) can be measured directly, while QoE-related parameters can be constructed from limited user input, e.g. preference sliders or big data analysis from user feedback. When the querying node receives the required information, the distance or QD vector $\mathbf{D}_N$ to node N is fed into the QD approximation model. The resulting distance (or QD) D_N, along with the timestamp T_N are stored in a QD cache, which contains the latest QD of each fog node in the neighbourhood. Both D_N and T_N are also fed into the QD prediction model, calculating an estimated future QD $D_{N,+t}$. The model is also run for every other known fog node, after which the closest one is returned as the optimal service provider for the short term future.

A. QD approximation

A metric tensor is a mathematical object that can calculate non-euclidean (i.e. curved) distances between positional vectors in arbitrary coordinate systems. While it is impossible to calculate a global metric tensor in an edge network due to varying topology features at each node [1], it is possible to learn an individual metric tensor at each edge node. This subsection uses a geometric approach to construct a neural network model for QD approximation suitable for inference on edge nodes.

Given some distance vector $\mathbf{d}$ between nodes, in a manifold containing all dimensions relevant to QD, their relative distance d can be calculated with a metric tensor g_{ij}, defined as representing the gradient products of each pair of dimensions in a manifold:

$$\mathbf{g} = g_{ij} dx^i dx^j \tag{2}$$

$$d = \int_{\mathbf{d}} \sqrt{ds^2} = \int_{t=0}^{\mathbf{d}} \sqrt{g_{ij} \frac{d\mathbf{d}^i}{dt} \frac{d\mathbf{d}^j}{dt}} \tag{3}$$

As the metric tensor can only calculate magnitude increments ds^2, it can not be used directly on vectors in any space apart from Euclidian coordinates and other orthogonal coordinate systems. Furthermore, Eq. 3 shows that if the components g_{ij} are learned directly as weights between two layers, treating $\mathbf{D}$ as input, this method would be limited to metrics that represent linear combinations of the input dimensions. At most, the activation function introduces some non-linearity, but complex functions can not be learned.

However, interpreting the weights tensor between (fully connected) layers of n neurons as a coordinate transform, and assuming a random metric tensor g_o of an n-dimensional manifold o representing the input data, there exists a sequence of m coordinate transforms in manifolds x_m which reduce the metric tensor to Euclidian distance g_e in space e:

$$\mathbf{g}_e = \left[\frac{de_i}{do_j}\right] \mathbf{g}_o = \prod_{k=1}^{m} \left[\frac{dx_{k,i}}{dx_{k-1,j}}\right] \mathbf{g}_o, i, j = 0..n \tag{4}$$

Each such a coordinate transform can be represented by a weight tensor $\mathbf{w}$, with the restriction that for the input vector $\mathbf{x}$ and output vector $\mathbf{y}$ of each layer:

$$\mathbf{y}^k = \frac{\mathbf{d}\mathbf{y}_k}{\mathbf{d}\mathbf{x}_l} \mathbf{x}^l = \mathbf{F}\left(\frac{\partial \mathbf{y}_k}{\partial \mathbf{x}_l} \mathbf{x}^l\right) = \mathbf{F}(\mathbf{w}_l^k \mathbf{x}^l) \tag{5}$$

Where each component of both $\mathbf{x}$ and $\mathbf{w}$ is a scalar. Thus, while technically the operation can be performed by a single tensor, the restriction makes those components too complex for individual neurons or layers to model. Instead, several layers of linear combinations are preferred, with the activation functions $\mathbf{F}$ for each layer introducing non-linearity. Hauser et al. [3] show that hyperbolic activation functions allow a network to learn the requisite transformations. The correctness of this method is confirmed by the dual nature of tensors, in which one-forms $\mathbf{d}$ and vectors $\mathbf{v}$ are inversely affected by coordinate transforms Λ, and as a result the magnitude m calculated by the metric tensor is constant for each layer l:

$$\mathbf{v_2}^i = \Lambda_j^i \mathbf{v_1}^j, \mathbf{d_1}_i = \Lambda_i^j \mathbf{d_2}_j \tag{6}$$

$$m = \mathbf{g}_{1,ij} \mathbf{x_1}^i \mathbf{x_1}^j = ct, \forall l \tag{7}$$

With the final layer conveniently representing Cartesian coordinates. The size of the input vector for each layer is identical, and the network is relatively shallow due to the modeling ability of hyperbolic functions. As such, this approach is acceptable for both **Req1** and **Req2**. Fig. 2 shows the proposed theoretical model to predict QD D from an m-dimensional distance vector $\mathbf{x}$ through hidden layers H_i.

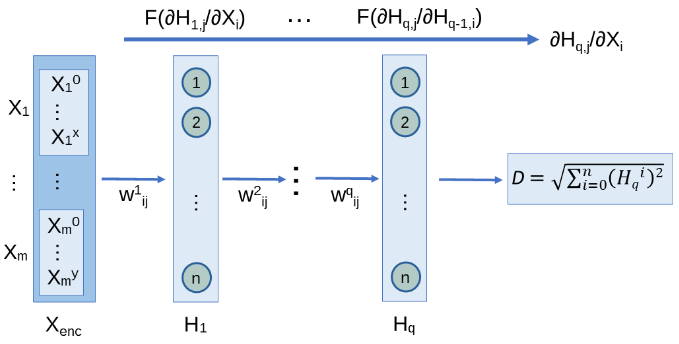

Fig. 2: General architecture of metric tensor-based QD approximation network.

Finally, the properties of the distance vector components should be considered. Spatio-temporal dimensions of the distance vector $\mathbf{d}$ are naturally associated with distance between nodes (e.g. geographic location, latency), but property dimensions merely represent target node properties (e.g. free memory, bandwidth). To calculate $\mathbf{D} = \mathbf{p}_{target} - \mathbf{p}_{source}$, the property dimensions of position vector $\mathbf{p}_{source}$ should be set to 0.

B. QD Prediction

The positional vectors and resulting QD of nodes at each timestep are essentially time-series data, ideally processed by an RNN. Reinforcement Learning (RL) is required, as the volume of training data for each node would be difficult or even impossible to label, depending on if and how the metric tensor for QD is defined. A deep network, however, is not required due to the limited complexity of the inputs. However, by default an RNN witn RL would merely predict the next value in a timeseries, whereas the model should be able to look ahead several timesteps, with a discount on the future. While an explicit time difference could be passed to the model to determine how far it should predict, such an input would significantly increase the number of training samples required. Additionally, it would be hard to discount such a value to any degree. Furthermore, the error for future predictions would grow increasingly larger, as like a metric tensor, an RNN is more suitable for predicting in smaller increments, i.e. the next timestep. To solve this, a property of Q-learning [15] is borrowed, which incorporates a discounted expected future reward using the Bellman equation:

$$Q(s_t, a_t) = r_t + \gamma \max \hat{Q}(s_{t+1}, a) \tag{8}$$

In which r_t can be considered the immediate reward for choosing a node (i.e. $D_{t+1} - D_t$) and γ is the discount factor for future rewards. This is converted to a more suitable form, by using D_t directly and extending to n timesteps:

$$D_{W,t+n} = \frac{\sum_{i=0}^{n} \gamma^i D_{t+i}}{\sum_{i=0}^{n} \gamma^i} \tag{9}$$

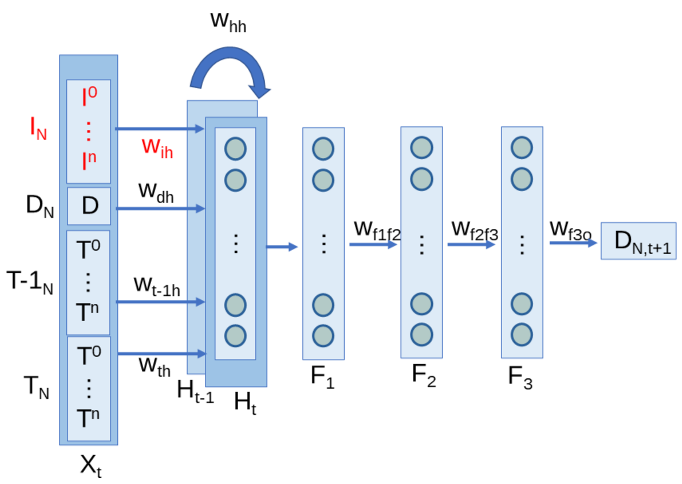

Fig. 3: Proposed architecture of QD prediction RNN.

Where D_{t+i} is the QD of a node at a specific timestep, and $D_{W,t+n}$ is the discount-weighted QD for the next n timesteps. Whereas the Bellman equation is used for maximizing rewards, Eq. 9 is more suitable for an RNN attempting to predict time-series data, for which Mean-Squared Error (MSE) is used as a loss function.

The proposed model is shown in Fig. 3. As the QD D_N of a fog node is already known, it is used directly as an input. Other inputs include node identification I_N, the QD timestamp T_N and previous QD timestamp $T{-}1_N$. As with the QD approximation model, a suitable encoding or embedding for each input must be constructed, although Sections IV and VI show that I_N is not always a requirement for accurate predictions. A single recurrent layer with GRU cells is used, to store state information and calculate complex time-dependent relations between measurements [16]. After the recurrent layer, 3 fully connected layers calculate up to third-order features using ReLU activation functions, which is enough to make accurate predictions from the recurrent inputs. A final neuron outputs the expected QD $D_{N,t+1}$ for the near future.

IV. Implementation

The architecture components are implemented in Python using TensorFlow, and executed as TensorFlow and TensorFlow Lite [1] models, the latter being explicitly designed for low-resource use.

For QD approximation, 3 fully connected hidden layers with tanh activation functions are used, which is enough to model most curvilinear spaces with acceptable accuracy. This assumption is confirmed using a version with 5 fully connected hidden layers. The depth of the network is independent of the number of input dimensions; each coordinate transform considers each pair of dimensions, and more layers only improve how accurately the model can represent random nonlinear dimensions. This model uses neither dropout nor regularization, as all parameters are potentially important for the end result. Furthermore, each input dimension is normalized into a single floating point input, removing the need for embedding and encoding. The final layer converting Cartesian coordinates to distance is slightly modified. Instead of learning coordinates H_3^i, the third hidden layer learns $(H_3^i)^2$, changing the output neuron to a sum and linear activation. The inputs of the output neuron are also weighted to allow a final scaling of each Cartesian dimension. The output does not represent the QD D directly, instead it results in D^2/D_{max}^2, from which D can be calculated given the maximum possible distance D_{max}.

The QD prediction network is modified so that no node ID inputs are required; during training the model is fed time series data for various nodes, learning general patterns from their individual behaviors. As such, the architecture is highly flexible and scalable in terms of discovered nodes. Again, the input layer is normalized, with D representing the fraction of maximum distance, and both T and $T+1$ representing a complete temporal "cycle". Such a "cycle" can be a day or a week, depending on application needs. All layers contain 30 neurons, apart from the last fully connected layer before the final output, which contains only 10 neurons.

V. Evaluation

This section describes the evaluation setup, evaluation scenarios, methodology and any tools used, as well as a description of simulated environments. The code for the models and training sample generator is made available on GitHub[2].

All evaluations are performed on an NVIDIA Jetson Nano, with kernel version 4.9.201-tegra, TensorFlow 2.4.0+nv21.5, and CuDNN 8.0. While generic edge devices are not necessarily as powerful, a Jetson Nano supports TensorFlow and is categorically an edge device.

A. Methodology

Both models are evaluated in terms of model size, memory consumption, and execution time. Model properties are shown as reported by the Keras Model Profiler[3]. Execution time is measured using TensorFlow for batched execution, and TensorFlow Lite for single inputs, each for 50.000 to 200.000 samples to amortize any overhead as much as possible. Additionally, QD approximation accuracy and the efficiency of QD prediction over SoSwirly are examined.

1) QD approximation accuracy: The position vector $\mathbf{p}_N$ for a node N consists of geographical x and y coordinates, a flag s_N showing service deployment status, free memory m_N and (estimated) available network bandwidth b_N. Memory and network bandwidth are considered property dimensions, so the distance between two nodes $N1$ and $N2$ is:

$$\mathbf{p}_{1,2}(x_{N2} - x_{N1}, y_{N2} - y_{N1}, s_{N2}, m_{N2}, b_{N2}) \tag{10}$$

Both component models of the architecture are evaluated separately to accurately gauge their efficiency.

[1]https://www.tensorflow.org/lite

[2]https://github.com/togoetha/rnnswirly

[3]https://pypi.org/project/model-profiler/

Although the primary use of the QD approximation model is to learn metric tensors which have no known representation, formally defined metrics provide both a ground truth and convenient training samples. Two distinct distance metrics are used for the evaluation:

- **DM1** uses only the geographical distance $s = \sqrt{x^2 + y^2}$. This metric is meant both as a control mechanism, and to evaluate how the model handles garbage input dimensions.
- **DM2** uses geographical distance, service deployment status, free memory and free bandwidth. Lower resources means higher distance, although an exponential discount is applied. Concretely, the distance metric is given by Eq. 11.

$$s^2 = x^2 + y^2 + 10^4 s_d + 36S(m, 1024) + 400S(b, 100) \quad (11)$$

$$S(x, s) = \left(1 - sigmoid\left(\frac{6x}{s}\right)\right) s \quad (12)$$

$S(x, s)$ is a support function based on the Sigmoid function, used for a scaled exponential discount of a certain resource. These distance metrics equate to the following metric tensors for the input data:

$$g_{DM1} = \delta_{ij}, i, j < 2 \quad (13)$$

$$g_{DM2} = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 100 & 0 & 0 \\ 0 & 0 & 0 & 6F(m, 1024) & 0 \\ 0 & 0 & 0 & 0 & 20F(b, 100) \end{vmatrix} \quad (14)$$

$$F(x, s) = \left(\frac{\partial S(x, s)}{\partial x}\right)^2 = \frac{36e^{12x/s}}{(e^{6x/s} + 1)^4} \quad (15)$$

For the evaluation, 200.000 training samples and 100.000 validation samples are generated, all random.

2) QD prediction efficiency: For QD prediction, a car trip simulator is created which generates datasets containing the necessary fields for QD calculation and QD prediction. Several such datasets are used as training samples, while others are used for validation. The geographical area used for the simulator is available in the GitHub repository.

The geographical area is defined as 255 by 255 units, with 50 fog nodes at random positions, and an edge node travelling along a number of predefined paths for 1000 timesteps. At each timestep, the edge node "receives" positional vectors from the fog nodes, and calculates the QD for each. Furthermore, a configurable number of static edge nodes are created, and assigned to the fog nodes. As such, some fog nodes may be randomly unavailable due to their resources being fully utilized. Model efficiency is evaluated by comparing QD of the travelling edge node for an entire path using two methods, both selecting the lowest QD fog node with available resources when switching to another fog node:

- Fog node selection based on SoSwirly, i.e. greedy. No switches to another fog node are allowed until the maximum allowable QD is reached to avoid overly frequent switching. No switching to other fog nodes over the maximum QD is allowed, as they offer no acceptable improvement.
- Fog node selection based on QD prediction. No switches to another fog node are allowed until maximum allowable QD is reached. However, as this method is based on predictions, switches to "better" fog nodes currently over the maximum QD are allowed.

For the evaluation, maximum QD is set at 100, and **DM2** is used for QD calculation, as it introduces randomness based on the available resources of each fog node. Finally, three separate situations are evaluated:

- **N**ormal, in which 750 edge nodes are generated and most fog nodes have few free resources left.
- **L**ow density, with only 400 edge nodes and plenty of opportunity to switch to other fog nodes for services.
- **H**igh density, a tipping point scenario with 800 edge nodes, resulting in a significant percentage of fog nodes with no free resources.

While the difference between Normal and High may seem minimal, the computational complexity of SoSwirly has an exponential drop-off near tipping points, depending on node densities.

Note that the evaluation of QD prediction is entirely independent of the QD approximation model; both methods would be equally affected by its use. Furthermore, Section VI shows that its detrimental impact on total QD (and thus QD prediction) is acceptably small.

TABLE I: QD approximation model properties for 5 and 20 dimensions. Memory is main memory + GPU memory.

Model	Size	Memory	Single	Batched
QD (5 dim)	2.59Kb	0.375Kb + 0.156Kb	109µs	75ns
QD (20 dim)	7.21Kb	5Kb + 1.49Kb	108µs	195ns
Prediction	32.61Kb	20.82Kb + 5.6Kb	138µs	1876ns

VI. Results

This section presents the results of the evaluations, and shows how they confirm that the architecture fulfills **Req1** through **Req3**.

A. Model Properties

All model properties are summarized in Table I. The QD approximation model is evaluated for the 5 input dimensions discussed in Section V, as well as 20 dimensions, showing how it performs in larger QD parameter spaces. The size of both versions is well under 10Kb, showing that they can be stored on even extremely resource-constrained devices. Memory consumption is similarly low at less than 10Kb for RAM and GPU combined. Execution times are extremely fast, around 100µs for single samples, while batched execution takes 75ns and 195ns per sample, for 5 and 20 dimensions respectively.

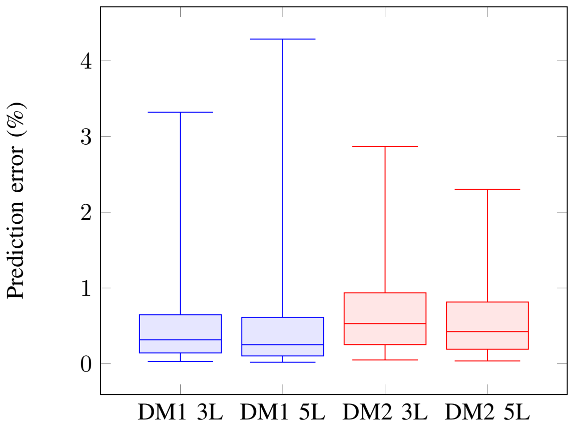

Fig. 4: Prediction error of QD approximation network, using both 3 and 5 hidden layers. For both metrics, over 75% of the outputs have less than 1% error.

The QD prediction model is similarly compact, requiring only 32.61Kb of disk space and 26.5Kb total memory. Its execution time for single samples is comparable to QD approximation, likely indicating that Python presents a significant overhead for single samples. Batched execution takes 1.876µs.

As such, the model properties satisfy **Req1** and **Req2** by several orders of magnitude.

B. QD approximation

The error of the QD approximation model for both **DM1** and **DM2** is shown in Fig. 4. Focusing on the 3 hidden layer version (3L), although over 50% of the outputs have only a 0.5% error and 75% of sample outputs have an error of less than 1%, there are some exceptions up to 3% error. However, large errors are without exception the result of low QD values (<10% of maximum QD), for which small absolute errors represent significant relative errors. Additionally, weight initialization and training samples have a significant influence on such a small model, and in a batch of 10 trained models the maximum error ranged from 2% to 5%. As such, accuracy can be made arbitrarily small by training a batch of models. A median error of 0.5% is acceptable as input for the QD prediction component. The results of the 5 hidden layer version (5L) confirm that 3 layers have sufficient modeling potential; barring some of the largest errors for **DM1**, the error rates improve by only 10-20% at the cost of a 60% increase in model size and computation. However, as the single sample execution time does not measurably increase, the 5 layer version can still prove useful.

C. QD prediction

Fig. 5 shows the range of QD for both (So)Swirly and the QD prediction model, for various scenarios. With a normal edge node density, the model performs around 1% better than SoSwirly overall, from lowest QD to highest QD. With a low edge node density, more fog nodes are available to switch to, and the model only improves the high end of QD (i.e. the worst cases), again by around 1%. For high edge

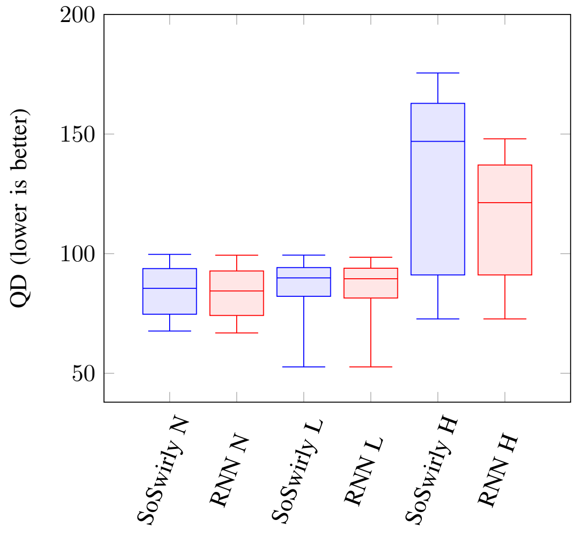

Fig. 5: QD distribution for a sample trip for both SoSwirly (blue) and the proposed RNN (red).

node density however, the model improves the median to highest QD values by up to 18%. While these numbers present an overall improvement, it is important to consider that the evaluation method only switches to a new fog node when maximum QD is exceeded, and at all other times the QD of both approaches is more or less equal.

As such, Fig. 6 shows the range of QD for only those moments when the algorithm decides to switch to a new fog node to improve QD, measured from 2 timesteps before a switch happens to 2 timesteps after. For normal and low edge node densities, the median QD is 12-20% lower when using the QD prediction model, as it accurately predicts which nodes will offer a lower QD in the near future, and allows pre-emptive switching. Maximum QD is not significantly lower, but unlike (So)Swirly, the model never allows it to exceed the limit of 100. For high edge node density, the model offers a significant improvement of 10% lower minimum QD and 16% lower maximum QD. Considering the resource overhead of the models, the results show that they can significantly improve QD at an insignificant computing cost, thus fulfilling **Req3**.

VII. Discussion

Aside from fulfilling the requirements from Section I, the results also illustrate the feasibility of highly accurate (R)NN learning and inferring on edge nodes. On one hand, QD approximation is shown to accurately learn two distance metrics using parameters spaces likely to occur in the edge. On the other hand, the RNN-based QD prediction is shown to accurately predict node movement for several timesteps ahead, allowing an application to avoid exceeding maximum QD. The concrete improvement offered by the models depends on how often an edge node is allowed to switch to other service providers, and the maximum QD. If no limit is imposed, the results of QD prediction are likely to keep the QD well below the maximum at all times. The results of a 20-dimensional

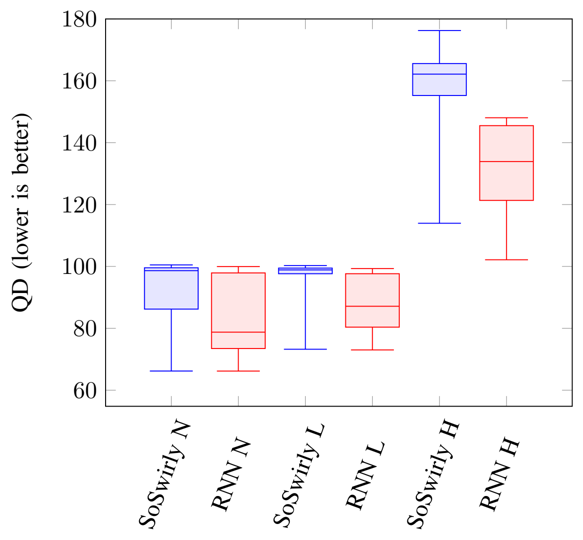

Fig. 6: QD distribution for a sample trip for both SoSwirly (blue) and the proposed RNN (red).

model show that scaling the number of QD parameters is not an issue. In terms of service providers, even if an edge node discovers 50 fog nodes, a final evaluation shows that QD approximation takes around 6ms and QD prediction 12ms with TensorFlow Lite, indicating that real-time operation is possible. The evaluation of a model with 5 hidden layers shows that the choice for 3 hidden layers is optimal when weighing computational cost versus execution time. Several topics for future work remain open, starting with optimization of the presented architecture. The QD prediction model does not currently use regularization or dropout, and could conceivably be both faster and more accurate. Additionally, the current architecture is rigid with respect to timestep look-ahead, as the discounted values are part of the training process. The prediction model would be more versatile if it could predict any number of timesteps ahead without multiplying the training data accordingly. Finally, the overall architecture would benefit from Gossip Learning [17], which is suitable for decentralized dissemination of weight updates.

VIII. Conclusion

This paper presents Qualitative Distance (QD) in relation to QoE, and a decentralized architecture for QoE optimization through QD approximation and prediction in the edge, resulting from a geometric interpretation of neural networks based on metric tensors. In the introduction, some requirements are listed for the architecture to run on resource-constrained edge devices. The theoretical concepts behind the architecture are elaborated and component models are developed, keeping in mind the stated requirements and the resource-constrained nature of edge devices. An evaluation setup is presented, in which data is generated by cars travelling around a road network trip simulator, switching fog nodes as they require better QD. The component models are evaluated in terms of resource requirements, execution time, and their accuracy in approximating and predicting QD. The results indicate that the models fulfill the requirements by a wide margin, providing a significant improvement over default SoSwirly fog node selection. The models require only 35Kb disk space and 27Kb memory combined, which combined with the measured execution times allows running them on devices far less powerful than the evaluation device. In conclusion, the results show that the architecture can accurately approximate and predict QD for at least 50 nearby service providers in real-time. Finally, some topics for future work are listed which can increase the efficiency, flexibility and adoption of the architecture.

References

[1] T. Goethals, F. D. Turck, and B. Volckaert, “Self-organizing fog support services for responsive edge computing,” *Journal of Network and Systems Management*, vol. 29, no. 2, jan 2021.

[2] J. Pennington and Y. Bahri, “Geometry of neural network loss surfaces via random matrix theory,” in *International Conference on Machine Learning*. PMLR, 2017, pp. 2798–2806.

[3] M. Hauser and A. Ray, “Principles of riemannian geometry in neural networks,” *Advances in neural information processing systems*, vol. 30, 2017.

[4] F. Fan, J. Xiong, and G. Wang, “Universal approximation with quadratic deep networks,” *Neural Networks*, vol. 124, pp. 383–392, 2020.

[5] Y. Tong, L. Yu, S. Li, J. Liu, H. Qin, and W. Li, “Polynomial fitting algorithm based on neural network,” *ASP Transactions on Pattern Recognition and Intelligent Systems*, vol. 1, no. 1, pp. 32–39, 2021.

[6] B. Magableh and M. Almiani, “A deep recurrent q network towards self-adapting distributed microservice architecture,” *Software: Practice and Experience*, vol. 50, no. 2, pp. 116–135, nov 2019.

[7] H. Lu, X. He, M. Du, X. Ruan, Y. Sun, and K. Wang, “Edge qoe: Computation offloading with deep reinforcement learning for internet of things,” *IEEE Internet of Things Journal*, vol. 7, no. 10, pp. 9255–9265, 2020.

[8] S. Li, J. Huang, J. Hu, and B. Cheng, “Qoe-deer: A qoe-aware decentralized resource allocation scheme for edge computing,” *IEEE Transactions on Cognitive Communications and Networking*, pp. 1–1, 2021.

[9] Z. Chen and X. Wang, “Decentralized computation offloading for multi-user mobile edge computing: a deep reinforcement learning approach,” *EURASIP Journal on Wireless Communications and Networking*, vol. 2020, no. 1, sep 2020.

[10] M. Mordacchini, L. Ferrucci, E. Carlini, H. Kavalionak, M. Coppola, and P. Dazzi, “Self-organizing energy-minimization placement of qoe-constrained services at the edge,” in *Economics of Grids, Clouds, Systems, and Services*, K. Tserpes, J. Altmann, J. Á. Bañares, O. Agmon Ben-Yehuda, K. Djemame, V. Stankovski, and B. Tuffin, Eds. Cham: Springer International Publishing, 2021, pp. 133–142.

[11] J. Nightingale, P. Salva-Garcia, J. M. A. Calero, and Q. Wang, “5g-qoe: Qoe modelling for ultra-hd video streaming in 5g networks,” *IEEE Transactions on Broadcasting*, vol. 64, no. 2, pp. 621–634, 2018.

[12] A. A. Barakabitze, N. Barman, A. Ahmad, S. Zadtootaghaj, L. Sun, M. G. Martini, and L. Atzori, “Qoe management of multimedia streaming services in future networks: A tutorial and survey,” *IEEE Communications Surveys Tutorials*, vol. 22, no. 1, pp. 526–565, 2020.

[13] K. Potdar, T. S. Pardawala, and C. D. Pai, “A comparative study of categorical variable encoding techniques for neural network classifiers,” *International journal of computer applications*, vol. 175, no. 4, pp. 7–9, 2017.

[14] Y. Tokuyama, Y. Fukushima, and T. Yokohira, “The effect of using attribute information in network traffic prediction with deep learning,” in *2018 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, oct 2018.

[15] B. Jang, M. Kim, G. Harerimana, and J. W. Kim, “Q-learning algorithms: A comprehensive classification and applications,” *IEEE Access*, vol. 7, pp. 133 653–133 667, 2019.

[16] P. T. Yamak, L. Yujian, and P. K. Gadosey, “A comparison between ARIMA, LSTM, and GRU for time series forecasting,” in *Proceedings of the 2019 2nd International Conference on Algorithms, Computing and Artificial Intelligence*. ACM, dec 2019.

[17] I. Hegedűs, G. Danner, and M. Jelasity, “Gossip learning as a decentralized alternative to federated learning,” in *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 2019, pp. 74–90.

Copyright