DeepViNE: Virtual Network Embedding with Deep Reinforcement Learning

Mahdi Dolati*†, Seyedeh Bahereh Hassanpour*, Majid Ghaderi† and Ahmad Khonsari*
* University of Tehran, Tehran, Iran; Email: {mahdidolati, b.hassanpour, a_khonsari}@ut.ac.ir
† University of Calgary, Calgary, Canada; Email: {mahdi.dolati, mghaderi}@ucalgary.ca

***Abstract*—Virtual Network Embedding (VNE) is a crucial problem in network virtualization. Prior work on VNE is mainly focused on optimization-based solutions that are carefully constructed and tuned under specific assumptions about resource demands brought by virtual networks. Recently, a few works have appeared on automating the design of VNE solutions that work well under general virtual resource demands using Deep Reinforcement Learning (DRL). These works, however, still rely on manual selection of relevant problem features required in the DRL approach. In this work, we develop a DRL-based VNE solution called DeepViNE, which automates the selection of problem features required in the DRL approach. The key idea is to encode physical and virtual networks as two-dimensional images, which are then perceivable by a convolutional deep neural network. To speed up learning and algorithm convergence, we also design a strategy to limit the number of actions required by the learning agent, while still allowing suitable exploration of the solution space. We evaluate the convergence and performance of DeepViNE using simulations, and compare it with several existing algorithms. The results show that DeepViNE learns an embedding policy that improves upon the performance of other simulated algorithms by at least 11%.**

I. Introduction

A. Background and Motivation

Network virtualization has emerged as one of the key technologies of future networks. In its general form, network virtualization enables multiple virtual networks (VNs) to co-exist on the same physical network (PN) through specific abstraction and isolation mechanisms. A VN is represented by a set of virtual nodes and virtual links. Virtual nodes and links require specific amount of resources, *i.e.*, processing power and bandwidth, which depend on the services provided by the corresponding VNs. The problem of mapping virtual resource requirements to physical resources is known as the virtual network embedding (VNE) problem.

The VNE problem is NP-hard and has been the subject of extensive research (see [1] for a comprehensive survey on the topic). The vast majority of works on VNE consider an *offline* setting, in which a given set of VN requests with pre-specified resource demands is embedded at once in the physical network. In real-word, however, requests for VNs arrive sequentially over time, where neither the exact timing of the requests nor their resource demands are known a priori. As such, VN embedding decisions have to be made at runtime as VN requests arrive. Once a VN is embedded, the embedding must not change in the future, as any change in the virtual to physical resource mapping at runtime triggers virtual machine migration and traffic rerouting causing significant disruptions to the VN operation. Thus, any solution for VNE has to operate in an *online* fashion. Most works on online VNE algorithms propose heuristics (*e.g.*, BestFit [2], FirstFit [3], and node ranking [4]) that do not provide any performance guarantee, which means that they may perform arbitrarily bad for certain VN arrival patterns. There are only a few works that propose algorithms with performance guarantees (see [5], [6]). Nevertheless, the worst-case performance of these algorithms, as characterized by their competitive ratios, can be significantly far from optimal. A major problem with such algorithms is their non-adaptive nature, where a pre-determined policy is executed regardless of the outcomes of previous embedding decisions.

Our objective in this work is to design an adaptive online VNE algorithm that learns to make better embedding decisions at runtime. Given the centralized structure of the VNE problem, and the repetitive nature of virtual resource mapping decisions, our hypothesis is that machine learning (ML) could be applied to solve the problem. Recently, due to simultaneous availability of large datasets and cheap processing, ML has been successfully applied to many complex practical problems, including a variety of problems in the networking area (see [7] and references therein). There also exist a few recent works on the application of ML to VNE (*e.g.*, [8], [9]). However, these works rely on a human expert to build a set of hand-crafted *features*, on which the behavior of the learning algorithm is heavily dependent. In contrast, our goal is to design an algorithm that automatically selects problem features with no manual intervention. We argue that automatic feature selection not only facilitates the solution design process, but also leads to better virtual network embedding decisions. Specifically, we formulate adaptive online VNE as a deep reinforcement learning (DRL) [10] problem, and design an algorithm to learn the optimal VNE algorithm autonomously. By using a convolutional deep neural network (DNN), our approach has the ability to automatically extract suitable problem features independent of the operating environment and without relying on a human expert.

B. Related Work

The works presented in [11]–[13] use ML to predict future resource demands and dynamically adjust the allocated resources to improve the utilization of physical resources. Specifically, in [11], the state of each allocated virtual node and virtual link is represented by three features (*e.g.*, percentage of unused physical resource), and then a look-up table is

used to design a learning algorithm. In a subsequent work [12], the look-up table is replaced with a Feed Forward Neural Network (FFNN). A Radial Basis Function (RBF) Neural Network is used in [13] to learn future demand fluctuations from historical records, but only the current demand of a VN is considered.

The works presented in [8], [9], [14], [15] directly apply ML to solve the VNE problem. In [14], Recurrent Neural Networks (RNNs) are used to design an admission control mechanism for the VNE problem that filters out the VN requests with unsatisfiable demands, which reduces the time it takes to compute an embedding for the feasible demands. In this work, 21 graph features and 8 resource features are used to represent the physical network, while 7 graph features are used to represent a VN. Using 4 features (*e.g.*, CPU capacity) for each physical node, an FFNN is used in [9] as the policy approximator, but no feature for VNs is considered in the learning algorithm. In another work [15], the same set of features as in [9] are used, but the spectrum method is applied to combine those features with the adjacency matrix of the network to obtain a state representation for use by the learning algorithm. Using residual CPU capacities and shortest paths between nodes to, respectively, compute a node and edge ranking, a Hopfield Network is employed in [8] to pre-process the problem and select a subset of physical nodes to serve VN requests, thereby reducing the search space for any optimization-based embedding algorithm (*e.g.*, the one proposed in [4]).

As mentioned earlier, these works rely on a human expert to build a set of hand-crafted features, while our goal is to design an algorithm that selects suitable features autonomously.

C. Our Work

We design an adaptive online VNE algorithm based on DRL. Specifically, we address two challenging problems when applying DRL to VNE. First, the convolutional layer of DNNs, with its origin in image feature extraction tasks, assumes that the inputs are organized as two-dimensional arrays [16], *i.e.*, they are images. With spatial locality being an important property of image data, we address the problem of how the VNE problem inputs, usually weighted graphs representing the physical network and VNs, can be encoded in order to be properly perceived by a DNN. Second, the DRL approach relies on exploring all available actions sufficiently enough in order to compute a policy close to the optimal. However, since the number of ways a VN can be embedded in the physical network is combinatorial, the DRL approach potentially requires a huge number of actions to derive an appropriate solution, which could lead to prohibitively long convergence time for the learning process [17]. Thus, we address the problem of how to shrink the action space of the VNE problem in order to provide sufficient flexibility for exploring different VN mappings, and yet retain the efficiency of the learning process.

Our main contributions can be summarized as follows:

- We develop an encoding method to present the state of the VNE problem as an image to feed into the convolutional layer of a DNN for automatic feature extraction.
- We design a DRL algorithm that uses only a constant number of actions (independent of the size of PN and VNs) to solve the VNE problem, which significantly improves the efficiency of the learning process.
- We define a suitable reward signal so that the learning algorithm achieves the long-term goal of minimizing the probability that an arriving VN can not be embedded.
- We provide simulation results to show the convergence and performance of our algorithm, and compare it with well-known existing VNE algorithms.

D. Paper Organization

The paper is organized as follows. We formally define the problem in Section II. We briefly explain the DRL framework and present the design of our algorithm in Section III. Performance evaluation results are presented in Section IV. Section V concludes the paper.

II. Problem Definition

In this section, we describe our network and resource models and formally define the VNE problem considered in the rest of this paper.

A. Physical Network

The physical network is modelled as an undirected graph $G^p = (\mathcal{N}^p, \mathcal{L}^p)$, where $\mathcal{N}^p = \{n_1^p, \dots, n_K^p\}$ and $\mathcal{L}^p = \{(n_i^p, n_j^p) | i \neq j\}$ denote the set of physical nodes (*i.e.*, servers) and physical links, respectively, and K denotes the number of physical nodes. Each physical node n_i^p has the computing capacity CPU(n_i^p) and each physical link $\ell_{i,j}^p = (n_i^p, n_j^p)$ has the bandwidth capacity BW($\ell_{i,j}^p$). We use real numbers to represent physical CPU and BW capacities with respect to a base unit.

B. Virtual Network

The virtual network v is represented by an undirected graph $G^v = (\mathcal{N}^v, \mathcal{L}^v)$, where $\mathcal{N}^v = \{n_1^v, \dots, n_M^v\}$ and $\mathcal{L}^v = \{(n_i^v, n_j^v) | i \neq j\}$ denote the set of virtual nodes and virtual links, respectively, and M denotes the number of virtual nodes. Each virtual node n_i^v can be considered as a virtual machine with computing demand CPU(n_i^v). Also, each virtual link $\ell_{i,j}^v = (n_i^v, n_j^v)$ has bandwidth demand BW($\ell_{i,j}^v$). Similarly, we use real numbers to represent virtual CPU and BW demands with respect to the base unit.

C. VNE Problem

The VNE problem involves computing a mapping from the virtual nodes and links to the physical nodes and paths with sufficient resource capacities. Assume that the current available capacity of physical node n_i^p and link $\ell_{i,j}^p$ are, respectively, $\overline{\text{CPU}}(n_i^p)$ and $\overline{\text{BW}}(\ell_{i,j}^p)$. Let $f(n_i^v)$ denote the physical node that embeds the virtual node n_i^v, and $g(\ell_{i,j}^v)$ denote the physical path between physical nodes $f(n_i^v)$ and $f(n_j^v)$ that embeds the virtual link $\ell_{i,j}^v$. A VN is embedded if

there are mapping functions f and g that satisfy the following constraints:

$$\overline{\text{CPU}}(n_i^p) \geq \sum_{j:f(n_j^v)=n_i^p} \text{CPU}(n_j^v), \tag{1}$$

$$\overline{\text{BW}}(\ell_{i,j}^p) \geq \sum_{(t,z):\ell_{i,j}^p \in g(\ell_{t,z}^v)} \text{BW}(\ell_{t,z}^v), \tag{2}$$

where, $\ell_{i,j}^p \in g(\ell_{t,z}^v)$ indicates that the path $g(\ell_{t,z}^v)$ includes the physical link $\ell_{i,j}^p$. If any of these constraints is violated, the embedding fails and the corresponding VN is blocked. In this work, the objective of the VNE algorithm is to *minimize the VN blocking probability* so that the physical network can serve the most number of VNs with its limited resource. This is achieved by learning to perform resource mappings efficiently so as to avoid wasting physical resources due to sub-optimal embedding decisions. Let $\mathcal{V}$ denote the set of all embedded VNs. The corresponding *Revenue* and *Cost* of the algorithm are then defined as follows:

$$\text{Revenue} = \sum_{v\in\mathcal{V}} \sum_{n_i^v \in \mathcal{N}^v} \text{CPU}(n_i^v) + \xi_r \sum_{\ell_{i,j}^v \in \mathcal{L}^v} \text{BW}(\ell_{i,j}^v), \tag{3}$$

$$\begin{aligned}\text{Cost} = &\sum_{n_i^p \in \mathcal{N}^p} \Big(\text{CPU}(n_i^p) - \overline{\text{CPU}}(n_i^p)\Big) \\ &+ \xi_c \sum_{\ell_{i,j}^p \in \mathcal{L}^p} \Big(\text{BW}(\ell_{i,j}^p) - \overline{\text{BW}}(\ell_{i,j}^p)\Big),\end{aligned} \tag{4}$$

where, $\xi_r \geq 0$ and $\xi_c \geq 0$ can be used to adjust the revenue and cost of BW relative to the revenue and cost of CPU.

III. Algorithm Design

A. Overview

Reinforcement learning (RL) deals with autonomous agents that learn to make better decisions by interacting with the environment. In reinforcement learning, the agent is given a choice of actions to take in each decision epoch, which changes the state of the environment in an unknown fashion, and receives feedback based on the consequence of the action. The feedback is typically given in the form of a reward or cost, and the objective of the agent is to choose a sequence of actions based on observations of the current environment that maximizes cumulative reward or minimizes cumulative cost. Under this framework, a VNE algorithm simply corresponds to a policy, which governs transitions between the states. Learning an optimal VNE algorithm then reduces to finding an optimal policy, which can be solved using any RL algorithm.

A popular RL algorithm is the *Q-learning* algorithm that learns a value $Q(s,a)$, called Q-value, for each state-action pair (s,a), which is equal to the maximum accumulated future reward when action a is performed in state s. The learning is based on the Bellman equation that estimates better Q-values when a new state transition and its corresponding reward are observed. Specifically, assume that in the decision epoch t, the action a_t is performed which takes the environment from state s_t to another state s_{t+1}, and results in receiving the reward r_t. Then, $Q(s_t,a_t)$ is updated as follows:

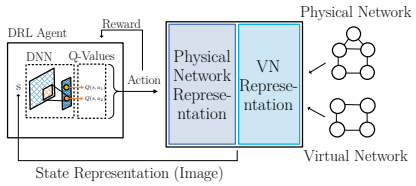


Fig. 1: A DNN approximates Q-values in DeepViNE.

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\Big(r_t + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)\Big), \tag{5}$$

where, $\alpha < 1$ and $\gamma < 1$ are called learning rate and discount factor, respectively. Observe that, the term $r_t + \gamma \max_a Q(s_{t+1},a)$ is the new estimate for $Q(s_t,a_t)$. Thus, the difference $(r_t + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t))$ can be interpreted as the error when updating the current Q-value.

A difficulty when applying Q-learning (and classic RL algorithms in general) is the size of the table needed to store the Q-values for every state-action pair. The size of the table becomes prohibitively large in problems such as VNE, where the number of state-action pairs is combinatorial. In DRL, instead of a table, a deep neural network (DNN) is used to efficiently approximate $Q(s_t,a_t)$ with the DNN's output $Q_\theta(s_t,a_t)$, where θ denotes the parameter vector of the neural network. Then, we can define an *error* function L_θ for updating the neural network's parameter vector θ as follows:

$$L_\theta = \mathbb{E}\left[r_t + \gamma \max_a Q_\theta(s_{t+1},a) - Q_\theta(s_t,a_t)\right], \tag{6}$$

where, the expectation is taken over the set of observed (s_t,a_t,r_t,s_{t+1})'s. Note that, L_θ is a differentiable function of DNN parameters, thus its gradients can be used by the stochastic gradient descent algorithm to minimize L_θ, and consequently train the DNN.

B. DRL Approach

The architecture of our DRL algorithm called DeepViNE for solving the VNE problem is presented in Fig. 1. When a new VN arrives, an image is constructed that encodes information about the physical resources and mapping of the current VN, *i.e.*, the image represents the state of the problem. The image is then fed to a deep neural network that selects an action which either i) marks a pair of virtual and physical nodes, or ii) embeds a previously marked virtual node into a physical node. As the result of selecting an action, the image is updated and is fed back to the neural network. This process continues until all virtual nodes of the VN are embedded or the number of iterations exceeds a pre-defined threshold. In each iteration, a reward value is revealed that shows the suitability of the selected action, and is used to train the neural network such that, in the future, it selects the actions that result in the most efficient mapping of VNs.

DeepViNE is presented in Algorithm 1. To reduce the number of actions, we define two *pointers* in lines 1 and 2 that mark a physical node and a virtual node at any moment. We call the pointers physical and virtual pointers, respectively. In Section III-C, we explain how to encode these

two pointers along with other information as an image, which is used in line 4. The procedures of selecting the actions from the DNN, exploring the solution space, and sequentially embedding virtual nodes of the VN are outlined in lines 7 to 25. The architecture of the employed DNN is described in Section III-F. Section III-D elaborates on how different actions manipulate the image and pointers, while Algorithm 2 describes the exact steps of the manipulation process. The rationale behind the reward values, in lines 19 to 24, and the training method, in lines 26 and 32, are discussed in Sections III-E and III-F, respectively.

C. Image Representation

Each image can be considered as a number of matrices, usually referred to as *channels*, where each channel represents the image from a specific viewpoint, *e.g.*, an RGB image has three channels to represent Red, Green, and Blue. In our approach, we use three channels to encode the image representing the VNE problem. Each channel is the concatenation of the PN and VN representations (see Fig. 1) with an emphasis on a special aspect of the system. For example, we use one channel to create a significant contrast between representations of those resources that have sufficient capacity to embed a VN and those that do not. This channel helps the DNN detect suitable physical resources more efficiently. Without loss of generality, we assume that the physical and virtual networks have grid topologies. We note that any planar graph can be transformed into a grid in linear time [18].

Algorithm 1 DeepViNE

Input: $G^p = (\mathcal{N}^p, \mathcal{L}^p)$, $G^v = (\mathcal{N}^v, \mathcal{L}^v)$, DNN, mem

1: $\hat{n}_0^p \leftarrow$ random node in $\mathcal{N}^p$ ▷ Physical pointer
2: $\hat{n}_0^v \leftarrow$ random node in $\mathcal{N}^v$ ▷ Virtual pointer
3: $\mathcal{E} \leftarrow \{\}$ ▷ Set of embedded virtual nodes
4: img $\leftarrow$ create_representation(G^p, G^v, $\hat{n}_0^p$, $\hat{n}_0^v$)
5: **for** $t \in \{0, \ldots,$ max_iteration$\}$ **do**
6: reward $\leftarrow 0$
7: a $\leftarrow \epsilon$-greedy(DNN, img)
8: **if** a $\in \{a_1, \ldots, a_4\}$ **then**
9: $\hat{n}_{t+1}^p \leftarrow$ update_pointer($\hat{n}_t^p$, a)
10: img$' \leftarrow$ update_representation($\hat{n}_t^p$, $\hat{n}_t^v$, $\hat{n}_{t+1}^p$, $\hat{n}_{t+1}^v$, $\mathcal{E}$)
11: **else if** a $\in \{a_5, \ldots, a_8\}$ **then**
12: $\hat{n}_{t+1}^v \leftarrow$ update_pointer($\hat{n}_t^v$, a)
13: img$' \leftarrow$ update_representation($\hat{n}_t^p$, $\hat{n}_t^v$, $\hat{n}_{t+1}^p$, $\hat{n}_{t+1}^v$, $\mathcal{E}$)
14: **else if** no negative value in img **then**
15: $\mathcal{E}$.add($\hat{n}_t^v$)
16: $f(\hat{n}_t^v) \leftarrow \hat{n}_t^p$ ▷ $f()$ is defined in Section II
17: img$' \leftarrow$ update_representation($\hat{n}_t^p$, $\hat{n}_t^v$, $\hat{n}_{t+1}^p$, $\hat{n}_{t+1}^v$, $\mathcal{E}$)
18: **if** All virtual nodes embedded **then**
19: reward = 1
20: **else**
21: reward = 0.01
22: **end if**
23: **else**
24: reward = -0.01
25: **end if**
26: mem.add(img, a, img$'$, reward)
27: img $\leftarrow$ img$'$
28: **if** $\mathcal{E} = \mathcal{N}^v$ **or** reward < 0 **then**
29: break
30: **end if**
31: **end for**
32: DNN $\leftarrow$ train(DNN, mem) ▷ Train DNN using the experience memory

Next, we describe the encoding of the three image channels used in DeepViNE to encode the VNR problem state.

Channel 1. First, we describe the encoding of a VN. To this end, each virtual node is assigned a unique identifier, which is encoded using a one-hot scheme, *i.e.*, the encoding of node n_i^v is a binary sequence of length M with a single 1 at the i^{th} place. See Fig. 2(a) for an example VN and its encoding. Notice how the data is arranged in rows and columns of a matrix that closely resembles the grid topology of the virtual network. The blank entries of the matrix are filled with appropriate number of 0's such that the dimensions of the matrix are correct. Each virtual node n_i^v is represented by four items: 1) its CPU demand (*i.e.*, CPU(n_i^v)), 2) its unique identifier, 3) a status variable which is equal to 1 if n_i^v is marked by the virtual pointer, and is equal to 0 otherwise, and 4) an indicator variable which is equal to 1 if n_i^v is embedded in a physical node, and is equal to 0 otherwise. Each virtual link is represented by its bandwidth demand (*i.e.*, BW($\ell_{i,j}^v$)). Assuming that a float number is 64 bits long, the number of bits required to store the matrix is $O(64(2M^2 + 8M))$.

Similar to the VN representation, the physical network is organized as a grid. Fig. 2(b) shows an example physical network and its corresponding encoding. Each link is represented by its available bandwidth (*i.e.*, $\overline{\text{BW}}(\ell_{i,j}^p)$) as a single entry in the corresponding matrix of Channel 1. However, each physical node n_i^p is represented by three items: 1) available CPU (*i.e.*, $\overline{\text{CPU}}(n_i^p)$), 2) an indicator variable that is set to 1 if the node is marked by the physical pointer, and 0 otherwise, and 3) a sequence of 0s and 1s that specifies the virtual nodes embedded in that physical node. Specifically, this sequence is the logical OR of the virtual node identifiers that are embedded in n_i^p. Recall that the virtual node identifiers are in one-hot scheme, and thus their logical OR uniquely specifies the individual embedded virtual nodes. Similarly, the number of bits required to store the matrix of the physical network representation is $O(64(2K^2 + 6K))$ bits.

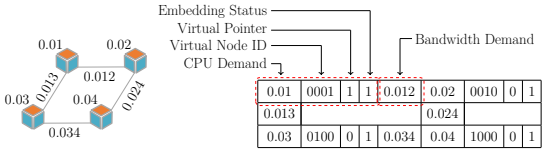
Channel 1 is constructed by concatenating these two representations horizontally. Note that, we need to add appropriate number of 0s to the VN representations such that the number of rows of both representations becomes equal.

Channel 2. This channel is designed to help the RL agent identify the nodes and links that are not embedded yet, which eventually increases the learning speed of the algorithm. The construction of Channel 2 is identical to that of Channel 1, except for when a virtual node or link is embedded. In that case, we replace its corresponding encoding in the VN representation with an array of appropriate number of 1's.

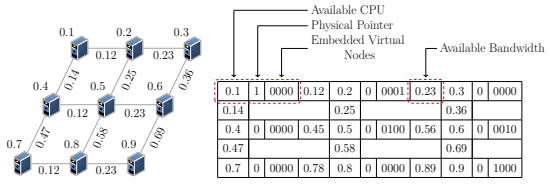
Channel 3. This channel helps the RL agent to quickly find the physical nodes and links with sufficient capacities. We construct Channel 3 in the same way as Channel 1, except for the physical nodes and links whose remaining capacities are not enough to embed any virtual node and link, respectively. These physical nodes and links are encoded by arrays of 0's with appropriate lengths. Otherwise, an array of 1's encodes a physical node or link.

D. Actions

Let $\hat{n}_t^v$ and $\hat{n}_t^p$ denote the virtual and physical nodes that are marked by the physical and virtual pointers in iteration t, respectively. Remember that in the grid topology each node



(a) Encoding of a virtual network. The numbers on nodes and links show the corresponding CPU and BW demands.



(b) Encoding of the physical network. The numbers on nodes and links show the corresponding CPU and BW capacities.

Fig. 2: Encoding scheme in Channel 1.

has at most 4 neighbours. Thus we define 4 actions to change $\hat{n}_t^v$ to one of its immediate neighbours. Similarly, we define 4 actions to update $\hat{n}_t^p$, and 1 action to embed $\hat{n}_t^v$ in $\hat{n}_t^p$.

Assume that a pointer is updated and the current values are $\hat{n}_{t+1}^p$ and $\hat{n}_{t+1}^v$ (one of them is the same as in the previous iteration), which leads to the following changes in the image representation (See Algorithm 2 for more details):

1) The available CPU capacity of $\hat{n}_{t+1}^p$ is set to $\overline{\text{CPU}}(\hat{n}_{t+1}^p) - \text{CPU}(\hat{n}_{t+1}^v)$. (line 1 in Algorithm 2)
2) The shortest paths from $\hat{n}_{t+1}^p$ to other physical nodes that embed a neighbour of $\hat{n}_{t+1}^v$ are computed, and the available bandwidth on each of these paths is updated with the bandwidth demand between $\hat{n}_{t+1}^p$ and the corresponding neighbor. (lines 2–9 in Algorithm 2)
3) The allocated resources due to the previous positions of the pointers are released. (lines 10–18 in Algorithm 2)

Algorithm 2 update_representation

Input: $\hat{n}_t^p$, $\hat{n}_t^v$, $\hat{n}_{t+1}^p$, $\hat{n}_{t+1}^v$, $\mathcal{E}$ ▷ $\mathcal{E}$ is the set of embedded virtual nodes

1: $\overline{\text{CPU}}(\hat{n}_{t+1}^p) \leftarrow \overline{\text{CPU}}(\hat{n}_{t+1}^p) - \text{CPU}(\hat{n}_{t+1}^v)$
2: $\mathcal{E}^n \leftarrow$ neighbours of $\hat{n}_{t+1}^v$ that are in $\mathcal{E}$
3: **for** $n_j^v \in \mathcal{E}^n$ **do**
4: path $\leftarrow$ shortest_path($f(\hat{n}_{t+1}^v)$, $f(n_j^v)$)
5: **for** $\ell^p \in$ path **do**
6: $\ell_{i,j}^v \leftarrow$ link between $\hat{n}_{t+1}^v$ and n_j^v
7: $\overline{\text{BW}}(\ell^p) \leftarrow \overline{\text{BW}}(\ell^p) - \text{BW}(\ell_{i,j}^v)$
8: **end for**
9: **end for**
10: $\overline{\text{CPU}}(\hat{n}_t^p) \leftarrow \overline{\text{CPU}}(\hat{n}_t^p) + \text{CPU}(\hat{n}_t^v)$
11: $\mathcal{E}^n \leftarrow$ neighbours of $\hat{n}_t^v$ that are in $\mathcal{E}$
12: **for** $n_j^v \in \mathcal{E}^n$ **do**
13: path $\leftarrow$ shortest_path($f(\hat{n}_t^v)$, $f(n_j^v)$)
14: **for** $\ell^p \in$ path **do**
15: $\ell_{i,j}^v \leftarrow$ link between $\hat{n}_t^v$ and n_j^v
16: $\overline{\text{BW}}(\ell^p) \leftarrow \overline{\text{BW}}(\ell^p) + \text{BW}(\ell_{i,j}^v)$
17: **end for**
18: **end for**
19: update *embedding status* and *embedded virtual nodes* of physical nodes based on $\mathcal{E}$ and $f(\cdot)$

Note that, we allow the agent to move the pointers arbitrarily. If it selects a physical node with insufficient capacity, the available capacity on that node actually becomes negative. In this case, if the agent tries to embed $\hat{n}_t^v$ in $\hat{n}_t^p$, the iteration is terminated, the algorithm fails and the VN is *blocked*. However, upon successful selection of action a_9, the status of $\hat{n}_t^v$ is changed to 1, and the binary sequence in the representation of $\hat{n}_t^p$ is updated by computing its logical OR with the identifier of $\hat{n}_t^v$ (See line 19). Also, whenever a virtual link or node is embedded, its representation is replaced with a series of 1's in Channel 2, and whenever the available capacity of a resource becomes negative a sequence of 0's is used for its representation in Channel 3.

We use ϵ-greedy strategy to choose an action: in each iteration, the action with highest Q-value is selected with probability $1 - \epsilon$, or an action is selected randomly with probability ϵ. In the beginning, we set $\epsilon = 1$, which allows the agent to explore different state-action pairs, then, linearly decrease its value to 0 after $50,000$ iterations.

E. Reward Signal

The reward signal is designed to encourage the RL agent to embed as many VNs as possible. To achieve this, whenever a VN is completely embedded, the agent receives 1 unit of reward. In the intermediate steps, whenever the agent embeds a virtual node in a physical node, if the processing and bandwidth constraints are met, it receives 0.01 unit of reward. This small reward signals to the agent that embedding a virtual node is a desirable action. However, it is far less than the reward for embedding a full VN, which is the ultimate goal of the agent. If the agent's choice violates the resource constraints, a reward of -0.01 unit is given and the episode is *terminated*. The reward of moving the virtual and physical pointers is set to 0, which allows the agent to explore and select the best embedding option possible. Clearly, the cumulative reward is maximized when the maximum number of VNs are embedded.

F. DNN Construction

We use a deep-neural network depicted in Fig. 3 to approximate the Q-values. This neural network has four convolution layers, where the number of neurons in the input layer depends on the size of the physical and virtual networks. The number of kernels and their sizes are selected such that the size of the last convolutional layer becomes equal to 512. We apply the dueling technique, which is observed to help learning better policies [19], to divide the output of last convolutional layer between two fully connected layers and then merge them in the last output. We store the observed state-action-rewards in a

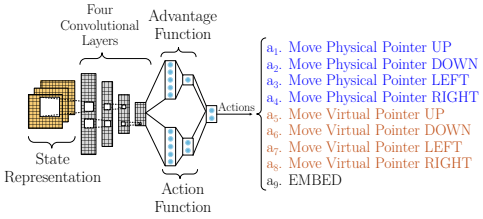
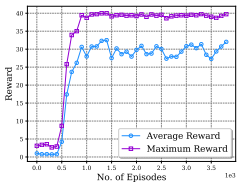
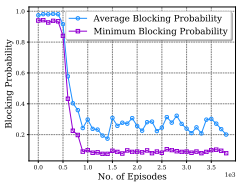


Fig. 3: DNN architecture and actions.



(a) Received reward.



(b) Blocking probability.

Fig. 4: Convergence behaviour of DeepViNE.

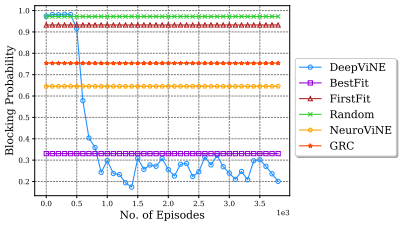


Fig. 5: Blocking probability of different algorithms.

memory, called the *experience memory*, of size $50,000$ entries and then train the DNN with randomly selected batches of size 32, by applying the method described in [10].

IV. Performance Evaluation

In this section, we present simulation results to study the convergence and performance of DeepViNE.

A. Simulation Setup

System Parameters. The simulated physical network is a grid with 25 nodes. Each virtual network is also represented as a grid with 9 nodes. The capacity of physical resources (*i.e.*, CPU and link bandwidth) is chosen randomly from the interval $[50, 100]$, while the virtual resource demands are generated randomly over the interval $[1, 10]$. We set $\xi_r = \xi_c = 1$. The system operates in an episodic manner, where each episode is a sequence of 800 iterations of action selection. In each episode, VNs arrive at the system sequentially and no information about the future VNs is available to the algorithms. Once a VN is fully processed (*i.e.*, embedded or blocked), after a number of iterations, the next VN arrives at the system. In our simulations, on average, 40 VNs arrive at the system in each episode. At the end of each episode, all VNs leave the system. Each simulation experiment is run for 4000 episodes.

Implemented Algorithms. In addition to DeepViNE, we have implemented three other agents:

- **Random** agent that selects actions randomly,
- **FirstFit** agent that embeds virtual nodes in the first physical node with sufficient capacity,
- **BestFit** agent that chooses the physical node with the maximum CPU capacity and packs it with virtual nodes.

Furthermore, we present the comparison of DeepViNE with two existing VNE algorithms:

- **GRC** [4]: a node-ranking-based greedy algorithm.
- **NeuroViNE** [8]: an RL algorithm that uses a Hopfield network to pre-select candidate physical nodes, and then uses GRC to embed VNs in the candidate physical nodes.

All the algorithms are implemented such that they allow multiple virtual nodes to be embedded on the same physical node. We used Python 2.7 and Tensorflow 1.9.0 [20] on Ubuntu 16.04 to implement the algorithms and DNN, respectively.

B. Learning Convergence

Fig. 4(a) plots the evolution of the "average" as well as "maximum" reward received by DeepViNE over a window of size 100 episodes. We observe that, at the beginning, until episode 500, the agent is in the exploration phase and its performance is poor. However, the agent gradually learns how to efficiently embed VNs. After 1200 episodes, its performance becomes stable. Specifically, after convergence, the agent consistently collects about 30 units of reward in each episode.

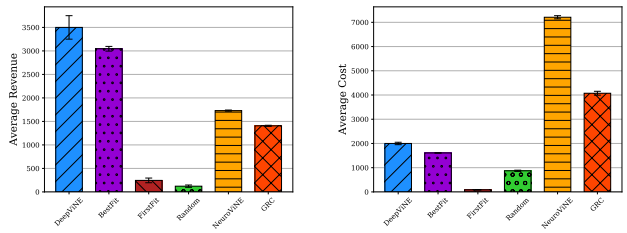
Fig. 4(b) shows the "average" as well as "minimum" VN blocking probability. The blocking probability refers to the probability of failing to embed a VN, and is computed over a window of size 100 episodes. We observe that the blocking probability closely follows the behaviour of the reward signal, which confirms that the designed reward signal is suitable for achieving the objective of minimizing the VN blocking probability. We also observe that in some episodes, the blocking probability is as low as 0.08%, which means the algorithm successfully embeds 37 VNs, which is close to the total number of VNs that arrive in a single episode (*i.e.*, 40).

C. Performance Comparison

Blocking Probability. Fig. 5 compares DeepViNE with the other algorithms in terms of the blocking probability. We see that in the beginning (*i.e.*, exploration phase) the performance of DeepViNE is similar to Random, which is worse than all other algorithms. However, after about 1200 episodes, where the learning completes, DeepViNE outperforms all other algorithms. Specifically, we see that DeepViNE is always better than BestFit and can reduce the blocking probability by up to 22%, while on average its performance is 11% better. This means that DeepViNE not only learns to better embed virtual nodes but also considers higher level metrics, such as the distance between the nodes with highest remaining capacity, to achieve better performance compared to BestFit. Interestingly, FirstFit performs only marginally better than Random, which indicates that it is crucial to select physical resources strategically in order to achieve a long-term goal such as minimizing the blocking probability.

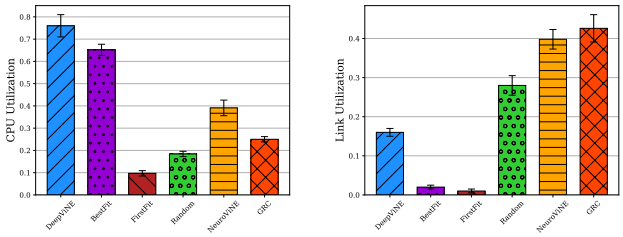
GRC considers the remaining capacity of physical nodes and links, and thus spreads the load with the objective of embedding more VNs in the future. However, it is evident that, at least in the simulated scenarios, this strategy is not efficient. NeuroViNE achieves a higher performance compared to GRC by employing a Hopfield network to intelligently choose a good set of candidate physical nodes for embedding. Nevertheless, NeuroViNE uses GRC to complete embedding. Thus, its performance is influenced by the limitations of GRC. Both algorithms perform worse than BestFit and DeepViNE.

Revenue and Cost. Figs. 6(a) and 6(b), respectively, compare DeepViNE with other algorithms in terms of their achieved average Revenue and Cost, as given by (3) and (4). Since



(a) Average revenue. (b) Average cost.

Fig. 6: Average revenue and cost.



(a) Average CPU utilization. (b) Average link utilization.

Fig. 7: Utilization of physical resources.

DeepViNE is more successful in embedding VNs in each episode, its revenue is also significantly higher compared to other algorithms. The closest competitor is BestFit, whose revenue nevertheless is significantly lower than that of DeepViNE, *i.e.*, by about 16%. BestFit achieves the least cost among the algorithms, because it tightly packs virtual nodes on physical nodes and thus avoids paying for connecting virtual nodes that are embedded in a single physical node. The cost incurred by DeepViNE is higher, which is justified as it also embeds more VNs. We see that the costs of GRC and NeuroViNE are significantly higher, which implies that they incur a higher cost for embedding virtual links (*i.e.*, they construct node embeddings that use longer paths to interconnect virtual nodes).

Resource Utilization. Average CPU and physical link utilization are depicted in Fig. 7. Specifically, Fig. 7(a) shows that DeepViNE achieves the highest CPU utilization, which confirms that it is the best algorithm for packing VNs, even better than BestFit. The achieved CPU utilization of NeuroViNE is below that of DeepViNE and BestFit. In particular, NeuroViNE's CPU utilization is almost half of that of DeepViNE. Fig. 7(b) depicts the average physical link utilization, which depends on the number of VNs that are successfully embedded in the physical network, and the mapping of virtual links to physical paths. We observe that, although NeuroViNE and GRC have higher VN block probabilities compared to DeepViNE, their link utilizations are significantly higher (almost 2x). This indicates that the virtual link mapping operations in NeuroViNE and GRC are not efficient. BestFit and FirstFit achieve very low link utilizations because they focus on packing virtual nodes in physical nodes and ignore the importance of node interconnections. However, DeepViNE considers the physical link and node resources simultaneously, which eventually leads to higher CPU and link utilization, but also lower blocking probability.

V. Conclusion

In this work, we presented the design and evaluation of an adaptive online virtual network embedding algorithm based on deep reinforcement learning. A key feature of our design is that it does not rely on hand-crafted features, rather automates feature extraction via a suitable encoding of the problem. We evaluated the convergence and performance of our algorithm using simulations, and compared it with several existing algorithms. The results showed that our algorithm learns an embedding policy that improves upon the performance of other algorithms by at least 11%. An interesting extension of this work is to modify the reward signal to model other system objectives such as minimizing energy consumption of the physical network.

References

[1] A. Fischer, J. F. Botero, M. T. Beck, H. De Meer *et al.*, "Virtual network embedding: A survey," *IEEE Commun. Surv. Tutor.*, vol. 15, no. 4, 2013.

[2] Z. Zhang, A. X. Liu, X. Cheng, Y. Wang *et al.*, "Energy-aware virtual network embedding," *IEEE/ACM Trans. Netw.*, vol. 22, no. 5, 2014.

[3] S. Zhang, Z. Qian, J. Wu, S. Lu, and L. Epstein, "Virtual network embedding with opportunistic resource sharing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 3, 2014.

[4] L. Gong, Y. Wen, Z. Zhu, and T. Lee, "Toward profit-seeking virtual network embedding algorithm via global resource capacity," in *Proc. IEEE INFOCOM*, 2014.

[5] M. Rost and S. Schmid, "Virtual network embedding approximations: Leveraging randomized rounding," *arXiv:1803.03622*, 2018.

[6] G. Even, M. Medina, G. Schaffrath, and S. Schmid, "Competitive and deterministic embeddings of virtual networks," *Theor. Comput. Sci.*, vol. 496, 2013.

[7] R. Boutaba, M. A. Salahuddin, N. Limam, and S. o. Ayoubi, "A comprehensive survey on machine learning for networking: evolution, applications and research opportunities," *Journal of Internet Services and Applications*, vol. 9, no. 1, 2018.

[8] A. Blenk, P. Kalmbach, J. Zerwas, M. Jarschel *et al.*, "NeuroViNE: A neural preprocessor for your virtual network embedding algorithm," in *Proc. IEEE INFOCOM*, 2018.

[9] H. Yao *et al.*, "A novel reinforcement learning algorithm for virtual network embedding," *Neurocomputing*, vol. 284, 2018.

[10] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, 2015.

[11] R. Mijumbi, J. Gorricho, J. Serrat, M. Claeys *et al.*, "Design and evaluation of learning algorithms for dynamic resource management in virtual networks," in *Proc. IEEE NOMS*, 2014.

[12] ——, "Neural network-based autonomous allocation of resources in virtual networks," in *Proc. IEEE EuCNC*, 2014.

[13] H. Zhang, X. Zheng, J. Tian, and Q. Xue, "A virtual network embedding algorithm based on RBF neural network," in *Proc. IEEE CSE*, 2017.

[14] A. Blenk, P. Kalmbach, P. Van Der Smagt, and W. Kellerer, "Boost online virtual network embedding: Using neural networks for admission control," in *Proc. CNSM*, 2016.

[15] H. Yao, B. Zhang, P. Zhang, S. Wu *et al.*, "RDAM: A reinforcement learning based dynamic attribute matrix representation for virtual network embedding," *IEEE Trans. Emerg. Topics Comput.*, 2018.

[16] O. Abdel-Hamid, A.-r. Mohamed, H. Jiang, L. Deng *et al.*, "Convolutional neural networks for speech recognition," *IEEE/ACM Trans. Audio, Speech, Language Process.*, vol. 22, no. 10, 2014.

[17] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proc. ACM HotNets*, 2016.

[18] W. Schnyder, "Embedding planar graphs on the grid," in *Proc. ACM-SIAM SODA*, 1990.

[19] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt *et al.*, "Dueling network architectures for deep reinforcement learning," in *Proc. ICML*, 2016.

[20] M. Abadi, P. Barham, J. Chen, Z. Chen *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proc. OSDI*, 2016.