Significant Sampling for Shortest Path Routing: A Deep Reinforcement Learning Solution

Yulin Shao†, *Student Member, IEEE*, Arman Rezaee†, *Student Member, IEEE*,
Soung Chang Liew, *Fellow IEEE*, Vincent W.S. Chan, *Life Fellow IEEE, Fellow OSA*

***Abstract*—We face a growing ecosystem of applications that produce and consume data at unprecedented rates and with strict latency requirements. Meanwhile, the bursty and unpredictable nature of their traffic can induce highly dynamic environments within networks which endanger their own viability. Unencumbered operation of these applications requires rapid (re)actions by Network Management and Control (NMC) systems which themselves depends on timely collection of network state information. Given the size of today's networks, frequent collection of network states is prohibitively costly for the network transport and computational resources. Thus, judicious sampling of network states is necessary for a cost-effective NMC system. This paper proposes a deep reinforcement learning (DRL) solution that learns the principle of *significant sampling* and effectively balances the need for accurate state information against the cost of sampling. Modeling the problem as a Markov Decision Process, we treat the NMC system as an agent that samples the state of various network elements to make optimal routing decisions. The agent will periodically receive a reward commensurate with the quality of its routing decisions. The decision on when to sample will progressively improve as the agent learns the relationship between the sampling frequency and the reward function. We show that our solution has a comparable performance to the recently published analytical optimal without the need for an explicit knowledge of the traffic model. Furthermore, we show that our solution can adapt to new environments, a feature that has been largely absent in the analytical considerations of the problem.**

I. Introduction

An important feature of any high performance, resilient, and scalable networking platform is its ability to monitor the state of the network and reconfigure network resources accordingly. More importantly, given the bursty and unpredictable nature of traffic in these networks it is paramount for these decisions to be made in real-time and without human interventions. Our recent works focus on the class of shortest path routing algorithms which are designed to identify the shortest routes between pairs of communicating nodes. Existing algorithms such as Dijkstra and Bellman-Ford can identify such paths if the underlying network is static or quasi-static. That is, the proof of their correctness requires the weights associated with network edges to be either fixed or converge to a fixed value.

†Authors have contributed equally to this work.

Y. Shao and S. C. Liew are with the Dept. of Information Engineering, The Chinese University of Hong Kong, Shatin, New Territories, Hong Kong (email: {sy016, soung}@ie.cuhk.edu.hk). A. Rezaee and V. W. S. Chan are with the Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA. (email: {armanr, chan}@mit.edu}).

This work was supported in part by the Innovation and Technology Fund (Project No. ITF/447/16FP) established under the Innovation and Technology Commission of Hong Kong Special Administrative Region, China. It is also sponsored in part by the U.S. National Science Foundation NeTS program, Grant No. #6936827

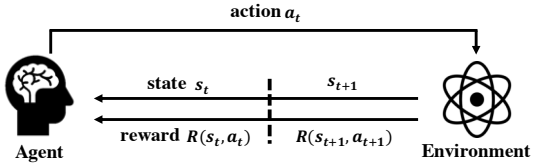

Figure 1: The interactions between agent and environment in reinforcement learning [2].

The first step in identifying shortest paths in stochastic networks is periodic collection of relevant network state information (NSI). Given the costs associated with the collection and dissemination of NSI, we introduced the notion of **significant sampling** in [1]. This technique enables the NMC system to collect NSI only when the information can be of significant value to the optimal operation of the network. Our work in [1] proposed the Ornstein-Uhlenbeck (OU) process as the underlying delay/traffic model and described the optimal sampling policy when the parameters of the OU process are known.

Despite its advantageous analytical nature, the work in [1] suffers from a few drawbacks that we wish to address in this paper. Firstly, the delay model assumed (the OU process) is not the right candidate for all networks and is thus narrow in scope. Secondly, even if OU process is a good approximation, it may be difficult to accurately estimate its parameters which degrades the performance of the system. We aim to address these shortcomings through an automated system that "learns" to collect the necessary NSI and use it to optimize the network performance. The learning agent should be able identify the optimal policy without an explicit knowledge of the traffic model or its parameters. This is achieved by formulating the significant sampling problem as a Markov Decision Process (MDP) which is then solved by a deep reinforcement learning method. We confirm that the performance of this approach is nearly identical to the performance of the examples shown in [1].

Note that reinforcement learning (RL) is a branch of machine learning known for its ability to derive solutions for MDPs [2], [3]. A salient feature of RL is "learning from interactions". As depicted in Fig. 1, the agent interacts with the environment in a sequence of time steps indexed by t. At time t, the environment is in state s_t. Given s_t, the agent takes action a_t, which steers the environment to s_{t+1} in the next time step. The environment will feedback a reward $R(s_t, a_t)$, from which the agent can measure the quality of action a_t at state s_t. A mapping between s_t and a_t is referred to as a policy function. The agent seeks to learn the optimal policy that maximizes the expected cumulative reward.

The latest trend in RL research is to integrate various tools from deep learning [4] into the RL framework [5], [6]. RL that makes use of deep neural networks to approximate the optimal policy function is referred to as deep reinforcement learning (DRL). DRL is often utilized when the set of possible state-action pairs is too large and traditional approaches lead to poor approximations of the policy function.

The main contribution of this paper is twofold:

- First, we propose a DRL solution to the significant sampling problem. This solution follows the actor-critic framework of RL and is model-free, requiring no prior knowledge of the traffic patterns. Assuming the traffic is an OU process, we demonstrate that our DRL solution can learn the optimal sampling policy derived in [1].
- Second, we extend our DRL solution so that it can adapt to non-stationary and changing environments. This improved framework, by re-sampling the past and generating artificial experiences, achieves 15 dB gain over the original DRL solution in terms of convergence time.

The rest of the paper is organized as follows: Section II describes the significant sampling framework and the shortest path routing solution of [1]. Section III describes our DRL formulation of the significant sampling problem and shows that its solution approaches the optimal solution derived in [1]. Section IV extends the DRL solution so that it can adapt to non-stationary and changing environments. A summary and concluding remarks are provided in Section V.

II. Problem Formulation

A. Significant Sampling for Shortest Path Routing

Similar to [1], we consider an Origin-Destination (OD) pair connected via two independent paths P_1 and P_2. The stochastically evolving weights of the two paths are denoted by $X_1(t)$ and $X_2(t)$, and $X(t) = X_1(t) - X_2(t)$ is the difference between the processes. In shortest path routing, the controller has to monitor the weights of both paths in order to identify the shorter one (the path with smaller weight). The process of monitoring the weights can be interpreted as a sampling process, that is, the controller queries the nodes on both paths, asking them to report their weights[1]. We associate a fixed cost c_s with the sampling process, which captures the efforts required to collect and disseminate the state information throughout the network.

As an example, suppose that we sample the weights of both paths at time t_{i-1}. Given these observations, we will route through P_1 if $X_1[t_{i-1}] < X_2[t_{i-1}]$ and route through P_2 otherwise. We will continue to use the same route until the next sampling time t_i, at which point realizations of $X_1[t_i]$ and $X_2[t_i]$ will be used to make future routing decisions. It is clear that identifying the shortest path is equivalent to identifying the sign of the difference process, $X(t)$; this is because $X_1[t_{i-1}] \lessgtr X_2[t_{i-1}]$ if and only if $X[t_{i-1}] \lessgtr 0$. Hence, in the remainder of this paper we will exclusively focus on sampling the difference process.

[1]Usually, weight refers to delay (queuing delay, especially), and it is known to the nodes on both paths.

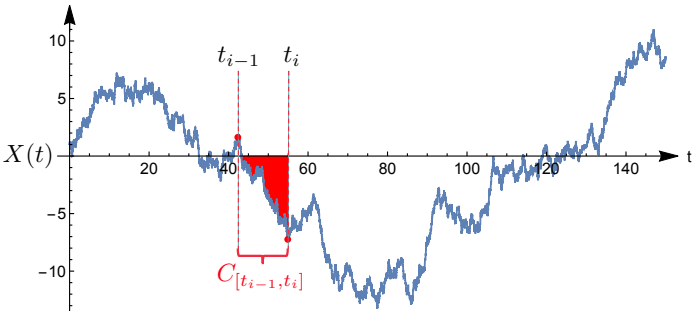

Figure 2: Visual depiction of cost of error, $C_{[t_{i-1},t_i]}$, between two consecutive sampling epochs.

Without loss of generality, let us suppose that $X[t_{i-1}] > 0$. As illustrated in Fig. 2, if the sign of $X(t)$ changes between sampling epochs t_{i-1} and t_i, we will incur an excess cost of

$$C_{[t_{i-1},t_i]} = -\int_{t_{i-1}}^{t_i} X^-(t)\, dt,$$

where $X^-(t)$ denotes the negative part of $X(t)$.

Note that, continuous sampling of the stochastic process can reduce the cost of error to zero, but will result in an extremely high sampling cost. On the other hand, sparse sampling incurs a low sampling cost, at the expense of higher error costs. The optimal tradeoff between sampling cost and cost of error can be described through a policy that specifies the best future sampling times based on the last observed value of $X(t)$. Using x to denote the realization of $X[t_{i-1}]$, we should choose the next sampling epoch $t_i = t_{i-1} + T_i$. More specifically we seek to determine the optimal sampling interval, T_i, as a function of x,

$$T_i^*(x) = \underset{T_i>0}{\operatorname{argmin}} \frac{c_s + \mathbb{E}\left[C_{[t_{i-1},t_{i-1}+T_i]}(x)\right]}{T_i}, \tag{1}$$

where $\mathbb{E}\left[C_{[t_{i-1},t_{i-1}+T_i]}(x)\right]$ is the expected cost of error as a function of x during $[t_{i-1}, t_{i-1} + T_i]$. This is known as the "significant sampling" problem.

B. Our Previous Solution

Our previous work in [1] models $X(t)$ as an Ornstein-Uhlenbeck (OU) process because it is the only non-trivial continuous random process that is simultaneously Gaussian, Markovian, and stationary; properties that are often necessary in obtaining analytical solutions. The OU process is governed by the following stochastic differential equation:

$$dX(t) = \theta(\mu - X(t))\, dt + \sigma\, dW(t)$$

where μ is the long-term mean of the process, $\theta > 0$ is the mean-reversion speed, and σ denotes its volatility. This process consists of two parts. The first part, $\theta(\mu - X(t))\, dt$, captures the mean reverting behavior, which is akin to a force that pulls the process towards its long-term mean. The second part, $\sigma\, dW(t)$, is a standard Wiener process scaled by volatility factor σ. This second part acts as additive noise and counteracts the mean reversion. Hence, the OU process can be roughly described as noisy oscillations around μ.

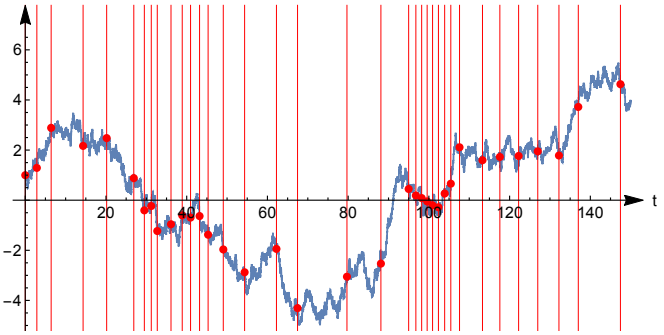

Figure 3: Optimal sampling of an OU process with parameters $\{\mu = 0, \sigma = 0.5, \theta = 0.025\}$, and sampling cost $c_s = 0.1$.

Since the OU process is Markovian and stationary, we can simplify our notation and denote the expected cost of error as $\mathbb{E}[C_T(x)]$, where x is the last observed value of $X(t)$, and T is the sampling interval. In [1] we illustrated that

$$\begin{aligned}\mathbb{E}\big[C_T(x)\big] &= \frac{x\left(\mathrm{e}^{-\theta T}-1\right)}{\theta} \\ &+ \frac{\sigma}{4\theta\sqrt{\theta\pi}}\int_0^{\mathrm{e}^{2\theta T}-1}\frac{\sqrt{y}}{(1+y)^{\frac{3}{2}}}\exp\left(-\frac{\theta x^2}{y\sigma^2}\right)dy \\ &+ \frac{x}{4\theta}\int_0^{\mathrm{e}^{2\theta T}-1}\frac{1}{(1+y)^{\frac{3}{2}}}\operatorname{erfc}\left(-\frac{\sqrt{\theta}x}{\sqrt{y}\sigma}\right)dy,\end{aligned}$$

Given this expression, the optimal policy $T_i^*(x)$ can be computed numerically by solving the non-convex optimization in Eq. (1). Fig. 3 depicts a realization of an OU process with parameters $\{\mu = 0, \sigma = 0.5, \theta = 0.025\}$ as well as the resultant optimal sampling policy when the sampling cost $c_s = 0.1$. As shown, the sampling frequency increases when $X(t)$ is close to zero, (corresponding to times when $X_1(t) \approx X_2(t)$). On the other hand, when $X(t)$ is far from zero (i.e when $X_1(t)$ and $X_2(t)$ are far apart) the algorithm will adopt a sparse sampling strategy.

The following section introduces a deep reinforcement learning (DRL) approach to solve the aforementioned significant sampling problem. The DRL solution makes no assumptions about the $X(t)$ process, and is thus more robust to various network conditions. The importance of the aforementioned analytical solutions is that they allow us to compare the performance of our DRL solution to the theoretically optimal benchmarks when $X(t)$ is an OU process. A satisfying and comparable performance enables us to extend the DRL framework beyond what was presented in [1] with a reasonable degree of confidence in its ability to perform close to optimal.

III. A DRL Solution to Significant Sampling

In this section, we introduce a deep reinforcement learning (DRL) framework to solve the significant sampling problem. We will show that the solution obtained via DRL matches the analytical results that were presented in [1]. This is an important achievement in and of itself because the DRL system is model-free and completely unaware of the underlying delay model, yet it "learns" to properly sample the process.

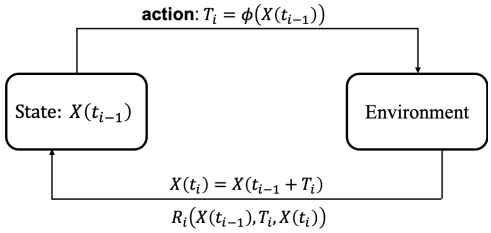

Figure 4: Modeling the significant sampling problem as a reinforcement learning problem.

A. Significant Sampling as an RL Problem

We shall first transform the original significant sampling problem to a reinforcement learning problem. The basic idea is to view the central controller as an agent, and the underlying random process $X(t)$ as the environment. Starting as a novice, the agent performs actions in the environment, and learns from the feedback of the environment. As the agent accumulates additional experiences, it learns to interpret the dynamics of the environment and to obtain the optimal policy that minimizes the cost in Eq. (1).

As shown in Fig. 4[2], the agent maintains a ledger that contains the most recently observed state of the environment, i.e., $X[t_{i-1}]$. Given this observation, the agent's policy ϕ will output an action T_i, which effectively specifies the next epoch for observing/sampling the state of the environment and thus the time for updating the ledger. That is, the ledger will be updated at $t_i = t_{i-1} + T_i$ with $X[t_i]$ and the cycle continues.

The agent's ability to improve its actions depends on a feedback system that can assign values to various actions; this allows the agent to differentiate good actions from bad ones. This feedback is often expressed in terms of a *reward* function that indicates the reward associated with taking a particular action when the environment is in state $X[t_{i-1}]$. Since our goal is to minimize the cost in Eq. (1), the reward function is defined as

$$R_i = -\frac{c_s + C_{[t_{i-1}, t_{i-1}+T_i]}}{T_i}. \tag{2}$$

Note that computing this reward requires full knowledge of $X(t)$ during $[t_{i-1}, t_i]$. To achieve this goal, the realization of the weights (i.e. $X(t)$) should be reported in its entirety during each sampling operation. In other words, at t_i, the agent will be provided with the realization of $X(t)$ for $t \in [t_{i-1}, t_i]$.

We shall refer to a complete cycle from t_{i-1} to t_i as one "time step". During a given time step, the policy function, ϕ, is often referred to as the "actor", as it controls the behavior of the agent; and the reward function, R, is referred to as the "critic", as it determines the value of the action. The learning process requires the execution of many time steps, and each time step gives the agent a new experience. With a growing set of experiences, the agent should be able to approximate the reward function, and learn the optimal policy function.

[2]The RL framework in Fig. 4 should only be used when the environment (i.e. $X(t)$) is Markovian. This is due to the fact that the ledger only tracks $X[t_{i-1}]$, and not $X(t)$ for all $t < t_{i-1}$. For non-Markovian processes in which the future trajectory of the process depends on its past, the ledger should maintain a longer history of $X(t)$. We leave this extension to future work.

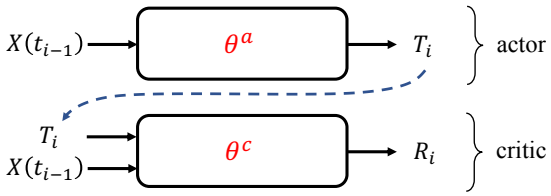

Figure 5: The logical interaction of the actor and critic networks in actor-critic RL [7].

B. An Actor-Critic Solution

The solution of actor-critic RL [2], [8] fits nicely into the above RL paradigm. Specifically, we can leverage two DNNs, an actor network and a critic network, to approximate the policy function and the reward function, respectively.

Fig. 5 depicts the logical interaction of the actor and critic networks [7]. The actor network, parameterized by θ^a, will output an action T_i when the network state $X[t_{i-1}]$ is used as its input, i.e. $T_i = \Psi(X[t_{i-1}]|\theta^a)$. Similarly, the critic network, parameterized by θ^c, will output a reward estimate R_i when the state-action pair, $\{X[t_{i-1}], T_i\}$, is provided as its input, i.e. $R_i = \Psi(X[t_{i-1}], T_i|\theta^c)$. Both DNNs are initialized randomly. As learning continues, we use the accumulated experiences to train both DNNs, such that the actor network approximates the optimal policy function, and the critic approximates the real reward function.

The pseudocode for this actor-critic solution to the significant sampling problem is given in Algo. 1. As shown, we start by initializing the actor and critic neural networks with parameters θ^a and θ^c respectively. Additionally, a FIFO buffer of size M is initialized for experience replay [5]. During the RL process, the accumulated experiences will be placed into and removed from this FIFO continuously. The initial sampling time is set to $t_0 = 0$, where $X[t_0]$ is given.

Algo. 1: The DRL solution to significant sampling.

Initialization:
Initialize a FIFO of size M, the actor network with parameter θ^a, and the critic network with parameter θ^c.
Set mini-batch size to K, evaluation cycle to Q, and time step $i = 1$, training step $j = 1$.
Set $t_0 = 0$, ask nodes on both paths to report their weights, and compute $X[t_0] = X_1[t_0] - X_2[t_0]$.
while 1 **do**
  Feed $X[t_{i-1}]$ into the actor network.
  Obtain $T_i = \Psi\left(X[t_{i-1}]\middle|\theta^a\right)$ from actor network.
  In epoch $t_i = t_{i-1} + T_i$:
    Ask nodes to report their weights in interval $[t_{i-1}, t_i]$.
    Compute $X(t) = X_1(t) - X_2(t), \forall t \in [t_{i-1}, t_i]$
    Compute reward $R_i = -(c + C_i)/T_i$
  Store experience $(X[t_{i-1}], T_i, R_i)$ into the FIFO.
  $i = i + 1$
  if $i > K \times 10$ **then**
    Sample a mini-batch of K experiences from FIFO.
    Update parameters θ^c (critic) using Eq. (3).
    Update parameters θ^a (actor) using Eq. (4).
    if $\text{mod}(j, Q) == 0$ **then**
      Assess the current policy (actor) network.
    end
    $j = j + 1$.
  end
end

The next phase is to "Gain Experience". At the beginning of a time step (i.e., at t_{i-1}), the agent feeds the current observation $X[t_{i-1}]$ into the actor network, which produces the output action T_i. We emphasize that this action is bad early on, but will improve as learning progresses. At the end of the time step (i.e., at t_i), the agent computes a reward R_i based on the feedback from the environment. Overall, one time step provides the agent with one experience, expressed as the triplet $\{X[t_{i-1}], T_i, R_i\}$, denoting the state, action, and reward. All experiences will be stored in the FIFO for experience replay.

When the number of experiences exceeds $10K$ (K is the mini-batch size set to 128), we will train the DNNs in the following manner [7]:

1) Randomly sample a mini-batch of K experiences $\{X[t_{k-1}], T_k, R_k\}$, $k \in \{i_1, i_2, \cdots, i_K\}$ from the FIFO.
2) Train the critic network by minimizing the mean square error (MSE) loss
$$\mathcal{L}_c = \frac{1}{K} \sum_{k \in \{i_1, \ldots, i_K\}} \left[\Psi\left(X[t_{k-1}], T_k\middle|\theta^c\right) - R_k\right]^2, \tag{3}$$
That is, for each experience, $\{X[t_{k-1}], T_k, R_k\}$, we feed the state-action pair, $\{X[t_{k-1}], T_k\}$, into the critic network and update parameters θ^c in such a way as to make the output of the critic network closely match the real reward R_k. This allows the critic network to approximate the reward function evermore closely, resulting in more accurate policy evaluations.
3) For state $X[t_{k-1}]$, we train the actor network to maximize the expected output of the critic network $\overline{R}$ (i.e., the expected reward) by sampled policy gradient. Specifically, the parameter θ^a will be updated in the gradient direction given by Eq. (4). This is a process of policy improvement, that is, we update the parameters θ^a in the direction that maximizes the expected output of the critic network for state $X[t_{k-1}]$.

$$\begin{aligned} \nabla_{\theta^a} \overline{R} &\approx \frac{1}{K} \sum_{k \in \{i_1, \ldots, i_K\}} \nabla_{\theta^a} \Psi\left(X[t_{k-1}], T\middle|\theta^c\right)\Big|_{T=\Psi\left(X[t_{k-1}]\middle|\theta^a\right)} \\ &= \frac{1}{K} \sum_{k \in \{i_1, \ldots, i_K\}} \nabla_T \Psi\left(X[t_{k-1}], T\middle|\theta^c\right)\Big|_{T=\Psi\left(X[t_{k-1}]\middle|\theta^a\right)} \\ &\quad \times \nabla_{\theta^a} \Psi\left(X[t_{k-1}]\middle|\theta^a\right) \end{aligned} \tag{4}$$

To monitor and illustrate the steady improvements of the DRL solution, we evaluate the performance of the actor network every Q training steps. The evaluation is performed on an independent trajectory of $X(t)$ (the evaluation set is completely independent from the sample path of the process that were observed during the training process).

C. Experiments

This subsection presents some experimental results to evaluate the performance of the actor-critic RL solution to the significant sampling problem. Let us start by assuming that the underlying process $X(t)$ is an OU process as in [1]. The optimal sampling policy derived in [1] will serve as a benchmark. We will confirm that, with no prior information on $X(t)$, the agent can learn the optimal policy using the actor-critic RL framework.

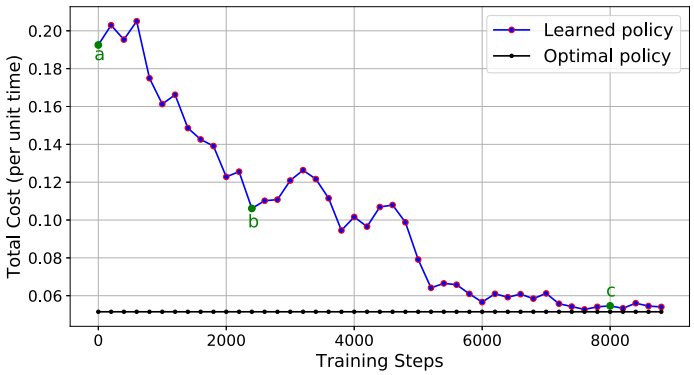

Figure 6: Total cost versus training steps in the RL process. The learned policies at points a, b, and c are plotted in Fig. 7.

In the experiment, the actor and critic networks are designed to be fully connected NNs. The actor network consists of five fully connected layers, and the number of neurons in each layer is $\{1, 32, 64, 32, 1\}$. The three hidden layers use rectifier non-linearity as activation functions, and the output layer uses hyperbolic tangent (tanh) non-linearity as activation function. The output of the actor network g takes value in $(-1, 1)$, corresponding to the sampling interval $T_i \in (0, T_{\max})$, where $T_{\max}$ is the maximum sampling interval (set to $T_{\max} = 40$). In other words, the next sampling interval $T_i = \frac{g+1}{2} \times T_{\max}$. The architecture of the critic network is the same as the actor network except for i) the input layer has two neurons instead of one; ii) the output layer uses no activation function. The output of the critic network is exactly the reward R_i (See Fig. 5 for additional reference).

As per hyper-parameters, we set the buffer size $M = 10000$, the mini-batch size $K = 128$, and the evaluation cycle $Q = 200$. We use Adam optimizer for both DNNs, the learning rates of actor and critic are set to 0.0005 and 0.001, respectively. As in [1], the sampling cost is set to $c_s = 0.1$, and we choose parameter set $\{\mu = 0, \sigma = 0.5, \theta = 0.025\}$ for the OU process $X(t)$. Given these parameters, we generate two independent trajectories of $X(t)$ for training and evaluation purposes with durations 2×10^5 and 2×10^4 time units, respectively.

Over the course of training, for every $Q = 200$ training steps, we assess the actor network on the evaluation trajectory. Specifically, we use the policy learned by the actor network on the evaluation trajectory, and record the total cost (sampling cost plus additional cost incurred by misrouting, i.e. cost of error). Fig. 6 presents the total cost (per unit time) as a function of training steps. As a comparative benchmark we have also plotted the minimum achievable cost (0.05); this is obtained by applying the optimal sampling policy given by Eq. (1) on the evaluation trajectory. We can see that the total cost of the learned policy decreases as training continues and approaches the optimal cost.

Figures 7(a-c) present the sampling policies of the actor network at different times during the learning process. The x-axis is the observation $X[t_{i-1}]$ (the range corresponds to two standard deviations from the mean), and the y-axis is the next sampling interval T_i. The benchmark is the analytically derived

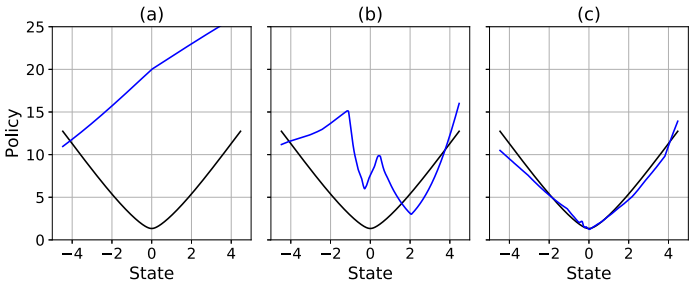

Figure 7: Policy improvements in the RL process. The blue curves in (a), (b), and (c) are the learned policies at points a, b, and c in Fig. 6. The black curves are the optimal policy.

optimal policy (its symmetry with respect to $X[t_{i-1}] = 0$ is because $\mu = 0$). It can be seen that the actor network learns the optimal policy after enough training steps.

IV. Environment Adaptation: Significant Sampling with Artificial Experiences

Realistic and robust communication networks should be able handle and smoothly transition through various operating regimes. Similarly, ideal monitoring systems should be able to identify and adapt to new conditions. In this section we will show that our DRL solution can automatically adjust the sampling policy and adapt to new environments in a reasonable amount of time.

There are two important indicators when we consider the performance of a controller in a non-stationary environment. First, we should ensure that the system can operate optimally in a wide range of environments, i.e. the system architecture is flexible enough to accommodate various network conditions. Second, we prefer a controller that can quickly transform its policy based on its perception of the environment. Unfortunately, these two goals are often at odds with one another. Said another way, an agent's ability to identify the optimal action depends on its ability to maintain and analyze past behaviors, on the other hand, adapting to a new environment requires the agent to "forget" the past states and "relearn" from the latest observations, and thus there is a clear tradeoff between optimality and adaptability.

A. Significant Sampling with Artificial Experiences

Let us define the phrase "transitional period" to denote the amount of time it takes for a given algorithm to adapt to a new environment. There are many ways to make this definition rigorous, for instance the amount of time it takes for the achieved cost to reach within 10% of the optimal cost, etc.

Without specifying a particular definition, note that the significant sampling framework in Algo. 1 does not lend itself to a short transitional period because of its inefficient experience collection process. For instance, in Algo. 1, one time step lasts for T_i time units, but it gives the agent (central controller) only one experience, hence is quite inefficient. This motivates us to ask whether there are other ways for the agent to generate more experiences, in addition to the regularly collected ones?

We answer the aforementioned question affirmatively by noting that in each time step, the continuously evolving weights $X(t)$ in interval $[t_{i-1}, t_i]$ are collected and reported to the central controller for reward calculation. As a result, at epoch t_i, all past weights $X(t)$, $t \in [0, t_i]$ are in fact known to the central controller. Hence the agent/central controller can re-sample the past to generate *artificial experiences*.

Algo. 2: Significant sampling with artificial experiences.

Initialization:
Initialize a FIFO of size M, the actor network with parameter θ^a, and the critic network with parameter θ^c.
Set mini-batch size to K, evaluation cycle to Q, and time step $i = 1$, training step $j = 1$.
Set $t_0 = 0$, ask nodes on both paths to report their weights, and compute $X[t_0] = X_1[t_0] - X_2[t_0]$.
Set resampling window to t_w, the number of artificial experiences to V, and parameter N.
while 1 **do**
  Feed $X[t_{i-1}]$ into the actor network,
  Obtain $T_i = \Psi\left(X[t_{i-1}]|\theta^a\right)$ from actor network,
  In epoch $t_i = t_{i-1} + T_i$, ask nodes to report their weights $\forall t \in [t_{i-1}, t_i]$, and compute reward $R_i = -(c_s + C_i)/T_i$.
  Store experience $(X[t_{i-1}], T_i, R_i)$ into FIFO, $i = i + 1$
  $v = 1$.
  while $v \leq V$ **do**
    Select a random starting epoch t_{v-1} in $[t_i - t_w, t_i]$,
    Sample value $X[t_{v-1}]$.
    Feed $X[t_{v-1}]$ into the actor network
    Obtain $T_v = \Psi\left(X[t_{v-1}]|\theta^a\right)$ from actor network,
    if $t_{v-1} + T_v < t_i$ **then**
      Compute reward $R_v = -(c_s + C_v)/T_v$,
      Store experience $(X(t_{v-1}), T_v, R_v)$ into the FIFO, $i = i + 1$, $v = v + 1$.
    end
  end
  if $i > M/2$ **then**
    Sample N mini-batches from FIFO.
    For each mini-batch:
      update parameters θ^c (critic) using Eq. (3).
      update parameters θ^a (actor) using Eq. (4).
    if mod$(j, Q) == 0$ **then**
      Assess the current policy (actor) network.
    end
    $j = j + N$.
  end
end

Algo. 2 presents an improved framework for significant sampling. This algorithm has two major differences compared with Algo. 1: i) In each time step, the agent not only collects the experience $\{X(t_{i-1}), T_i, R_i\}$, but also generates V artificial experiences from the near past $[t_i - t_w, t_i]$, where t_w is a resampling window. Specifically, each time an experience is collected at t_i, the agent will randomly select V starting epochs in period $[t_i - t_w, t_i]$. For each starting epoch t_{v-1}, the agent will sample $X[t_{v-1}]$, feed it into the current actor network for action T_v, and then calculate the reward R_v, which constitutes an artificial experience $\{X[t_{v-1}], T_v, R_v\}$. This allows the agent to collect V more experiences in each time step. ii) Since one time step gives the agent many more experiences, we can now speed up the training of the two networks. In Algo. 2, when the number of experiences is more than $M/2$ (M is the FIFO size), after each time step, we sample N mini-batches instead of one to train the actor and critic networks.

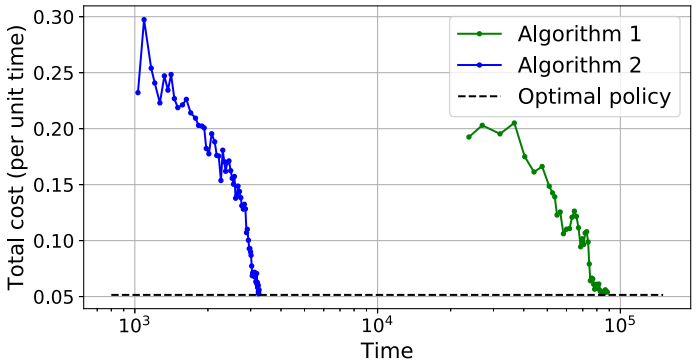

Figure 8: The total costs of Algorithms 1 and 2 over time.

B. Experiments

Given Algo. 2, we evaluate the agent's agility in adapting to new environments in this subsection.

1) Compare Algorithm 1 and 2: To see that Algo. 2 has a significantly superior experience collection method, we first repeat the experiments in Section III using Algo. 2. This time, our focus is on comparing the evolution of cost over time. The "time" here refers to the real time rather than time steps.

In this repeated experiment, we use the same training and evaluation data, the same hyper-parameter settings and DNN designs as that in Section III. For resampling, we set $t_w = 900$ and $V = 1800$: in each time step, the agent will first collect one experience by moving forward and then resample 1800 experiences from the past 900 time units. Since Algo. 2 can generate more experiences than Algo. 1, we increase the buffer size from $M = 10000$ to $M = 10(V + 1)$ (it contains experiences collected in the last 10 time steps). In each training step, we sample $N = 100$ mini-batches to train the DNNs.

During the learning process, for every $Q = 200$ training steps, we evaluate the current actor network on the evaluation trajectory and output the total cost per unit time. The cost versus time curves are shown in Fig. 8. Let us first focus on the curve corresponding to the performance of Algo. 1. This curve is indeed the same curve as that of Fig. 6, with the caveat that the x-axis has been changed to time (instead of time-steps). For Algo. 1, the first training starts at $t = 26000$, because the agent needs to collect $10K = 1280$ experiences before it starts to train the DNNs. The training ends at around $t = 85000$, and the whole training process takes 59000 time units. On the other hand, the first training of Algo. 2 starts at $t = 1300$, because the agent has already collected $M/2 = 9000$ experiences. The training ends at around $t = 3200$, and the whole training process takes only 1900 time units. Comparatively speaking, Algo. 2 learns the optimal sampling policy faster than Algo. 1 by a factor of $59000/1900 \approx 31$ (15 dB). This performance gain should not be surprising because in each time step Algo. 2 generates 1800 additional (artificial) experiences.

This raises an interesting question: given the performance improvements resulting from artificial experiences, would there be a limit to these gains if we choose to generate many more experiences in the latest $t_w = 900$ time units? The answer is indeed yes. To understand why note that given a constant

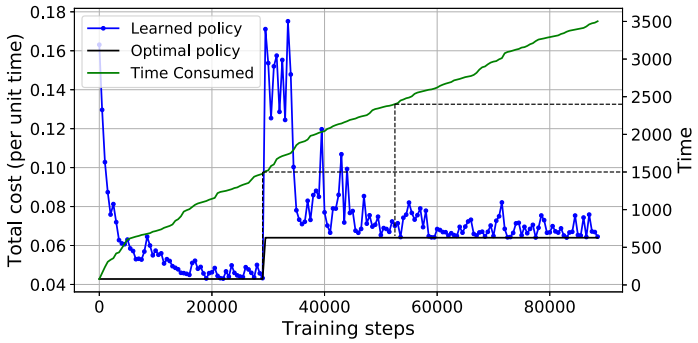

Figure 9: Total cost and required time versus training steps in the RL process.

resampling window t_w, the artificially generated experiences are in fact correlated. As a result, we will be learning from a set of highly correlated experiences which often leads to over-fitting of the DNNs, or may even cause a divergence in the learning process. Our experimental results also confirm that further increases in V does not result in significant improvements.

2) Environment adaptation: Given Algo. 2, let us evaluate the performance of the DRL solution when the underlying process experiences a sudden change.

Training data was generated in two steps: the first parameter set, $\{\mu = 1, \sigma = 0.5, \theta = 0.025\}$, was used to get a sample path of an OU process of duration $D_1 = 1500$ and the second parameter set, $\{\mu = 0, \sigma = 1, \theta = 0.025\}$, was used to generate an independent OU process of duration $D_2 = 2000$. The training data is obtained by concatenating these two sample paths. To evaluate the performance of the actor network fairly, we should employ different evaluation data for different periods of the learning process. Specifically, in the first $D_1 = 1500$ time units, we use an evaluation trajectory generated by the first parameter set, and in the remaining $D_2 = 2000$ time units, we use another evaluation trajectory generated by the second parameter set. Both evaluation trajectories have a duration of 20000 time units. For resampling, we set $t_w = 900$, $V = 1800$, $M = 10(V + 1)$, $N = 100$, and $c_s = 0.1$.

Over the course of training, for every $Q = 500$ training steps, we assess the actor network on the evaluation trajectory. Fig. 9 presents the total cost per unit time and time versus training step curves. Let us first focus on the required time curve (the second y-axis on the right). The agent starts to collect and resample experiences from $t = 0$. The first period is from $t = 0$ to $t = 1500$ (i.e., training steps 0 to 29000), where the agent learns to adapt to the OU process generated by the first parameter set. The second period is from $t = 1500$ to $t = 3500$ (i.e., training steps 29500 to the end), where the agent learns to adapt to the OU process generated by the second parameter set.

In the first period, the actor network takes approximately 15000 training steps (1100 time units) to approach the optimal cost (the cost yielded by the optimal policy). As training progresses, the realized cost oscillates near the optimal cost (the cause of this oscillatory behavior will be discussed shortly). Then, at $t = 1500$, the underlying process changes abruptly. In particular, the σ of the process is doubled, hence a more frequent sampling policy is needed. Clearly, the policy learned during the first period performs poorly when applied to the second process. Moreover, the poor performance persists for several hundred time units. This is because in interval $t \in [1500, 1500 + t_w]$, the agent will still resample experiences from the first period. These bad experiences will contribute negatively to the training of DNNs and hence prolong the transitional period. Overall, various experiments have shown that the transitional period is nearly equal to t_w. After $t = 1500 + t_w$, the system has nearly reached the optimal cost and will oscillates near this value.

The oscillatory behavior observed in both periods are related to the resampling window t_w. Recall that in the framework of Algo. 2, each time when we train the DNNs, the experiences contained in the buffer are collected from the latest 10 time steps (because the buffer size $M = 10V$). Thus, the experiences are selected from the last $(t_w + \sum_{l=0}^{9} T_{i-l}) \approx 1100$ time units. This interval is quite small and if the underlying process experiences an atypical behavior in this small interval, the DNN will be trained in the wrong direction, causing oscillations on the total cost. A simple solution to alleviate the oscillations is to increase the resampling window, or increase the buffer size. However, both solutions can further prolong the transitional period. As discussed earlier there is a fundamental tradeoff between optimality and quick adaptability.

V. Conclusion

In this paper, we propose a deep reinforcement learning solution to the significant sampling problem introduced in [1]. This model-free approach enables the NMC system to collect network state information when they are of significant value to optimal operation of the network. We have shown the feasibility of this approach without any restrictions or assumptions on the underlying nature of the traffic, and demonstrated that when the traffic is an OU process, our DRL solution matches the optimal sampling policy derived in [1]. In addition, we demonstrated that our DRL solution is capable of adapting to non-stationary environments; a desirable and powerful feature that has been out of reach in many previous attempts on this problem. We also introduced the idea of artificial experiences to speed up the learning process and showed that in one instance this technique can improve the convergence time of the algorithm by approximately 15 dB.

References

[1] A. Rezaee and V. Chan, "Cognitive network management and control with significantly reduced state sensing," in *Global Telecommunications Conference (GLOBECOM 2018)*, pp. 1–7, IEEE, 2018.

[2] R. Sutton, A. Barto, and F. Bach, *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning series, MIT Press, 2018.

[3] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

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

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[6] Y. Li, "Deep reinforcement learning: An overview," *arXiv preprint arXiv:1701.07274*, 2017.

[7] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[8] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *ICML*, 2014.